HEWLETT PACKARD

# Domain Pascal Language Reference

# Domain Pascal Language Reference

Order No. 000792–A01

Apollo Systems Division
A subsidiary of
HEWLETT
PACKARD

# Preface

The *Domain Pascal Language Reference* explains how to code, compile, bind, and execute Domain Pascal programs.

We've organized this manual as follows:

| | |
|---|---|
| **Chapter 1** | Introduces Domain Pascal and provides an overview of its extensions. |
| **Chapter 2** | Defines Domain Pascal building blocks (like the length of an identifier) and describes the structure of the main program. |
| **Chapter 3** | Explains all the Domain Pascal data types. |
| **Chapter 4** | Contains alphabetized listings describing all the functions, procedures, statements, and operators that you can use in the code portion of a program. |
| **Chapter 5** | Explains how to declare and call procedures and functions. |
| **Chapter 6** | Details compiling, binding, debugging, and executing. |
| **Chapter 7** | Describes how you can break your program into two or more separately compiled modules (which can be in Domain Pascal, Domain FORTRAN, or Domain/C). |
| **Chapter 8** | Contains an overview of the I/O resources available to Domain Pascal programmers. |
| **Chapter 9** | Covers compiler diagnostic messages and how to handle them. |
| **Appendix A** | Contains a table of Domain Pascal reserved words and predeclared identifiers. |
| **Appendix B** | Contains an ISO Latin-1 table that includes ASCII characters. |
| **Appendix C** | Describes Domain Pascal's extensions to ISO/ANSI standard Pascal. |
| **Appendix D** | Describes Domain Pascal's deviations from ISO/ANSI standard Pascal. |

| | |
|---|---|
| **Appendix E** | Describes built-in routines available for systems programmers. |
| **Appendix F** | Describes how to obtain the best floating-point performance on MC68040-based Domain workstations. |

## Audience

We wrote this manual to serve programmers at a variety of levels of Pascal expertise. Our goal is to keep the writing as simple as possible, but to assume that you know the fundamentals of Pascal programming. If you are totally inexperienced in a block-structured language like Pascal or PL/I, you probably should study a Pascal tutorial before using this manual. If you have a little experience with a block-structured language, you will probably benefit most by experimenting with the many examples we provide (particularly in Chapter 4). If you are an expert Pascal programmer, turn to Appendix C first for a list of our extensions to standard Pascal.

## Summary of Technical Changes

This manual describes Version 8.8 of the Domain Pascal compiler. The last update to the Domain Pascal manual was at SR10. The following list summarizes the features added to Domain Pascal since SR10:

- The introduction of new Boolean operators **and then** and **or else**
- Modified operator precedence rules (to accommodate the **and then** and **or else** Boolean operators)
- The following new compiler directives:
  - **%begin_inline** and **%end_inline**
  - **%push_alignment** and **%pop_alignment**
- The following new compiler options:
  - **-bounds_violation** and **-no_bounds_violation**
  - **-compress** and **-ncompress**
  - The **-cpu** arguments **mathlib_sr10**, **mathlib**, **mathchip**, **fpa1**, **a88k**, and **m68k**
  - **-nclines**
  - **-prasm** and **-nprasm**

- Compatibility with NFS (Network File System)
- Larger maximum set size
- Enforcement of name compatibility when assigning the address of an actual routine to a procedure or function pointer

Change bars in the margin indicate technical changes since the last revision of this manual. Because Appendix F is completely new with this revision, it does not have change bars.

## 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) for a complete list of related documents. For more information on topics related to the Domain Pascal compiler, refer to the following documents:

- *Getting Started with Domain/OS* (002348) explains the fundamentals of the Domain system.
- The *Domain/OS Call Reference, Volume 1* (007196) and *Volume 2* (012888) describe the system service routines provided by the operating system and explain how to call these routines from user programs.
- *Programming with Domain/OS Calls* (005506) covers writing Domain Pascal programs that use stream calls and many other important system calls.
- The *Domain/C Language Reference* (002093) describes the Domain implementation of the C language.
- The *Domain FORTRAN Language Reference* (000530) describes the Domain implementation of FORTRAN.
- The *Domain/C++ Programmer's Guide* (017874), the *AT&T C++ Language System* (017823), and the *C++ Primer* by Stanley Lippman (017997) describe the Domain implementation of C++.
- The *Domain Floating-Point Guide* (015853) describes floating-point calculations on Domain systems.

- The *Aegis Command Reference* (002547) describes the Aegis environment.
- The *BSD Command Reference* (005800) describes the BSD environment.
- The *SysV Command Reference* (005798) describes the SysV environment.
- The *Domain/OS Programming Environment Reference* (011010) describes how to use the **bind** utility to link object modules and the librarian utility to create library files.
- The *BSD Unix Programmer's Manual* (017272) and the *SysV Programmer's Guide, Volume II* (017625) describe the **make** and **dbx** utilities and the Source Code Control System (SCCS).
- The *Domain Distributed Debugging Environment Reference* (011024) describes the high-level language debugger.
- The *Domain Software Engineering Environment (DSEE) Reference* (003016), *Getting Started with DSEE* (08788), and *Engineering in the DSEE Environment* (008790) describe the DSEE product.
- The *Domain/Dialogue User's Guide* (004299) describes the Domain/Dialogue product.
- The *Open Dialogue Reference* (012807), *Creating User Interfaces with Open Dialogue* (011167), the *MOTIF Style Guide* (017153), and *Customizing Open Dialogue* (011166) describe the Open Dialogue product.
- *Analyzing Program Performance with Domain/PAK* (008096) describes the Domain Performance Analysis Kit.
- *Using NFS on the Domain Network* (010414) describes the use of the Network File System (NFS).

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

## Pascal Tutorials

- Jensen, K. and N. Wirth, revised by Mickel, A. and J. Miner. *Pascal User Manual and Report.* Third Edition. Springer-Verlag, New York: 1985.
- Grogono, Peter. *Programming in Pascal.* Revised Edition. Reading, Massachusetts: Addison-Wesley, 1980.
- Cooper, D., and M. Clancy. *Oh! Pascal!* New York: WW Norton, 1982.

## Does This Manual Support Your Software?

This manual was released with Version 8.8 of Domain Pascal. Domain Pascal Version 8.8 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, then type

```
/com/bldt
```

To check the version of Domain Pascal, type:

```
/com/pas -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. It has one of the following pathnames:

  **/install/doc/apollo/pas.v.***version_number***.m__notes**
  **/install/doc/apollo/pas.v.***version_number***.mpx__notes**
  **/install/doc/apollo/pas.v.***version_number***.p__notes**
  **/install/doc/apollo/pas.v.***version_number***.pmx__notes**

- 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 a question or problem 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 also use the Reader's Response Form at the back of this manual to submit comments about 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 Pascal appear in color. |
| `output` | `Information that the system displays appears in this typeface.` |
| $\Big[\quad\Big]$ | Large square brackets enclose optional items in formats and command descriptions. |
| [ ] | Regular sized square brackets in Pascal statements 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 F is completely new with this revision, it does not have change bars. |
| ——— ⊞ ——— | This symbol indicates the end of a chapter. |

——— ⊞ ———

# Contents

# Chapter 6 Program Development

# Chapter 8 Input and Output

# Chapter 9 Diagnostic Messages

## Figures

## Tables

# Chapter 1

# Introduction

You should be somewhat familiar with Pascal before attempting to use this manual. If you are not, please consult a good Pascal tutorial. (We've listed some good tutorials in the Preface.) If you are somewhat familiar with Pascal or if you are expert in a highly block-structured language such as PL/I, then you should be able to write programs in Domain Pascal after reading this manual.

## 1.1 A Sample Program

The best way to get started with Domain Pascal is to write, compile, and execute a simple program. Figure 1-1 shows a simple program that you can use to get started.

```
PROGRAM getting_started;
 {A simple program to try out.}

 VAR
  x : integer16;
  y : integer32;

 BEGIN
  write('Enter an integer -- ');
  readln(x);
  y := x * 2;
  writeln;
  writeln(y:1, ' is twice ', x:1);
 END.
```

*Figure 1-1. Sample Program*

Although you are welcome to type in this program yourself, it is also available through the **getpas** utility. The following section contains details about using **getpas** to run sample programs.

Suppose you store the program in the file **easy.pas**. (Although it is not required that the filename end with the **.pas** extension, we recommend its use so that you can readily identify Pascal source programs.)

To compile a program, simply enter the shell command **pas** followed by the filename. If you do use the **.pas** extension, you can include or omit that extension at this step. Domain Pascal doesn't care which way you type it. For example, to compile easy.pas, you can enter either of the following commands:

```
$ pas easy
```

or

```
$ pas easy.pas
```

The compiler creates an executable object in filename **easy.bin**. To execute this object you merely enter its name. For example:

```
$ easy.bin
 Enter an integer -- 15

 30 is twice 15
```

## 1.2 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 Pascal product. They illustrate features of the Domain Pascal language, and demonstrate programming with Domain graphics calls and system calls. You retrieve these online sample programs with the **getpas** utility.

### 1.2.1 What You Get When You Install Sample Programs

When you (or your system administrator) load the Domain Pascal product, the install procedure will ask if you want to install sample programs. We recommend that you answer "y" (for yes). If you answer "y", the install program will store the following three files in the **/domain_examples/pascal_examples** directory:

| | |
|---|---|
| **examples** | This is a master file containing all the sample Pascal programs. |
| **list_of_examples** | This is a help file describing all the sample Pascal programs. |
| **getpas** | This is the utility that retrieves the sample programs out of the examples file. |

Note that the sample programs take up a lot of disk space (> 600 Kbytes), so we recommend that you install the sample programs in only one site on the network and have users link to them.

## 1.2.2 Creating Links to the Sample Programs

If you want to be able to invoke **getpas** from any directory on the system, you must create the appropriate links. The links differ according to your type of environment. Also, before you can set up the links, you must find out the name of the disk on which the examples are stored, and use that name as *node* in the command lines shown below.

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

```
$ crl ~/com/getpas //node/domain_examples/pascal_examples/getpas
```

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

```
$ ln -s //node/domain_examples/pascal_examples/getpas a_dir/getpas
```

where *node* 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* is not your node, then you should also create one of the following links, depending on your operating system environment:

```
$ crl /domain_examples/pascal_examples //node/domain_examples/pascal_examples

$ ln -s //node/domain_examples/pascal_examples /domain_examples/pascal_examples
```

**NOTE:** Instead of creating links you can set your working directory to the *//node*/**domain_examples/pascal_examples** directory and invoke **getpas** from there.

## 1.2.3 Invoking getpas

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

There are two different ways to invoke **getpas**. The first, and simplest, way is to specify its name on any shell command line.

For example, in an Aegis environment:

```
$ getpas
```

After you invoke **getpas**, the utility will prompt you for the appropriate information.

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

```
$ getpas 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/pascal_programs/getting_started.pas**:

```
$ getpas getting_started //dolphin/pascal_programs/getting_started.pas
```

For full details on using **getpas**, issue the following command:

```
$ getpas -usage
```

## 1.3 Overview of Domain Pascal Extensions

Domain Pascal supports many extensions to ISO/ANSI standard Pascal. The purpose of this section is to provide an overview of these extensions. For a complete list of all the extensions, see Appendix C. All extensions to the standard are marked in color like this or are noted explicitly in text as an extension. For a list of omissions from standard Pascal, see Appendix D.

Naturally, the more you take advantage of Domain Pascal extensions, the less portable your code will be. Therefore, if you are very concerned with portability, you should avoid using the extensions.

### 1.3.1 Extensions to Program Organization

Chapter 2 describes the organization of a Domain Pascal program contained within one file. Following is an overview of the extensions described within the chapter. You can

- Specify an underscore (_) or dollar sign ($) in an identifier.
- Specify integers in any base from 2 to 16.
- Specify comments in three ways.
- Specify that the compiler assign the code or data in your program to nondefault named sections.
- Declare the **label**, **const**, **type**, and **var** declaration parts in any order.
- Declare **define** and **attribute** parts in addition to the standard declaration parts.
- Use constant expressions when declaring constants, as long as the components of the expressions are constants.
- Use both identifiers and integers as labels.

### 1.3.2 Extensions to Data Types

Chapter 3 describes the data types supported by Domain Pascal. Domain Pascal supports the following extensions that allow you to

- Specify two additional pointer types. The first is a special pointer to procedures and functions. The second is a universal pointer type that will hold a pointer to a variable of any data type.
- Initialize static variables in the **var** declaration part of your program.
- Group variables into named sections for better run-time performance.
- Specify variable and type attributes that let you better control compiler optimization and data layout.
- Embed unprintable characters in string constants.
- Create variable-length strings.
- Specify two additional record types—**aligned record** and **unaligned record**. You can use the first type to make sure that records are always naturally aligned, and you can use the second to make sure that records are always aligned on word boundaries.

### 1.3.3 Extensions to Code

Chapter 4 describes the action portion of your program. Domain Pascal supports the following extensions to executable statements:

- An **addr** function that returns the address of a specified variable
- An **align** function that returns a correctly aligned copy of an expression passed as a routine parameter
- An **append** procedure for concatenating strings
- Bit operators or functions for bitwise *and*, *not*, *or*, and *exclusive or* operations
- Three bit-shift functions (**rshft, arshft,** and **lshft**)
- Many compiler directives that enable features like **include** files and conditional compilation
- A **ctop** procedure for converting a C-style string into a Domain Pascal variable-length string
- A **discard** procedure for explicitly discarding an expression's value and so suppressing some compiler optimizations

- An **exit** statement for jumping out of the current loop
- An exponentiation operator (**)
- A **find** procedure for locating a specified element in a file
- A **firstof** and a **lastof** function for returning the first and last possible value of a specified data type
- Some additional capabilities for the **if** statement
- An **in_range** function for determining whether a specified value is within the defined range of an integer subrange or enumerated type
- A **max** and a **min** function for finding the greater and lesser of two specified expressions
- A **next** statement for skipping over the current iteration of a loop
- An **open** procedure for opening files and a **close** procedure for closing files
- A **ptoc** procedure for converting a Domain Pascal variable-length string into a C-style string
- A **replace** procedure that allows you to modify an existing element in a file
- A **return** statement for causing a premature return to a calling procedure or function
- A **sizeof** function for returning the size (in bytes) that a specified data type requires in storage
- A **substr** function for extracting a substring from a string
- Additional type transfer functions that transform the data type of a variable or expression into some other data type
- Some additional capabilities for the **with** statement

### 1.3.4 Extensions to Routines

Chapter 5 describes procedures and functions (routines). This chapter documents extensions that allow you to:

- Specify the direction of parameter passing with the special **in**, **out**, and **in out** keywords.
- Use the **univ** keyword to suppress parameter type checking.
- Specify routine attribute clauses, routine options clauses, and a routine option declaration part to control how the compiler processes a routine.
- Specify argument lists on both the **forward** declaration and the routine heading of routines declared past the calling statement.

### 1.3.5 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 a Pascal implementation; that is, there is no standard for these tools.

### 1.3.6 External Routines and Cross–Language Communication

Chapter 7 explains how to write a program that accesses code or data written in another separately compiled module or library. It also describes how to access routines written in Domain FORTRAN or Domain/C. The entire chapter describes features that are implementation–dependent.

### 1.3.7 Extensions to I/O

Chapter 8 describes input and output from a Domain Pascal programmer's point of view. Domain Pascal supports all the standard I/O procedures. In addition, it supports the **open**, **close**, **find**, and **replace** procedures. As a further extension to the standard, Domain Pascal permits you to access the operating system's I/O and formatting system calls.

### 1.3.8 Diagnostic Messages

Chapter 9 lists compile–time and run–time messages and explains how to deal with them. Diagnostic messages are an implementation–dependent feature of Pascal.

# Chapter 2

# Blueprint of a Program

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

## 2.1 Building Blocks of Domain Pascal

In this section we describe the basic building blocks or elements of the Domain implementation of Pascal. We define identifiers, integers, real numbers, comments, and strings, and we explore case-sensitivity and spreading source code across multiple lines.

### 2.1.1 Identifiers

In this manual, the term "identifier" refers to any sequence of characters that meets the following criteria:

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

  A...Z and a...z (ASCII values 65 through 90 and 97 through 122)
  0...9
  _ (underscore)
  $ (dollar sign)

Identifiers are case-insensitive. An identifier can have up to 4096 characters.

## 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. (See the "Integers" section in Chapter 3 for a description of the range of various integer data types.)

An unsigned integer must begin with a digit. Each successive character must be a digit.

Note that Pascal prohibits two consecutive mathematical operators. If you want to divide 9 by –3, you might be tempted to use the following expression:

```
9 DIV -3          {WRONG!}
```

However, this produces an error, since Pascal interprets the negative sign as the subtraction operator (and that makes two mathematical operators in a row). Where the sign of an integer can be confused with an addition or subtraction operator, enclose the integer within parentheses. Thus, the correct expression for 9 divided by –3 is:

```
9 DIV (-3)        {RIGHT!}
```

Pascal assumes a default of base 10 for integers. If you want to express an integer in another base, use the following syntax:

*base#value*

For *base*, enter an integer from 2 to 16. For *value,* enter any integer within that base. If the *base* is greater than 10, use the letters A through F (or a through f) to represent digits with the values 10 through 15.

For example, consider the following declarations of integer constants:

```
half_life := 5260;      /* default     (base 10) */
hexagrams := 16#1c6;    /* hexadecimal (base 16) */
luck      := 2#10010;   /* binary      (base 2)  */
wheat     := 8#723;     /* octal       (base 8)  */
```

## 2.1.3 Real Numbers

Domain Pascal supports the standard Pascal definition of a real number literal, which is

*integer.unsigned_integer*E*integer*

In other words, a valid real number literal may contain a decimal point, but it doesn't have to. If the real number literal contains a decimal point, you must specify at least one digit to the left of the decimal point and at least one digit to the right of the decimal point.

To express expanded notation (powers of 10), use the letter e or E followed by the exponent; for example:

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

Compare the right and wrong way for writing decimals in your program:

```
.5     {wrong}
0.5    {right}
5E-1   {right}
```

Note that although using .5 in your source code causes an error at *compile* time, entering .5 as input data to a real variable does *not* cause an error at *run* time.

## 2.1.4 Comments

You can specify comments in any of the following three ways:

{ comment }
(* comment *)
"comment"

For example, here are three comments:

```
{ This is a comment. }
(* This is a comment. *)
 "This is a comment."
```

The spaces before and after the comment delimiters are for clarity only; you don't have to include these spaces.

> **NOTE:** If you use a compiler directive within comment delimiters you *cannot* use spaces. Also, surrounding a compiler directive by comment delimiters does not necessarily cause it to be treated as a comment by the compiler. This is because you can specify a compiler directive anywhere a comment is valid by specifying its name inside a comment or as a statement; see the listing for "Compiler Directives" in Chapter 4 for details.

Unlike standard Pascal, the comment delimiters of Domain Pascal must match. For example, a comment that starts with a left brace doesn't end until the compiler encounters a right brace. Therefore, you can nest comments, for example:

```
{ You can (*nest*) comments inside other comments. }
```

Standard Pascal does not permit nested comments. If you want to use unmatched comment delimiters, as standard Pascal allows, you must compile with the **-iso** option. (See Section 6.4.18 for details about this option.)

The Domain Pascal compiler ignores the text of the comment, and interprets the first matching delimiter as the end of the comment. Use quotation marks to set off comments only if you are converting existing applications to the Domain system. In all other programs, you should use either of the other two methods.

Note that Pascal comments can stretch across multiple lines; for example, the following is a valid comment:

```
{ This is a comment
  that stretches across
  multiple lines. }
```

> NOTE: You can use the **-comchk** compiler option (described in Chapter 6) to warn you if a new comment starts before an old one finishes. This option can help you find places where you forgot to close a comment.

## 2.1.5 Strings

We refer to strings throughout this manual. In Domain Pascal, a string is a sequence of characters. A string can be represented by a string literal, which is formed by enclosing a sequence of characters in single quotes (''). Strings differ from identifiers in that you can use any character within a string. Here are some sample strings:

```
'This is a string.'
'18'
'b[2~{q^%pl'
'can''t'
```

To include a single quote in a string, write the single quote twice; for example:

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

> NOTE: Within a string, Domain Pascal treats the comment delimiters as ordinary characters rather than as comment delimiters.

### 2.1.5.1 Embedding Special Characters in Strings

The Domain Pascal compiler allows you to embed any ISO Latin-1 character in a string by placing the decimal ISO Latin-1 value in parentheses. This is especially useful for embedding unprintable characters. (The ISO Latin-1 character set, described in Appendix B, includes the ASCII character set.)

For example, the following statements cause the compiler to output a tab (ISO Latin-1 value 9) following the text line:

```
CONST
     TAB = 9;
     .
     .
     .
     writeln('Print a tab: '(TAB));
```

In essence, the compiler concatenates two string literals, one designated by the normal single quotes, and the other designated by the new ISO Latin-1 code syntax. There is no limit to the number of special characters and strings that you can concatenate provided that the total number of characters is not greater than 1024 and that the first component is a string within single quotation marks. For example, all of the following definitions are legal:

```
CONST
     NUL = 0;
     LF = 10;
     CR = 13;
     S1 = 'newline'(CR)(LF);
     S2 = 'three carriage returns' (13)(CR)(13) 'followed by more text';
     S3 = 'A null-terminated string'(NUL);
```

However, it is illegal to begin a string with an ASCII code:

```
    S3 := (CR)(LF)'newline';  {ILLEGAL!}
```

The compiler supports an alternative syntax that allows you to specify more than one ISO Latin-1 code at a time by enclosing all of the ISO Latin-1 codes in parentheses, separated by commas.

For instance, the expression

```
'newline'(CR)(LF)
```

can also be written

```
'newline'(CR,LF)
```

The following example illustrates some uses of embedded characters:

```
PROGRAM print_special_strings;
 CONST
     NUL = 0;
     BEL = 7;
     TAB = 9;
     LF = 10;
     CR = 13;


 BEGIN
     writeln('Output four bells' (BEL)(BEL)(BEL)(BEL));
     writeln('Print two tabs' (TAB,TAB) 'followed by text');
     writeln('Print a carriage return'(CR) 'and linefeed' (LF));
 END.
```

## 2.1.6 Case-Sensitivity

Domain Pascal, like standard Pascal, is case-insensitive to keywords and identifiers (i.e., variables, constants, types, and labels), but case-sensitive to strings. That is, Domain Pascal makes no distinction between uppercase and lowercase letters except within a string. For example, the following three uses of the keyword **begin** are equivalent:

```
BEGIN
begin
Begin
```

However, the following two character strings are not equivalent:

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

**NOTE:** At SR10, Domain/OS is case-sensitive for pathnames. Be sure that strings containing pathnames store the pathnames in the correct case.

## 2.1.7 Spreading Source Code Across Multiple Lines

In Pascal, you can start a statement or declaration at any column and spread it over as many lines as you want. However, note that you cannot split a token (keyword, identifier, or string) across a line. For example, consider the **writeln** statement which can take character strings as an argument. The following use of **writeln** is wrong because it splits the string across a line:

```
WRITELN('This is an uninteresting
          long string');
```

Instead, the line should appear this way:

```
WRITELN('This is an uninteresting long string');
```

You can also break the string into two strings and separate them by a comma:

```
WRITELN('This is an uninteresting'
        ,' long string');
```

This works because the **writeln** procedure takes more than one argument.

**NOTE:** By default, any text file you **open** for reading can have a maximum of 256 characters per line. You can specify an optional buffer size when you **open** the file, however, to change that default.

## 2.2 Organization

You can write a Domain Pascal program in one file or across several files. This section explains the proper structure for a program that fits into one file. Chapter 7 details the structure for a program that is spread over several files.

A Domain Pascal program takes the format shown in Figure 2-1. Note that routines are themselves declarations.

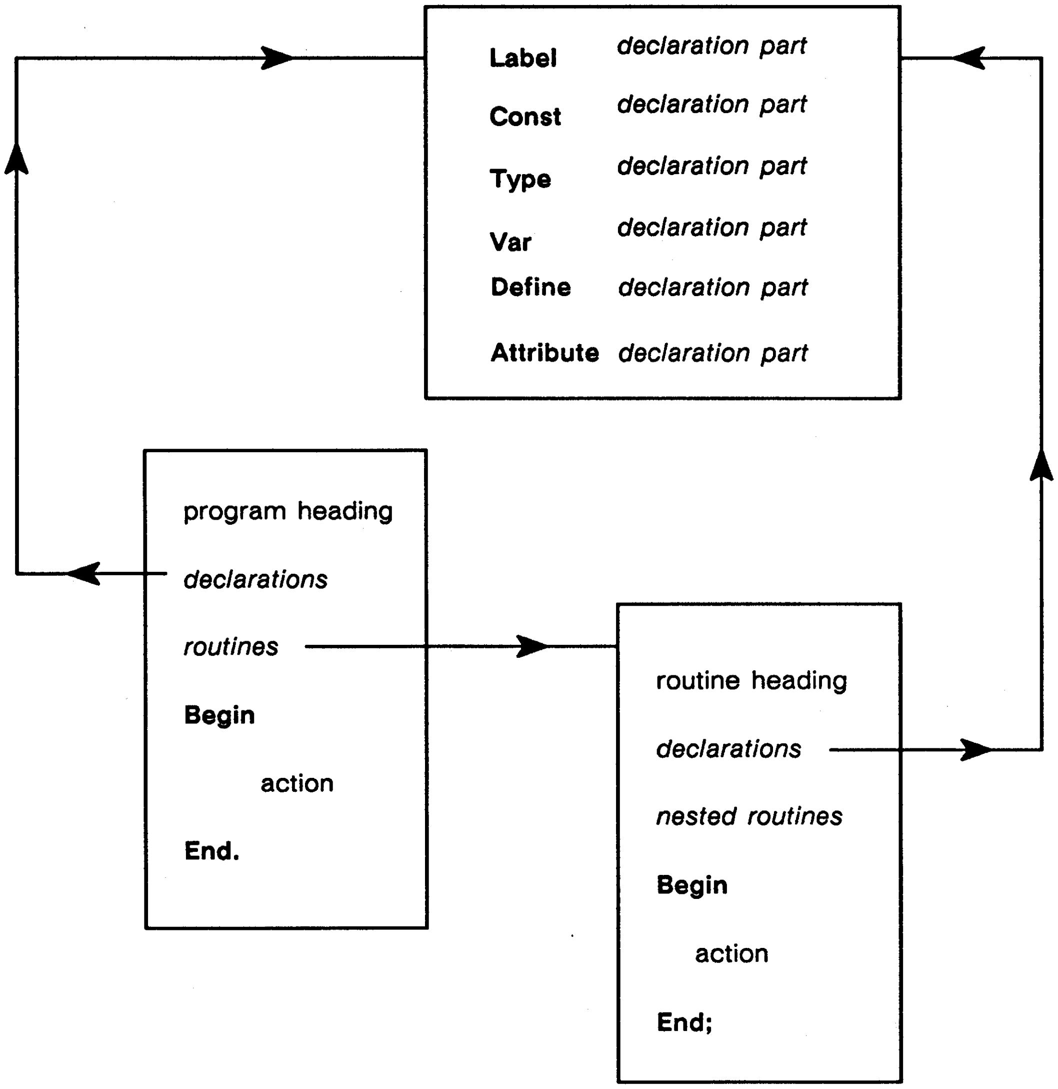

*Figure 2-1. Format of Main Program in Domain Pascal*

Figure 2-2 is a labeled program designed to help you understand the structure of a Domain Pascal program. The following subsections detail the parts of a program.

```
PROGRAM labeled;                                   {program heading}

 {Start of the declaration part of the main program.      }
 {These declarations will be global to the entire program.}
   LABEL                                    {LABEL declaration part}
      finish;

   CONST                                    {CONST declaration part}
      axiom = 'Clarity is wonderful!';

   TYPE                                     {TYPE declaration part}
      flavors = (mint, lime, orange, beige);

   VAR                                      {VAR declaration part}
      x, y, z : integer;
      ice_cream : flavors;
 {End of the declaration part of the main program.}


 {Start of the roots procedure.}
   Procedure roots;                         {routine heading}

 {Start of the declaration part of roots.}
 {These declarations will be local to roots.}
   VAR                                      {VAR declaration part}
      q : real;
 {End of the declaration part of roots.}

   BEGIN          {Start of the action part of roots.}
      write('Enter a number - '); readln(q);
      writeln('The square root of ',q, ' is ',sqrt(q));
   END;           {End of the action part of roots.}
 {End of the roots procedure.}


 BEGIN             {Start of the action part of the main program.}
   writeln(axiom);
   x := 5;  y := 7;  z := x + 5;
   if z > 100 then goto finish;
   for ice_cream := mint to beige do
      writeln(ice_cream);
   roots;
 finish:
 END.              {End of the action part of the main program.}
```

*Figure 2–2. Labeled Main Program*

## 2.2.1 Program Heading

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

**program** *name* [(*file_list*)] [, *code_section_name*] [, *data_section_name*];

In Domain Pascal, as in standard Pascal, you must supply a *name* for the program. *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* manual for details on these utilities for Aegis. For the UNIX utilities **ld** and **ar**, refer to the *SysV Command Reference* and the *BSD Command Reference*.)

In standard Pascal you can supply an optional *file_list* to the program heading. The *file_list* specifies the external files (including standard input and output) that you are going to access from the program. However, unlike standard Pascal, the *file_list* in a Domain Pascal program has no effect on program execution; the compiler ignores it. (For details on I/O, see Chapter 8.)

*Code_section_name* and *data_section_name* are optional elements of the program heading. Use them to specify the names of the sections in which you want the compiler to store your code and data. A section is a named contiguous area of memory in which all entities share the same attributes. (See the *Domain/OS Programming Environment Reference* for details on sections and attributes.) By default, Domain Pascal assigns all the code in your program to the **.text** section and all the data in your program to the **.data** section. To assign your code and data to nondefault sections, specify a *code_section_name* and a *data_section_name*.

> NOTE: In addition to nondefault code and data section names for the entire program, you can also specify a nondefault section name for a procedure, a function, or a group of variables. See the "Section" section of Chapter 5 for an explanation of how to assign section names to procedures and functions, and see the "Putting Variables into Sections" section of Chapter 3 to learn how to assign section names to groups of variables.

To specify the default **.text** section together with an alternate **.data** section, use the following syntax:

**program** *name* [(*file_list*)], , *data_section_name*;

Let's now consider some sample program headings. Despite the options available, most Domain Pascal program headings can look as simple as the following:

```
Program trapezoids;
```

Those of you desiring to write standard Pascal programs will also probably want to supply a *file_list* as in the next example:

```
Program trapezoids (input, output, datafile);
```

Finally, those of you wanting to capitalize on certain run-time features may wish to define your own section names. For example, if you want the compiler to store the code into section **mycode** and the data into section **mydata**, you would issue the following program heading:

```
Program trapezoids, mycode, mydata;
```

## 2.2.2 Declarations

The declarations part of a program is optional. It can consist of zero or more **label** declaration parts, **const** declaration parts, **type** declaration parts, **var** declaration parts, **define** declaration parts, and **attribute** declaration parts. Domain Pascal allows you to specify these parts in any order.

### 2.2.2.1 Label Declaration Part

You define labels in the **label** declaration part. A label has only one purpose—to act as a target for a **goto** statement. (See Chapter 4 for a description of the **goto** statement.) In other words, the statement

```
GOTO 40;
```

works only if you have defined 40 as a label.

The format for a **label** declaration part is

**label**
    *label1* [, ... *labelN*];

A **label** is either an identifier or an unsigned integer. If there are multiple **labels**, you must separate them with commas. Remember, though, to put a semicolon after the final **label**.

For example, the following is a sample label declaration:

```
LABEL
    40, reprompt, finish, 9999;
```

### 2.2.2.2 Const Declaration Part

You define constants in the **const** declaration part. A constant is a synonym for a value that will not (and cannot) change during the execution of the program.

The **const** declaration part takes the following syntax:

**const**
    *identifier1* = *value1*;
    [   .        .
        .        .
        .        .
    *identifierN* = *valueN*; ]

An **identifier** is any valid Domain Pascal identifier. A **value** must be a real, integer, string, char, or set constant expression. **Value** can also be the pointer expression **nil**.

For example, here is a sample **const** declaration part:

```
CONST
    pi = 3.14;                                        {A real number}
    cup = 8;                                            {An integer}
    key = 'Y';                                        {A character}
    blank = ' ';                                      {A character}
    words = 'To be or not to be';                        {A string}
    vowels = ['a', 'e', 'i', 'o', 'u'];                     {A set}
    ptr1 = nil;                                         {A pointer}
```

The preceding sample involves simple expressions; however, you can also specify a more complex expression for *value*. Such an expression can contain the following types of terms:

- A real number, an integer, a character, a string, a set, a Boolean, or **nil**
- A constant that has already been defined in the **const** declaration part (note that you cannot use a variable here)
- Any predefined Domain Pascal function (for example, **chr**, **sqr**, **lshft**, **sizeof**, but only if the argument to the function is a constant)
- A type transfer function

You can optionally separate these terms with any of the operators shown in Table 2-1.

*Table 2-1. Domain Pascal Mathematical Operators*

| Operator | Data Type of Operand |
|---|---|
| +, -, * | Integer, real, or set |
| / | Real |
| **mod, div, !, &, ~** | Integer |
| ** | Exponentiation |

Chapter 4 describes these operators.

For example, the following **const** declaration part defines eight constants:

```
CONST
    x = 10;
    y = 100;
    z = x + y;
    current_year = 1994;
    leap_offset  = (current_year mod 4);
    bell         = chr(7);
    pathname     = '//et/go_home';
    pathname_len = sizeof(pathname);
```

### 2.2.2.3 Type Declaration Part

Chapter 3 details the many predeclared data types Domain Pascal supports. In addition to these Pascal-defined data types, you can create your own data types in the **type** declaration part. After creating your own data type, you can then declare variables (in the var declaration part) that have these data types. The format for a **type** part is as follows:

**type**
    *identifier1* = *typename1*;
    [ .     .
      .     .
      .     .
    *identifierN* = *typenameN*; ]

An *identifier* is any valid Domain Pascal identifier. A *typename* is any predeclared Domain Pascal data type (like **integer** or **real**), any data type that you create, or the identifier of a data type that you created earlier in the **type** declaration part.

For example, here is a sample **type** declaration part:

```
type
    long = integer32;               {A predeclared Domain Pascal data type}
    student_name = array[1..20] of long;       {A user-defined data type}
    colors = (magenta, beige, mauve);          {A user-defined data type}
    hue = set of colors;                       {A user-defined data type}
    table = array[magenta..mauve] of real;     {A user-defined data type}
```

### 2.2.2.4 Var Declaration Part

Declare variables in the **var** declaration part. A variable has two components — a name and a data type. The format for the **var** declaration part is:

**var**
    *identifier_list1* : *typename1*;
    [    . . .            ;
    *identifier_listN* : *typenameN*;]

An *identifier_list* consists of one or more identifiers separated by commas. Each identifier in the *identifier_list* is assigned the data type of *typename*. *Typename* must be one of these:

- A predeclared Domain Pascal data type
- A data type you declared in the **type** declaration part
- An anonymous data type (that is, a data type you define for the variables in this *identifier_list* only)

For example, consider the following **type** declaration part and **var** declaration part:

```
type
   names = array[1..20] of char;
   colors = (red, yellow, blue);
var
   counter, x, y:integer;
                 {integer is a predeclared Domain Pascal data type.}
   angles:real;        {real is a predeclared Domain Pascal data type.}
   a_letter:char;      {char is a predeclared Domain Pascal data type.}
   couch_colors:colors;          {colors is defined in the type part.}
   evil;boolean;    {boolean is a predeclared Domain Pascal data type.}
   mystery_guest:names;           {names is defined in the type part.}
   seniors:67..140;                 {An anonymous subrange data type.}
   pet:(cat, dog);                {An anonymous enumerated data type.}
```

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

### 2.2.2.5 Define Declaration Part—Extension

In addition to the **const**, **type**, **var**, and **label** declaration parts of standard Pascal, Domain Pascal also supports an optional **define** declaration part, which is described in the first three sections of Chapter 7.

### 2.2.2.6 Attribute Declaration Part—Extension

Domain Pascal supports an optional **attribute** declaration part, which is described in the "Attribute Declaration Part" section of Chapter 3.

## 2.2.3 Routines

A program can contain zero or more routines. There are two types of routines in Domain Pascal: procedures and functions. A routine consists of three parts: a routine heading, an optional declaration part, and an action part.

### 2.2.3.1 Routine Heading

Routine headings take the following format:

[*attribute_list*] **procedure** *name* [*(parameter_list)*] ; [*routine_options*;]

or

[*attribute_list*] **function** *name* [*(parameter_list)*] : *typename*; [*routine_options*;]

where:

- *attribute_list* is optional. Inside the *attribute_list*, you can specify nondefault section names for the routine's code and data. For a description, see Chapter 5.
- *name* is an identifier. You call the routine by this name.
- *parameter_list* is optional. It is here that you declare the names and data types of all the parameters that the routine expects from the caller. See Chapter 5 for details on the *parameter_list*.
- *typename* is the data type of the value that the function returns. The difference between a procedure and a function is that the *name* of a procedure is simply a name, but the *name* of a function is itself a variable with its own *typename*. 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 procedure. (It is an error if you do.)
- *routine_options* is an optional element of the routine heading. You can specify characteristics of the routine such as whether or not it can be called from another routine. Chapter 5 describes the *routine_options*.

### 2.2.3.2 Declaration Part of a Routine

The optional declaration part of a routine follows the same rules (with one exception) as the optional declaration part under the program heading. The constants, data types, variables, and labels are local to the routine declaring them and to any routines nested within them. (See the "Global and Local Variables" and "Nested Routines" sections at the end of this chapter for details.)

One difference between the declaration part of a routine and the declaration part of the main program is that the declaration part of a routine cannot contain a **define** declaration part. Another difference is that you cannot initialize non-static variables in a routine, though you can initialize them in the main program.

#### 2.2.3.3 Nested Routines

You can optionally nest one or more routines within a routine. See the "Nested Routines" section at the end of this chapter for details.

#### 2.2.3.4 Action Part of a Routine

The action part of a routine starts with the keyword **begin** and finishes with the keyword **end**. Between **begin** and **end** you supply one or more Domain Pascal statements. (See Chapter 4 for a description of Domain Pascal statements.) You must place a semicolon after the final **end** in a routine.

For example, consider the following sample action part of a routine:

```
BEGIN
  x := x * 100;
  writeln(x);
END;
```

### 2.2.4 Action Part of the Main Program

The action part of the main program is almost identical to the action part of a routine. Both start with **begin**, both finish with **end**, and both contain Domain Pascal statements in between. The only difference is that you must place a period (rather than a semicolon) after the final **end** in the main program. For example, consider the following sample action part of the main program:

```
BEGIN
    x := x * 100;
    writeln(x);
 END.
```

## 2.3 Global and Local Variables

The declarations in the declaration part of the main program are global to the entire program. The declarations in the declaration part of a routine are local to that routine (assuming no nesting). For example, consider the following program. In it, variable **g** is global and variable **l** is local to procedure **add100**.

```
Program scope;
 VAR
    g : integer16;

    Procedure add100;
    VAR
       l : integer16;
    BEGIN
       l := g + 100;
         {Variable l is accessible within this procedure only,}
         {while g is global and so is accessible anywhere.    }
    END;


 BEGIN
    g := 10;          {Variable g is accessible because it is global. }
    add100;           {Call the procedure.                            }

                      {Variable l is not accessible here because it is}
                      {local to procedure add100.                     }
 END.
```

What happens when you specify a local variable with the same name as a global variable? To answer this question, see Figure 2–3 for two more programs. In the program on the left (**global_example**), x is declared as a global variable. In the program on the right (**local_example**), x is declared twice. The first declaration specifies x as a global variable. The second declaration declares x as local to procedure **convert**.

```
Program global_example;                   Program local_example;

VAR {global declarations}                 VAR {global declarations}
   x : integer16;                            x : integer16;


   PROCEDURE convert;                        PROCEDURE convert;
                                             VAR {local declarations}
                                                x : integer16;
   BEGIN                                     BEGIN
      x := -10;                                 x := -10;
      writeln('In convert, x=',x:1);            writeln('In convert, x=',x:1);
   END;                                      END;


BEGIN  {main}                             BEGIN {main}
   x := +10;                                 x := +10;
   convert;                                  convert;
   writeln('In main, x=',x:1);               writeln(In main, x=', x:1);
END.                                      END.
```

*Figure 2–3. Global Variables Example*

If you execute the programs in Figure 2-3, you get the following results:

```
Execution of global_example        Execution of local_example
    In convert, x= -10                 In convert, x= -10
    In main, x= -10                    In main, x= 10
```

In program **local_example**, within procedure **convert**, the declaration of the local variable **x** overrides the global declaration of **x**. Within **convert**, the fact that the local variable and the global variable have the same name (x) prevents procedure **convert** from accessing the global variable **x** at all.

Both programs are available online and are named **global_example** and **local_example**.

## 2.4 Nested Routines

A nested routine is a routine that is declared inside another routine. A nested routine can access any declared object (label, constant, type, or variable) in a routine outside it, provided that the object is not hidden by a local declaration. The reverse is not true; that is, a routine cannot access an object in a routine nested inside it. Thus, the purpose of nesting routines is to create a hierarchy of access. You might view declared objects in the following way:

- Global to the entire program.
- Local to a single routine.
- Local to the routine it is defined in and to all routines nested within it (that is, neither truly local nor truly global). This is termed an "intermediate level" object.

Note that the main program is itself a routine, and that all routines are nested at least one level inside it. A routine can call any routine nested one level inside it, but cannot explicitly call any routine nested two or more levels inside it. A routine can also call any routine at its level or outside it, though a routine cannot explicitly call the main program.

For example, consider the program in Figure 2-4. Procedure **one** is nested inside the main program. Procedures **twoa** and **twob** are both nested inside procedure **one**. The most-nested procedures (**twoa** and **twob**) can access the most variables. The least-nested procedure (the main program) can access the least number of variables.

```
Program  nesting_example;

 VAR
    g : integer16;

    procedure one;
    VAR
       l : integer16;

       procedure twoa;
       VAR
          n1 : integer16;
       BEGIN {twoa}
       {can access g, l, and n1.}
          n1 := l + g + 500;
       END;  {twoa}


       procedure twob;
       VAR
          n2 : integer16;
       BEGIN  {twob}
       {can access g, l, and n2.}
          n2 := l + g + 1000;
       END;   {twob}


    BEGIN {one}
    {can access g and l.}
       l := g + 10;
       twob;
    END;  {one}

 BEGIN  {main program}
 {can only access g.}
    g := 1;
    g := g * 2;
    one;
 END.   {main program}
```

*Figure 2-4. Nesting Example*

Note that the main program can call procedure **one**, but cannot call procedure twoa or **twob** (since they are nested two levels inside it). Procedure **one** can call procedure **twoa** or **twob**. Procedure **twob** can call procedure **twoa** or **one**. In Pascal, you cannot make a forward reference to a routine unless you declare the routine with the **forward** option (described in Chapter 5). If you used **forward** in this example, procedure **twoa** could call **twob** or **one**.

# Chapter 3

# Data Types

This chapter explains Domain Pascal data objects. It tells you how to declare variables using the predeclared Domain Pascal data types and how to define your own data types. The chapter also shows how Domain Pascal represents each data type internally. Finally, the chapter describes attributes for variables and types and an attribute declaration part. You can use these attributes to define characteristics in addition to the data type.

## 3.1 Data Type Overview

Domain Pascal supports data types that can be sorted into three groups—the **simple, structured,** and **pointer** data types. Furthermore, Domain Pascal provides extensions to the standard in each category of data type. In this section, we list all the Domain Pascal data types according to category.

The following list shows the **simple** data types:

- Integers—Domain Pascal supports the three predeclared integer data types **integer, integer16,** and **integer32.**
- Real numbers—Domain Pascal supports the three predeclared real number data types **real, single,** and **double.**
- Boolean—Domain Pascal supports the predeclared data type **boolean.**
- Character—Domain Pascal supports the predeclared **char** data type.
- Enumerated—Domain Pascal supports **enumerated** data types.
- Subrange—Domain Pascal supports a **subrange** of scalar data types. The **scalar** data types are integer, Boolean, character (**char**), and enumerated.

You can use the simple data types to build the following **structured** data types:

- Sets—Domain Pascal permits you to create a set of elements of a scalar data type.
- Records—Domain Pascal supports the **record, aligned record, unaligned record,** and **packed record** data types.
- Array—Domain Pascal supports the **array** and **packed array** data types. It also supports a predeclared character array type called **string,** and variable-length array type declared with the **varying** keyword.
- Files—Domain Pascal supports the **file** and **text** data types.

You can declare any of three kinds of **pointer** data types:

- Type-specific pointer—points to any previously defined data type.
- Universal pointer—Domain Pascal supports **univ_ptr,** a predeclared pointer data type that is compatible with *any* pointer type.
- Procedure and function pointers—Domain Pascal supports a special data type that points to procedures and functions.

The program shown in Figure 3-1 contains sample declarations of the above data types. This program is available online and is named **sample_types.**

```
PROGRAM  sample_types;
 TYPE
  real_pointer =    ^real;                   {This is a pointer type.    }
  writers =         (Amy, Phil, Janice);     {This is an enumerated type.}
  element =         record                   {This is a record type.     }
                    atomic_number : INTEGER16;
                    atomic_weight : SINGLE;
                    half_life     : DOUBLE;
                    end;
 VAR
  i1                : INTEGER;
  i2                : INTEGER16;
  i3                : INTEGER32;
  r1                : REAL;
  r2                : SINGLE;
  r3                : DOUBLE;
  consequences      : BOOLEAN;
  onec              : CHAR;
  teenage_years     : 13..19;     {teenage_years is a subrange variable.}
  good_writers      : writers; {good_writers is an enumerated variable.}
  tw                : SET OF writers;            {tw is a set variable.}
  e                 : element;                {e is a record variable.}
  cat_nums          : array[1..5] of INTEGER16;
                                        {cat_nums is an array variable.}
  a_sentence        : STRING;
                     {a_sentence is an array variable of 80 characters.}
  hamlets_soliloquy : TEXT; {hamlets_soliloquy is a text file variable.}
  periodic_table    : FILE OF element;
                                    {Periodic_table is a file variable.}
  r1_ptr            : real_pointer;       {r1_ptr is a pointer variable.}
  Any_Ptr           : UNIV_PTR;
                              {Any_ptr is a universal pointer variable.}
  pp                : ^PROCEDURE (IN x : INTEGER);
                              {pp is a pointer to a procedure variable.}
 BEGIN
  writeln('Greetings.');
 END.
```

*Figure 3-1. Program Declaring All Available Data Types*

## 3.2 Integers

This section explains how to declare variables as integers, how to initialize integer variables, and how to define integer constants. It also explains how Domain Pascal represents integers internally.

### 3.2.1 Declaring Integer Variables

Domain Pascal supports the following predeclared integer data types:

- **Integer**—Use it to declare a signed 16-bit integer. A signed 16-bit integer variable can have any value from -32768 to +32767.
- **Integer16**—Use it to declare a signed 16-bit integer. (**Integer** and **integer16** have identical meanings.)
- **Integer32**—Use it to declare a signed 32-bit integer. A signed 32-bit integer variable can be any value from -2147483648 to +2147483647.

For example, consider the following integer declarations:

```
VAR
    x, y, z : INTEGER;
    quarts  : INTEGER16;
    social_security_number : INTEGER32;
```

If you want to define unsigned integers, you must use a subrange declaration. (Refer to the "Subrange" section later in this chapter.)

### 3.2.2 Initializing Integer Variables—Extension

Domain Pascal permits you to initialize the values of integers within the variable declaration in most cases. You initialize a variable by placing a colon and equal sign (:=) immediately after the data type. For example, the following excerpt initializes **X** and **Y** to 0, and **Z** to 7000000:

```
VAR
    X,Y : INTEGER16 := 0;
    Z   : INTEGER32 := 7000000;
```

If the variable declaration occurs within a procedure or function, you *cannot* initialize the variable at the declaration unless it has been declared **static**. This is because storage within routines is dynamic and so variables in them do not necessarily retain their values between executions.

For example, the following is incorrect:

```
FUNCTION do_nothing(IN OUT x : INTEGER) : BOOLEAN;
 VAR
    init_value : INTEGER := 0;         {Wrong!}
```

This is the correct way to initialize the variable at its declaration in a routine:

```
   init_value : STATIC INTEGER := 0;  {Right!}
```

See the "Accessing a Variable Stored in Another Pascal Module" section of Chapter 7 for more information on the **static** attribute.

### 3.2.3 Defining Integer Constants

When you declare an integer constant, Domain Pascal internally represents the value as a 32-bit integer. For example, in the following declarations, Domain Pascal represents both **poco** and **grande** as 32-bit integers.

```
CONST
    poco = 6;
    grande = 6000000;
```

You can specify an integer constant anywhere in the range -2147483648 to +2147483647.

It is also possible to compose constant integers as a mathematical expression. (Refer to the "Const Declaration Part" section in Chapter 2 for details.)

The predeclared integer constant **maxint** has the value +32767.

### 3.2.4 Internal Representation of Integers

Domain Pascal represents a 16-bit integer (types **integer** and **integer16**) as two contiguous bytes, as shown in Figure 3-2. Bit 15 contains the most significant bit (MSB), and bit 0 contains the least significant bit (LSB). If the integer is signed, bit 15 contains the sign bit.

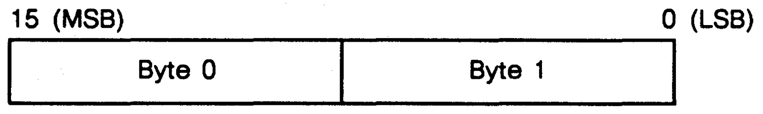

*Figure 3-2. 16-Bit Integer Format*

Domain Pascal represents a 32-bit integer (type **integer32**) in four contiguous bytes as illustrated in Figure 3-3. The most significant bit in the integer is bit 31; the least significant bit is bit 0. If the integer is signed, bit 31 contains the sign bit.

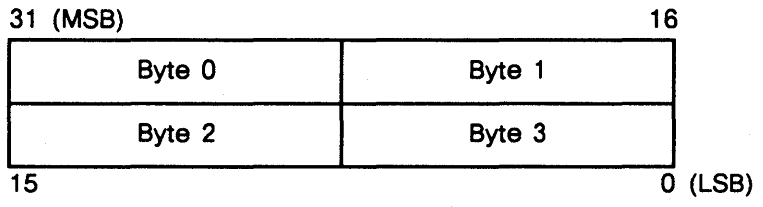

*Figure 3-3. 32-Bit Integer Format*

By default, Domain Pascal aligns freestanding 16-bit integers on word boundaries and 32-bit integers on longword boundaries. (See the "Internal Representation of Records" section of this chapter for details about alignment for integers that are part of records.)

## 3.3 Real Numbers

This section describes how to declare variables as real numbers, how to define real numbers as constants, and how Domain Pascal represents real numbers internally.

### 3.3.1 Declaring Real Variables

Domain Pascal supports the following real data types:

- **Real**—Use it to declare a signed single-precision real variable. Domain Pascal represents a single-precision real number in 32 bits. A single-precision real variable has approximately seven significant digits.

- **Single**—Same as **real**.

- **Double**—Use it to declare a signed double-precision real variable. Domain Pascal represents a double-precision real number in 64 bits. A double-precision real variable has approximately 16 significant digits.

For example, consider the following declarations:

```
VAR
    l, m, n : REAL;
    winning_time : SINGLE;
    cpu_time : DOUBLE;
```

### 3.3.2 Initializing Real Variables—Extension

Domain Pascal permits you to initialize the values of real numbers within the variable declaration in most cases. You initialize a value by placing a colon and equal sign (:=) immediately after the data type. For example, the following excerpt initializes variable **pi** to 3.14:

```
VAR
    pi  : SINGLE := 3.14;
```

If you declare a variable as **single** or **real**, and if you attempt to initialize it to a number with more than seven significant digits, then Domain Pascal rounds (it does not truncate) the number to the first seven significant digits.

For example, if you try to initialize **pi** this way:

```
VAR
    pi  : SINGLE := 3.1415926535;
```

Domain Pascal rounds **pi** to 3.141593.

As with integers, if the variable declaration occurs within a procedure or function, you can initialize the variable at the declaration *only if* it has been declared **static**. This is because storage within routines is dynamic and so variables in them do not necessarily retain their values between executions. For example, the following is incorrect:

```
FUNCTION do_nothing(IN OUT x : REAL) : BOOLEAN;
 VAR
    init_value : REAL := 0.0;                {Wrong!}
```

This is the correct way to initialize the variable at its declaration in a routine:

```
   init_value : STATIC REAL := 0.0;          {Right!}
```

See the "Accessing a Variable Stored in Another Pascal Module" section of Chapter 7 for more specific information on the **static** attribute.

## 3.3.3 Defining Real Constants

When you use a real number as a constant, Domain Pascal automatically defines the constant as a double-precision real number. This is true even if the constant can be accurately represented as a single-precision real number. However, when you use a real constant in a mathematical operation with a single-precision number, Domain Pascal automatically rounds the constant to a single-precision number to produce a more accurate result. The following fragment defines four valid (and one invalid) real constants:

```
CONST
      N  = 24.57;                       { Valid real number. }
      N2 = 2E19;              { Valid, symbolizes 2.0 * (10^19) }
      G  = 6.67E-11;        { Valid, symbolizes 6.67 * (10^-11) }
      X  = .5;       { Not a valid real literal because it does }
                       { not contain a digit to the left of the }
                                              { decimal point. }
      X2 = 0.5;                           {Valid real literal.}
```

## 3.3.4 Internal Representation of Real Numbers

Single-precision floating-point numbers (types **real** and **single**) occupy four contiguous bytes of a longword, as shown in Figure 3-4. Domain Pascal uses the IEEE standard format for representing 32-bit real values. Bit 31 is the sign bit with "1" denoting a negative number. If the value is normalized, the next eight bits contain the exponent plus 127, and the remaining 23 bits contain the mantissa of the number without the leading 1. (Domain Pascal stores the mantissa in magnitude form, not in two's-complement.)

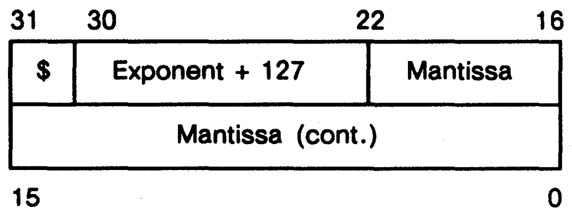

*Figure 3-4. Single-Precision Floating-Point Format*

For example, Pascal represents +100.5 in the following manner:

```
0100001011001001
0000000000000000
```

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

| | |
|---|---|
| sign | 0 (positive) |
| exponent | 10000101 (133 in decimal) |
| significant part of mantissa | 1001001 |

The exponent is 133; 133 is equal to 127 plus 6. Therefore, you can view the mantissa bits as follows:

```
bit 22 represents 2 to the fifth power
bit 21 represents 2 to the fourth power
bit 20 represents 2 to the third power
  .
  .
  .
bit 16 represents 2 to the negative first power.
```

You get 100.5 by adding ($2^6 + 2^5 + 2^2 + 2^{-1}$ ). (The implicit leading 1 of the mantissa corresponds to $2^6$.)

A number with a negative exponent is stored differently. Pascal represents 5E−2 (+0.05) as follows:

```
0011110101001100
1100110011001101
```

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

| | |
|---|---|
| sign | 0 (positive) |
| exponent | 01111010 (122 in decimal) |
| significant part of mantissa | 10011001100110011001101 |

The exponent is 122; 122 is equal to 127 plus -5. Therefore, you can view the mantissa bits as follows:

```
bit 22 represents 2 to the -6 power
bit 21 represents 2 to the -7 power
bit 20 represents 2 to the -8 power
  .
  .
  .
bit 0 represents 2 to the -29 power
```

You get 5E–2 by adding $2^{-5} + 2^{-6} + 2^{-9}$ and so on.

Domain Pascal represents double-precision floating-point numbers (type **double**) in eight bytes of a longword (64 bits). Figure 3-5 illustrates the format. The first bit (bit 63) contains the sign bit. If the value is normalized, the next 11 bits contain the exponent plus 1023. The remaining 52 bits hold the mantissa, without the leading 1.

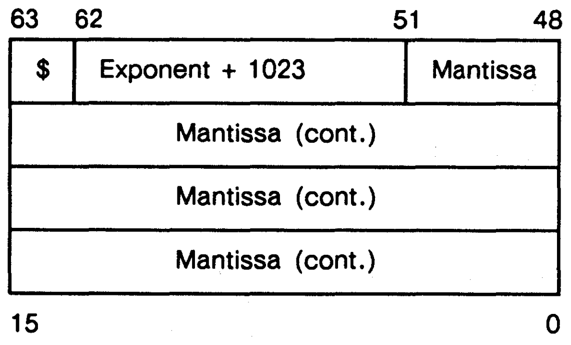

*Figure 3-5. Double-Precision Floating-Point Format*

By default, Domain Pascal stores single-precision floating-point numbers (types **real** and **single**) on longword boundaries. It stores double-precision floating-point numbers (type **double**) on quadword boundaries. (See the "Internal Representation of Records" section of this chapter for details about alignment for real numbers that are part of records.)

For complete information about floating-point formats, see the *Domain Floating-Point Guide*.

## 3.4 Unsigned Types

Although Domain Pascal does not have a true unsigned data type, it does offer unsigned ranges. Through the use of large unsigned subranges (from $0..2^{30}$ up to $00..2^{31}–1$), you can achieve much of the effect of having true unsigned types.

The following code fragment illustrates the simulation of unsigned types through the use of unsigned subranges:

```
TYPE
    signed_32   = -2147483648..2147483647;
    unsigned_32 = 0..2147483647;

VAR
    s32 : signed_32;         { Signed 32-bit integer. }
    u32 : unsigned_32;       { Unsigned 31-bit integer.  Not
                               exactly the full 32 bits, but
                               enough to convince the compiler
                               that u32 is much like an unsigned
                               32-bit integer. }
```

**NOTE:** The true full range of unsigned 32-bit integers cannot be expressed because constants larger than 2147483647 are not valid. You cannot get around this restriction by using constants with an explicit base with the signed bit set because the compiler treats them as negative numbers (for example, 16#FFFFFFFF).

When you have an unsigned subrange that is large enough to require 31 bits, the compiler allocates 32 bits.

## 3.5 Booleans

A Boolean variable can have only one of two values—**true** or **false**. This section describes how you declare Boolean variables, how you define Boolean constants, and how Domain Pascal represents Boolean variables internally.

### 3.5.1 Initializing Boolean Variables—Extension

Domain Pascal permits you to initialize the values of Boolean variables within the variable declaration in most cases. You initialize a value by placing a colon and equal sign (:=) immediately after the data type. For example, the following excerpt declares **liar** to be a Boolean variable with an initial value of **false**:

```
VAR
    liar : boolean := false;
```

If the variable declaration occurs within a procedure or function, you *cannot* initialize the variable at the declaration unless it has been declared **static**. This is because storage within routines is dynamic and so variables in them do not necessarily retain their values between executions. For example, the following is incorrect:

```
FUNCTION do_nothing(IN OUT x : INTEGER) : BOOLEAN;
 VAR
    liar : BOOLEAN := false;         {Wrong!}
```

This is the correct way to initialize the variable at its declaration in a routine:

```
    liar : STATIC BOOLEAN := false;     {Right!}
```

See Chapter 7 for information on the **static** attribute.

### 3.5.2 Defining Boolean Constants

To define a Boolean constant, simply write the name of the constant, followed by an equal sign, and concluding with either **true** or **false**. For instance, the following excerpt defines constant **virtue** and sets it to **true**:

```
CONST
    virtue = true;
```

Notice that you do *not* enclose **true** or **false** inside a pair of apostrophes.

### 3.5.3 Internal Representation of Boolean Variables

Domain Pascal represents Boolean values in one byte. The system sets all eight bits to 1 for **true** and sets all eight bits to 0 for **false**. By default, Domain Pascal stores freestanding Boolean objects on byte boundaries. However, a Boolean field in a packed record will have a different allocation (see the "Internal Representation of Packed Records" section later in this chapter for details).

## 3.6 Characters

This section describes how you declare a variable as a character data type, how you define characters as constants, and how Domain Pascal represents characters internally.

### 3.6.1 Declaring Character Variables

Use the **char** type to declare a variable that holds one character; for example:

```
VAR
    a_letter, a_better_letter : CHAR;
```

To declare a variable that holds more than one character you must use an array or the predefined type **string** (both of which are detailed in the "Arrays" section later in this chapter).

## 3.6.2 Initializing Character Variables—Extension

Domain Pascal permits you to initialize the values of character variables within the variable declaration in most cases. You initialize a value by placing a colon and equal sign (:=) immediately after the data type. For example, the following excerpts each declare **c1** as a **char** variable with an initial value of **a**:

```
VAR
    c1 : CHAR := 'a'; {you must enclose the character in single quotes}
```

```
VAR
    c1 : CHAR := chr (65);
```

If the variable declaration occurs within a procedure or function, you *cannot* initialize the variable at the declaration unless it has been declared **static**. This is because storage within routines is dynamic and so variables in them do not necessarily retain their values between executions.

For example, the following is incorrect:

```
FUNCTION do_nothing (IN OUT x : INTEGER) : BOOLEAN;
 VAR
    best_grade : CHAR := 'A';                 {Wrong!}
```

This is the correct way to initialize the variable at its declaration in a routine:

```
   best_grade : STATIC CHAR := 'A';           {Right!}
```

See the "Accessing a Variable Stored in Another Pascal Module" and "Accessing a Routine Stored in Another Pascal Module" sections of Chapter 7 for more information on the **static** attribute.

## 3.6.3 Defining Character Constants

There are two common methods of assigning character constants. The first is to simply enclose a character inside a pair of single quotes. For example:

```
CONST
    c1 = 'b';
```

This first method works only if the character is printable, but the second method works for *all* ISO Latin-1 characters (printable or not). The second method uses the **chr** function (which is detailed in Chapter 4). As an example, suppose you want constant **bell** to contain the bell ringing character. The bell ringing character has an ISO Latin-1 value of 7, so to assign this value to constant **bell** you can make the following declaration:

```
CONST
    bell = CHR(7);
```

### 3.6.4 Internal Representation of Char Variables

Domain Pascal stores the ISO Latin–1 value of a **char** variable in one 8–bit byte. By default, freestanding **char** variables are byte aligned.

## 3.7 Enumerated Data

An enumerated data type consists of an ordered group of identifiers. The only value you can assign to an enumerated variable is one of the identifiers from its group of identifiers. Here are declarations for four enumerated variables:

```
VAR
    citrus          : (lemon, lime, orange, carambola, grapefruit);
    primary_colors  : (red, yellow, blue);
    Beatles         : (John, Paul, George, Ringo);
    German_speaking_countries : (Germany, Switzerland, Austria);
```

In the code portion of your program, you can only assign the values **red**, **yellow**, or **blue** to variable **primary_colors**.

Notice that the elements of an enumerated type must be identifiers. Identifiers cannot begin with a digit, so, for example, the following declaration produces an "Improper enumerated constant syntax" error:

```
VAR
    first_six_primes : (2, 3, 5, 7, 11, 13);  {error}
```

### 3.7.1 Internal Representation of Enumerated Variables

Domain Pascal represents an enumerated variable in one 16–bit word. In this word, Domain Pascal stores an integer corresponding to the ordinal position of the current value of the enumerated variable. For example, consider the following declaration:

```
VAR
    pets : (cats, dogs, dolphins, gorillas, pythons);
```

**Pets** has five elements; Domain Pascal represents those five elements as integers from 0 to 4 as shown in Table 3–1.

*Table 3-1. Representation of an Enumerated Variable*

| **Variable** | **Representation** |
|---|---|
| `pets := cats` | 0000000000000000 |
| `pets := dogs` | 0000000000000001 |
| `pets := dolphins` | 0000000000000010 |
| `pets := gorillas` | 0000000000000011 |
| `pets := pythons` | 0000000000000100 |

## 3.8 Subrange Data

A variable with the subrange type has a valid range of values that is a subset of the range of another type called the base type. When you define a subrange, you specify the lowest and highest possible value of the base type. You can specify a subrange of integers, characters, or any previously defined enumerated type. The following fragment declares four different subrange variables:

```
TYPE
   mountains = (Wachusett, Greylock,
      Washington, Blanc, Everest);    {Mountains is an enumerated type.}

 VAR             {The following four variables all have subrange types.}
   teenage_years     : 13..19;                   {Subrange of INTEGER.}
   positive_integers : 1..MAXINT;                {Subrange of INTEGER.}
   capital_letters   : 'A'..'Z';                    {Subrange of CHAR.}
   small_mountains   : Wachusett..Washington;  {Subrange of MOUNTAINS.}
```

Currently, Domain Pascal does not support subrange checking. For example, if you try to assign the value 25 to **teenage_years**, Domain Pascal does not report an error. However, you can use the **in_range** function to determine whether 25 is within the declared subrange. (See the "In_range" section of Chapter 4 for information on the **in_range** function.)

### 3.8.1 Internal Representation of Subranges

The storage allocation for subrange variables is the same as that for their base types. However, a subrange field in a packed record will have a different allocation. (See the "Internal Representation of Packed Records" section later in this chapter for details.)

## 3.9 Sets

A set in Domain Pascal is similar to a set in standard mathematics. For instance, Domain Pascal can compute unions and intersections of Domain Pascal set variables just as you can find unions and intersections of two mathematical sets. Refer to the "Set Operations" listing in Chapter 4 for information on using sets in the action part of your program.

### 3.9.1 Declaring Set Variables

The format for specifying a set variable is as follows:

**set of** **boolean** | **char** | *enumerated_type* | *subrange_type*

For example, consider the following set declarations:

```
TYPE
    very    = (ochen, sehr, tres, muy);
    lowints = 0..100;

VAR
    ASCII_values : set of char;          {Char is base type.}
    possibilities : set of boolean;      {Boolean is base type.}
    capital_letters : set of 'A'..'Z';   {Subrange of CHAR is base type.}
    lots : set of very;                  {Enumerated type is base type.}
    digits : set of lowints;             {lowints is base type.}
```

If the base type is a subrange of integers, then the low end of the subrange cannot be a negative number. Also, the high end of the subrange cannot exceed 1023.

NOTE: Although Domain Pascal lets you declare **packed** set variables or **packed** set types, the compiler ignores the designation. (That is, the **packed** designation does not affect the amount of memory the compiler uses to represent the set.) See Section 3.9.3 for further information about the internal representation of sets and a description of a technique for the packing of small sets.

### 3.9.2 Initializing Set Variables—Extension

In most cases, you can initialize set variables with an assignment statement in the variable declaration. For example, consider the following set variable initializations:

```
TYPE
    unstable_elements = (U, Pl, Ei, Ra, Xe);

VAR
    letters : set of CHAR := ['A', 'E', 'I', 'O', 'U'];
    humanmade_elements : set of unstable_elements := [Pl, Ei];
```

If the variable declaration occurs within a procedure or function, you *cannot* initialize the variable at the declaration unless it has been declared **static**. This is because storage within routines is dynamic and so variables in them do not necessarily retain their values between executions. For example, the following is incorrect:

```
FUNCTION assign_grades(IN OUT score : INTEGER) : BOOLEAN;
 VAR
    grades : set of CHAR := ['A', 'B', 'C', 'D', 'E'];          {Wrong!}
```

This is the correct way to initialize the variable at its declaration in a routine:

```
   grades : STATIC set of CHAR := ['A', 'B', 'C', 'D', 'E'];  {Right!}
```

See the "Accessing a Variable Stored in Another Pascal Module" and "Accessing a Routine Stored in Another Pascal Module" sections of Chapter 7 for more information on the **static** attribute.

Refer to the "Set Operations" listing in Chapter 4 for more information on set assignment.

## 3.9.3 Internal Representation of Sets

A set can contain up to 1024 elements; their ordinal values are 0 to 1023. Sets are stored as bit masks, with one bit representing one element of the set. The number of bits that Domain Pascal allocates to a set is the number of elements in the set, rounded up to a multiple of 16 bits. That is, a set occupies the minimum number of words that provides one bit per element. Consequently, the minimum storage size for a set is one word (16 bits) and the maximum size is 64 words (1024 bits). The one exception to this is small sets within a **packed** records. See Section 3.10.7 for a technique that packs small sets within a **packed** record.

For example, suppose you define an enumerated type named **Greek_letters**, with values **Alpha, Beta, Gamma,** and so forth, up to **Omega**. You can then declare a set of **Greek_letters** as follows:

```
VAR
    Greek_alphabet : SET of Greek_letters
```

**Greek_alphabet** has 24 values, and therefore, **Greek_alphabet** requires at least 24 bits. The nearest word boundary is 32 bits, so Domain Pascal allocates 32 bits (2 words) for the variable. It then stores the values as shown in Figure 3-6:

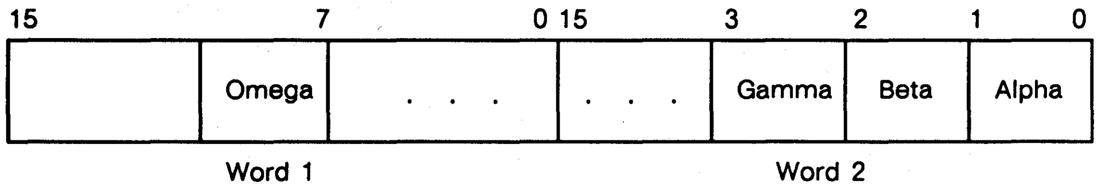

*Figure 3-6. Storage of Sample Set*

If the base type of the set is a subrange of integers or a subrange of **char**, then the ordinal value of the high end of the subrange determines the amount of space required to store the set. For example, consider the following two set declarations:

```
TYPE
    possible_values = 80..170;
    small_letters   = 'a'..'z';

 VAR
    pos : set of possible_values;
    sma : set of small_letters;
```

Domain Pascal stores variable **pos** in 11 words (176 bits). That is because the highest ordinal value of the base type (**possible_values**) is 170. The next word boundary up from 170 is 176.

Domain Pascal stores variable **sma** in eight words (128 bits). In the base type (**small_letters**), the ordinal value of **z** is 122. The next word boundary up from 122 is 128.

## 3.10 Records

A record variable is composed of one or more different components (called **fields**) which may have different types. Domain Pascal supports the two standard kinds of records: fixed records and variant records. The following subsections describe both kinds.

### 3.10.1 Fixed Records

A fixed record consists of zero or more fields. Each field can have any valid Domain Pascal data type. To declare a fixed record type, issue a declaration of the following format:

```
type
    record_name = record
        field1
        .         .
        .         .
        .         .
        fieldN
    end;
```

Each **field** has the following format:

*field_name1, ... field_nameN : typename;*

For example, consider the following three record declarations:

```
TYPE
    student = record                                {Contains two fields.}
              name  : array[1..30] of char;
              id    : INTEGER16;
    end;

    element = record                               {Contains four fields.}
              name           : array[1..15] of char;
              symbol         : array[1..2] of char;
              atomic_number  : 1..120;
              atomic_weight  : real;
    end;

    weather = record                               {Contains five fields.}
              station         : array[1..3] of char;
              sky_condition   : (fair, ptly_cloudy, cloudy);
              windspeed       : 1..100;
              winddirection   : 1..360;
              pressure        : single;
    end;

 VAR
    new_students : student;
    noble_gases  : element;
    w1           : weather;
```

Note that you can declare a record type as the data type of a field. For example, notice the changes in the declaration of the record **weather**:

```
TYPE
    wind = record
           speed : 1..100;
           direction : 1..360;
    end;

    weather = record
              station         : array[1..3] of char;
              sky_condition   : (fair, ptly_cloudy, cloudy);
              gradient        : wind;
              pressure        : single;
    end;
```

**NOTE:** A common mistake is to misuse the equal sign (=) and the colon (:). When declaring a record in the **type** declaration part, put an *equal sign* between the name of the record and the keyword **record**. For example:

```
type
weather =  record...
```

When declaring a record in the **var** declaration part, put a *colon* between them. For example:

```
var
w1 : weather;
```

## 3.10.2 Variant Records

A variant record is a record with multiple data type possibilities. When you declare a variant record, you specify all the possible data types that the record can have. You also specify the condition for selecting among the multiple possibilities.

In other words, at run time a *fixed* record variable has the *same* group of data types from one use of the variable to another. However, a *variant* record variable has a *flexible* group of data types from one use of the variable to another.

The variant record has the following format:

```
type
    record_name = record
        fixed_part;
        variant_part;
    end;
```

The *fixed_part* of a variant record is optional. It looks just like a fixed record. In other words, the *fixed_part* consists of one or more fields each having the following format:

*field_name1*, ... *field_nameN* : *typename*;

The *variant_part* of a variant record takes the following format:

```
case [tag_field] typename of
    constantlist1 :  (field; ... fieldN);
            .
            .
            .
    constantlistN :  (field; ... fieldN);
```

The *constantlist* is one or more constants that share the same data type. For instance, if *typename* is **integer**, then every constant in *constantlist* must be an integer. You associate one or more **fields** with each *constantlist*. With one exception, each field has the same syntax as a field in the fixed part. The one exception is that *fieldN* can itself be a **variant_part**.

Note that you can optionally associate a *tag_field* with the *typename*. The *tag_field* is simply an identifier followed by a colon (:). You can use the *tag_field* to select the desired variant at run time. For more information on *tag_fields*, see the "Record Operations" listing in Chapter 4.

Consider the following declaration for variant record type **worker. Worker** contains a fixed part and a variant part. The fixed part contains two fields (**employee** and **id_number**). The *typename* of the variant part is **worker_groups**, which is an enumerated type. Wo has two possible values, **exempt** and **non_exempt**. When **wo** is **exempt**, the field name is **yearly_salary** which is an **integer32** data type, and when **wo** is **non_exempt**, the field name is **hourly_wage** which has a real data type.

```
TYPE
   worker_groups = (exempt, non_exempt);                    {enumerated type}
   worker = record                                              {record type}
            employee : array[1..30] of char;           {field in fixed part}
            id_number : integer16;                     {field in fixed part}
   CASE wo : worker_groups OF                                  {variant part}
                 exempt : (yearly_salary   : integer32);
                 non_exempt : (hourly_wage : real);
   end:
```

Consider the following declaration for **my_code**, a variant record that does not contain a fixed part. The data type of the variant part is integer, so the **case** portion declares integer constants. Choosing 1, 2, 3, and 4 as the constants is totally arbitrary; you could pick any four integers. These constants serve no purpose except to establish the fact that there are four choices. The fields themselves provide four different ways to view the same 4-byte section of main memory.

```
my_code = record
     CASE integer OF                                           {variant part}
               1 : (all : array[1..4] of char);
               2 : (first_half : array[1..2] of char;
                    second_half : array[1..2] of char);
               3 : (x1   : integer16;
                    x2   : boolean;
                    x3   : char);
               4 : (rall : single);
     end;
```

**NOTE:** The preceding example shows four parts that take up exactly four bytes. However, it is perfectly valid to declare parts that take up differing numbers of bytes.

### 3.10.3 Unpacked Records and Packed Records

Domain Pascal supports regular (unpacked) records and "packed" records. You declare a packed record by putting the keyword **packed** prior to **record** in the record declaration; for example:

```
VAR
   student : PACKED record
             ages : 10..20;
             grade : (seventh, eighth, ninth, tenth, eleventh, twelfth);
             graduating : boolean;
   end;
```

The advantage to declaring a packed record is that it can save space. The disadvantage is that you cannot pass a field from a packed record as an argument to a procedure (including predeclared procedures like **read**). The next subsection details the space savings of packing. Note that you should not directly manipulate fields in a packed record. If you want to perform some operation that changes the value of an existing field in a packed record, use the following steps:

1. Assign the value of the field to a variable of the same type.
2. Perform the operation on the variable.
3. Assign the value of the variable to the field of the packed record.

### 3.10.4 Initializing Data in a Record—Extension

Domain Pascal permits you to initialize a record in the variable declaration portion of the program unless that declaration comes within a procedure or function and the record has not been declared **static**. (See the "Accessing a Variable Stored in Another Pascal Module" and "Accessing a Routine Stored in Another Pascal Module" sections of Chapter 7 for more information on the **static** attribute.) You can initialize some or all of the fields in a record.

To initialize a field in a record, enter a declaration with the following format:

**var**
    *name_of_record_variable* : *type_of_record* :=
        [*init*,
            . ,
            . ,
            . ,
        *init* ];

where *init* is a statement having one of the following formats:

*field_name* := *initial_value*

or

*initial_value*

If you use the second format, Domain Pascal assumes that the *initial_value* applies to the next *field_name* in the record definition. For example, consider this record initialization:

```
TYPE
    messy = record
            rx  : real;
            c   : char;
            abc : array[1..3] of integer;
            case integer of
               0 : (i32 : integer32);
               1 : (i16 : integer);
               2 : (hb, lb : char);
            end;

VAR
    very : MESSY :=
           [ c := 'X',
             [-1, -2, -3],
             rx := 123.456,
             'Y',
             lb := 'a',
             hb := 'z' ];
```

The preceding example initializes field **c** to 'X'. The next declaration [-1, -2, -3] applies to field **abc** (because it follows field **c**). Field **rx** gets initialized to 123.456. Then, field **c** gets reinitialized to 'Y' (because it follows field **rx**). Finally, the third field in the variant portion of the record gets initialized, with field **lb** getting the value 'a' and field **hb** getting set to 'z'.

### 3.10.5 Internal Representation of Unpacked Records

In order to understand the internal representation of unpacked records, you need to understand the following concepts:

- Alignment
- Natural alignment
- Guaranteed default alignment
- Default alignment
- Layout of records
- Memory allocation
- Arranging record fields in descending order by size

We describe each of these concepts in the following sections.

### 3.10.5.1 Alignment

An object's **alignment** is the set of addresses at which the compiler can allocate the object. For example, the compiler can allocate **byte aligned** objects on any byte boundary; it can allocate **word aligned** objects only at addresses that are evenly divisible by 2 (word boundaries, or **shortword** boundaries); and it can allocate **longword aligned** objects only at addresses that are evenly divisible by four (longword boundaries).

### 3.10.5.2 Natural Alignment

An object is **naturally aligned** if it begins at an address that is a multiple of its size in bytes. For example, a 2-byte integer is naturally aligned if it starts on an even address boundary. Similarly, an 8-byte double-precision floating-point number is naturally aligned if it starts on an address divisible by 8. A record is considered to be naturally aligned when it starts on a boundary that results in natural alignment for its fields.

Since **char** and **boolean** type values are one byte long, their natural alignment is byte aligned. Similarly, since **integer** and **integer16** values are two bytes long, their natural alignment is word aligned. Since **real, single, integer32,** and **pointer** types are four bytes long, their natural alignment is longword aligned. And, since **double** types are eight bytes long, their natural alignment is quadword aligned. Figure 3-7 illustrates natural alignment for these simple data types.

### 3.10.5.3 Guaranteed Default Alignment of Record Fields

A type's guaranteed default alignment is not necessarily the same as its natural alignment. A type's **guaranteed default alignment** is the alignment it is guaranteed if a field of that type is a component of a record. The guaranteed default alignment of types **char** and **boolean** is byte alignment, which also happens to be the natural alignment for these types. However, the guaranteed default alignment for the other unstructured types (**integer, integer16, real, single, integer32, pointer,** and **double**) is word alignment. Table 3-2 compares the guaranteed default and natural alignments of the simple data types.

Table 3-2. *Guaranteed Default and Natural Alignment for Simple Data Types*

| Data Type | Guaranteed Default Alignment | Natural Alignment |
|---|---|---|
| **char** | byte | byte |
| **boolean** | byte | byte |
| **integer** | word | word |
| **integer16** | word | word |
| **real** | word | longword |
| **single** | word | longword |
| **integer32** | word | longword |
| **pointer** | word | longword |
| **double** | word | quadword |

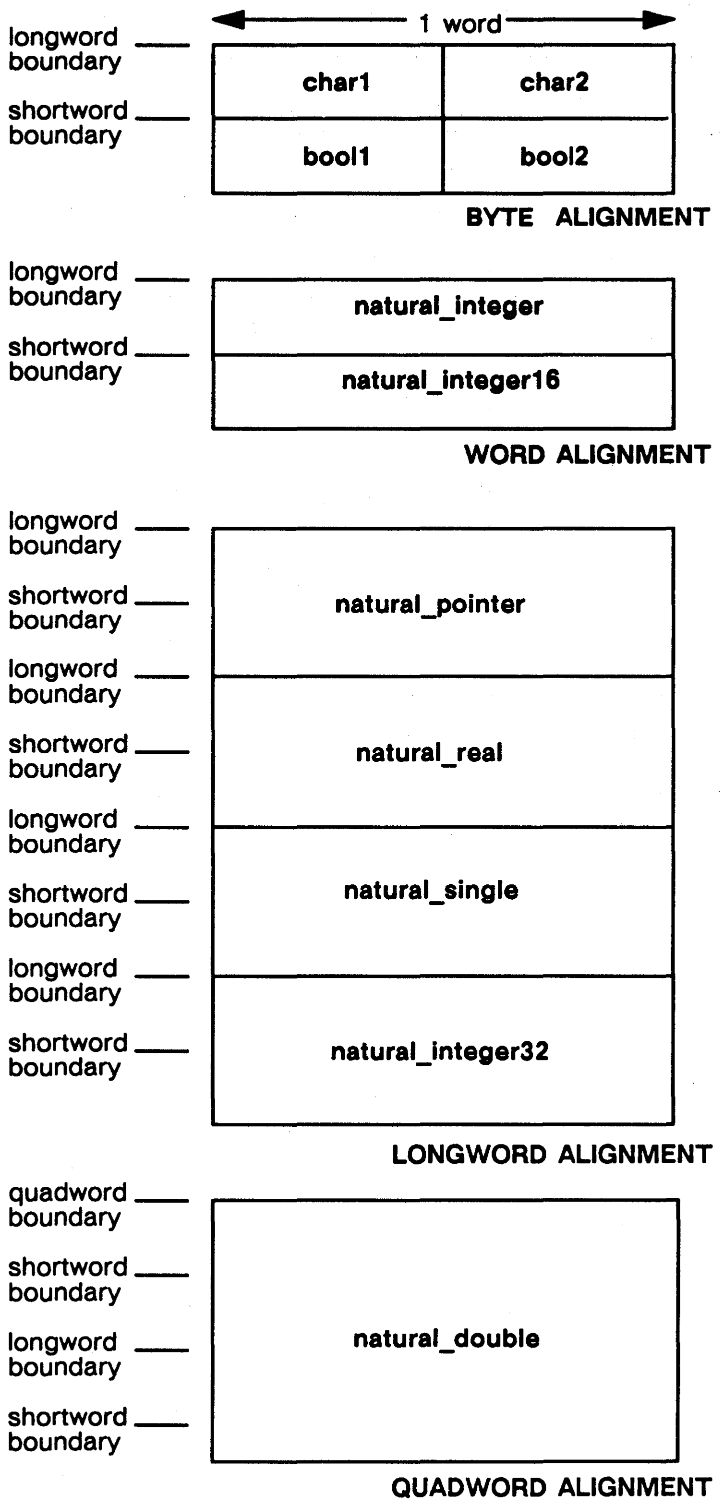

*Figure 3-7. Natural Alignment for Pascal Simple Data Types*

### 3.10.5.4 Default Alignment

The default alignment of a simple data type is the same as its natural alignment.

For the structured data types, such as records, default alignment is somewhat complex. The default alignment rules affect two properties of records:

- How fields are laid out in the record (whether padding is inserted between fields).
- How memory for the entire record is allocated.

We describe these two properties in the next two sections.

### 3.10.5.5 Layout of Unpacked Records

Domain Pascal follows these rules for the layout of unpacked records:

- The size of a record must be an even number of bytes. There is no way to override this rule. This means that the smallest possible record is 2 bytes.
- The default alignment of the beginning of a record is at least word aligned.
- By default, Domain Pascal allocates the same amount of space for each field in a record that the field would have required if it were not part of the record. For example, the compiler allocates one byte for a **char** field.
- By default, the compiler lays out record fields based on their guaranteed default alignment. Thus, all objects longer than a byte are aligned on word boundaries. Objects which are **chars** and **booleans** may be aligned on byte boundaries.
- By default, the compiler aligns a record according to the largest alignment of its fields.
- A byte aligned field may not cross two word boundaries. This means that a 32-bit object, such as an **integer32** type variable, cannot be byte aligned.

The default alignment rules for the layout of records can produce padding (also called "holes" or "gaps") in a structure. However, each gap is never larger than one byte.

For example, consider the following record declaration:

```
S1 = record
         a: integer32;
         b: char;
         c: integer16
     end;
```

Figure 3-8 shows how the fields for **S1** records are laid out. Note that there is a byte of padding inserted after **b** to ensure that **c** is aligned on a word boundary.

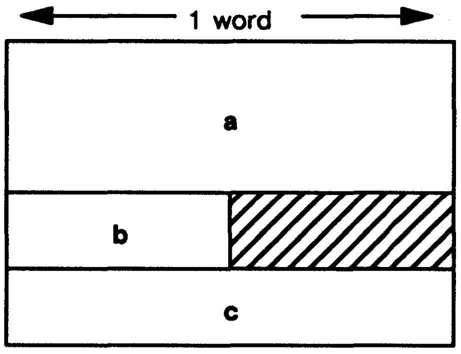

*Figure 3-8. Default Layout of Record S1*

Since the total size of a record must be a multiple of two bytes, records sometimes have a byte of padding at the end. Figure 3-9 shows the layout of a record that contains a gap in the middle and a gap at the end as a result of the default alignment rules.

```
S2 = record
        c1: char;
        s1: integer16;
        c2: char
     end;
```

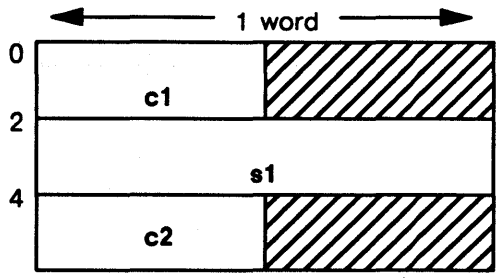

*Figure 3-9. Layout of Record S2*

### 3.10.5.6 Memory Allocation

Domain Pascal always allocates a record on at least a word boundary. It allocates a record on a larger boundary if that larger allocation produces natural alignment for any of the record's fields. Specifically, the compiler uses the following algorithm to allocate records:

1. It assumes that the starting address of the record is zero.
2. It lays out the fields in the order they are declared, noting which fields are naturally aligned.
3. It looks for the largest naturally aligned field.
4. The compiler allocates the entire record on a boundary that matches the natural alignment for the field it identified in Step 3.

Note that the compiler must lay out the record before it allocates memory for it.

Consider the following record type:

```
S3 = record
        a : integer32;
        b : char;
        c : integer16
     end;
```

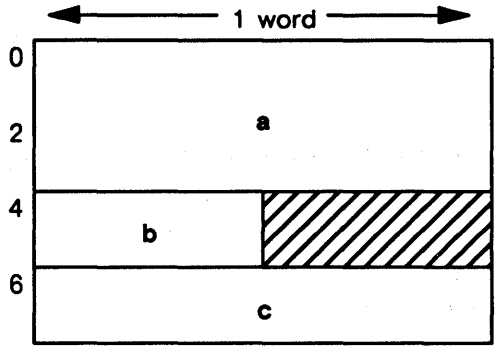

*Figure 3-10. Naturally Aligned Record S3 with 1-Byte Padding*

Figure 3-10 shows the layout for S3. The compiler assumes a beginning address of 0 and lays out the fields according to the default alignment rules. For this record, the default alignment rules produce a layout in which *all* elements are naturally aligned. The compiler then searches for the largest field that is naturally aligned, which is **a**. Since **a**'s natural alignment is longword, the compiler allocates records of type S3 on longword boundaries.

Consider a second example:

```
S4 = record
        a : integer16;
        b : integer32
     end;
```

Figure 3-11 shows the layout for S4. In this case, **a** is naturally aligned. However, **b** is not naturally aligned because its offset from the start of the record, which is 2, is not evenly divisible by its size, which is 4. The largest field in S4 that is naturally aligned, therefore, is **a**. The compiler uses word alignment, which is the natural alignment of **a**, to allocate records of type S4.

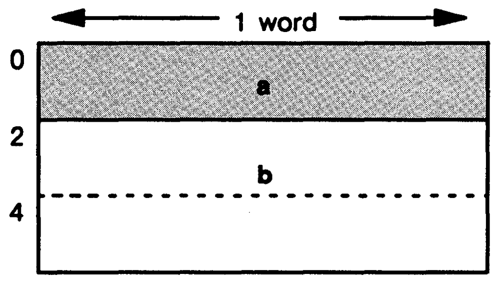

*Figure 3–11. Layout of S4 Using Word Alignment*

### 3.10.5.7 Arranging Record Fields in Descending Order by Size

You can usually improve the efficiency of memory accesses significantly by carefully arranging fields within records. The best guideline to improve access efficiency when you set up natural alignment for records is to *arrange record fields in descending order by size.* Consider the following example:

```
FOO = record
          a:char;
          b:integer16;
          c:char;
          d:double;
          e:integer32
      end;
```

By default, this declaration will produce the memory arrangement shown in Figure 3–12. Due to a poor ordering of fields, there are 4 bytes of gaps and members **d** and **e** are not naturally aligned.

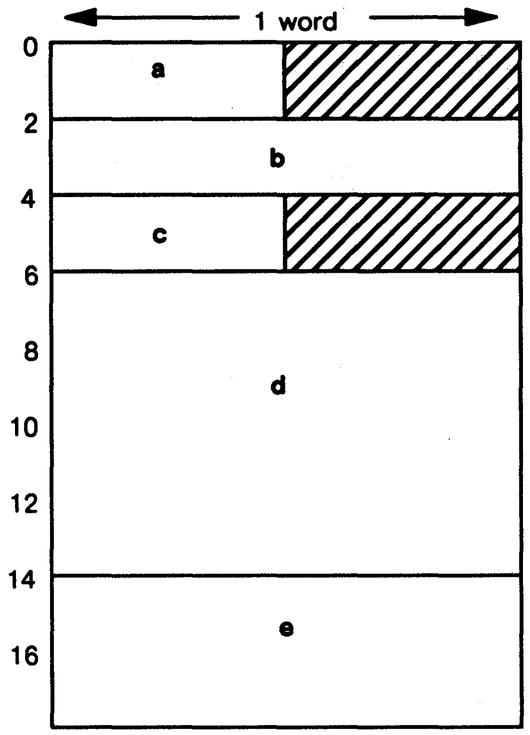

*Figure 3-12. Memory Arrangement for Record with Poorly Arranged Fields*

Following the preceding rule of arranging fields according to descending order by size, you change the declaration to:

```
FOO = record
         d: double;
         e: integer32;
         b: integer16;
         a, c: char
      end;
```

The above declaration produces the memory arrangement shown in Figure 3-13. All fields are naturally aligned and padding fields are not necessary to achieve natural alignment for the record.

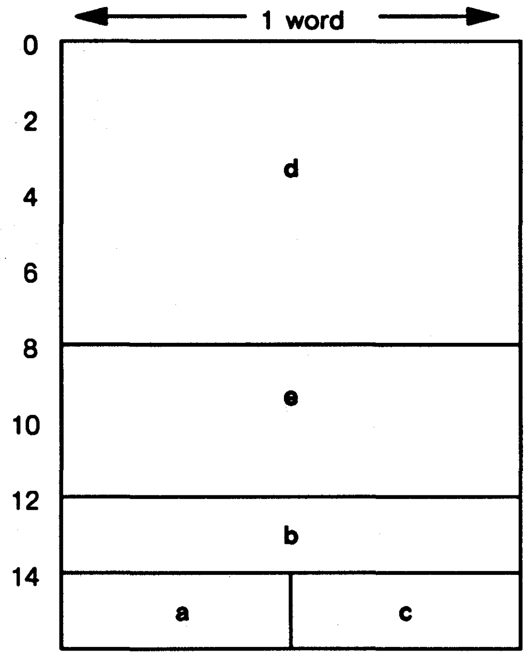

*Figure 3-13. Memory Arrangement According to Decreasing Size of Fields*

**NOTE:** You can usually guarantee that all fields of a record will be naturally aligned by arranging the fields in descending order of size. This will always work if all the fields are scalar objects. This technique may not work if one or more of the record fields is a record or array. Arranging fields in decreasing order of size also guarantees that there will be no padding between record fields, although there might still be a byte of padding at the end of the record to make it an even number of bytes.

In some instances, a record that would normally be allocated on a longword or quadword boundary receives a different allocation because the record is part of a larger aggregate type (e.g., a record or array). For example, consider the declaration of **S1**:

```
S1 = record
        x: integer32;
        y: integer16
     end;
```

The compiler can guarantee that an individual record of type **S1** will be allocated on a longword boundary (so that x and y will be naturally aligned), but if you declare an array of three **S1** records, only two-thirds of them will be aligned on longword boundaries:

```
array_of_s1_records : array [1..3] of S1;
```

Figure 3–14 shows the layout of an array of three **S1** records. Note that the second element is aligned on a word (2-byte) boundary, not a longword (4-byte) boundary, and so the second element is not naturally aligned.

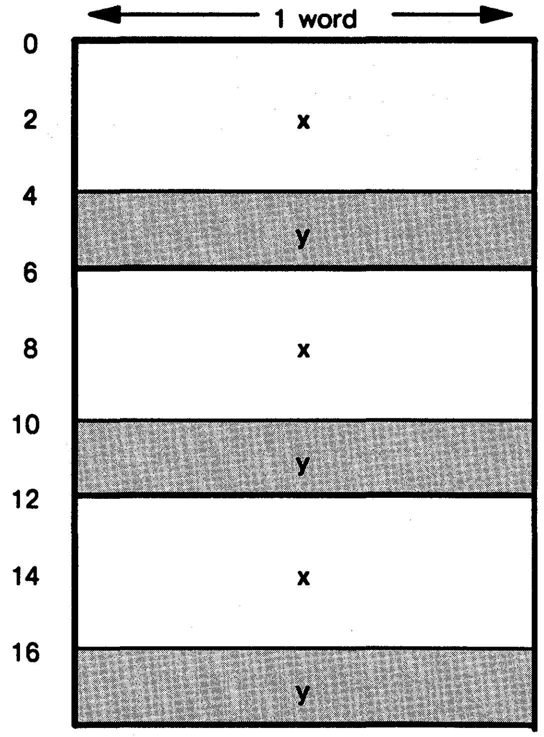

*Figure 3–14. Array of S1 Records, Not Naturally Aligned*

To ensure that all elements of **array_of_s1_records** are naturally aligned, you need to insert an additional word of padding at the end of **S1**. You could do this explicitly, as shown in the following declaration:

```
S1 = record
        x: integer32;
        y: integer16;
        padding: integer16
     end;
```

You could also tell the compiler to add the padding by using the **natural** or **aligned** attributes in the record declaration or by specifying a **%natural_alignment** compiler directive. See the "Attributes" section of this chapter for details about the **natural** and **aligned** attributes. See the "Compiler Directives" section of Chapter 4 for details about the **%natural_alignment** compiler directive.

## 3.10.6 Aligned Record and Unaligned Record Data Type

To make sure that records are always naturally aligned regardless of the alignment environment, you can use the **aligned record** data type. The alignment environment is the alignment that you set when you use compiler directives or the default alignment that is set by the compiler.

For example, the **%natural_alignment** directive tells the compiler to use natural alignment for any data that does not have an alignment attribute in its declaration. (See the "Compiler Directives" section of Chapter 4 for more details about the **%natural_alignment** and **%word_alignment** directives.)

To declare an **aligned record** data type, use the following syntax:

```
type

record_name =  aligned record
               field1;
               field2;
               ...
               fieldN;
               end;
```

Note that the **aligned record** data type sets alignment for records only. The **aligned** and **natural** attributes set alignment for records and fields.

The syntax for the **aligned record** data type is just like the syntax for a regular Domain Pascal record except that you use **aligned record** instead of **record**. For example, consider the following declaration:

```
TYPE
        nat_rec =  ALIGNED RECORD
                   a : integer;
                   b : integer32;
                   END;
```

The above declaration defines an object of type **nat_rec** to be a record laid out as shown in Figure 3-15.

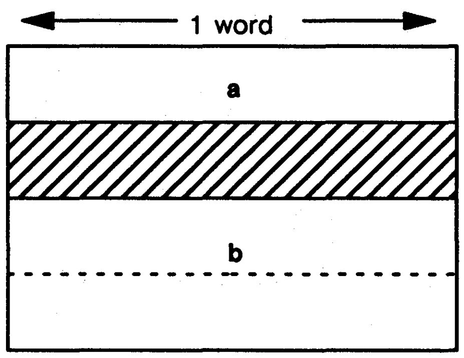

*Figure 3-15. An* **Aligned Record** *Type Record*

All **nat_rec** type objects will have the layout shown in Figure 3-15 even if they are in programs compiled with a **%word_alignment** directive.

To make sure that records are aligned according to word alignment rules regardless of their alignment environment, use the **unaligned record** data type. To declare an **unaligned record** type, use the following syntax:

```
type
record_name =   unaligned record
                field1;
                field2;
                ...
                fieldN;
                end;
```

Note that the syntax for the **unaligned record** data type is just like the syntax for a regular Domain Pascal record except that you use **unaligned record** instead of **record**. For example, consider the following declaration:

```
TYPE
        not_nat_rec = UNALIGNED RECORD
                      a : integer;
                      b : integer32;
                      END;
```

The preceding declaration defines an object of type **not_nat_rec** to be a record laid out as shown in Figure 3-16.

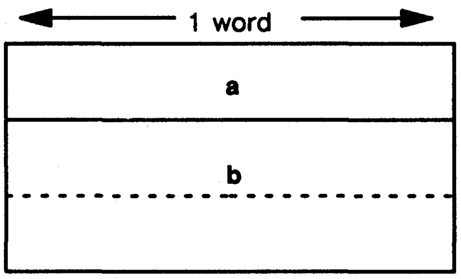

*Figure 3-16. An* **Unaligned Record** *Type Record*

All **not_nat_rec** type objects will have the layout shown in Figure 3-16 even if they are in programs compiled with a **%natural_alignment** directive.

## 3.10.7 Internal Representation of Packed Records

Table 3-3 shows the space required for fields in packed records.

Domain Pascal always starts the first field of a packed record on a word boundary. After the first field, if the exact number of bits required for the next field crosses zero or one 16-bit boundary, the field starts in the next free bit. If the field would cross two or more 16-bit boundaries, it starts at the next 16-bit boundary. Pascal allocates fields left to right within bytes and then by increasing byte address.

The minimum size of a packed record is 16 bits.

In packed records, characters are byte aligned. Structured types, except for sets, are aligned on word boundaries. Sets are aligned only if they cross two or more 16-bit boundaries.

*Table 3–3. Storage of Packed Record Fields*

| **Data Type of Field** | **Space Allocation** |
|---|---|
| Integer; Integer16 | 16 bits; word aligned. |
| Integer32, Real, Single | 32 bits; word aligned. |
| Double | 64 bits; word aligned. |
| Boolean | 1 bit; bit aligned. |
| Char | 8 bits; byte aligned |
| Enumerated | Number of bits required for largest ordinal value; bit aligned. |
| Subrange | Subrange of char fields require eight bits; all other subrange fields take up the number of bits required for their extreme values. Subrange of char fields are byte aligned. All other subrange fields are bit aligned. |
| Set | If fewer than 32 elements, then exactly one bit per element; if more then 32 elements, then same size as set. Bit aligned. |
| Array | May or may not be packed; requires the same space as an array outside of a packed record. (See the "Internal Representation of Arrays" section.) |
| Pointer | 32 bits; word aligned. |

The following type declaration, along with Figure 3–17, illustrates the storage of a packed record type.

```
TYPE
    Shapes = (Sphere, Cube, Ovoid, Cylinder, Tetrahedron);
    Uses = (Toy, Tool, Weapon, Food);
    Characteristics = PACKED RECORD
                      Mass     : Real;
                      Shape    : Shapes;
                      B        : Boolean;
                      Purpose  : SET OF Uses;
                      Low_temp : -100..40;
                      Class    : 'A'..'Z';
    end;
```

The fields require the following number of bits:

```
Mass                 32 bits  (word aligned)
Shape                 3 bits  (bit aligned)
B                     1 bit   (bit aligned)
Purpose               4 bits  (bit aligned)
Low_temp              8 bits  (bit aligned)
Class                 8 bits  (word aligned)
```

(The variable **low_temp** requires eight bits because it can take a range of 140 values (-100 to +40) and seven bits can represent only 128 values.)

Domain Pascal represents fields in the same order you declared them, as shown in Figure 3-17.

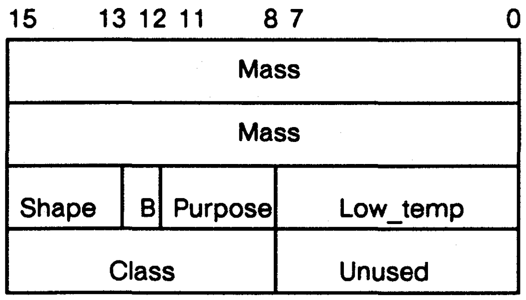

*Figure 3-17. Sample Packed Record*

In this example, the order of field declaration has been chosen very carefully. The whole record takes up only eight bytes, and out of the eight bytes, only eight bits are unused. If the fields had been declared in a different order, the record might have taken up 10 or 12 bytes.

## 3.11 Arrays

Like standard Pascal, Domain Pascal supports array types. An array consists of a fixed number of elements of the same data type. This data type is called the component type. The component type can be any predeclared or user-declared data type.

Use an index type to specify the number of elements the array contains. The index type must be a subrange expression. Domain Pascal permits arrays of up to eight dimensions. Specify one subrange expression for each dimension.

This fragment includes declarations for five arrays:

```
TYPE
    elements = (H, He, Li, Be, B, C, N, O, Fl, Ne);
                                          {elements is an enumerated type.}
VAR
                                    {Here are the five array declarations.}
    test_data      : array[1..100] of INTEGER16;
    atomic_weights : array[H..Be] of REAL;         {Range defined in TYPE}
                                                   {declaration.         }
    last_name      : array[1..15] of CHAR;
    a_thought      : STRING;
    lie_test       : array[1..4,1..2] of BOOLEAN;  {2-dimensional array.}
```

Notice that variable **a_thought** is of type **string. String** is a predefined Domain Pascal array type. Domain Pascal defines **string** as follows:

```
TYPE
    string = array[1..80] of CHAR;
```

In other words, **string** is a data type of 80 characters. Use **string** to handle lines of text conveniently.

See the "Array Operations" listing in Chapter 4 for a description of array bound checking.

## 3.11.1 Initializing Array Variables—Extension

If an array has been declared **static** or its declaration is not in a function or procedure, you can initialize array components in Domain Pascal in a variable declaration statement. (See the "Accessing a Variable Stored in Another Pascal Module" and "Accessing a Routine Stored in Another Pascal Module" sections of Chapter 7 for more information on the **static** attribute.) Domain Pascal initializes only those components for which it finds initialization data; it does not initialize the other components. For example, if an array consists of 10 components and you specify six initialization constants, Domain Pascal initializes the first six components and leaves the remaining four components uninitialized.

This section describes the following three basic methods of initializing single-dimensional and multidimensional arrays:

- Initializing multiple components to a single value
- Initializing components individually
- Initializing using repeat counts

In addition, this section describes controlling the order of initialization and setting a default for the size of the array.

### 3.11.1.1 Initializing Multiple Components with a Single Expression—Extension

To initialize multiple components with a single expression, specify an assignment operator (:=) followed by the value. This initialization method is especially useful for components of type **char**, where the value is a string of characters. You can also initialize the **char** components individually, but it's usually easier to do it as a string.

For example, consider the following single-dimensional initializations:

```
CONST
    msg1 = 'This is message 1';

VAR
    s1          : array[1..40] of CHAR := 'Quoted strings are ok';
    s2          : array[1..30] of CHAR := msg1;
    blank_line  : STRING               := chr(10);  {Newline}
```

These initializations assign elements 1 through 21 of array **s1** with the string "Quoted strings are ok", elements 1 through 17 of array **s2** with "This is message 1", and element 1 of **blank_line** with the newline character.

You can also use this method to initialize multidimensional arrays. To do this, you must assign the value to each row. For example the following initialization statement initializes columns 1 through 6 in rows 1 and 2 to 0:

```
VAR
    I2 : array[1..2, 1..6] of INTEGER16 := [
                                            [1..6 := 0],
                                            [1..6 := 0],
                                           ];
```

### 3.11.1.2 Initializing Components to Individual Values—Extension

To initialize array components individually, specify an assignment operator (:=) followed by the values to which the components should be initialized. You must enclose the values inside a pair of brackets and separate each value with a comma.

For example, consider the following single-dimensional array initializations:

```
VAR
    I : array[1..6] of INTEGER16 := [1, 2, 4, 8, 16, 32];
    R : array[1..3] of SINGLE := [-5.2, -7.3, -2E-3];
    B : array[1..5] of BOOLEAN := [true, false, true, true, true];
```

You can also use this method (and the method described in Section 3.11.1.1) to initialize multidimensional arrays. For example, the following fragment initializes two multidimensional arrays (**I2** and **B2**). **I2** has two subrange index types (1..2 and 1..6). The first index type consists of two values, so you must supply two rows of brackets. The second index is 1..6, so you must specify six values for each row, initializing a total of 12 components.

```
VAR
    I2 : array[1..2, 1..6] of INTEGER16 := [
                                           [1, 2, 3, 4, 5, 6],
                                           [7, 5, 9, 1, 2, 8],
                                          ];

    B2 : array[1..4, 1..3] of BOOLEAN := [
                                         [true, true,  true],
                                         [true, false, false],
                                         [false, true, false],
                                         [false, false, false],
                                        ];
```

### 3.11.1.3 Initializing Arrays Using Repeat Counts—Extension

**Repeat counts** let you initialize groups of elements in an array. There are two forms of repeat counts.

The first form takes the following syntax:

*n* **of** *constant*

where **of** is a keyword, *n* is an integer, and *constant* is any valid constant that corresponds to the data type specified in the declaration of the array.

This form tells the compiler to initialize *n* components of the array to the value of *constant*. *n* can be an integer or an expression that evaluates to an integer. The following initializations demonstrate this form of the repeat count:

```
CONST
    x = 50;
VAR
    a : array[1..1024] of INTEGER16 := [512 OF 0, 512 OF -1];
    b : array[1..400] of REAL := [x of 3.14, 400-x OF 2.7];
```

In the preceding example, Domain Pascal initializes the first 512 values of array **a** to 0 and the second 512 values to −1. Domain Pascal also initializes the first 50 components of array **b** to 3.14 and the remaining 350 components to 2.7.

The second form of the repeat count takes the following syntax:

*** of** *constant*

The asterisk (*) tells Domain Pascal to initialize the remainder of the components in the array to the value of *constant*. The following initializations demonstrate this form of the repeat count:

```
VAR
    c : array[1..2000] of INTEGER16 := [* of 0];
    d : array[1..50] of BOOLEAN := [12 of true, * of false];
```

In the preceding example, Domain Pascal initializes all 2000 components in array **c** to 0. Domain Pascal also initializes the first 12 components of array **d** to **true** and the remaining 38 components to **false**.

You can use repeat counts to initialize multidimensional arrays. You must initialize a multidimensional array column by column rather than all at once. For example, compare the right and wrong ways to initialize a 2-dimensional array:

```
VAR
    x : array[1..2, 1..5] of INTEGER16 := [10 of 0];   {Wrong!}
    y : array[1..2, 1..5] of INTEGER16 := [* of 0];    {Wrong!}
    z : array[1..2, 1..5] of INTEGER16 := [
                                           [5 of 0],
                                           [5 of 0],
                                          ];           {Right!}
    q : array[1..2, 1..5] of INTEGER16 := [
                                           [* of 0],
                                           [* of 0],
                                          ];           {Right!}
```

### 3.11.1.4 Initializing Components in Any Order—Extension

You can initialize array components in any order. Consider the following initialization statements:

```
VAR
    C1 : array[1..7] of INTEGER :=
           [
            3..4 := 2,
            7    := -1
            1    := 3
           ];
```

These statements produce the following initializations:

Component 1 = 3
Component 2 = undefined
Component 3 = 2
Component 4 = 2
Component 5 = undefined
Component 6 = undefined
Component 7 = -1

### 3.11.1.5 Defaulting the Size of an Array—Extension

When you initialize an array in the **var** declaration part of a program, you can let Domain Pascal determine the size of the array for you. To do this, put an asterisk (*) in place of the upper bound of the array declaration.

For example, in the following fragment, the upper bound of **init4** is 18; the upper bound of **init5** is 22; and the upper bound of **init6** is 4. The compiler defines the upper bound once it has counted the number of initializers.

```
CONST
    msg5 = 'And this is message 5.';

VAR
    init4 : ARRAY[1..*] of CHAR := 'This is message 4.';
    init5 : ARRAY[1..*] of CHAR := msg5;
    init6 : ARRAY[1..*] of INTEGER16 := [1, -17, 35, 46];
```

> NOTE: You can use an asterisk in the index type only if you supply an initialization value for the array. For example, the following fragment causes a "Size of TABLE1 is zero" warning:
>
> ```
> VAR
>     table1 : array[1..*] of integer16;
> ```

### 3.11.1.6 Mixing Methods of Array Initialization—Extension

You can combine all methods of array initialization described in this section. The following example illustrates the use of the methods to initialize an array named **array_example**:

```
CONST
      x = 100;

TYPE
      subr = ( one, two, three, four, five, six, seven, eight, nine,
               ten, eleven, twelve );

VAR
      array_example ; ARRAY [subr] OF integer :=
           [
             2 of -1,                 {repeat count method}

             four..five := 47,        {initializing multiple components
                                      to a single value}

             2 of x,                  {repeat count method}

             42,                      {initializing components
                                       individually}

             ten..* := 814            {initializing multiple components
                                      to a single value}

            ];
```

The results of the preceding declarations are:

Component 1 = -1
Component 2 = -1
Component 3 = undefined
Component 4 = 47
Component 5 = 47
Component 6 = 100
Component 7 = 100
Component 8 = 42
Component 9 = undefined
Component 10 = 814
Component 11 = 814
Component 12 = 814

### 3.11.2 Variable-Length Arrays—Extension

Domain Pascal supports an array data type that enables you to construct and manipulate variable-length strings. A **variable-length string** is a string whose length can change dynamically during program execution. This is in contrast to a fixed-length string whose length is specified by the subrange expression in its declaration. Although the size of a variable-length string can change, it cannot exceed a maximum, which you specify in the declaration.

The syntax for declaring a variable-length string is:

> **varying[*max_length*] of char;**

where *max_length* is a positive integer between 1 and 65535.

For example, the following are legal declarations:

```
VAR
    short_string     : VARYING[10] of CHAR;
    long_string      : VARYING[10000] of CHAR;
    array_of_var_str : ARRAY[1..5] of VARYING[20] of CHAR;
```

It is illegal to use the **varying** type specifier with any type other than **char**.

When you declare a variable-length string, the compiler allocates enough memory to hold a string of the maximum length. You can assign strings of any length not greater than the maximum to the array. When the variable-length string is read, only as many characters as indicated by the current length are accessed.

Internally, the compiler translates the **varying** type into a record of the form:

```
RECORD
     length : 0..65535;
     body : PACKED ARRAY[1..max_length] of CHAR;
     {unnamed filler}
END;
```

The **length** field contains the current length of the string. When you assign a string to a variable-length character array, the compiler adjusts the value of **length** to reflect the size of the assigned string. The **body** field contains the actual string. Following the body field, the compiler allocates additional bytes, if necessary, for a trailing null character and padding. The trailing null character and pad bytes are described in the following section.

You can reference the **length** and **body** fields explicitly by using the same syntax and semantics you would use for a normal record. For example:

```
cur_length := short_string.length;
last_char := short_string.body[short_string.length];
```

Be aware that if you update the character string in the **body** field, it will not automatically update the value in the **length** field.

It is illegal to subscript a variable-length array. For example:

```
VAR
     var_string   :  VARYING[100] of CHAR;
          .
          .
     var_string[5] := 'a';    { ILLEGAL }
```

To access a particular element in a variable-length string, subscript into the **body** field.

## 3.11.3 Packed Arrays

Like standard Pascal, Domain Pascal allows you to use packed arrays to store sequences of characters, byte integers, and boolean variables. To declare a packed array, use the following format:

**var**

*name_of_array* : **packed array** [*low_value..high_value*] **of** *variable_type*;

For example, consider the following variable declarations:

```
VAR
    switches : PACKED ARRAY [1..100] OF boolean;
    names : PACKED ARRAY [1..25] OF char;
    numbers : PACKED ARRAY [0..100] OF integer;
```

Although you can save space by using packed arrays, you may pay a price in the efficiency of loading from and storing to elements in the arrays.

## 3.11.4 Internal Representation of Arrays

Packed arrays usually require less storage space than arrays that are not packed. In this section we describe how both types of arrays are represented internally.

### 3.11.4.1 Non-Packed Arrays

With two exceptions, the total amount of memory required to store an array that is not packed equals the number of elements in the array times the amount of space required to store one element. The amount of space for one element depends on the component type of the array, as shown in Table 3-4.

*Table 3-4. Size of One Element of an Array*

| Base Data Type | Size of One Element |
|---|---|
| Integer16 or Integer | 16 bits |
| Integer32 | 32 bits |
| Single or Real | 32 bits |
| Double | 64 bits |
| Boolean | 8 bits |
| Char | 8 bits |
| Subrange | size of base type of subrange |
| Enumerated | 16 bits |

The two exceptions to this rule are arrays of **booleans** and **varying** arrays of **chars**.

If the component type of an array is **boolean** and an odd number of elements are declared, the compiler adds an extra pad byte to the storage space for the array so that the storage is an even number of bytes. For example, if you declare **boolean** array **b** as

```
VAR
    b  : array[1..5] of boolean;
```

Domain Pascal reserves six bytes of memory for **b**.

A variable-length string always begins with a 2-byte **length** field, followed by the **body** field, followed by padding. The **body** field must always be capable of holding a null character at the position just beyond the current length. Consequently, the compiler always allocates to the **body** field one byte more than the maximum length specified in the declaration. The compiler adds 2 or 3 bytes of padding to make the total size of the string an even number of longwords (4 bytes).

In summary, to figure the total number of bytes allocated for a variable-length string: add either 5 or 6 bytes to the number of bytes specified as the maximum length for the string such that the total number of bytes is an even number. For instance:

```
VAR
         v2:  VARYING[2] OF CHAR;   {8 bytes are allocated}
         v7:  VARYING[7] OF CHAR;  {12 bytes are allocated}
         v8:  VARYING[8] OF CHAR;  {14 bytes are allocated}
```

## Multidimensional Arrays

Multidimensional arrays are stored in row major order. Given a 2-dimensional array of the following declaration:

```
a : array[1..2, 1..3] of integer16;
```

Domain Pascal represents it in the following order:

```
a[1,1] first
a[1,2] second
a[1,3] third
a[2,1] fourth
a[2,2] fifth
a[2,3] sixth
```

### 3.11.4.2 Packed Arrays

Table 3-5 shows the space required for each type of element in a packed array.

*Table 3-5. Storage of Packed Array Elements*

| Type of Element | Space Allocation |
|---|---|
| Subranges of integers<br>Enumerated<br>Subrange of enumerated | Exact number of bits required *;<br>bit aligned ** |
| Integer<br>Integer16 | 16 bits; word aligned |
| Boolean | 1 bit; bit aligned |
| Real<br>Integer32<br>Pointer | 32 bits; word aligned |
| Double | 64 bits; word aligned |
| Character<br>Subrange of character | 8 bits; byte aligned |
| Character array | Exact number of bytes required; byte aligned |
| Record<br>Array of non-characters | Exact number of words required; word aligned |
| Set | Exact number of bits required [1];<br>bit aligned [2] |

[1] If the element size is less than 16 bits, the element is padded up to the nearest power of 2 bits.
[2] If the element size is greater than 16 bits, then the elements are word aligned.

In packed arrays, if the element size is less than 16 bits, then the element is padded up to the nearest power of 2 bits. Thus, in the following example, 4 bits would be used to store each element of **array1** and 2 bytes would be used to store the entire **array1**:

```
array1: PACKED ARRAY [1..3] OF 0..7
```

Contrast this to the following declaration:

```
array2: ARRAY [1..3] OF 0..7
```

Each element of **array2** requires 16 bits of storage, and 48 bits would be used to store the entire **array2**.

## 3.12 Files

When you open a file for I/O access, you must specify a file variable that will be the pseudonym for the actual pathname of the file. Thereafter, you specify the file variable (not the pathname) to refer to the file. Domain Pascal supports the **file** data type and the **text** data type. (Throughout this manual, the word "**file**," in boldface type, means the **file** data type, and the word "file," in roman type, means a disk file.) Files of both **file** type and **text** type are stored as Domain **unstruct** (unstructured ISO Latin–1) files. These files are compatible with text files produced under UNIX systems.

The following declaration establishes variable **f1** as an identifier of a **text** file:

```
VAR
    f1 : text;
```

A **text** file contains sequences of ISO Latin–1 characters representing variable-length lines of text. You can read or write entire lines of a **text** file. You can read from or write to a **text** file the values of a variable of any type (except **pointer** and **file**). Chapter 8 describes **text** files in more detail.

You specify a **file** variable with the following format:

*variable* : **file of** *base_type*;

A variable with the **file** type specifies an **unstruct** binary file composed of values having the **base_type**. That is, the only permissible values in such a file all have the same data type, that of the **base_type**. The **base_type** can be any type except a pointer, **file**, or **text** type. For example, the following declaration creates a file type corresponding to a file that consists entirely of student records:

```
TYPE
    student = record
              name : array[1..30] of char;
              id   : integer32
    end;
    U_of_M  : FILE OF student;
```

The Domain/OS operating system stores each occurrence of **student** in 34 bytes: 30 bytes for **name** and 4 bytes for **id**.

**NOTE:** Older versions of Domain Pascal created special record structured Domain files (called "**rec**" files) when you opened a file with **file** type. For compatibility with older versions, the current version of Domain Pascal allows you to manipulate **rec** files, but you cannot create them. When you open an existing file with the **file** type, Domain Pascal checks whether it is a **rec** or **unstruct** file, and accesses it appropriately. Whenever you open a *new* file, however, Domain Pascal creates an **unstruct** file.

## 3.13 Pointers

A pointer variable points to a dynamic variable. In Domain Pascal, the value of a pointer variable is a variable's virtual address. Domain Pascal supports the pointer type declaration of standard Pascal as well as a special **univ_ptr** data type and procedure and function pointer types. This section details the declaration of pointer types. You should also refer to the "Pointer Operations" listing of Chapter 4 for information on using pointers in your programs.

### 3.13.1 Standard Pointer Type

To declare a pointer type, use the following format:

**type**
    *name_of_type = ^typename;*

You can specify any data type for *typename*. The pointer type can point only to variables of the given *typename*. For example, consider the following pointer type and variable declarations:

```
TYPE
    ptr_to_int16 = ^integer16;      {Points only to integer16 variables.}
    ptr_to_real  = ^real;                {Points only to real variables.}
    studentptr = ^student;     {Points only to student record variables.}
    student = record
              name : array[1..25] of char;
              id   : integer;
              next_student : studentptr;
    end;

 VAR
    x  : integer16;
    p_x  : ptr_to_int16;
    half_life : real;
    p_half_life : ptr_to_real;
    Brown_Univ : student;
```

## 3.13.2 Univ_ptr—Extension

The predeclared data type **univ_ptr** is a universal pointer type. A variable of type **univ_ptr** can hold a pointer to a variable of any type.

You can use a **univ_ptr** variable *only* in the following contexts:

- Comparison with a pointer of any type
- Assignment to or from a pointer of any type
- Formal or actual parameter for any pointer type
- Assignment to the result of a function

Note that you *cannot* dereference a **univ_ptr** variable. Dereferencing means finding the contents at the logical address that the pointer points to. You must use a variable of an explicit pointer type for the dereference. Please see the "Pointer Operations" listing in Chapter 4 for more information on **univ_ptr**.

## 3.13.3 Procedure and Function Pointer Data Types—Extension

Domain Pascal supports a special pointer data type that points to a procedure or a function. By using procedure and function data types, you can pass the addresses of routines obtained with the **addr** predeclared function. (See the **addr** listing of Chapter 4 for a description of this function.) You may obtain the addresses only of top-level procedures and functions; you cannot obtain the addresses of nested or explicitly declared **internal** procedures and functions. Note that the compiler checks pointers for data type and name compatibility when addresses are assigned to a pointer or procedure. (See Chapter 5 for details about declaring **internal** procedures and details about compatibility checking of pointers. See Chapter 7 for details about using **internal**.)

Procedure and function pointer type declarations are the same as regular procedure and function declarations, except for the following:

- The procedure or function has no identifier; in other words, the procedure or function does not have a name.
- The type declaration begins with a caret (^), as in standard pointer type declarations.

You can declare procedure and function pointers in either of two ways: in the **type** and **var** declaration parts or just in the **var** declaration part.

Here is an example of declaring a procedure pointer and a function pointer using both the **type** and **var** declaration parts:

```
TYPE
        proc_ptr = ^procedure (a,b,c: integer);
        func_ptr = ^function (x,y: real): real;
VAR
        my_proc_ptr: proc_ptr;
        my_func_ptr: func_ptr;
```

And here is an example of declaring the same pointers as above in just the **var** declaration part:

```
VAR
         my_proc_ptr: ^procedure (a,b,c: integer);
         my_func_ptr: ^function (x,y: real): real;
```

See the "Pointers to Routines—Extension" section of Chapter 5 for a sample program showing how to pass function pointers as parameters.

## 3.13.4 Initializing Pointer Variables—Extension

Domain Pascal permits you to initialize the values of pointer variables within its variable declaration in most cases. You initialize a value by placing a colon and equal sign (:=) immediately after the data type. For example, the following fragment declares **my_ptr** as a type **ptr_to_int16** with an initial value of **NIL**:

```
TYPE
    ptr_to_int16 = ^integer16;
VAR
    my_ptr : ptr_to_int16 := NIL;
```

If the variable declaration occurs within a procedure or function, you *cannot* initialize the variable at the declaration unless it has been declared **static**. This is because storage within routines is dynamic and so variables in them do not necessarily retain their values between executions. For example, the following is incorrect:

```
TYPE
    ptr_to_int16 = ^integer16;

FUNCTION do_nothing(IN OUT x : INTEGER) : BOOLEAN;
VAR
    my_ptr : ptr_to_int16 := NIL;              {Wrong!}
```

This is the correct way to initialize the variable at its declaration in a routine:

```
my_ptr : STATIC ptr_to_int16 := NIL;  {Right!}
```

See Chapter 7 for information on the **static** attribute.

## 3.13.5 Internal Representation of Pointers

Domain Pascal stores pointer variables in the 32-bit record shown in Figure 3-18.

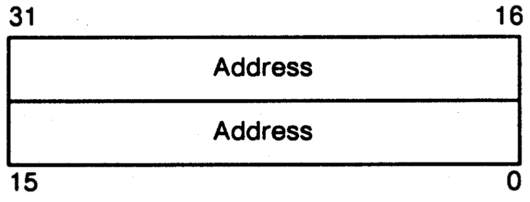

*Figure 3-18. Pointer Variable Format*

By default, pointer-type objects are stored on longword boundaries.

A pointer to a procedure or function (a Domain Pascal extension) points to the starting address of that routine.

## 3.14 Putting Variables into Overlay Sections—Extension

A **section** is a named area of code or data. An **overlay** section is a section whose contribution from a module "overlays" that of other modules. A Domain Pascal overlay section is just like a named common block in FORTRAN.

At run time, the code or data in a particular section occupies contiguous logical addresses. By default, all variables that you declare in a **var** declaration part are stored in the **.data** section. However, Domain Pascal lets you assign variables to sections other than **.data**.

To specify an overlay data section, place the section name inside a set of parentheses after the reserved word **var**. The section name can be any valid identifier or a string within quotation marks. For example, both of the following are valid names for a section:

```
this_is_a_section_name

'this is a section name'
```

The following is the format to declare a section name for a **var** declaration part.

```
var (section_name)
    identifier_list1 : typename1;
  [      ...
    identifier_listN : typenameN;]
```

All the variables named in all the **identifier_lists** will be stored in **section_name**. Since you can put multiple **var** declaration parts in the same program, you can create multiple named sections. If you do not specify a **section_name,** Domain Pascal puts the variables in the **.data** section.

Domain Pascal allocates variables defined in a **var** declaration part sequentially within the specified section. If more than one **var** declaration specifies the same section name, the subsequent declarations are considered to be continuations of the first declaration.

By forcing certain variables into the same section, you can reduce the number of page faults and thus make your program execute faster. For example, suppose you declare the following three variables:

```
VAR
    x      : integer16;
    b_data : array[1..5000] of integer16;
    y      : single;
```

Further suppose that whenever you need the value of **x**, you also need the value of **y**. By default, Domain Pascal places **x**, **b_data**, and **y** inside the **.data** section. The **.data** section encompasses 10 pages (1 page = 1024 bytes). There is no way to ensure that **x** and **y** will be on the same page in **.data** because Domain Pascal might place **b_data** in between **x** and **y**. However, by putting **x** and **y** in the same named section, you can improve the odds to over 99%. For example, to put **x** and **y** into section **important**, you must issue the following declarations:

```
VAR (important)
    x : INTEGER16;                    {will go into section "important"}
    y : REAL;                         {will go into section "important"}

VAR
    b_data : array[1..5000] of INTEGER16;  {will go into section ".data"}
```

See the "Section—Extension" section of Chapter 5 for information about using sections within routines. Sections are important at bind time. For complete information on the Domain binder, see the *Domain/OS Programming Environment Reference.*

## 3.15 Attributes for Variables and Types—Extension

Domain Pascal supports attributes for variables and types. These attributes supply additional information to the compiler when you declare a variable or a type. The attribute names are:

- **volatile**
- **atomic**
- **device**
- **address**
- **bit, byte, word, long, quad** (size attributes)
- **aligned(n), natural** (alignment attributes)

The **volatile, atomic,** and **device** attributes enable you to turn off certain compiler optimizations that would otherwise ruin programs that access device registers or shared memory locations. The **address** attribute associates a variable with a specific virtual address. Alignment and size attributes enable you to enhance your program's performance by specifying storage allocation and data layout information.

Specify **volatile, device**, alignment, and size attributes inside a pair of brackets immediately before the type in the **type** or **var** declaration. For example:

```
TYPE
    int_array = [VOLATILE] array[1..10] of integer;

VAR
    x : [DEVICE] integer16;
```

Specify **address** and **atomic** attributes inside a pair of brackets immediately prior to the type in the **var** declaration. For example:

```
VAR
    x: [ATOMIC] integer;
```

To specify more than one attribute for a particular data type or variable, separate the attributes with commas. For example:

```
VAR
    x: [ATOMIC, DEVICE] integer;
```

Table 3-6 summarizes the attributes that Domain Pascal supports. The following subsections provide details about the attributes.

Table 3-6. Summary of Attributes for Variables and Types

| Attribute | Example of Syntax | Purpose |
|---|---|---|
| **volatile** | `TYPE`<br>`  int_array = [volatile]`<br>`       array[1..10]of integer;`<br>`VAR`<br>`  x: [volatile] integer;` | Prevent certain default optimizations based on assumptions about value. |
| **atomic** | `VAR`<br>`  x: [atomic] integer;` | Implies **volatile**. Also perform updates with single instruction when possible. |
| **device** | `TYPE`<br>` keyboard = [device] char;`<br>`VAR`<br>` x: [device] integer16;` | Implies **volatile**. Also prevent additional optimizations. |
| **address** | `VAR`<br>` peb_page: [address`<br>`   (16#ff7000),device]char;` | Bind a variable to a specified virtual address (use with **volatile** or **device**). |
| **SIZE ATTRIBUTES**<br>**bit**<br>**byte**<br>**word**<br>**long**<br>**quad** | `TYPE`<br>`  big_boo =`<br>`       [long]boolean;`<br>`VAR`<br>` large_boo:`<br>`       [long]boolean;` | Specify amount of storage for types and/or objects. |
| **ALIGNMENT ATTRIBUTES**<br>**aligned**<br>**natural** | `TYPE`<br>` word_aligned_integer32 =`<br>`     [aligned(1)] integer32;`<br>`VAR`<br>`  natural_int:[natural]integer;` | Control the data layout. |

## 3.15.1 Volatile—Extension

**Volatile** informs the compiler that memory contents may change in a way that the compiler cannot predict. There are two situations, in particular, where this might occur:

- The variable is in a shared memory location accessed by two or more processes.
- The variable is accessible through two different access paths. (That is, multiple pointers with different base types refer to the same memory locations.)

In both of these situations, it is crucial that you tell the compiler *not* to perform certain default optimizations.

For example, the following module causes optimizations leading to erroneous behavior:

```
Module volatile_example;

 VAR
    p : ^integer;

 Procedure Init(VAR v : integer);
 BEGIN
    p := addr(v);
 END;

 Procedure Update;
 BEGIN
    p^ := p^ + 1; {anonymous path.}
 END;

 Procedure Top;
 VAR
    i : integer;
 BEGIN
    Init(i);
    i := 0;            {Visible modification.     }
    while i < 10 do    {Visible reference.        }
       update;         {Hidden modification to i.}
 END;
```

However, you can prevent these destructive optimizations if you change the declaration of variable **i** to:

```
VAR
   i : [volatile] integer;
```

## 3.15.2 Atomic—Extension

You declare a variable to be **atomic** if you want to make sure that its value does not change unpredictably as a result of multiprocessing. **Atomic** prevents the same optimizations as **volatile.** In addition, the compiler handles any assignment statements whose left side is a variable specified as **atomic** and whose right side contains the same variable in a special way that protects the value of the variable from being changed by other processes.

**Atomic** attributes can only be used with scalar variables. The scalar data types are integer, Boolean, character, and enumerated.

For example, consider the following declaration:

```
VAR
         x: [atomic] integer;
```

The above declaration tells the compiler to make sure that the value of **x** is not changed by other processes while it completes an assignment statement such as the following:

```
x := x + 2;
```

## 3.15.3 Device—Extension

**Device** informs the compiler that a device register (control or data) is mapped to a specific virtual address. Device registers are memory locations bound to a specific device, such as a disk drive. The **device** attribute prevents the same optimizations that **volatile** prevents, and it also prevents two other optimizations, which are described below.

By default, the compiler optimizes certain adjacent references by merging them into one large reference. The **device** attribute prevents this optimization.

For example, consider the following fragment:

```
VAR
    a,b : integer16;
BEGIN
    a := 0;
    b := 0;
```

By default, the compiler optimizes the two 16-bit assignments by merging them into one 32-bit assignment. (That is, at run time, the system assigns a 32-bit zero instead of assigning two 16-bit zeros.) By specifying the **device** attribute, you suppress this optimization.

The **device** attribute also prevents the compiler from generating gratuitous read-modify-write references for device registers. That is, specifying a variable as **device** causes the compiler to avoid using instructions that do unnecessary reads.

Now, consider an example. Suppose **kb** in the following fragment is a device register that accepts characters from the keyboard.

```
TYPE
    keyboard = char;

 VAR
    c, cl : char;
    kb    : ^keyboard;

 BEGIN
    c  := kb^;
    cl := kb^;
```

The purpose of the program is to read a character from the keyboard and store it in **c**, then read the next character and store it in **c1**. However, the compiler, unaware that the value of **kb** can be changed outside of the block, optimizes the code as follows: it stores the value of **kb** in a register, and thus assigns the same value to both **c** and **c1**. Obviously, this is not what the programmer intended since Domain Pascal assigns the same character to both **c** and **c1**. To ensure that Domain Pascal reads **kb** twice, declare it as:

```
TYPE
    keyboard = [DEVICE] char;
```

Another situation when normal optimization techniques can change the meaning of a program is in loop-invariant expressions. For instance, using the keyboard example again, suppose you have the program segment:

```
TYPE
    keyboard = char;

 VAR
    x  : integer;
    c  : char;
    kb : ^keyboard;
         .
         .
         .
    while (x < 10) do
       begin
       c := kb^;
       foo(c);
       x := x + 1;
       end;
```

The purpose of the block is to read 10 successive characters from the keyboard and pass each to a function called **foo**. However, to the compiler, it looks like an inefficient program since **c** will be assigned the same value 10 times. To optimize the program, the compiler may translate it as if it had been written as follows:

```
   c := kb^;
   while (x<10) do
       begin
       foo(c);
       x := x + 1;
       end;
```

To ensure that the compiler does not optimize your program in that manner, declare **kb** as follows:

```
TYPE
    keyboard = [DEVICE] char;
VAR
    kb : ^keyboard;
```

In addition to suppressing optimizations, you can also use **device** to specify that a device is either exclusively read from or exclusively written to. You achieve this by using the **read** and **write** options which have the following meanings:

- **Device**(read)—This attribute specifies read-only access for this variable or type. That is, if you attempt to write to this variable, the compiler flags the attempt as invalid and issues an error message.
- **Device**(write)—This attribute specifies write-only access for this variable or type. That is, if you attempt to read from this variable, the compiler flags the attempt as invalid and issues an error message.
- **Device**(read, write)—This attribute specifies both read and write access for this variable. This attribute is identical to the **device** attribute without any options.
- **Device**(write, read)—Same as **device**(read, write).

For example, here are some sample declarations using the **device** attributes:

```
TYPE
    truth_array : [DEVICE] array[1..10] of boolean;
VAR
    c  : [DEVICE(read)] char;                          {read-only access.}
    c2 : [DEVICE(write)] char;                        {write-only access.}
    t  : truth_array;                             {read and write access.}
```

### 3.15.4 Address—Extension

**Address** takes one required argument.

The **address** specifier binds a variable to the specified virtual address, specified by a constant. You can only use **address** in a **var** declaration, not in a **type** declaration.

**Address** is useful for referencing objects at fixed locations in the address space, such as device registers, the PEB (Performance Enhancement Board) page, or certain system data records. Typically, the compiler generates absolute addressing modes when accessing such an operand. You cannot specify **define, extern,** or **static** when you use this option.

Using **address** by itself (without **device** or **volatile**) does not suppress any compiler optimizations. You should use it in conjunction with **volatile** or **device**. The example below associates the variable **peb_page** with the hexadecimal virtual address FF7000.

```
VAR
    peb_page : [ADDRESS(16#FF7000), DEVICE(read)]  char;
```

## 3.15.5 Size—Extension

Size attributes specify the amount of storage to be reserved for the following:

- Variables
- Formal parameters
- Function results
- Record fields
- Array components

You declare size attributes in either of two ways—as part of a type-identifier or as part of an object. Declaring a size attribute as part of a type identifier indicates the amount of storage to be reserved for every object of that type. Declaring a size attribute as part of an object declaration specifies the amount of storage reserved for that particular object.

Domain Pascal supports five size attribute-names:

- **bit** (1 bit)
- **byte** (8 bits)
- **word** (16 bits)
- **long** (32 bits)
- **quad** (64 bits)

Each attribute-name represents a unit of storage. You indicate the number of units of storage to be reserved by including an integer-expression, *n*, in parentheses immediately after the attribute-name, as follows:

$$[\textit{attribute-name}\left[(n)\right]]\ \textit{typename}$$

If (*n*) is missing, then the compiler uses a default value of 1.

For example, each of these two variable declarations reserves 32 bits of memory for **original_value**:

```
TYPE
    big_number = [long] integer16;
VAR
    original_value : big_number;
```

and

```
VAR
    original_value : [long] integer16;
```

Compare the above declarations to the following declaration, which reserves 16 bits for **original_value**:

```
VAR
    original_value: integer16;
```

In Domain Pascal, the following size rules apply:

- Every data type has a minimum size.
- A size attribute must specify *at least* the number of bits required for the type to which it is applied.
- A size attribute can specify a number of bits *greater than* the minimum for **integer**, **boolean**, and **set** data types. If you use a size attribute to specify a size larger than the minimum for a data type, then the compiler uses all of the bits in the larger object to represent the value of an object of that type.

Table 3-7 shows the minimum number of bits for Domain Pascal data types.

*Table 3-7. Size of Simple Data Types*

| **Data Type** | **Minimum Size** |
|---|---|
| Integer16 or Integer | 16 bits |
| Integer32 | 32 bits |
| Single or Real | 32 bits |
| Double | 64 bits |
| Boolean | 8 bits |
| Char | 8 bits |
| Subrange | Number of bits needed to store subrange |
| Enumerated | 16 bits |

You can use size attributes to create bit arrays. Domain Pascal supports arrays of 1–, 2–, and 4–bit elements. The order of elements within words is from most significant bit to least significant bit. The only aggregate bit array operation that Domain Pascal supports is aggregate assignment. The following program illustrates the use of bit arrays:

```
program bit_array;
type
    element = [bit (4)]  0..1;
       {Declares a subrange type that occupies 4 bits.}
    arr = packed array [1..6] of element;
       {You need to say 'PACKED' to get the intended size, 3 bytes}
VAR
    a1 : arr;
begin
    a1[4]  := 2;
    writeln('The size of a1 is: ', sizeof(a1):4, ' bytes');
    writeln('          a1[4] = ', a1[4]:4 );
end.
```

The output of this program is as follows:

```
The size of a1 is:    3 bytes
           a1[4] =    2
```

You can also use size attributes to declare an array of byte integers, such as:

```
TYPE
   byte_integer = [byte] 0..255;
   byte_int_array = array[1..1000] of byte_integer;
```

Finally, size attributes are useful for declaring variables in Domain Pascal that you want to correspond to data types in other languages. For example, the Domain Pascal type **[byte] 0..255** corresponds to the **unsigned char** data type in Domain/C. (See Chapter 7 for a more detailed discussion of cross–language communication.)

Notice that you use size attributes in the declaration part of your program, whereas you use type transfer functions in the action part of your program. (See the "Type Transfer Functions" listing in Chapter 4 for details about type transfer functions.)

### 3.15.6 Alignment—Extension

Domain Pascal supports two alignment attributes: **aligned** and **natural**. In the following sections we tell you:

- The format for the **aligned** and **natural** attributes.
- How to use **natural** and **aligned** to align objects on natural boundaries.
- How to use **aligned** to prevent padding by default.

- How to use **aligned** to ensure the same record layout in all alignment environments.
- How to suppress informational messages about alignment by using the alignment attributes.
- How and why to use alignment attributes to inform the compiler that an object is not naturally aligned. This is especially important in connection with dereferencing pointers and passing arguments by reference.

### 3.15.6.1 Format for the aligned and natural Attributes

The format for the **aligned** attribute is:

**[aligned** [ (*n*) ] **]** *typename*;

where ***n*** is the number of low order zeros in the address (in binary representation). If you omit a value for *n*, then the compiler will default to a value of 0 (byte aligned). Note that:

| | | |
|---|---|---|
| **aligned** | specifies | byte alignment |
| **aligned(1)** | specifies | word alignment |
| **aligned(2)** | specifies | longword alignment |
| **aligned(3)** | specifies | quadword alignment |
| **aligned(4)** | specifies | octaword alignment |

Thus, **aligned**(*n*) implies 2***n*-byte alignment. For example, the declaration

```
TYPE
    word_aligned_integer32 = [ALIGNED(1)] integer32;
```

tells the compiler that all objects of type **word_aligned_integer32** are at least word aligned.

The format for the **natural** attribute is as follows:

**[natural]** *typename*;

For example, the declaration

```
TYPE
    natural_integer32 = [NATURAL] integer32;
```

tells the compiler that all objects of type **natural_integer32** are at least longword aligned.

The **natural** attribute has one main use:

- Overriding the default alignment rules to ensure that objects are stored on natural boundaries.

Similarly, you can use the **aligned** attribute to ensure natural alignment by specifying the correct value for *n*. In addition, the **aligned** attribute has these uses:

- Preventing the compiler from inserting padding in records.
- Ensuring the same layout in all alignment environments.
- Suppressing informational messages.
- Dereferencing pointers to unaligned objects.
- Passing arguments by reference.

Each of these uses is described in the following sections.

#### 3.15.6.2 Aligning Objects on Natural Boundaries

In general, natural alignment produces faster executable code, although the efficiency savings vary a great deal from one processor to another. (See Section 3.10.5 for a description of natural alignment.) Code for the 68000 family of processors runs slightly faster if objects are naturally aligned. Code for the Series 10000 workstations runs *significantly faster* if objects are naturally aligned. Moreover, on the Series 10000, if the compiler assumes that an object is naturally aligned when it is not, then the loss of efficiency is severe.

Although natural alignment often results in faster code, it can also produce holes in structures such as arrays and records, which can have an impact on memory efficiency. Before naturally aligning record fields, you need to weigh these two efficiency concerns.

The **aligned** and **natural** attributes override the default alignment rules (described in the "Internal Representation of Unpacked Records" section of this chapter) as well as the alignment rules specified by any compiler alignment directive that is currently in effect. (See Chapter 4 for more information about specifying alignment rules with compiler directives.)

Note that *if all of the record fields are scalar, it is always possible to guarantee natural alignment of the entire record by arranging the fields in decreasing order of size.* This method is preferable to using the **aligned** and **natural** attributes because it is portable. (See the "Internal Representation of Unpacked Records" section of this chapter for an example of using this rule.)

Although it is usually possible to align record fields naturally by arranging them in descending order of size, the arrangement of fields does not guarantee that a record will remain aligned if it is used within another aggregate (that is, in an array or record). Consider the following example:

```
TYPE
        S = record
                  a: integer32;
                  b: integer16
            end;
```

The layout is shown in Figure 3-19. Note that both **a** and **b** are naturally aligned, and that the entire record is aligned on a longword boundary.

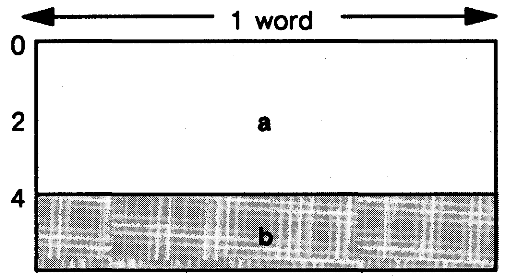

*Figure 3-19. Naturally Aligned Record S*

Note what happens, however, if we declare an array of three S records:

```
VAR
        bunch_of_records: ARRAY [1..3] OF S;
```

The memory layout for this array is shown in Figure 3-20.

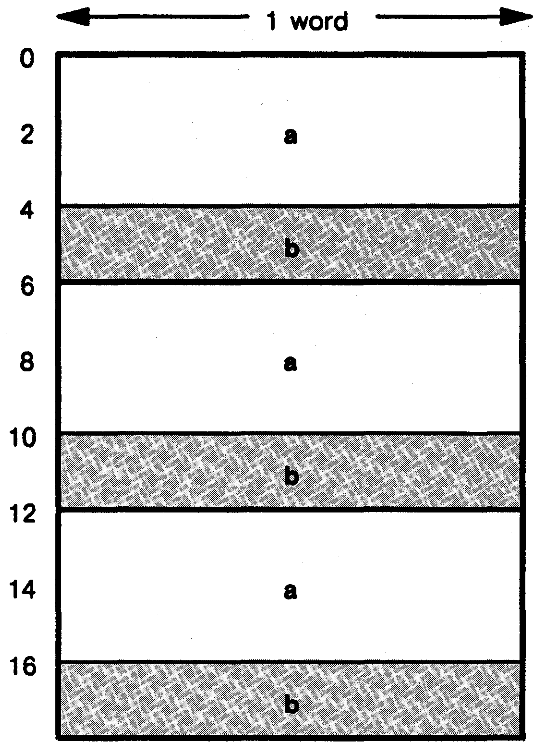

*Figure 3–20. Array of S Records*

Note that the second element of the array, which starts at address 6, is not naturally aligned. This alignment results from the fact that each element is six bytes long and array elements must be laid out contiguously in memory. There are several solutions to this problem. They are as follows:

- Explicitly enter a word of padding in the record so that the total size of the record is divisible by four. Specifically, change the declaration of S–type records to:

```
TYPE
          S = record
                    a: integer32;
                    b: integer16;
                    padding: integer16
              end;
```

- Use the **natural** attribute for the record. Specifically, change the declaration to:

```
TYPE
          S = [NATURAL] record
                    a: integer32;
                    b: integer16
              end;
```

- Use the **aligned** attribute for the record. Specifically, change the declaration to:

```
TYPE
            S = [ALIGNED(2)] record
                    a: integer32;
                    b: integer16
                end;
```

- Use the **natural** attribute for field **a**. This works because a record inherits the largest alignment specification of its fields. Since **a** is declared to be aligned on a longword boundary, the entire record will be also be longword aligned. Specifically, change the declaration to:

```
TYPE
            S = record
                    a: [NATURAL] integer32;
                    b: integer16
                end;
```

- Use the **aligned** attribute for field **a**. This method also works because of the rule that a record inherits the largest alignment specification of its fields. Specifically, change the declaration to:

```
TYPE
            S = record
                    a: [ALIGNED(2)] integer32;
                    b: integer16
                end;
```

Figure 3–21 shows the memory allocation for S records resulting from all of the above methods of declaring S.

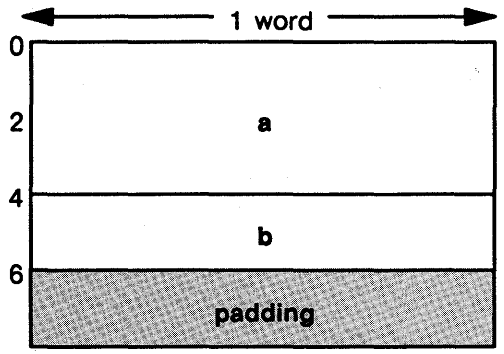

*Figure 3–21. Naturally Aligned Record S*

The principal difference between declaring a padding field and using an alignment attribute is that the padding word is accessible if you explicitly declare it. It is inaccessible if the compiler includes it to satisfy an alignment attribute. Furthermore, if you use padding instead of the attribute, your record declaration is portable.

**NOTE:** Assigning attribute specifications to a record or field can have unexpected repercussions. Consider the following declarations:

```
TYPE
        S =  record
                        a: [NATURAL] integer32;
                        b: integer16
             end;
        S1 = record
                        x: integer16;
                        y: S;
                        z: integer16
             end;
```

The layout for record **S** is shown in Figure 3–21 and the layout for **S1** is shown in Figure 3–22. Note that record **S** inherits the alignment of field **a** (longword alignment) and that this alignment requirement causes an additional word of padding to be inserted between **x** and **y**. In addition, record **S1** inherits longword alignment from record **S**, causing a word of padding to be added at the end of **S1**.

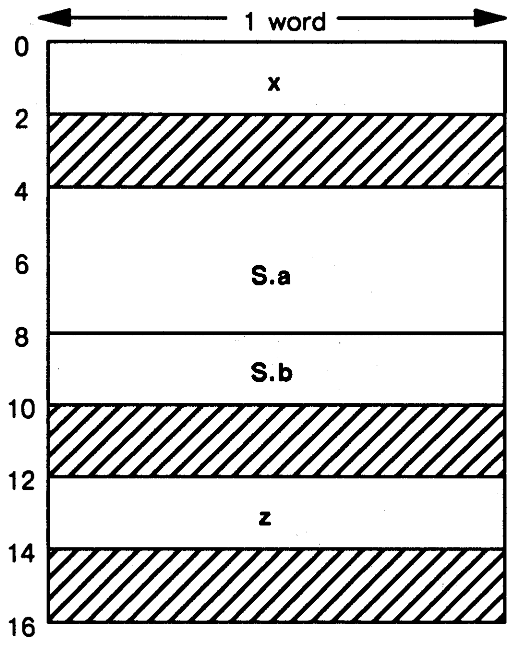

*Figure 3–22. Naturally Aligned Record S1*

### 3.15.6.3 Using the aligned Attribute to Prevent Padding

In the previous example, we used the **aligned** attribute to insert padding in a record so that the size of the record would be evenly divisible by the size of its largest field (four bytes). You can also use the **aligned** attribute to *prevent* the compiler from inserting padding. This is particularly useful if you need to declare a record that maps onto an existing layout (for example, a shared memory area).

Suppose, for example, that you want to declare a record that consists of a **char** followed by two integers:

```
TYPE
        S2 = record
                 a: char;
                 b, c: integer16
             end;
```

By default, the compiler produces the layout shown in Figure 3-23.

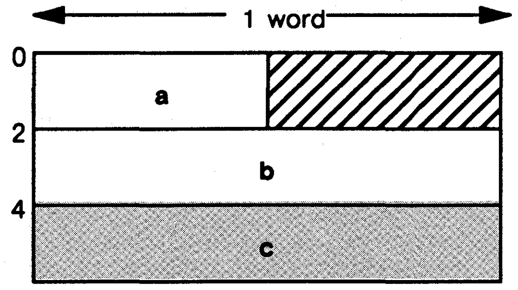

*Figure 3-23. Default Layout for S1*

The compiler inserts a byte of padding after **a** to ensure that **b** starts on a word boundary. If you want to create a record without padding at this position, you need to use the **aligned** specifier as shown in the following declaration:

```
TYPE
        S2 = record
                 a: char;
                 b, c: [ALIGNED(0)] integer16
             end;
```

The **aligned(0)** attribute tells the compiler that these fields may be aligned on byte boundaries. This declaration results in the layout shown in Figure 3-24.

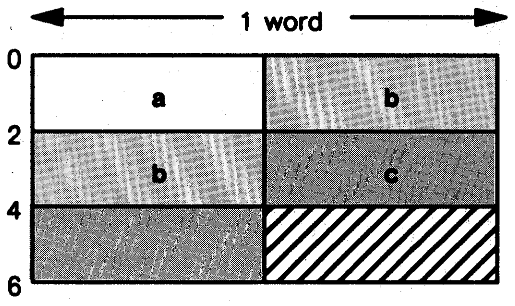

*Figure 3–24. Layout for S1 with Byte Alignment Specified*

Note that the compiler still inserts a byte of padding so that the size of the record is evenly divisible by two. There is no way to suppress this trailing byte. Records created by Domain Pascal are *always* 2–byte multiples.

Suppressing padding sometimes becomes more important if a **%natural_alignment** directive is in effect. The **%natural_alignment** directive tells the compiler to use natural alignment for any data that does not have an alignment attribute in its declaration. (See the "Compiler Directives" section of Chapter 4 for more details about the **%natural_alignment** directive). Consider the following declaration:

```
%NATURAL_ALIGNMENT
TYPE
         S1 = record
                    a: char;
                    b: integer32;
                    c: char
              end;
```

The **%natural_alignment** directive applies not only to each field, but also to the entire record. A record is considered to be naturally aligned if it starts on a boundary that ensures natural alignment for its largest field (and every other field). The layout for **S1** is shown in Figure 3–25.

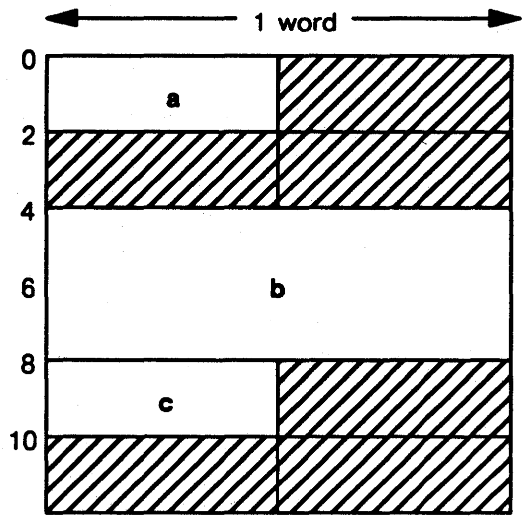

*Figure 3-25. Layout for S1 with Natural Alignment Specified*

By specifying word alignment for the record, we can remove the final word of padding, as shown in Figure 3-26.

```
%NATURAL_ALIGNMENT
TYPE
          S1 = [ALIGNED(1)] record
                     a: char;
                     b: integer32;
                     c: char
               end;
```

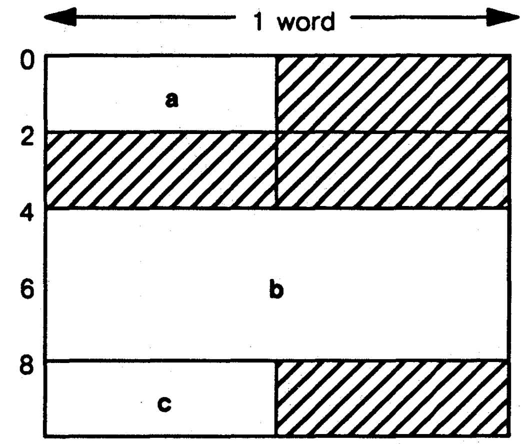

*Figure 3-26. Layout for S1 with Word Alignment Specified*

To remove at least some of the padding between **a** and **b**, we need to specify word alignment for **b**. Figure 3-27 shows the layout if we specify word alignment for **b** as well as for the whole record.

```
%NATURAL_ALIGNMENT
TYPE
         S1 = [ALIGNED(1)] record
                   a: char;
                   b: [ALIGNED(1)] integer32;
                   c: char
              end;
```

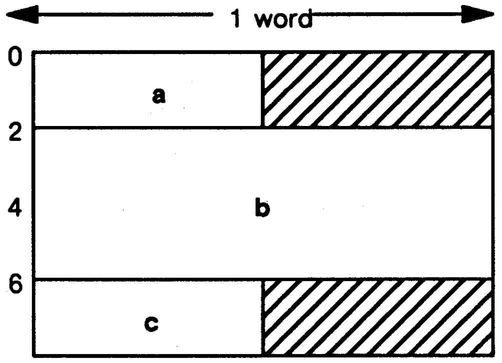

*Figure 3-27. Layout for S1 with Word Alignment for B Specified*

Note that specifying byte alignment for **b** produces the same layout as specifying word alignment for **b**—namely, the layout in Figure 3-27. The results are the same because of the rule that a byte aligned object may not cross two word boundaries. (See the "Internal Representation of Unpacked Records" section of this chapter for more details about the rules for record layout.)

### 3.15.6.4 Ensuring the Same Layout in All Alignment Environments

Alignment attributes can be particularly useful for ensuring that a record receives the same layout regardless of what alignment environment is in effect due to a compiler directive or a compiler option.

Suppose, for example, that you want to declare a record that consists of an **integer16** followed by an **integer32**, and you want to guarantee that there is no padding in the structure regardless of the alignment environment. Consider the following declaration:

```
. . .
S2 = record
         a: integer16;
         b: integer32
     end;
```

If the alignment environment is natural alignment, then the layout of **S2** type records will be the layout shown in Figure 3–28. If the alignment environment is word alignment, then the layout will be that shown in Figure 3–29.

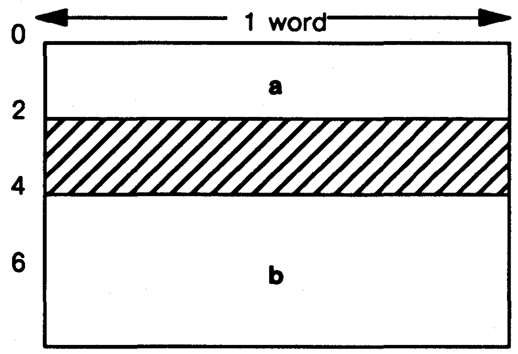

*Figure 3–28. Naturally Aligned Structure S2*

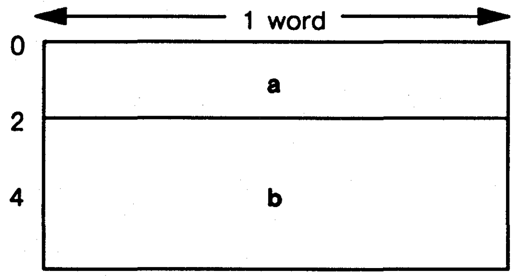

*Figure 3–29. Word Aligned Structure S2*

You can use the **aligned** attribute to ensure that **S2** never receives padding between **a** and **b**:

```
...
S2 = record
          a: integer16;
          b: [ALIGNED(1)] integer32
     end;
```

**NOTE:** You can also use the **aligned record** data type or the **unaligned record** data type to ensure that your records have the same layout in all alignment environments.

### 3.15.6.5 Suppressing Informational Messages about Alignment

In general, the compiler assumes that all objects are naturally aligned. If the compiler encounters an object that it knows is not naturally aligned, it issues an informational message. This happens, for example, when the compiler is forced to use word alignment rules for a record field that is larger than a word, or when a record cannot be aligned because it is embedded in another aggregate object. The following declarations, for example, will produce two informational messages:

```
TYPE
        S3 = record
                 a: integer16;
                 b: integer32;
                 c: double
             end;
```

The messages inform you that **b** and **c** are not naturally aligned. If you attempt to declare an array of **S3** records, you will receive another informational message telling you that the array elements are not naturally aligned.

The best way to suppress these messages is to rearrange the record so all the fields are naturally aligned. If rearrangement is impossible, however, you can suppress these messages by specifying the alignment and telling the compiler that the objects are not naturally aligned:

```
TYPE
        S3 = record
                 a: integer16;
                 b: [ALIGNED(1)] integer32;
                 c: [ALIGNED(1)] double
             end;
```

Note that these alignment specifications do not affect the record's layout—they merely reaffirm that the compiler should use word alignment rules rather than natural alignment rules.

> **NOTE:** If you compile with **–info 4**, you will receive informational messages even if you specify alignment. See Chapter 6 for more information about the **–info** option.

### 3.15.6.6 Informing the Compiler that an Object Is Not Naturally Aligned (Series 10000 Only)

In many instances, the compiler can determine whether an object is naturally aligned. When the compiler knows that an object is *not* naturally aligned, it produces code that accesses the object as if it consisted of separate parts, where each part *is* naturally aligned.

For example, suppose a program accesses a 4–byte integer that starts on a 2–byte boundary. Because the computer can't access the entire integer at once, the compiler treats the 4–byte integer as if it were composed of two contiguous 2–byte integers, each of which *is* naturally aligned. The compiler produces code that accesses each half of the 4-byte object and then recombines the two halves to obtain the 4-byte integer value.

Obviously, this decomposition and recomposition of objects is less efficient than accessing the object in a single instruction. Still, it is considerably better than taking a hardware trap, which is what occurs if the compiler assumes that an object is naturally aligned when, in fact, the object is not naturally aligned. The trap invokes a software routine to handle the unaligned data, which results in a significant loss of efficiency.

There are some situations where the compiler *cannot* determine whether an object is or is not naturally aligned:

- You declare a pointer to point to a naturally aligned object, and then assign it the address of an object that is not naturally aligned.
- You pass an argument to a routine by reference.

In both of these cases, the compiler assumes that the object is naturally aligned. If you know that this is not the case, you should inform the compiler with the **aligned** attribute or with the **align** function. (See the "Align" listing in Chapter 4 for details about the **align** function). This causes the compiler to use the decomposition/recomposition technique instead of suffering a hardware fault at run time.

**NOTE:** If you run your Domain Pascal program on a Series 10000 workstation, you should make sure that the compiler is informed about any objects that are not naturally aligned. If the compiler assumes that an object *is* naturally aligned when, in fact, the object *is not* naturally aligned, your program will suffer a severe loss of efficiency when you run it on Series 10000 workstations.

### 3.15.6.7 Dereferencing Pointers

When you declare a pointer to an object, the compiler assumes that the object pointed to is naturally aligned unless you tell it otherwise. Consider the following declarations:

```
TYPE
    rec =  record
             int16 : integer16;
             int32 : [ALIGNED (1)] integer32
           end;
VAR
        iptr : ^integer32;
        r: rec
BEGIN
        iptr := ADDR (rec.int32)
END;
```

In this example, the assignment of **addr(rec.int32)** to **iptr** will cause a compile-time warning. The program declared **iptr** as a pointer to an **integer32** type, which is assumed to be naturally aligned. However, the address of **rec.int32** has been declared to be word aligned.

You can correct this problem by declaring **iptr** as pointing to a word aligned **integer32** type variable, as follows:

```
VAR
          word_int32 : [ALIGNED (1)] integer32;
          iptr : ^word_int32
                         {iptr is a longword aligned pointer; it points
                                    to a word aligned 4-byte integer}
```

Note that the above declaration tells the compiler that **iptr** points to a word aligned **integer32**-type variable. It is different from the following declaration, which tells the compiler that **iptr** is a *word aligned pointer variable* and *also* that what it points to is a word aligned **integer32**-type variable:

```
VAR
          word_int32 : [ALIGNED (1)] integer32;
          int_ptr: ^word_int32;
          iptr : [ALIGNED(1)] int_ptr;
                         {iptr is a word aligned pointer variable;
                            it points to a word aligned 4-byte integer}
```

#### 3.15.6.8 Passing Arguments by Reference

The compiler is unable to determine the alignment of objects passed by reference as arguments to routines. By default, the compiler assumes that such objects are naturally aligned. Therefore, if you know that an argument passed by reference is *not* naturally aligned, you should specify its alignment.

### 3.15.7 Attribute Inheritance—Extension

Types and variables inherit the **device** attribute, and in some cases the **volatile** attribute, from more primitive data types. If you define a data type in terms of a more primitive data type declared with **device** or **volatile**, the new data type may inherit the attributes of that more primitive data type. For example, in the following declarations, **resource** inherits the **volatile** attribute from **semaphore**:

```
TYPE
    semaphore = [VOLATILE] integer;
    resource  = array[1..10] of semaphore;
```

If you define a record type as **volatile** or **device**, all the fields within the record inherit the attribute. And if you designate any one field within a record as having the **device** attribute, the entire record itself inherits the **device** attribute. However, the same is *not* true for a **volatile** field within a record; the entire record is not considered **volatile** just because one field is declared that way. Consider the following:

```
TYPE
    lock  = [VOLATILE] integer;
    queue = RECORD
            key : lock;
            users : integer;
                .
                .
                .
    end;
VAR
    wait : queue;
```

In this example, all references to **wait.key** are volatile, because the lock type is declared as **volatile**, but references to **wait.users** are not volatile. If you want all the fields to be volatile, insert the following after the record definition:

```
volque = [VOLATILE] queue;
```

**NOTE:** Pointer types do not inherit the **device** or **volatile** attributes of their base type. However, when pointer variables are dereferenced, the system applies any attributes of their base type.

## 3.15.8 Special Considerations—Extension

A common mistake is to associate an attribute with a pointer type. For example, we do not recommend that you use the following declaration:

```
VAR
    iodata : [DEVICE] ^integer16;
```

The memory location of **iodata** is normally on the stack or in the **.data** section. You don't want to make the local variable a device; you want to make the local variable a pointer to a device. Specify the following declarations instead:

```
TYPE
    DevInt = [DEVICE] integer;

VAR
    iodata : ^DevInt;
```

## 3.16 Attribute Declaration Part—Extension

Domain Pascal supports an **attribute** declaration part that allows you to define your own attributes. The syntax for the **attribute** declaration part is as follows:

**attribute**

*identifier1* = *attribute_name1* [, ... , *attribute_nameN*];

.
.
.

*identifierN* = *attribute_name1* [, ... , *attribute_nameN*];

An *identifier* is any valid Domain Pascal identifier. An *attribute_name* is any predeclared Domain Pascal attribute (such as **aligned(0)** or **long**) or the identifier of an attribute that you created earlier in the **attribute** declaration part.

For example, the following is a sample **attribute** declaration part:

```
ATTRIBUTE
        integer_attributes = long, natural;
        keyboard_attributes = device;
        array_attributes = volatile;
        peb_page_attributes = address(16#ff7000), keyboard_attributes;
```

And here is an example of **type** and **var** declaration parts that correspond to the above **attribute** declaration part:

```
TYPE
        int1 = [integer_attributes] integer32;
        int2 = [integer_attributes] integer16;

VAR
        i,j: int1;
        k: int2;
        peb_page: [peb_page_attributes] char;
        int_array: [array_attributes] ARRAY[1..10] of  int1;
```

You can use attributes that you define in an **attribute** declaration part in any context that you can use the predeclared attributes that are included in the definition. The compiler follows the same scope rules for attributes as it does for variables; the compiler evaluates attributes when they are declared.

# Chapter 4

# Code

This chapter describes the statements, procedures, functions, and operators constituting the action part of a Domain Pascal program or routine. The beginning of the chapter provides an overview of what's available. The remainder of the chapter is a Domain Pascal encyclopedia complete with many examples. If you are a Pascal beginner, you should read a good Pascal tutorial before trying to use this chapter.

The overview of Domain Pascal is divided into the following categories:

- Conditional branching
- Looping
- Mathematical operators
- Input and output
- Miscellaneous functions and procedures
- Systems programming functions and procedures

## 4.1 Overview: Conditional Branching

Domain Pascal supports the two standard Pascal conditional branching statements—**if** and **case**.

## 4.2 Overview: Looping

Domain Pascal supports **for**, **repeat**, and **while**—the three looping statements of standard Pascal. All three looping statements support the **next** and **exit** extensions. **Next** causes a jump to the next iteration of the loop, and **exit** transfers control to the first statement following the end of the loop.

## 4.3 Overview: Mathematical Operators

Domain Pascal supports all the standard arithmetic, logical, and set operators, as well as three additional operators for bit manipulation, two additional Boolean operators, and one additional operator for exponentiation. Table 4-1 lists these operators.

*Table 4-1. Domain Pascal Operators*

| Data Types | Operator | Meaning |
|---|---|---|
| Numeric | + | Addition |
| | - | Subtraction |
| | * | Multiplication |
| | / | Division (real values) |
| | **div** | Division (integer values) |
| | **mod** | Modulus (returns remainder of integer division) |
| | ** | Exponentiation |
| Integer | & | Bitwise and |
| | ! | Bitwise or |
| | ~ | Bitwise negation |
| Set | + | Set union |
| | * | Set intersection |
| | - | Set exclusion |
| | = | Set equality |
| | <> | Set inequality |
| | <= | First operand is subset of second |
| | >= | First operand is superset of second |
| | **in** | First operand is element of second |
| Boolean | **and** | Logical and |
| | **and then** | Logical and (short-circuit) |
| | **or** | Logical or |
| | **or else** | Logical or (short-circuit) |
| | **not** | Logical negation |
| Non-pointer types | > | Greater than |
| | >= | Greater than or equal to |
| | < | Less than |
| | <= | Less than or equal to |
| All types | = | Equal to |
| | <> | Not equal to |

**NOTE:** For the Boolean "short-circuit" operators, if the system can determine the value of the expression after evaluating the first operand, it does not check the second operand.

The exponentiation operator has the following syntax:

*mantissa* ** *exponent*

Table 4–2 shows the meaning of various expressions that use the exponentiation operator. An important restriction on the exponentiation operator is that you may not use a negative *mantissa* with a noninteger *exponent*.

*Table 4–2. Exponentiation Expressions*

| Expression | Meaning | |
|---|---|---|
| x ** y | $x^{y}$ | Raise x to the power y |
| x ** y ** z | $x^{y^{z}}$ | Raise y to the power z; then raise x to the result |
| x ** (y + z) | $x^{(y + z)}$ | Evaluate (y + z); then raise x to the result |

When evaluating expressions, Domain Pascal uses the order of precedence rules found in Table 4–3. The operators grouped together have the same precedence. Note that some operators work as both mathematical operators and as set operators. Nevertheless, the precedence rules are the same no matter how the operator is used.

*Table 4–3. Order of Precedence in Evaluating Expressions*

| Operator | Order of Precedence |
|---|---|
| **not** | highest precedence |
| ** | |
| & * / **div** **mod** **and** | |
| ! + – **or** | |
| = <> > < >= <= **in** | |
| **and then** | |
| **or else** | lowest precedence |

Domain Pascal permits the mixing of real and integer types in arithmetic expressions. For such mixed operations, Domain Pascal promotes the integers to reals before performing the operation.

### 4.3.1 Expansion of Operands

The compiler computes operands smaller than 32 bits with 32 bits of precision when necessary to achieve correct arithmetic. This means **integer16** operands sometimes are expanded to **integer32** before calculations. These data expansions produce more accurate results; however, the compiler tries to avoid the extra code produced by data expansion.

### 4.3.2 Predeclared Mathematical Functions

In addition to the mathematical operators, you can use any of the predeclared mathematical functions listed in Table 4-4. Note that although the **arctan**, **cos**, **exp**, **ln**, **sin**, and **sqrt** functions permit integer arguments, the compiler converts an integer argument to a real number before calculating the function. Therefore, when possible, it is better to supply real, rather than integer, arguments to these functions.

*Table 4-4. Mathematical Functions*

| Function | Argument(s) | Result | Meaning |
|---|---|---|---|
| **abs(x)** | integer or real | same type as *x* | Absolute value of *x*. |
| **arctan(x)** | integer or real | real | Arctangent of *x*. |
| arshft(x,n) | both are integer | integer | Shifts the bits in *x* to the right *n* places. Preserves the sign of *x*. |
| **cos(x)** | integer or real | real | Cosine of *x*. |
| **exp(x)** | integer or real | real | Raises exponential function e to the *x* power. |
| **ln(x)** | integer or real | real | Natural log of *x*; *x* >0 |
| lshft(x,n) | both are integer | integer | Shifts the bits in *x* to the left *n* places. |
| **odd(x)** | integer | boolean | True if *x* is an odd value. |
| **round(x)** | real | integer | Rounds *x* up or down to nearest integer. |
| rshft(x,n) | both are integer | integer | Shifts the bits in *x* to the right *n* places. |
| **sin(x)** | integer or real | real | Sine of *x*. |
| **sqr(x)** | integer or real | same type as *x* | Square of *x*. |
| **sqrt(x)** | integer or real | real | Square root of *x*. |
| **trunc(x)** | real | integer | Truncates the fractional part of *x* (rounds *x* towards zero). |
| xor(x,n) | both are integer | integer | Bit exclusive or. |

## 4.3.3 Mixing Signed and Unsigned Operands in Expressions

Although Domain Pascal does not have an unsigned type, it does support unsigned ranges. (See Section 3.4 for further information on unsigned types.)

Mixing signed and unsigned integer operands is tricky for the following operations:

- >
- >=
- <
- <=

- **min**
- **max**
- **div**
- **mod**

The compiler interprets these operations as signed except under the following circumstances:

- Both operands are unsigned run–time values.
- One operand is an unsigned run–time value and the other is a constant in the positive subrange 0..2147483647.

For these two cases, the compiler uses unsigned operations.

You can use a type transfer to force the desired type of operation if it does not result from the above rules. (See the section on "Type Transfer Functions" later in this chapter for further information.) The following code fragment illustrates the use of type transfer for operations involving mixed signedness:

```
TYPE
    u_type = 0..2147483647;

VAR
     s32 : integer32;
     u32 : u_type;
     a   : integer32;
   .
   .
   .
     a := s32 DIV integer32(u32);    { Compiler generates a signed
                                       divide. }

     { OR }

     a := u_type(s32) DIV u32;      { Compiler generates an unsigned
                                      divide. }
```

## 4.4 Overview: I/O

Domain Pascal supports the I/O procedures described in Table 4-5. For details on these routines, consult the encyclopedia later in this chapter and Chapter 8.

*Table 4-5. Predeclared I/O Procedures*

| Name | Action |
|---|---|
| close | Closes a file. |
| **eof** | Tests whether the stream marker is pointing to the end of the file. |
| **eoln** | Tests whether the stream marker is pointing to the end of a line. |
| find | Sets the stream marker to the specified record. |
| **get** | Reads from a file. |
| open | Opens a file for future access. |
| **page** | Inserts a formfeed (page advance) into a file. |
| **put** | Writes to a file. |
| **read** | Reads information from the specified file (or from the keyboard) into the specified variables. After reading the information, **read** positions the stream marker so that it points to the character or component immediately after the last character or component it read. |
| **readln** | Similar to **read** except that after reading the information, **readln** positions the stream marker so that it points to the character or component immediately after the next end-of-line character. |
| replace | Substitutes a new record component for an existing record. |
| **reset** | Specifies that an open file be open for reading only. |
| **rewrite** | Specifies that an open file be open for writing only, or tells the system to open a temporary file. |
| **write** | Writes the specified information to the specified file (or to the screen). |
| **writeln** | Same as **write** except that **writeln** always appends a linefeed to its output. |

## 4.5 Overview: Miscellaneous Routines and Statements

Table 4-6 lists several Domain Pascal elements that do not fit neatly into categories.

Table 4-6. *Miscellaneous Elements*

| Element | Action |
|---|---|
| addr | Returns the address of the specified variable. |
| append | Concatenates two or more strings. |
| **chr** | Finds the character whose ISO Latin-1 value equalshe specified number. |
| ctop | Converts a C-style string into a Domain Pascal variable-length string. |
| discard | Explicitly discards a computed value. |
| **dispose** | Deallocates the storage space that a dynamic record was using. |
| exit | Transfers control to the first statement following a **for**, **while**, or **repeat** loop. |
| firstof | Returns the first possible value of a type or a variable. |
| **goto** | Unconditionally jumps to the first command following the specified label. |
| in_range | Tells you if the specified value is within an enumerated variable's defined range. |
| lastof | Returns the last possible value of a type or a variable. |
| max | Returns the larger of two expressions. |
| min | Returns the smaller of two expressions. |
| **new** | Allocates space for storing a dynamic record. |
| next | Transfers control to the test for the next iteration of a **for**, **while**, or **repeat** loop. |
| **nil** | A special pointer value that points to nothing. |
| **ord** | Finds the ordinal value of a specified integer, Boolean,enumerated, or char type. |
| **pack** | Copies unpacked array elements to a packed array. |
| **pred** | Finds the predecessor of a specified value. |
| ptoc | Converts a Domain Pascal variable-length stringinto a C-style string. |
| return | Causes program control to jump back to the callingprocedure or function. |
| sizeof | Returns the size (in bytes) of the specified data type. |
| substr | Extracts a substring from a string. |
| **succ** | Finds the successor of a specified value expression in the code portion of your program. |
| type transfer functions | Permits you to change the data type of a variable orexpression in the code portion of your program. |
| **unpack** | Copies packed array elements to an unpacked array. |
| **with** | Lets you abbreviate the name of a record. **With** is standard, but Domain Pascal includes an extension that supports a name tag. |

## 4.6 Overview: Systems Programming Routines

Several Domain Pascal routines are available for systems programmers' use. Table 4-7 lists these routines. Because only a few programmers will need to use these routines, they are not described in the encyclopedia section that follows. Instead, they appear in Appendix E.

*Table 4-7. Systems Programming Routines*

| Routine | Action |
|---|---|
| disable | Turns off the interrupt enable in the hardware status register. |
| enable | Turns on the interrupt enable in the hardware status register. |
| set_sr | Saves the current value of the hardware status register and then inserts a new one. |

## 4.7 Encyclopedia of Domain Pascal Code

The remainder of this chapter contains an explanation of the concepts and keywords that you can use in the action part of a Domain Pascal program or routine. These items are listed alphabetically.

The concepts that we include are as follows:

- Array operations
- Bit operators
- Compiler directives
- Expressions
- Pointer operations
- Record operations
- Set operations
- Statements
- Type transfer functions
- Variable-length string operations

The keywords that we include are:

| | | | | | |
|---|---|---|---|---|---|
| abs | cos | for | new | ptoc | sqr |
| addr | ctop | get | next | put | sqrt |
| align | discard | goto | nil | read, readln | substr |
| and | dispose | if | not | repeat/until | succ |
| and then | div | in | odd | replace | trunc |
| append | end | in_range | open | reset | unpack |
| arctan | eof | lastof | or | return | while |
| arshft | eoln | ln | ord | rewrite | with |
| begin | exit | lshft | or else | round | write, writeln |
| case | exp | max | pack | rshft | xor |
| chr | find | min | page | sin | |
| close | firstof | mod | pred | sizeof | |

## Abs

**Abs** Returns the absolute value of an argument.

### FORMAT

**abs**(*number*) {**abs** is a function.}

### ARGUMENTS

| | |
|---|---|
| *number* | Any real or integer expression. |

### FUNCTION RETURNS

The **abs** function returns a real value if *number* is real and an integer value if *number* is an integer.

### DESCRIPTION

The **abs** function returns the absolute value of the argument. The absolute value is the number if it is nonnegative, and the negative of the number if it is negative. Note that *number* cannot be -2147483648 (which is the lowest negative integer).

### EXAMPLE

```
program abs_example;
{ This program displays the absolute values for two numbers }
VAR
   x  : INTEGER;
   y  : REAL;
BEGIN
   x := -3;        x := ABS(x);
   y := -456.78;   y := ABS(y);
   WRITELN(x,y);
END.
```

### USING THIS EXAMPLE

If you execute the sample program named **abs_example**, you get the following output:

```
3    4.567800E+02
```

**Addr** Returns the address of the specified variable. (Extension)

## FORMAT

**addr**(*x*)    {**addr** is a function.}

## ARGUMENTS

| | |
|---|---|
| *x* | Can be a variable declared as any data type except as a procedure or function data type having the **internal** attribute. *x* can also be a string constant but it cannot be a constant of any type other than string. |

## FUNCTION RETURNS

The **addr** function returns an **univ_ptr** value. (Chapter 3 describes the **univ_ptr** data type.)

## DESCRIPTION

Use **addr** to return the address at which variable *x* is stored. If *x* is a variable-length string, **addr** returns the address of the entire record, not the address of the string component. **Addr** is particularly useful with variables defined as pointers to functions or procedures.

Using **addr** can prevent some compiler optimizations. If you apply **addr** to a variable that is local to a routine, and the variable is not a **set**, **record**, or **array**, you do not get optimizations and register allocation for that variable or any expressions using the variable. This means the routine's code might be larger and slower than it otherwise would be.

Applying **addr** to a variable is equivalent to declaring the variable **volatile**. See Chapter 3 for more information on **volatile.**

Refer to the "Pointer Operations" listing later in this chapter for an example of **addr**.

**NOTE:** The compiler issues a warning if you assign the result of **addr** to a pointer type variable that expects an alignment greater than the alignment of the **addr** result. For example,

```
TYPE
       natural_integer = [natural] integer32;
       rec = record
             int16 : integer16;
             int32 : [ALIGNED (1)] integer32;
             end;
VAR
       iptr : ^natural_integer;
       r: rec
BEGIN
       iptr := ADDR (r.int32)
END;
```

In this example, the assignment of **addr(r.int32)** to **iptr** causes a compile-time warning. The code in the example declares **iptr** as a pointer to a naturally aligned **integer32** type, but assigns to it the address of a word-aligned object, **r.int32**. (See the "Internal Representation of Records" and "Alignment" sections of Chapter 3 for further details about alignment.)

## EXAMPLE

```
Program addr_example;
{This program displays the contents stored at an address returned}
{by the addr function}
TYPE
    ptr_to_real = ^real;
VAR
    y, y2       : real;
    ptr_to_y    : ptr_to_real;
BEGIN
    write('Enter a real number -- ');
    readln(y);
    ptr_to_y := ADDR(y);
             { Set ptr_to_y to the address at which y is stored. }
    y2 := ptr_to_y^;
             { Set y2 to the contents stored at y's address;     }
             { i.e., set y2 equal to y. }
    writeln(y2);
END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **addr_example**:

```
Enter a real number -- 5.3
     5.300000E+00
```

## Align

**Align** Causes the compiler to copy an expression that is being passed as a parameter. (Extension)

### FORMAT

**align** (*expression*);　　　　{**align** is a function.}

### ARGUMENTS

| | |
|---|---|
| *expression* | Any valid Domain Pascal expression that is being passed as an input parameter to a routine. |

### FUNCTION RETURNS

The **align** function returns a correctly aligned copy of *expression*.

### DESCRIPTION

The **align** function tells the compiler to make a copy of an *expression* passed as an **in** parameter to an external routine. The alignment of the copy matches the alignment specified for the formal parameter. The compiler uses the copy as the actual parameter when the external routine is called.

The **align** function is useful for making sure that expressions are correctly aligned when they are passed as arguments to functions and procedures. An expression is correctly aligned if its alignment matches the alignment specified for it in the formal parameter definition.

The main use of the **align** function is to pass a record field that is not naturally aligned to a routine that expects the parameter to be naturally aligned. This use is illustrated in the example.

Although correct alignment for parameters passed by reference generally produces at least somewhat faster executable code on all Apollo workstations, the improvement is very significant on Series 10000 workstations. If you run your program on a Series 10000 workstation and the compiler assumes that an object is naturally aligned when in fact, it is not naturally aligned, your program will suffer a severe loss of efficiency.

**EXAMPLE**

```
PROGRAM align_example;
{This program shows how to use the ALIGN function, a Domain Pascal    }
{extension that causes parameters passed as IN parameters to external }
{functions to be aligned according to the expectations of the called  }
{routine.                                                              }
{NOTE:  You must also compile the add_em Pascal program and bind it   }
{       with align_example to get an executable file.                 }
TYPE
    rec = record
           short_num : integer16;
           long_num  : integer32; {this field is not naturally aligned}
          end;
VAR
    sum : rec;
    i,j : integer32;
FUNCTION add_em (IN x    : integer32;
                    y, z : integer32 ) :integer32; extern;
BEGIN
    with sum do
        begin
          short_num := 7;
          long_num := 3;
        end;
    i := 1;
    j := 1942;

    write('The sum of the numbers is ');
    writeln(add_em(ALIGN(sum.long_num), i,j));
          {use the ALIGN function to make sure that sum.long_num is  }
          {naturally aligned when it is passed to the add_em function}
          {If you omit the ALIGN function, you get a warning from    }
          {the compiler.                                             }
END.

MODULE add_em;
{This function is called by the align_example program}
FUNCTION add_em (IN x   : integer32;
                    y,z : integer32) : integer32;
     BEGIN
        add_em:= x+y+z;
     END;
```

These programs are available online and are named **align_example** and **add_em**. A sample run of the program is shown below:

```
The sum of the numbers is       1946
```

## And, And Then Calculate the logical **and** of two Boolean arguments.

### FORMAT

| | |
|---|---|
| *x* **and** *y* | {**and** is an operator.} |
| *x* **and then** *y* | {**and then** is an operator.} |

### ARGUMENTS

| | |
|---|---|
| *x, y* | Any Boolean expressions. |

### OPERATOR RETURNS

The result of an **and** or an **and then** operation is a Boolean value.

### DESCRIPTION

Sometimes **and** is called Boolean multiplication. Use it to find the logical and of expressions *x* and *y*. Here is the truth table for **and**:

| x | y | Result |
|---|---|---|
| true | true | true |
| true | false | false |
| false | true | false |
| false | false | false |

(See also the listings for the logical operators **or** and **not** later in this encyclopedia).

NOTE: Some programmers confuse **and** with &. & is a bit operator; it causes Domain Pascal to perform a **logical and** on all the bits in its two arguments. For example, compare the following results:

the result of (true **and** false) is false
the result of (75 & 15) is 11

(Refer to "Bit Operators" later in this encyclopedia.)

The Boolean operator **and then** is a Domain extension to standard Pascal. You can use **and then** in any statement where you use **and** in standard Pascal. The choice between **and** and **and then**, however, affects the run-time evaluation of a statement.

When **and then** appears between two Boolean operands in an expression, the system begins evaluating the operands in the order in which they appear. If the first operand is false, the system does not evaluate the second. If one operand is false, then the entire expression is false.

Hence, **and then** guarantees "short-circuit" evaluation. That is, at run time, the system evaluates an operand only if necessary.

For example, in the statement

```
IF boolean_1 AND THEN boolean_2
    THEN ...
```

the system first evaluates **boolean_1**. If **boolean_1** is false, the system does not evaluate **boolean_2**. In this statement, the operands **boolean_1** and **boolean_2** can be any valid Pascal Boolean expressions.

The operator **and then** can be more efficient than **and**. For example, in the statement

```
IF boolean_1 AND boolean_2
    THEN ...
```

the system may evaluate both **boolean_1** and **boolean_2** to test if the statement is true. Also, there is no guarantee that the system will evaluate the two operands in the order in which they appear.

The **and then** operator helps you avoid nested constructions. For example, compare the standard Pascal code on the left with the equivalent Domain Pascal code on the right:

```
Standard Pascal              Domain Pascal

WHILE c1 DO                  WHILE c1 AND THEN c2 DO
    WHILE c2 DO                  s1;
        s1;
```

In this example, the standard Pascal code contains one **while** loop nested within another. The Domain Pascal code, however, contains only one loop.

## And, And Then

The following example illustrates how to avoid referencing a NIL pointer through the use of the **and then** operator:

```
WHILE p <> NIL AND THEN NOT p^.flag DO
    p := p^.next;
```

### EXAMPLE

The following example uses an **and** operation to calculate the gravitational force between two objects.

```
Program and_example;

CONST
    g = 6.6732e-11;
VAR
    mass1, mass2, radius, force : single;
BEGIN
write('This program finds the gravitational force between two ');
writeln('objects.');
write('Enter the mass of the first object (in Kg) -- ');
readln(mass1);
write('Enter the mass of the second object (in Kg) -- ');
readln(mass2);
write('Enter the dist. between their centers (in M) -- ');
readln(radius);
if (mass1 > 0.0) AND (mass2 > 0.0)
     then force := (g * mass1 * mass2) / sqr(radius)
     else begin
            writeln('The data you have entered seems inappropriate');
            return;
          end;
writeln('The force between these two objects is ', force:9:7, ' N');
END.
```

### USING THIS EXAMPLE

This program is available online and is named **and_example**.

Append Concatenates two or more strings. (Extension)

## FORMAT

**append**(*dst_string, s1, s2, s3, s4, s5, s6, s7, s8, s9*); {**append** is a procedure.}

## ARGUMENTS

| | |
|---|---|
| *dst_string* | A variable-length character string. |
| *s1* | A variable-length string, character array, or character-string constant that will be appended to the destination string. |
| *s2..s9* | Optional arguments. Up to nine variable-length strings, character arrays, and character-string constants may be appended to the destination string. |

## DESCRIPTION

**append** builds a destination string by concatenating the destination string and up to nine additional strings.

An error trap is generated if concatenating the source string(s) with the destination string results in a string that is larger than the maximum size of the destination string.

## EXAMPLE

```
PROGRAM append_example;

 VAR
     str1    :   varying[100] of char;
     str2    :   varying[20] of char;
     str3    :   array[1..20] of char;

 BEGIN
     str1 := 'one...';
     str2 := 'two...';
     str3 := 'three...';
     append(str1, str2, str3, 'four');
     writeln(str1);
 END.
```

## USING THIS EXAMPLE

Executing this program, named **append_example,** results in the following output:

```
one...two...three...               four
```

Note that the fixed-length array, **str3,** is padded with spaces, whereas the variable-length strings are not padded. Also note that the first parameter *must* be a variable-length string.

**Arctan** Returns the arctangent of a specified number.

## FORMAT

**arctan**(*number*) {**arctan** is a function.}

## ARGUMENTS

| | |
|---|---|
| *number* | Any real or integer expression. |

## FUNCTION RETURNS

The **arctan** function returns a real value for the angle in radians.

## DESCRIPTION

The **arctan** function returns the arctangent (in radians) of *number*. The arctangent of a number has the following relationship to the tangent:

```
y = arctan(x) means that x = tan(y)
```

Note that Pascal does not support a predeclared tangent function. However, you can find **tangent**(x) by dividing **sin**(x) by **cos**(x).

## EXAMPLE

```
PROGRAM arctan_example;
  { This program demonstrates the ARCTAN function. }

 CONST
    degrees_per_radian = 180.0 / 3.14159;

 VAR
    q, answer_in_radians : REAL;
    answer_in_degrees    : INTEGER16;

 BEGIN
    q := 2.0;
  {First, find the arctangent of 2.0 in radians. }
  answer_in_radians := ARCTAN(q);
  writeln('The arctan of ', q:5:3, ' is', answer_in_radians:6:3,
          ' radians');

 {Now, convert the answer to degrees. }
    answer_in_degrees := round(answer_in_radians * degrees_per_radian);
    writeln('The arctan of ', q:5:3, ' is ', answer_in_degrees:1,
            ' degrees');
 END.
```

## USING THIS EXAMPLE

If you execute the sample program named **arctan_example**, you get the following output:

```
The arctan of 2.000000E+00 in radians is 1.107149E+00
 The arctan of 2.000000E+00 in degrees is 63
```

## Array Operations

Chapter 3 explains how to declare and initialize an array. In this listing, we explain how to use arrays in the code portion of your program. See "Variable-Length String Operations" in this chapter for information about accessing varying arrays of chars.

### 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 brackets. The value of the index expression must be within the declared subrange of the index type.
- A value of the component type.

For example, the following program fragment assigns values to four arrays:

```
TYPE
   {elements is an enumerated type.}
    elements = (H, He, Li, Be, B, C, N, O, Fl, Ne);
    student  = record
        name : packed array [1..50] of char;
        id   : integer16;
        class : (freshman, sophomore, junior, senior);
    end;

 VAR
   {Here are four array declarations.}
    test_data      : array[1..100] of INTEGER16;
    atomic_weights : array[H..Be] of REAL;
    lie_test       : array[1..4, 1..2] of BOOLEAN; {2-dimensional array}
    enrollment     : array[1..500] of student;

 BEGIN
    test_data[37]       := 9018;
    atomic_weights[He]  := 4.0;
    lie_test[3, 2]      := true;
    enrollment[30].name   := 'Betsy Ross';
    enrollment[30].id     := 8245;
    enrollment[30].class  := senior;
    .
    .
    .
```

## Array Operations

There are a few exceptions to the rule that you must supply an index expression.

The first exception is that you can assign a string to an array of **char** variable without specifying an index expression; for example, consider the following assignments to **greeting** and **farewell**:

```
CONST
    hi = 'aloha';

 VAR
    greeting, farewell : array[1..12] of CHAR;

 BEGIN
    greeting := hi;
    farewell := 'a bientot';
```

The only restriction on this kind of assignment is that the number of bytes in the string must be less than or equal to the declared number of declared components in the array. For example, you cannot assign the string 'auf wiedersehen' to **farewell** because the string contains 15 bytes and the array is declared as only 12 bytes. If you do try that assignment, the compiler will give you the following error message:

```
Assignment statement expression is not compatible with the
 assignment variable.
```

There is a second exception to the rule that you must specify an index expression when assigning a value to an array. The exception is that you can assign the value of one array to another array if both arrays are **char** arrays that contain the same number of bytes. For example, in the following program fragment, **a**, **b**, and **e** are the same size, while **c** and **d** are uniquely declared:

```
CONST
     quote = 'Ottawa!';
VAR
     a : array[1..20] of CHAR;
     b : array[1..20] of CHAR;
     c : array[1..21] of CHAR;
     d : array[1..19] of CHAR;
     e : array[21..40] of CHAR;

BEGIN
     a := quote;  {Assign the string 'Ottawa!' to array a.     }
     b := a;  {This is a valid assignment.                     }
     e := a:  {This is a valid assignment.                     }
     c := a;  {WRONG!}
              {This is not a valid assignment because a and c }
              { have different declared lengths.               }
     d := a;   {WRONG!}
               {This is not a valid assignment because a and d }
               { have different declared lengths.              }
```

The assignment **b := a** causes Domain Pascal to assign all components of array a to the corresponding indices in array **b**; that is, **b := a** is equivalent to the following 20 assignments:

```
b[1]  := a[1];
b[2]  := a[2];
  .        .
  .        .
  .        .
b[20] := a[20];
```

**NOTE:** In standard Pascal, before assigning a string to an array, you must explicitly pad the string to the length of the array. Domain Pascal automatically pads with spaces any string of fewer than 4096 characters.

## USING ARRAYS

You can specify an array component wherever you can specify a component variable of the same data type. In other words, if the compiler expects a real number, you can specify any real expression including a component of an array of real numbers.

## EXAMPLE

```
PROGRAM array_example;

 {This simple example reads in five input values, assigns the values to}
 {elements of an array, and then finds their mean.                     }

 CONST
      number_of_elements = 5;

 VAR
      a : array[1..number_of_elements] of single;
      running_total : single := 0.0;
      n : integer16;

 BEGIN
  for n := 1 to number_of_elements do
        begin
           write('Enter a value -- ');
           readln(a[n]);
        end;

  for n := 1 to number_of_elements do
        running_total := running_total + a[n];

  writeln(chr(10),'The mean is ', running_total/number_of_elements:3:1);

 END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **array_example**:

```
Enter a value -- 4.3
Enter a value -- 10.3
Enter a value -- 9.5
Enter a value -- 6.2
Enter a value -- 1.5

The mean is 6.4
```

**Arshft** Shifts the bits in an integer to the right by a specified number of bits. Preserves the sign of the integer. (Extension)

## FORMAT

**arshft**(*num, sh*) {**arshft** is a function.}

## ARGUMENTS

| | |
|---|---|
| *num, sh* | Must be integer expressions. *sh* should be nonnegative. |

## FUNCTION RETURNS

The function returns an integer value.

## DESCRIPTION

**Arshft** does an arithmetic right shift of an integer. The **arshft** function tells the compiler to preserve the sign bit of *num* and shift the other bits *sh* positions to the right. The expression *num* can be any integer expression smaller than 32 bits.

Say, for example, *num* is a 16–bit integer and the result of the function is to be stored in a 16–bit integer variable. In this case, **arshft** expands *num* to a 32–bit integer, performs the shift, and then converts it back to a 16–bit integer.

First examine how **arshft** shifts a positive integer. Consider the effect of **arshft** on the 16–bit positive integer +100 in the following table:

```
unshifted              0000000001100100 = +100
ARSHFT(+100,1)         0000000000110010 = +50
ARSHFT(+100,2)         0000000000011001 = +25
ARSHFT(+100,3)         0000000000001100 = +12
```

Notice three things in the preceding table. First, the sign bit (the leftmost bit) never changes. Second, notice that the bits move to the right. Third, notice that the bits do not wrap around from right to left; the absolute value always gets smaller.

Now, examine how **arshft** shifts a negative integer. Consider the effect of **arshft** on the 16-bit negative integer -100 in the following table:

```
unshifted                    1111111110011100 = -100
ARSHFT(-100,1)               1111111111001110 = -50
ARSHFT(-100,2)               1111111111100111 = -25
ARSHFT(-100,3)               1111111111110011 = -13
```

In contrast to the preceding table, notice that **arshft** fills the leftmost bits with ones rather than zeros as the rightmost bits are shifted off the right end of the number.

Results are unpredictable if *sh* is negative.

## EXAMPLE

```
PROGRAM arshft_example;

 { This program compares ARSHFT with RSHFT. }

 VAR
     original_number, spaces_to_shift, r, ar : integer32 := 0;

 BEGIN
   write('Enter a positive or negative integer -- ');
   readln(original_number );

   for spaces_to_shift := 1 to 5 do
      BEGIN
         writeln;
         writeln('When shifted ', spaces_to_shift:1, ' spaces.');

         r := RSHFT(original_number, spaces_to_shift);
         writeln('     The rshft result is ', r:1);

         ar := ARSHFT(original_number, spaces_to_shift);
         writeln('     The arshft result is ', ar:1);
      END;
 END.
```

## USING THIS EXAMPLE

This program is available online and is named **arshft_example**.

**Begin** Marks the start of a compound statement.

## FORMAT

**begin** is a reserved word.

## DESCRIPTION

**Begin** and **end** establish the limits of a sequence of Pascal statements. A program must contain at least as many **ends** as **begins**. (Note that a program can contain more **ends** then **begins**.) You must use a **begin/end** pair to indicate a compound statement. (Refer to the "Statements" listing later in this encyclopedia.)

## EXAMPLE

```
PROGRAM begin_end_example;

{This program does very little work, but does have lots of BEGINs }
{and ENDs.                                                         }

TYPE
    student = record
       age : 6..12;
       id  : integer16;
    end;  {student record definition}
 VAR
    x    : integer32;

PROCEDURE do_nothing;
BEGIN    {do_nothing}
writeln('You have triggered a procedure that does absolutely nothing.');
writeln('Though it does do nothing with elan.').
END;     {do_nothing}

FUNCTION do_next_to_nothing(var y : integer32) : integer32;
BEGIN  {do_next_to_nothing}
    do_next_to_nothing := abs(y);
END;   {do_next_to_nothing}
```

```
BEGIN  {main procedure}
    write('Enter an integer -- ');    readln(x);
    if x < 0
      then BEGIN
             writeln('You have entered a negative number!!!');
             writeln('Its absolute value is ', do_next_to_nothing(x):1);
            END
      else if x = 0
             then BEGIN
                    writeln('You have entered zero');
                    do_nothing;
                  END
             else
               writeln('You have entered a positive number!!!');
END.   {main procedure}
```

## USING THIS EXAMPLE

This program is available online and is named **begin_end_example.**

**Bit Operators** Calculate **and**, **or**, and **not** on a bit-by-bit basis. (Extension)

## FORMAT

| | |
|---|---|
| *op1* **&** *op2* | {& (an ampersand) is bit **and**.} |
| *op1* **!** *op2* | {! (an exclamation point) is bit **or**.} |
| *˜op1* | {˜ (a tilde) is bit **not**.} |

## ARGUMENTS

| | |
|---|---|
| *op1, op2* | Must be integer expressions. |

## OPERATOR RETURNS

All three operators return integer results.

## DESCRIPTION

Domain Pascal supports three bit operators, all of which are extensions to standard Pascal. The operators perform operations on a bit-by-bit level using the following truth tables:

*Table 4-8. Truth Table for & (Bitwise And Operator)*

| **& (and)** | | |
|---|---|---|
| **bit x of op1** | **bit x of op2** | **bit x of result** |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

*Table 4-9. Truth Table for ! ( Bitwise Or Operator)*

| **! (or)** | | |
|---|---|---|
| **bit x of op1** | **bit x of op2** | **bit x of result** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

*Table 4-10. Truth Table for ~ (Bitwise Not Operator)*

| **~ (not)** | |
|---|---|
| **bit x of op1** | **bit x of result** |
| 0 | 1 |
| 1 | 0 |

Don't confuse these bit operators with the logical operators. Bit operators take integer operands; logical operators take Boolean operands.

In addition to the three bit operators, Domain Pascal supports the following bit functions: **lshft**, **rshft**, **arshft**, and **xor**. All of these functions have their own listings in the encyclopedia.

NOTE: If one of the operators is declared as **integer32**, and the other operator is declared as **integer16**, Domain Pascal extends the **integer16** to an **integer32** before calculating the answer.

When performing these bitwise operations, Domain Pascal treats the sign bit just as it treats any other bit.

## EXAMPLE

```
PROGRAM bit_operators_example;

 { This program demonstrates bitwise AND, OR, and NOT. }

 CONST
 { The 2# prefix specifies a base 2 number. }
     x = 2#0000000000001010; {10}
     y = 2#0000000000010111; {23}

 VAR
     result1, result2, result3 : integer16;

 BEGIN
     result1 := x & y;  writeln(x:1, ' AND ', y:1, ' = ', result1:1);
     result2 := x ! y;  writeln(x:1, ' OR ', y:1, ' = ', result2:1);
     result3 := ~x;     writeln('NOT ', x:1, ' = ', result3:1);
 END.
```

## USING THIS EXAMPLE

If you execute the sample program named **bit_operators_example**, you get the following output:

```
10 AND 23 = 2
 10 OR 23 = 31
 NOT 10 = -11
```

**Case** A conditional branching statement that selects among several statements based on the value of an ordinal expression.

## FORMAT

There are two different forms of the **case** statement. Here, we describe the use of **case** in the body of your program. The other use of **case** is in the variable or type declaration portion of the program. (See the "Variant Records" section in Chapter 3 for details on this use.)

**Case** takes the following syntax:

```
case expr of                              {case is a statement.}
        constantlist1 : stmnt1;
            .         .
            .         .
            .         .
        constantlistN : stmntN;
        otherwise  stmnt_list;
end;
```

## ARGUMENTS

| | |
|---|---|
| *expr* | Any ordinal expression (variable, constant, etc.) The ordinal types are integer, Boolean, char, enumerated, and subrange. You cannot specify an array as an expr, though you can specify an element of an array (assuming the element has an ordinal type). Also, you cannot specify a record, though you can specify a field of the record (assuming the field has an ordinal type). |
| *constantlist* | One or more values (separated by commas) having the same data type as *expr*. |
| *stmnt* | A simple statement or a compound statement (refer to the "Statements" listing later in this encyclopedia). |
| *stmnt_list* | One or more statements associated with the optional **otherwise** clause. (The **otherwise** clause tells the system to execute *stmnt_list* if *expr* matches none of the constants in any of the *constantlists*.) The *stmnt_list* differs from a compound statement in that you do not have to bracket the *stmnt_list* with a **begin/end** pair (though doing so does not cause an error). |

## DESCRIPTION

The **case** statement performs conditional branching. It is very useful in situations involving a multi-way branch. When the value of *expr* equals one of the constants in a *constantlist,* the system executes the associated *stmnt*.

Note that **case** and **if/then/else** serve nearly identical purposes. The differences between **case** and **if/then/else** are:

- **Case** can compare only ordinal values. **If/then/else** can compare values of any data type.
- The system can sometimes execute a **case** statement faster than an equivalent **if/then/else** statement. That's because the Domain Pascal compiler sometimes translates a **case** statement into a dispatch table and always translates an **if/then/else** statement into a series of conditional tests.

Also, note that a **case** statement is often more readable than an **if/then/else** statement. For instance, compare the following **if/then/else** statement to its equivalent **case** statement:

```
IF grade = 'A' THEN
   write('Excellent')
 ELSE IF grade = 'B' THEN
   write('Good')
 ELSE IF grade = 'C' THEN
   write('Average')
 ELSE IF grade = 'D' THEN
   write('Poor')
 ELSE IF grade = 'F' THEN
   write('Failing');
```

```
CASE grade OF
     'A' : write('Excellent');
     'B' : write('Good');
     'C' : write('Average');
     'D' : write('Poor');
     'F' : write('Failing');
 end;
```

### Otherwise—Extension

As an extension to the **case** statement, Domain Pascal supports the **otherwise** clause. The **otherwise** clause tells the compiler to execute *stmnt_list* if *expr* matches none of the constants in any of the *constantlists*. For example, you can write the preceding **case** example as follows. Notice that you do not put a colon (:) after the keyword **otherwise**.

```
CASE grade OF
    'A' : write('Excellent');
    'B' : write('Good');
    'C' : write('Average');
    'D' : write('Poor');
    OTHERWISE write('Failing');
 end;
```

As mentioned earlier, the **begin/end** pair is optional in an **otherwise** clause. Therefore, these two **case** statements are equivalent:

```
CASE number OF
1, 2, 3 : writeln('Good');
OTHERWISE writeln('Great.');
          writeln('Encore.');
end;
```

```
CASE number OF
1, 2, 3 : writeln('Good');
OTHERWISE begin
          writeln('Great');
          writeln('Encore');
          end;
end;
```

**EXAMPLE**

```
PROGRAM case_example;
{ This program demonstrates the use of the case statement }
VAR
    a_letter : char;
    sale     : boolean;
    price    : array[1..5] of char;
BEGIN
    write('Is whole wheat bread on sale today? -- ');
    readln(a_letter);
    CASE a_letter OF
        'y', 'Y'  : sale := true;
        'n', 'N'  : begin
                        sale := false;
                        writeln('Remember to tell them it''s organic.');
                    end;
        OTHERWISE   begin
                        writeln('You have made a mistake.');
                        writeln('The correct response was YES or NO');
                        writeln('Please rerun the program');
                        return;
                    end;
    end; {CASE}
    if sale then
        price := '$1.99'
    else
        price := '$2.99';
    writeln('Mark it as ', price:5);
END.
```

**USING THIS EXAMPLE**

This program is available online and is named **case_example**.

**Chr** Returns the character whose ISO Latin–1 value corresponds to a specified ordinal number.

## FORMAT

**chr**(*number*) {**chr** is a function.}

## ARGUMENTS

| | |
|---|---|
| *number* | An integer. |

## FUNCTION RETURNS

The **chr** function returns a value with the **char** data type.

## DESCRIPTION

The **chr** function returns the character that has an ISO Latin–1 value equal to the value of the low eight bits of *number*. Appendix B contains a table of ISO Latin–1 values.

**Chr** produces a character with the bit pattern

```
number & 16#FF
```

Usually, *number* is between 0 and 127, in which case the character that **chr** returns is simply the character that has the ISO Latin–1 value of *number*. If *number* is greater than 127, **chr** returns the character having the ISO Latin–1 value of

```
number MOD 256
```

See the **mod** listing later in this encyclopedia.

Note that the **ord** function is the inverse of **chr** when **ord**'s argument type is **char**. (See the **ord** listing later in this encyclopedia.)

## EXAMPLE

```
PROGRAM chr_example;
  { This program demonstrates three uses for the CHR function.         }
 VAR
     capital_letter : 65..90;
     y : CHAR;
     age : 10..99;
     c_array : array[1..2] of char;
 BEGIN
 { First, we'll use CHR to convert an integer to its ISO Latin-1 value.}
  write('Enter an integer from 65 to 90 -- ');
  readln(capital_letter);
  y := CHR(capital_letter);
  writeln(capital_letter:1,   corresponds to the ', y:1, ' character');
  writeln;

 { Second, we'll use CHR to ring the node bell.                      }
   write(chr(7));

 { The graphics primitive function gpr_$text writes character arrays }
 { to the display.  But suppose you want gpr_$text to write an       }
 { integer. In order to accomplish this task, you would write a      }
 { routine similar to the following. which converts a 2-digit integer}
 { into a 2-character array.  Note that 48 is the ISO Latin-1 value  }
 { for the '0' character, 49 for the '1' character, and so on up to  }
 { 57 for the '9' character.                                         }
  write('Enter an integer from 10 to 99 -- '); readln(age);
  c_array[1] := CHR((age DIV 10) + 48);
  c_array[2] := CHR((age MOD 10) + 48);
  writeln('The first digit is ', c_array[1]:1);
  writeln('The second digit is ', c_array[2]:1);
  writeln('The entire array is ', c_array);
 END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **chr_example**:

```
Enter an integer from 65 to 90 -- 83
 83 corresponds to the S character

 Enter an integer from 10 to 99 -- 71
 The first digit is 7
 The second digit is 1
 The entire array is 71
```

Close Closes the specified file. (Extension)

## FORMAT

**close**(*filename*) {**close** is a procedure.}

## ARGUMENTS

| | |
|---|---|
| *filename* | A file variable. |

## DESCRIPTION

Use the **close** procedure to close the file *filename* that you opened with the **open** procedure. By closing, we mean that the operating system unlocks it. When a program terminates (naturally or as a result of a fatal error), the operating system automatically closes all open files. So the **close** procedure is optional.

You cannot close the predeclared files **input** and **output**, but if you try, Domain Pascal does not issue an error.

If *filename* is a temporary file, **close**(*filename*) deletes it.

Please see Chapter 8 for an overview of I/O.

**NOTE:** For permanent text files, your program should issue a **writeln** to the file just before closing it in order to flush the file's internal output buffer. If you don't include that **writeln**, the last line of the file may not be written.

## EXAMPLE

```
PROGRAM close_example;
 { This program demonstrates the CLOSE procedure. }

 CONST
     pathname = 'primates';
 VAR
     class  : text;   {a file variable}
     name   : array[1..20] of char;
     status : integer32;
```

```
begin
    writeln('This program writes data to file "primates"');

    open(class, pathname, 'NEW', status);    {Open a file for writing.}
    if status = 0 then
        rewrite(class)
    else
        return;

    writeln('Enter the names of the children in your class -- ');
    writeln('The last entry should be "end"');
    repeat
        readln(name);
        if name <> 'end' then
            writeln(class, name)
        else
            exit;
    until false;

    CLOSE(class);                            {Close the file for writing.}

{    . . .   }
{ Execute some time-consuming routines that do not access 'primates'. }
{    . . .  }


{Now, re-open the file for reading.}
    open(class, pathname, 'OLD', status);
    reset(class);

    writeln;
    writeln('Here are the names you entered:');
    repeat
        readln(class, name);
        writeln(name);
    until eof(class);

    CLOSE(class);
end.
```

## USING THIS EXAMPLE

This program is available online and is named **close_example**.

**Compiler Directives** Specify a variety of special services including conditional compilation and include files. (Extension)

## FORMAT

The Domain Pascal compiler understands the directives shown in Table 4-11. All directives begin with a percent sign (%). You can specify a directive anywhere a comment is valid. To use a directive, specify its name as a statement or inside a comment. (There is one exception: directives associated with the **-config** option cannot be used as comments.) For example, all of the following formats are valid:

*%directive*

{*%directive*}

(**%directive**)

If you specify a *directive* within a comment, the percent sign must be the first character after the delimiter (where spaces count as characters). In addition, you do not need to put a semicolon at the end of the *directive*.

You *must* place a semicolon after some *directives* if you use them as statements. Those *directives* are:

- **%begin_inline;**
- **%begin_noinline;**
- **%debug;**
- **%eject;**
- **%end_inline;**
- **%end_noinline;**
- **%include** '*pathname*';
- **%list;**
- **%natural_alignment;**
- **%nolist;**

- %**slibrary** '*pathname*';
- %**word_alignment;**

Table 4-11. Compiler Directives

| Directive | Action |
|---|---|
| **%begin_inline;** | Directs the compiler to expand subsequent routines inline, if **-opt** 3 or **-opt** 4 is specified. |
| **%begin_noinline;** | Directs the compiler not to expand subsequent routines inline, even if **-opt** 4 is specified. |
| ☆ **%config** | Lets you easily set up a warning message if you forget to compile with the **-config** compiler option. |
| **%debug;** | Directs Domain Pascal to compile lines prefixed with this directive when you use the **-cond** compiler option. If you do not use **-cond** when you compile, lines prefixed with **%debug** are not compiled. |
| **%eject;** | Directs Domain Pascal to put a formfeed in the listing file at this point. |
| ☆ **%else** | Specifies that a block of code should be compiled if the preceding **%if** predicate **%then** is false. |
| ☆ **%elseif** *predicate* **%then** | Directs the compiler to compile the code until the next **%else**, **%elseif**, or **%endif** directive, if and only if the predicate is true. |
| ☆ **%elseifdef** *predicate* **%then** | Checks whether additional predicates have been declared with a **%var** directive. |
| ☆ **%enable;** | Sets compiler directive variables to true. |
| **%end_inline;** | Directs the compiler to stop inline expansion, if **-opt** 3 or **-opt** 4 is specified. |
| **%end_noinline;** | Allows the compiler to resume inline expansion, if -opt 4 is specified. |
| ☆ **%endif** | Marks the end of a conditional compilation area of the program. |
| ☆ **%error** '*string*' | Prints '*string*' as an error message whenever you compile. |

☆ is a directive described in "Directives Associated with the -Config Option" later in this chapter.

*(Continued)*

*Table 4-11. Compiler Directives (Cont.)*

| Directive | Action |
|---|---|
| ☆ **%exit** | Directs the compiler to stop conditionally processing the file. |
| ☆ **%if** *predicate* **%then** | Directs the compiler to compile the code 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 Pascal to read input from the specified file. |
| **%list;** | Enables the listing of source code in the listing file. |
| **%natural_alignment;** | Sets environment to natural alignment. |
| **%nolist;** | Disables the listing of source code in the listing file. |
| **%slibrary** *'pathname'*; | Causes Domain Pascal to incorporate a precompiled library into the program. |
| **%pop_alignment;** | Saves the current alignment by pushing it onto a stack. |
| **%push_alignment;** | Restores the alignment saved by the last **%push_alignment.** |
| ☆ **%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. |
| **%word_alignment;** | Sets environment to default alignment. |

☆ is a directive described in "Directives Associated with the -Config Option" later in this chapter.

## 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. Unlike the other compiler directives, conditional directives cannot be used as comments.

Several of the directives take a predicate. A predicate can contain any of the following:

- Special variables that you declare with the **%var** directive
- Optional Boolean keywords **not**, **and**, or **or**
- A predeclared conditional variable, **_BFMT__COFF**

  **_BFMT__COFF** is a Boolean variable. The value of **_BFMT__COFF** is set to *true* whenever the compiler is generating COFF (Common Object File Format) files. Otherwise, the value of **_BFMT__COFF** is set to *false*.

  Beginning with SR10, the Domain Pascal compiler generates COFF files whenever it compiles your source code.

  NOTE: There are *two* underscore (_) characters between 'BFMT' and 'COFF' in the name of this variable.

- A pair of predeclared conditional variables that you can use to find out whether the compiler is generating code for the 68000 family of workstations or for the Series 10000. These variables are:

  **_ISP__M68K** (for 68000 code generation)

  **_ISP__A88K** (for Series 10000 code generation)

  NOTE: There are *two* underscore (_) characters after '_ISP' in the names of these variables.

For example, given that **color** and **mono** are special variables that you defined using **%var**, here are some possible predicates:

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

**%if** *predicate* **%then**

If the *predicate* is true, Domain Pascal 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. %Else** 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**

**%Elseif** *predicate* **%then** is used in conjunction with **%if** *predicate* **%then**. It serves an analogous purpose to the Pascal 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 Pascal compiles the code for color nodes since it appears first. Note that you can put multiple **%elseif** directives in the same block.

Or, suppose that you want to tailor a source module for a specific application, depending on whether the compiler is generating code for the 68000 family of workstations or the Series 10000. Consider the following fragment:

```
%IF _ISP__M68K %THEN PROCEDURE do_68K ...

%ELSEIF _ISP__A88K  %THEN PROCEDURE do_100000 ...

%ENDIF;
```

The above fragment tells the compiler to compile the **do_68K** procedure if it is generating 68K code and to compile the **do_10000** procedure if it is generating code for the Series 10000.

> NOTE: Since **_ISP__M68K** and **_ISP__A88K** are predeclared, you cannot use the **%enable** directive or the **-config** option with them.

### %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 was already declared with a **%var** directive. If you accidentally declare the same variable more than once, Domain Pascal issues an error message. **%Ifdef** is a way of avoiding this error message. **%Ifdef** 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 }
     %VAR color             {with %VAR, declare it now.     }
%ENDIF;
```

> **NOTE:** 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 whether or not 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  {If mono has not been declared with     }
    %VAR mono               {%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. (**%Enable** 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 Pascal assumes that it is false.

For example, the following example declares three variables named **code.sr9**, **code.sr8**, and **code.sr7**, then it sets **code.sr9** and **code.sr7** to true:

```
%VAR     code.sr9  code.sr8  code.sr7
%ENABLE code.sr9  code.sr7
```

The compiler preprocessor 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 use **%config** only in a predicate. The Domain Pascal 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 to use 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:** You cannot 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 all 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 do compile with the **-config mono** option, Domain Pascal prints out the following error message:

```
(0011)   %ERROR 'I have not finished the code for a monochromatic node.'
******** Line 11:  Conditional compilation user error.
 1 error, no warnings, Pascal Rev n.nn
```

Because of the error, Domain Pascal does not create an executable object.

**%warning** *'string'*

This directive causes the compiler to print *'string'* as a warning message. You must place this directive on a line all 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 Pascal prints out the following error message:

```
 (0011)       %WARNING 'You forgot to use the -CONFIG color option.
 ******** Line 11: Warning:  Conditional compilation user warning.
 No errors, 1 warning, Pascal Rev n.nn
```

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

### %exit

**%Exit** directs the compiler to stop processing the file. For example, if you put **%exit** in an include file, Domain Pascal only reads in the code up until **%exit**. (It ignores the code that appears after **%exit**.)

**%Exit** has no effect if it is in a part of the program that does not get compiled.

## 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** directives are delimiters that define routines for inline expansion. Inline expansion means that the compiler generates code for a given routine wherever a call to that routine appears.

Inline expansion of a given routine allows you to avoid the overhead of a procedure or function call. When used with small routines, inline expansion can increase execution speed. It also increases the size of the executable code, however.

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 procedure or function definitions.
- Place **%end_inline** on the line following the last routine that you define for inline expansion.

Suppose that a program contains this function declaration:

```
%begin_inline;
function test_pos (number: real) : boolean;
begin
    test_pos := (number >= 0.0);
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.

You cannot nest these directives, and you must have matching pairs to begin and end the specification. The compiler detects recursion and will not use inline expansion if doing so would cause the compiler to loop indefinitely.

The **%begin_inline** and **%end_inline** directives are effective only if you compile with an optimization level of 3 or 4. With Domain Pascal, level 3 is the default.

At optimization level 3, the compiler expands all routines that are enclosed between **%begin_inline** and **%end_inline** directives, so long as they are not recursive. At optimization level 4, the compiler expands all of these functions, and also selects other functions that are suitable for inline expansion.

### %begin_noinline; and %end_noinline;

The **%begin_noinline** and **%end_noinline** directives are delimiters for routines that the compiler must never expand inline. These delimiters are the converse of **%begin_inline** and **%end_inline**. Use **%begin_noinline** and **%end_noinline** if you specify an optimization level of 4, but want to restrict inline expansion of certain functions.

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

- Place **%begin_noinline** on a line in the source file before you begin any appropriate procedure or function definitions.
- Place **%end_noinline** on the line following the last routine that you define for no inline expansion.

The following code fragment tells the compiler not to use inline expansion for the **for_loop** procedure under any circumstances.

```
%begin_noinline;
procedure for_loop;
var
    i : integer32;
begin
    for i := 1 to 100 do
        writeln('i is ', i);
end;
%end_noinline;
```

## %debug;

The **%debug** directive marks source code for conditional compilation. The "condition" is the compiler option, **-cond**. If you compile with the **-cond** option, then the compiler compiles the lines that begin with **%debug**. If you do not compile with the **-cond** switch, Domain Pascal does not compile the lines that begin with **%debug**. The reason this directive is called **%debug** is that it can help you debug your program.

For instance, consider the following fragment:

```
          value := data + offset;
 %DEBUG;   writeln('Current value is ', value:3);
```

The preceding fragment contains one **%debug** directive. If you compile with the **-cond** option, then the system executes the **writeln** statement at run time. If you compile without the **-cond** option, the system does not execute the **writeln** 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 **writeln** message.

The **%debug** directive applies to one physical line only, not to one Domain Pascal statement. Therefore, in the following example, **% debug** applies only to the **for** clause. If you compile with **-cond**, Domain Pascal compiles both the **for** statement and the **writeln** procedure. If you compile without **–cond**, Domain Pascal compiles only the **writeln** procedure (and thus there is no loop).

```
%DEBUG;    FOR j := 1 to max_size do
                WRITELN(barray[j]);
```

If you **%debug** within a line, text to the left of the directive is always compiled, and text to the right of the directive is conditionally compiled.

**%eject;**

The **%eject** directive does not affect the **.bin** file; it only affects the listing file. (The **–l** compiler option causes 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 Pascal 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, procedure, 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 **/sys/ins** directory; see Chapter 6 for details.

Domain Pascal permits the nesting of include files. That is, an include file can itself contain an **%include** directive.

The compiler option **–idir** enables you to select alternate pathnames for insert files at compiletime. See Chapter 6 for details.

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

**%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.pas';
 %INCLUDE '/sys/ins/ios.ins.pas';
 %LIST;
  ...
```

**%List** is the default.

%slibrary **'pathname';**

The **%slibrary** directive is analogous to the **%include** directive. While **%include** tells the compiler to read in Domain Pascal source code, **%slibrary** tells it to read in previously-compiled code.

The **%slibrary** directive tells the compiler to read in a precompiled library residing at *'pathname'*. The compiler inserts the precompiled library where you place the **%slibrary** directive. The compiler acts as if the files that were used to produce the precompiled library were included at this point, except that any conditional compilation will have already occurred during precompilation.

Precompiled libraries can only contain declarations; they may *not* contain routine bodies and may *not* declare variables that would result in allocating storage in the default data section, **.data**. This means the declarations must either put variables into a named section, or must use the **extern** variable allocation clause. See Chapter 3 for more information about named sections, and Chapter 7 for details on **extern**.

Use the **–slib** compiler option (described in Chapter 6) to precompile a library and then insert **–slib**'s result in *'pathname'*. For example, if you create a precompiled library called **mystuff.ins.plb**, this is how to include it in your program:

```
%SLIBRARY 'mystuff.ins.plb';
```

Precompiled library pathnames by default end in **.plb**.

## %natural_alignment; and %word_alignment;

Use the **%natural_alignment** and **%word_alignment** directives to tell the compiler how to align any data *that does not have an alignment attribute in its declaration.* (See the "Alignment—Extension" section of Chapter 3 for details about the alignment attributes). Specifically,

- Use the **%natural_alignment** directive to set data alignment to natural.
- Use the **%word_alignment** directive to set all data in the environment to word alignment.

By default, the Domain Pascal compiler aligns all objects larger than a byte in word-alignment mode. If you want to change this, you can use the **%natural_alignment** directive. Conversely, if you want the compiler to resume word-alignment mode, you can specify the **%word_alignment** directive, which overrides the previous directive. In any case, the alignment directive you specify stays in effect until you specify another one.

You can use these directives in combination with the predeclared conditional variables **_ISP__68K** and **_ISP__A88K** to compile source modules according to whether they will be run on 680x0 or Series 10000 workstations. For example, one way to compile your program so that it will run efficiently on a Series 10000 workstation would be to add these lines to the beginning of the program:

```
%IF _ISP__A88K
%THEN
  %NATURAL_ALIGNMENT;
%ENDIF
```

**NOTE:** You can use an alignment directive to set the alignment for programs or program modules written prior to SR10 that did not include alignment attributes. Thus, you can gain the improved performance that results from natural alignment without rewriting all your variable and type declarations.

However, you *must* be careful when using these directives with files on disk that have pre-SR10 record formats. The **%natural_alignment** and **%word_alignment** directives *change record layout*. This means that the fields of records are in different positions and you may be unable to access them. (See the "Internal Representation of Unpacked Records" section of Chapter 3 for details about the alignment of data in records.)

## %push_alignment; and %pop_alignment;

The **%push_alignment** and **%pop_alignment** directives are designed to save and restore the current alignment mode while another alignment mode is used by a particular structure or file. The **%push_alignment** directive saves the current alignment mode by pushing it onto a stack. The **%pop_alignment** directive restores the alignment mode saved by the last **%push_alignment** by popping it off the stack.

These directives are useful for controlling the alignment of **%include** files. For example, a **%push_alignment** directive placed at the top of an **%include** file tells the compiler to save the current alignment mode by pushing it onto the stack. A **%pop_alignment** directive placed at the end of the **%include** file restores the alignment mode saved by the last **%push_alignment**.

Cos Calculates the cosine of the specified number.

## FORMAT

**cos**(*number*) {**cos** is a function.}

## ARGUMENTS

| | |
|---|---|
| *number* | Any real or integer value in radians (not degrees). |

## FUNCTION RETURNS

The **cos** function returns a real value (even if *number* is an integer).

## DESCRIPTION

The **cos** function calculates the cosine of *number*.

## EXAMPLE

```
PROGRAM cos_example;

 { This program demonstrates the COS function. }

 CONST
     pi = 3.1415926535;

 VAR
     degrees : INTEGER;
     q, c1, c2, radians : REAL;

 BEGIN
     q := 0.5;
     c1 := COS(q);  { Find the cosine of one-half radians. }
     writeln('The cosine of ', q:5:3, ' radians is ', c1:5:3);
```

```
{The following statements show how to convert from degrees to radians.}
{More specifically, they find the cosine of 14 degrees.}
    degrees := 14;
    radians := ((degrees * PI) / 180.0);
    c2 := COS(radians);
    writeln('The cosine of ', degrees:1, ' degrees is ', c2:5:3);
END.
```

## USING THIS EXAMPLE

If you execute the sample program **cos_example**, you get the following output:

```
The cosine of 0.500 radians is 0.878
 The cosine of 14 degrees is 0.970
```

**Ctop** Converts a C–style string to a variable–length string. (Extension)

## FORMAT

**ctop**(*string*); {**ctop** is a procedure.}

## ARGUMENTS

| | |
|---|---|
| *string* | A variable–length string. |

## DESCRIPTION

The **ctop** procedure converts a C–style null–terminated string into a variable–length string by searching the **body** field of the string for a terminating null byte and setting the string's length field accordingly. (See the "Variable–Length Arrays—Extension" section of Chapter 3 for details about variable–length strings.) Note that this function does not remove the null byte; it simply sets the length field to one less than the null byte position.

See the description of **ptoc** for information about converting a variable–length string into a null–terminated string.

## EXAMPLE

(See the description of the **ptoc** procedure.)

---

**Discard** Explicitly discards the return value of an expression. (Extension)

---

## FORMAT

**discard**(*exp*) {**discard** is a procedure.}

## ARGUMENTS

| | |
|---|---|
| *exp* | Any expression, including a function call. |

## DESCRIPTION

In its effort to produce efficient code, the compiler sometimes issues warning messages concerning optimizations it performs. Those optimizations might not be right for your particular situation. For example, if you compute a value but never use it, the compiler may eliminate the computation, or the assignment of the value, and issue a warning message.

However, there are times when you call a function for its side effects rather than its return value. You don't need the value, but a Pascal function *always* returns a value to retain legal program syntax. You must keep the function call in your program, but if you don't use the value, the compiler's optimizer automatically discards the return value and issues a warning message.

Since you know the return value is useless, in such a case you may want to eliminate this particular warning message. Domain Pascal's **discard** procedure explicitly throws away the value of its *exp* and so gets rids of the warning. For example, to call a function that returns a value in **arg1** without checking that value, use **discard** as follows:

```
DISCARD(my_function(arg1));
```

## EXAMPLE

```
PROGRAM discard_example;

 VAR
    payment, monthly_sal : real;

 {  The following function figures out whether a user can afford the   }
 {  mortgage payments for a given house based on the rule that no more }
 {  than 28% of one's gross monthly income should go to housing costs. }
```

```
FUNCTION enough(in        payment : real;
                in out monthly_sal : real) : boolean;
VAR
    amt_needed : real;

BEGIN
writeln;
amt_needed := monthly_sal * 0.28;

if amt_needed < payment then
    begin
    enough := false;
    monthly_sal := payment / (0.28);
    writeln ('Your monthly salary needs to be ', monthly_sal:6:2);
    end
else
    begin
    enough := true;
    writeln('Amazing! You can afford this house.');
    end
END;            {end function enough}

BEGIN           {main program}
    write ('How much is the monthly payment for this house? ');
    readln (payment);
    write ('What is your gross monthly salary? ');
    readln (monthly_sal);

{ The function enough can change the value of the global variable }
{ monthly_sal, so the function call is important, but its return  }
{ value is not. DISCARD that return value.                        }

    DISCARD (enough(payment,monthly_sal));
END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **discard_example**:

```
How much is the monthly payment for this house? 928
 What is your gross monthly salary? 2400

 Your monthly salary needs to be 3314.29
```

## Dispose

**Dispose** Deallocates the storage space that a dynamic variable was using. (Refer also to **New**.)

### FORMAT

**Dispose** is a predeclared procedure that takes one of two formats. The format you choose depends on the format you use to call the **new** procedure. If you create a dynamic variable with the short form of **new**, then you must use the short form for **dispose**, which is:

**dispose**(*p*) {**dispose** is a procedure.}

If you create a dynamic variant record with the long form of **new**, then you must use the long form of **dispose**, which is:

**dispose**(*p, tag1..tagN*);

### ARGUMENTS

| | |
|---|---|
| *tag* | One or more constants. The number of constants in a **dispose** call must match the number of constants in the **new** call. |
| *p* | A variable declared as a pointer. After you call **dispose**(*p*), Domain Pascal sets *p* to **nil**. |

### DESCRIPTION

If *p* is a pointer, then **dispose**(*p*) causes Pascal to deallocate space for the occurrence of the record that *p* points to. Deallocating means that Pascal permits the memory locations occupied by the dynamic record to be occupied by a new dynamic record. For example, consider the following declarations:

```
TYPE
    employeepointer = ^employee;
    employee = record
       first_name : array[1..10] of char;
       last_name  : array[1..14] of char;
       next_emp   : employeepointer;
    end;

 VAR
    current_employee : employeepointer;
```

To store employee records dynamically, call **new(current_employee)** for every employee. If an employee leaves the company, and you want to delete his or her record, you can call **dispose(current_employee)**. **Dispose** returns the storage occupied by that record for reuse by a subsequent **new** call.

If you create a dynamic record using a long-form **new** procedure, then you must call **dispose** with the same constants. For example, if you create a dynamic record by calling **new(widget, 378, true)**, then to deallocate the stored record, you must call **dispose(widget, 378, true)**.

Note that the **dispose** procedure merely deallocates the record. If this disconnects a linked list, then it is up to you to reset the pointers. If some other variable points to this record and another program uses **dispose** to deallocate the record, then you get erroneous results.

> NOTE: If you call **dispose**(*p*) when *p* is **nil**, Domain Pascal reports an error. It is also an error to call **dispose** when *p* points to a block of storage space that you already deallocated with **dispose**. Finally, if you use a pointer copy that points to deallocated space, the results are unpredictable.

## EXAMPLE

For a sample program that uses **dispose**, refer to the **new** listing later in this encyclopedia.

**Div** Calculates the quotient (excluding the remainder) of two integers.

## FORMAT

*d1* **div** *d2* {**div** is an operator.}

## ARGUMENTS

| | |
|---|---|
| *d1, d2* | Any integer expression. |

## OPERATOR RETURNS

The result of a **div** operation is always an integer.

## DESCRIPTION

The expression (**d1 div d2**) produces the integer (nonfractional) result of dividing **d1** by **d2.** The **div** operator uses the division rules of standard mathematics regarding negatives. For example, consider the following results:

```
 9 DIV 3 is equal to 3            -9 DIV 3 is equal to -3
10 DIV 3 is equal to 3           -10 DIV 3 is equal to -3
11 DIV 3 is equal to 3           -11 DIV 3 is equal to -3
12 DIV 3 is equal to 4           -12 DIV 3 is equal to -4
13 DIV 3 is equal to 4           -13 DIV 3 is equal to -4

 9 DIV (-3) is equal to -3        -9 DIV (-3) is equal to 3
10 DIV (-3) is equal to -3       -10 DIV (-3) is equal to 3
11 DIV (-3) is equal to -3       -11 DIV (-3) is equal to 3
12 DIV (-3) is equal to -4       -12 DIV (-3) is equal to 4
13 DIV (-3) is equal to -4       -13 DIV (-3) is equal to 4
```

To find the remainder of an integer division operation, use the **mod** operator. (See the **mod** listing later in this encyclopedia.)

See the "Expressions" listing later in this encyclopedia for information on using binary and unary operators together.

## EXAMPLE

```
PROGRAM div_example;
 { This program converts a 3-digit integer to a 3-character array.     }
 { Note that the character 0 has an ISO Latin-1 value of 48, the       }
 { character 1 has an ISO Latin-1 value of 49, and so on up until the  }
 { character 9, which has an ISO Latin-1 value of 57.                  }

 VAR
     x        : 100..999;
     digits   : array[1..3] of char;

 BEGIN
    write('Enter a three-digit integer -- ');
    readln(x);

    digits[1] :=chr(48 + (x DIV 100));
    x := x MOD 100;
    digits[2] := chr(48 + (x DIV 10));
    digits[3] := chr(48 + (x MOD 10));

    writeln(digits);
 END.
```

## USING THIS EXAMPLE

This program is available online and is named **div_example**.

**Do** Refer to the **For** or **While** listings later in this encyclopedia.

**Downto** Refer to **For** later in this encyclopedia.

## Else

**Else** Refer to **If** later in this encyclopedia.

**End** Signifies the end of a group of Pascal statements.

## FORMAT

**End** is a reserved word.

## DESCRIPTION

**End** is the terminator for a sequence of Pascal statements. A Pascal program must contain an **end** to match every **begin**.

Pascal requires a **begin/end** pair to indicate a compound statement. (Refer to the "Statements" listing later in this encyclopedia.)

Pascal requires **end** (without an accompanying **begin**) in the following situations:

- To terminate a **case** command.
- To terminate a **record** declaration.

## EXAMPLE

```
PROGRAM begin_end_example;

{This program does very little work, but does have lots of BEGINs }
{and ENDs.                                                         }

TYPE
    student = record
       age : 6..12;
       id  : integer16;
    end;  {student record definition}
 VAR
    x   : integer32;

PROCEDURE do_nothing;
BEGIN   {do_nothing}
writeln('You have triggered a procedure that does absolutely nothing.');
writeln('Though it does do nothing with elan.').
END;    {do_nothing}
```

```
FUNCTION do_next_to_nothing(var y : integer32) : integer32;
BEGIN  {do_next_to_nothing}
    do_next_to_nothing := abs(y);
END;   {do_next_to_nothing}

BEGIN  {main procedure}
    write('Enter an integer -- ');    readln(x);
    if x < 0
      then BEGIN
             writeln('You have entered a negative number!!!');
             writeln('Its absolute value is ', do_next_to_nothing(x):1);
            END
      else if x = 0
             then BEGIN
                    writeln('You have entered zero');
                    do_nothing;
                  END
             else
               writeln('You have entered a positive number!!!');
END.   {main procedure}
```

### USING THIS EXAMPLE

This program is available online and is named **begin_end_example**.

**Eof** Tests the current file position to see if it is at the end of the file.

## FORMAT

**eof**(*filename*) {**eof** is a function.}

## ARGUMENTS

|  |  |
|---|---|
| *filename* | A file variable symbolizing the pathname of an open file. The *filename* argument is optional. If you do not specify *filename*, Domain Pascal assumes that the file is standard input (**input**). |

## FUNCTION RETURNS

The **eof** function returns a Boolean value.

## DESCRIPTION

The **eof** function returns true if the current file position is at the end of file *filename*; otherwise, it returns false. With one exception, *filename* must be open for either reading or writing when you call **eof**. The one exception occurs when *filename* is **input**; for a description of this exception, see the "Interactive I/O" section in Chapter 8.

## EXAMPLE

```
PROGRAM eof_example;
{NOTE: Before running this program, you must obtain file "annabel_lee" }
{      and store it in the same directory as the program.               }

CONST
    title_of_poem = 'annabel_lee';
VAR
    poetry  : text;
    stat    : integer32;
    a_line  : string;
BEGIN
{Open file anabel_lee for reading.}
open(poetry, title_of_poem, 'OLD', stat);
if stat = 0 then
    reset(poetry)
else
    return;
{Read each line from the file and write each line to the screen. }
{Halt execution when end of file is reached.                     }
while not EOF(poetry) do
    begin
    readln(poetry, a_line);
    writeln(output, a_line);
    end;
END.
```

## USING THIS EXAMPLE

This program is available online and is named **eof_example**.

**Eoln** Tests the current file position to see if it is pointing to the end of a line.

## FORMAT

**eoln**(*f*)  {**eoln** is a function.}

## ARGUMENTS

| | |
|---|---|
| *f* | A variable having the **text** data type. *f* is optional; if you do not specify it, **eoln** tests the standard input (**input**) file. |

## FUNCTION RETURNS

The function returns a Boolean value.

## DESCRIPTION

The **eoln** function returns true when the stream marker points to an end-of-line character; otherwise, with two exceptions, **eoln** returns false. The two exceptions are:

- **Eoln** causes a run-time error if *f* was not opened for reading (with **reset**) or for writing (with **rewrite** ). However, you do not need to open **input** or **output** for reading or for writing. (See the "Interactive I/O" section in Chapter 8 for details on **input** and **output**.)
- **Eoln** causes a run-time error if **eof(f)** is true.

## EXAMPLE

```
PROGRAM eoln_example;
{NOTE: Before running this program, you must obtain file "annabel_lee" }
{      and store it in the same directory as the program.              }

CONST
    title_of_poem = 'annabel_lee';

VAR
    poetry  : text;
    stat    : integer32;
    a_char  : char;
```

```
BEGIN
{ Open file annabel_lee for reading. }
    open(poetry, title_of_poem, 'OLD', stat);
    if stat = 0
        then reset(poetry)
        else return;
{ Read in the first line of the poem one character at a time,  }
{ and write each character to the screen.                      }
    repeat
          read(poetry, a_char);
          writeln(output, a_char);
    until EOLN(poetry);
END.
```

## USING THIS EXAMPLE

This program is available online and is named **eoln_example**.

**Exit** Transfers control to the first statement following a **for, while,** or **repeat** loop. (Extension)

## FORMAT

**Exit** is a statement that neither takes arguments nor returns values.

## DESCRIPTION

Use **exit** to terminate a loop prematurely; that is, to jump out of the loop you're in. In nested loops, **exit** applies to the innermost loop in which it appears. You can use **exit** within a **for, while,** or **repeat** loop only. If **exit** appears elsewhere in a program, Domain Pascal issues an error.

It is preferable to use **exit** for jumping out of a loop prematurely rather than **goto**. That's because **goto** inhibits some compiler optimizations that **exit** does not.

## EXAMPLE

```
PROGRAM exit_example;
{This program demonstrates the exit statement.  }
VAR
    i, j        : integer16;
    data        : real;
    geiger      : array[1..5, 1..3] of real := [[* of 0.0],[* of 0.0],];
BEGIN
for i := 1 to 4 do
  begin
    writeln;
    for j := 1 to 3 do
      begin
      writeln(chr(10), 'Enter the data for coordinates', i:2, ',', j:1);
      write('(or enter -1 to jump down to the next row) -- ');
      readln(data);
      if data = -1 then
          EXIT
      else
          geiger[i,j] := data;
      end; {for j}
  end; {for i}
END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **exit_example**:

```
Enter the data for coordinates 1,1
 (or enter -1 to jump down to the next row) -- 1.2

 Enter the data for coordinates 1,2
 (or enter -1 to jump down to the next row) -- -1


 Enter the data for coordinates 2,1
 (or enter -1 to jump down to the next row) -- 3.2

 Enter the data for coordinates 2,2
 (or enter -1 to jump down to the next row) -- 1.2

 Enter the data for coordinates 2,3
 (or enter -1 to jump down to the next row) -- 4.3


 Enter the data for coordinates 3,1
 (or enter -1 to jump down to the next row) -- -1


 Enter the data for coordinates 4,1
 (or enter -1 to jump down to the next row) -- 1.3

 Enter the data for coordinates 4,2
 (or enter -1 to jump down to the next row) -- 4.2

 Enter the data for coordinates 4,3
 (or enter -1 to jump down to the next row) -- -5
```

**Exp** Calculates the value of e, the base of natural logarithms, raised to the specified power. (See also **Ln**.)

## FORMAT

**exp**(*number*) {**exp** is a function.}

## ARGUMENTS

| | |
|---|---|
| *number* | Any real or integer expression. |

## FUNCTION RETURNS

The **exp** function returns a real value.

## DESCRIPTION

The **exp** function returns *e* raised to the power specified by *number*.

*e* to 16 significant digits is 2.718281828459045.

Note that Domain Pascal supports an exponentiation operator. (See the "Overview: Mathematical Operators" section earlier in this chapter for details about the exponentiation operator.)

## EXAMPLE

```
PROGRAM exp_example;
{This example demonstrates the use of EXP in calculating the}
{exponential growth of bacteria.                            }

CONST

c1  = 0.3466;

VAR

starting_quantity   : INTEGER;
ending_quantity, elapsed_time : REAL;
```

```
BEGIN

write('How many bacteria are there at zero hour? -- ');
readln(starting_quantity);
write('How many hours pass? -- ');
readln(elapsed_time);

ending_quantity := starting_quantity * EXP(c1 * elapsed_time);

writeln('There will be approximately ', ending_quantity:1,' bacteria.');

END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **exp_example**:

```
How many bacteria are there at zero hour? -- 10500
 How many hours pass? -- 5.6
 There will be approximately 7.313705E+04 bacteria.
```

## Expressions

Throughout this encyclopedia, we refer to expressions. Here, we define expressions. An expression can be any of the following:

- A constant declared in a **const** declaration part
- A variable declared in a **var** declaration part
- A constant value
- A function call
- Any one of the above preceded by a unary operator appropriate to its data type
- Any two of the above separated by a binary operator appropriate to their data types

You can organize expressions into more complex expressions with parentheses. For example, the **odd** function requires an integer expression as an argument. The following program fragment demonstrates several possible arguments to **odd**:

```
CONST
    century := 100;

VAR
    x, y : integer;
  result : boolean;

BEGIN
    . . .
    result := ODD(century);           {a constant}
    result := ODD(x);                 {a variable}
    result := ODD(15);                {a value}
    result := ODD(sqr(25));           {a function}
    result := ODD(x + y);             {an operation}
    result := ODD((x * 3) + sqr(y));  {several operations}
    . . .
```

## Expressions

NOTE: You cannot follow a binary operator with a unary operator of lower precedence. For example, consider the following proper and improper expressions:

```
9 DIV -3   {improper expression}
9 DIV (-3) {proper expression}
5 * -100   {improper expression}
5 * (-100) {proper expression}
```

Table 4–3 shows the order of precedence of operators.

**Find** Sets the file position to the specified record. (Extension)

## FORMAT

**find**(*file_variable, record_number, error_status*); {**find** is a procedure.}

## ARGUMENTS

| | |
|---|---|
| *file_variable* | Must be a variable having the **file** data type. The *file_variable* argument cannot be a variable having the **text** data type. |
| *record_number* | Must be an integer between 1 and *n* or between -1 and -*n*, where 1 denotes the first record of the file and *n* denotes the last record. |
| *error_status* | Must be declared as a variable with the **integer32** data type. Domain Pascal returns a hexadecimal number in *error_status* which has the following meaning: |

```
0 - no error or warning occurred.

greater than 0 - an error occurred.

less than 0 - a warning occurred.
```

**NOTE:** Your program is responsible for handling the error. We detail error handling in Chapter 9.

## DESCRIPTION

Before reading this, make sure you are familiar with the description of I/O in Chapter 8.

When you open a file for reading, the operating system sets the stream marker to the beginning of the file. You can call **read** to move this stream pointer sequentially, or you can call **find** to move it randomly.

Before you can call **find**, you must have first opened the file symbolized by *file_variable* for reading. (See Chapter 8 for a description of opening files for reading.) When you call **find**, Domain Pascal sets the stream marker to point to the record specified by *record_number*.

If you specify a *record_number* between 1 and *n*, where *n* is the number of records in the file, **find** locates that number record. If *record_number* is between −1 and −*n*, **find** counts backward from the end of the file to locate the proper record. For example, if there are five records in the file and you specify −4 for *record_number*, Domain Pascal counts back four from the end of the file and retrieves record number 2.

If you specify *record_number* as zero, the compiler returns an error code in **error_status.**

If you specify a *record_number* that is one greater than the number of records stored in the file, Domain Pascal does not return an error code, but does not change the stream marker either.

After executing a **find,** Domain Pascal sets the stream marker to point to the beginning of the next record. For example, if *record_number* is 2, then after executing a **find,** Domain Pascal sets the stream marker to point to record 3.

Frequently, programmers use the **find** procedure with the **replace** procedure (which is described later in this encyclopedia).

**NOTE:** The term "record," as it applies to files of **file** type, refers to a data object of the file's base type. This is not necessarily a Domain Pascal record type.

## EXAMPLE

```
PROGRAM find_and_replace_example;

{This program demonstrates the FIND and REPLACE procedures.  }
{ NOTE: File 'his101' must exist before you run get_example. }
{       To create 'his101', you must run put_example.        }

%NOLIST;  { We need these include files for error checking.  }
%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%LIST;

CONST
    pathname = 'his101';

TYPE
    student = RECORD
                name   : array[1..12] of char;
                age    : integer16;
              END;
```

```
VAR
    class              : FILE OF student;
    a_student          : student;
    st                 : status_$t;
    more_corrections   : char;
    particular_record  : integer16 := 0;
    n                  : integer16;

PROCEDURE print_records;
BEGIN
    n := 0;
    writeln(chr(10), 'Here are the records stored in the file:');
    reset(class);
    repeat
        n := n + 1;
        read(class, a_student);
        writeln('record ', n:2, '      ', a_student.name, a_student.age);
    until eof(class);
END;

PROCEDURE correct_errors;
BEGIN
    write('Enter the number of the record you wish to change -- ');
    readln(particular_record);
    if particular_record = n+1 then
        writeln ('There are only ', n:2, ' records in the file.')
    else
        BEGIN
        FIND(class, particular_record, st.all);
        if st.code = 0 then
            BEGIN
            write('What should this name be -- ');
            readln(a_student.name);
            write('What should this age be -- ');
            readln(a_student.age);
            class^ := a_student;
            REPLACE(class);
            END
        else if st.code = stream_$end_of_file then
            BEGIN
            write('You specified a number greater than the number of ');
            writeln ('records in the file.');
            END
        else
            error_$print(st);
        END;
END;
```

```
BEGIN  {main procedure}
    open(class, pathname, 'OLD', st.all);
    if st.code = 0 then
        BEGIN
        repeat
        print_records;
        write('Do you want to correct any records? (enter y or n) -- ');
        readln(more_corrections);
        if more_corrections = 'y' then
            correct_errors
        else
            exit;
        until false;
        END
    else if st.code = stream_$name_not_found then
        writeln('Did you remember to run put_example to create his101?')
    else
        error_$print(st);
END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **find_and_replace_example** :

```
Here are the records you have entered:
record  1     Kerry                 28
record  2     Barry                 26
record  3     Jan                   25
Do you want to correct any records? (enter y or n) -- y
Enter the number of the record you wish to change -- 2
What should this name be -- Sandy
What should this age be -- 27

Here are the records you have entered:
record  1     Kerry                 28
record  2     Sandy                 27
record  3     Jan                   25
Do you want to correct any records? (enter y or n) -- n
```

**Firstof** Returns the first possible value of a type or a variable. (Extension)

## FORMAT

**firstof**(*x*) {**firstof** is a function.}

## ARGUMENTS

| | |
|---|---|
| *x* | Is either a variable or the name of a data type. The data type can be a predeclared Domain Pascal data type, or it can be a user-defined data type. *x* cannot be a record, file, or pointer type. |

## FUNCTION RETURNS

The **firstof** function returns a value having the same data type as *x*.

## DESCRIPTION

The **firstof** function returns the first possible value of *x* according to the following rules:

| Data Type of x | Firstof Returns |
|---|---|
| **integer** or **integer16** | -32767 |
| **integer32** | -2147483647 |
| **char** | The character represented by **chr(0)** called **nul**. |
| **boolean** | False. |
| **enumerated** | The first (leftmost) identifier in the data type declaration. |
| **array** | The lower bound of the subrange that defines the array's size. |
| **varying array** | 1 |

The **firstof** function is particularly useful for finding the first element of an enumerated type (as in the example).

## EXAMPLE

```
PROGRAM firstof_lastof_example;

{This program demonstrates the use of the firstof and lastof functions}
TYPE
   astronomers = (aristotle, galileo, newton, tycho, kepler);
VAR
   stargazers  : astronomers;
BEGIN
   writeln('The following is a list of great astronomers:');
   for stargazers := firstof(astronomers) to lastof(astronomers) do
      writeln(stargazers);
END.
```

## USING THIS EXAMPLE

If you execute the sample program named **firstof_lastof_example**, you get the following output:

```
The following is a list of great astronomers:
       ARISTOTLE
         GALILEO
          NEWTON
           TYCHO
          KEPLER
```

**For** Repeatedly executes a statement a fixed number of times.

## FORMAT

**for** *index_variable* **:=** *start_exp* **to | downto** *stop_exp* **do**
*stmnt;* {**for** is a statement}

## ARGUMENTS

| | |
|---|---|
| *index_variable* | Any variable declared as an ordinal type. The ordinal types are enumerated, subrange, integer, Boolean, and char. Note that *index_variable* cannot be a real number. As an extension to standard Pascal, Domain Pascal permits the *index_variable* to be declared in a scope other than the scope of the routine immediately containing the **for** loop. |
| *start_exp* | An expression matching the type of the *index_variable*. |
| *stop_exp* | An expression matching the type of the *index_variable*. |
| *stmnt* | A simple statement or compound statement. (Refer to the "Statements" listing later in this encyclopedia.) |

## DESCRIPTION

**For, repeat,** and **while** are the three looping statements of Pascal. With **for**, you explicitly define both a starting and an ending value to the *index_variable*.

When executing a **for** loop, Pascal initializes the *index_variable* to the value of the *start_exp,* and then either increments (**to**) or decrements (**downto**) the value of the *index_variable* by 1 until its value equals that of the *stop_exp*. When the *index_variable* equals the value of the *stop_exp,* Pascal executes the statements in the loop one final time before exiting the loop. You may not assign a value to *index_variable* within the body of the **for** loop.

If **index_variable** is an integer or subrange variable, **for** increments or decrements its value by 1 for each cycle. If **index_variable** is a **char** variable, then **for** increments or decrements its ISO Latin-1 value by 1 for each cycle. If **index_variable** is an enumerated variable, then incrementing means selecting the next element in sequence and decrementing means selecting the preceding element. If **index_variable** is a Boolean, then true has a value greater than false.

The keyword **to** causes incrementing; the keyword **downto** causes decrementing.

If you want to jump out of a **for** loop prematurely (i.e., before the value of the *index_variable* equals the value of the *stop_exp*), you have the following choices:

- Execute an **exit** statement to transfer control to the first statement following the **for** loop.
- Execute a **goto** statement to transfer control to outside of the loop.
- Execute a **return** statement to transfer control back to the calling routine.

In addition to these measures, you can also execute a **next** statement to skip the remainder of the statements in the loop and proceed to the next iteration. Here are some tips for using the **for** statement:

- Within the *stmnt*, you are not allowed to change the value of the *index_variable*.
- If you set up a meaningless relationship between the *start_exp* and the *stop_exp* (for example, **for x := 8 to 5** or **for x := 10 downto 20**), Pascal does not execute the loop even once.

## EXAMPLE

```
PROGRAM for_example;
{This program demonstrates several uses of for loops}

 VAR
     time, year, zeta : integer16 := 0;
     hurricanes  : (king, donna, cleo, betsy, inez);
     scores : array[1..5, 1..3] of integer16;
     i, j : integer16;

 BEGIN

 {If you do not use a BEGIN/END pair, FOR assumes that the loop }
 {consists of the first statement following it. }
     FOR time := 1 TO 3 DO
         writeln(time);
```

```
{To create a loop consisting of multiple statements, enclose the }
{ loop in a BEGIN/END pair. }
    FOR time := 21 TO 30 DO
       begin
       year := year + time;
       writeln(year:5);  { Write a running total. }
       end;
{Here's an example of DOWNTO. }
    FOR time := year DOWNTO (year - 100) DO
       zeta := zeta + (time * 3);
    writeln;
    writeln(zeta,' is the result of the downto for loop');
    writeln;

{Here's an example of an enumerated index_variable. }
    FOR hurricanes := donna TO inez DO
       writeln(hurricanes);

{And finally, we use nested FOR loops to load a 2-dimensional array.}
    FOR i := 1 TO 5 DO
      begin      {for i}
       FOR j := 1 TO 3 DO
        begin  {for j}
        write('Enter the score for player ',i:1,' game ',j:1,' -- ');
        readln(scores[i,j]);
        end;   {for j}
       writeln;
      end;       {for i}
END.
```

## USING THIS EXAMPLE

This program is available online and is named **for_example**.

## Get Advances the stream marker to the next component of a file.

### FORMAT

**get**(*f*) {**get** is a procedure.}

### ARGUMENTS

| | |
|---|---|
| *f* | A variable having the **file** or **text** data type. |

### DESCRIPTION

If *f* is a **file** variable, calling **get** causes the operating system to advance the stream marker so that it points to the next record in the file. If *f* is a **text** variable, calling **get** causes the operating system to advance the stream marker so that it points to the next character in the file.

After calling **get** to advance the stream marker, you can use another statement to read in the data that the stream marker points to and assign it to a variable from your program. Therefore, the sequence for reading in data looks like the following:

```
GET(f);          { Advance the stream marker. }
 variable := f^; { Set variable equal to whatever the stream marker }
                 { points to.                                       }
```

For example, the following program fragment demonstrates input via the get procedure:

```
VAR
     primes   : file of integer16;
     poem     : text;
     a_number : integer16;
     a_letter : char;

 BEGIN
     . . .
     GET(primes);
     a_number := primes^; {Set a_number equal to next record in primes}
     . . .

     GET(poem);
     a_letter := poem^;   {Set a_letter equal to next character in poem}
     . . .
```

Note that the two statements

```
GET(poem);
 a_letter := poem^;  {Set a_letter equal to next character in poem  }
```

are identical to the single statement

```
READ(poem, a_letter);
```

Also notice that unlike **read**, **get** allows you to save the contents of *f*^.

You must open *f* for reading (with **reset**) before calling **get**. If **eof**(*f*) is true, calling **get**(*f*) causes a "read past end of file" error trap.

## EXAMPLE

```
PROGRAM get_example;

 { This program demonstrates the GET procedure.          }
 { File 'his101' must exist before you run get_example. }
 { To create 'his101', you must run put_example.         }

 %NOLIST;
 %INCLUDE '/sys/ins/base.ins.pas';
 %INCLUDE '/sys/ins/error.ins.pas';
 %INCLUDE '/sys/ins/streams.ins.pas';
 %LIST;

 CONST
     file_to_read_from = 'his101';

 TYPE
     student =
        record
             name   : array[1..12] of char;
             age    : integer16;
        end;

 VAR
     class            : file of student;
     a_student        : student;
     st               : status_$t;
```

```
BEGIN
{Open a file for reading.}
    open(class, file_to_read_from, 'OLD', st.all );
    if st.code = 0
      then reset(class)
      else if st.code = stream_$name_not_found
         then begin
                writeln('Did you forget to run put_example?');
                return;
              end
         else error_$print(st);

{Now that the file is open, read all the records from it. }
    repeat
       a_student := class^;
       GET(class);
       write(chr(10), a_student.name);
       writeln(a_student.age:2);
    until eof(class);
END.
```

## USING THIS EXAMPLE

This program is available online and is named **get_example**.

**Goto** Unconditionally jumps to a specified label in the program.

## FORMAT

**goto** *lbl*; {**goto** is a statement.}

## ARGUMENTS

| | |
|---|---|
| *lbl* | Is an unsigned integer or identifier that you have previously declared as a label. (For information on declaring labels, see the "Label Declaration Part" section in Chapter 2.) |

## DESCRIPTION

A **goto** statement breaks the normal sequence of program execution and transfers control to the statement immediately following *lbl*.

A declared *lbl* usually is local to the block in which it is declared. That is, if you know you declared a label, but the compiler still reports the following error, you must move your label declaration to the correct procedure or function:

```
(Name_of_Label) has not been declared in routine (name_of_routine)
```

It is illegal to use **goto** to jump inside a structured statement (for example, a **for**, **while**, **case**, **with**, or **repeat**) from outside that statement. This means a fragment like this produces an error:

```
if error_flag = true then
     goto cleanup;
          .
          .
          .
 for i := 1 to 10 do
     begin
          .  .  .
 cleanup:                    {WRONG!}
          .  .  .
     end;                    {close for statement}
```

It is illegal to jump into an **if/then/else** statement if you compile with the **-iso** option. See Chapter 6 for more details.

## Goto

**Gotos** are useful for handling exceptional conditions (such as an unexpected end of file).

Nonlocal **gotos**, whose target *lbl* is in the main program or some other routine at a higher level, have a great effect on the generated code. They generally shut off most compiler optimizations on the code near the target *lbl*. In order to produce the most efficient code, you should try to use **goto** as infrequently as possible.

You cannot jump into a structured statement from outside that statement. For example, the **goto 100** statements in the **bad_gotos** program below are illegal. This program also contains a **goto 900** statement that is illegal if you compile with the **–iso** option.

```
PROGRAM bad_gotos;
VAR
    z,x,value : integer16;
    a_char    : char;
LABEL 900;

PROCEDURE foo;
LABEL 100;

BEGIN
for x := 1 to 100 do
    begin
    writeln ('value ', x);
100: write ('Enter a value ');
    readln(value);
    z := z + value;
    end;
GOTO 100;                {ILLEGAL: cannot jump to a label inside}
                         {the for loop.                         }
END;

BEGIN
write ('Do you want to use the program? ');
readln (a_char);
if a_char = 'y' then
    GOTO 100              {ILLEGAL: cannot jump to a label in    }
                          {another routine.                      }
else if a_char = 'n' then
    GOTO 900              {ILLEGAL IF COMPILED WITH -ISO SWITCH:}
                          {cannot jump to a label that's inside }
                          {another statement.                   }
else if a_char = 'o' then
    writeln ('o is not a legal response')
else
    900: writeln ('ok, we won''t use the program');
END.
```

Note that you can use **goto** to jump directly from a nested routine to an outer routine. For example, procedure **xxx** issues a valid **goto** in the following program:

```
Program non_local_goto;
 Label
      900;


 Procedure xxx;
 BEGIN
      . . .
      GOTO 900;
      . . .
 END;


 BEGIN
      . . .
 900: writeln('back in main program.');
      . . .
 END.
```

**EXAMPLE**

```
PROGRAM goto_example;
{This program demonstrates the use of the goto statement}

 TYPE
     possible_values = 10..25;
 VAR
     x : possible_values;
 LABEL
     100;

 BEGIN
     writeln('You will now enter the experimental data.', chr(10));
 100:
     write('Please enter the obtained value for x -- ');
     readln(x);
     if in_range(x) then
         writeln('This value seems possible.')
     else
         begin
         writeln('This value seems suspicious.');
         GOTO 100;
         end;
 END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **goto_example:**

```
You will now enter the experimental data.

 Please enter the obtained value for x -- 35
 This value seems suspicious.
 Please enter the obtained value for x -- 17
 This value seems possible.
```

**If** Tests one or more conditions and executes one or more statements according to the outcome of the tests.

## FORMAT

You can use **if, then,** and **else** in the following two ways:

**if** *cond* **then** *stmnt;* {first form}

**if** *cond* **then** *stmnt1* **else** *stmnt2*; {second form}

## ARGUMENTS

| | |
|---|---|
| *cond* | Any Boolean expression. |
| *stmnt* | A simple statement or a compound statement. (Refer to the "Statements" listing in this encyclopedia.) Note that *stmnt* can itself be another **if** statement. |

## DESCRIPTION

The **if** and **case** statements are the two conditional branching statements of Pascal.

In an **if/then** statement, if *cond* evaluates to true, Pascal executes *stmnt*. If *cond* is false, Pascal executes the first statement following *stmnt*.

In an **if/then/else** statement, if *cond* is true, Pascal executes *stmnt1*. However, if *cond* is false, Pascal executes *stmnt2*.

You often use an **if** statement to evaluate multiple conditions. To do so, just remember that a *stmnt* can itself be an **if** statement. For example, consider the following **if** statement which evaluates multiple conditions:

```
IF age < 3 THEN
     price = 0.0
 ELSE IF (age >= 3) AND (age <= 6) THEN
     price = 1.00
 ELSE IF (age > 6)  AND (age <=12) THEN
     price = 2.00
 ELSE
     price = 4.00;
```

## EXAMPLE

```
PROGRAM if_example;

 {This program demonstrates IF/THEN and IF/THEN/ELSE.}

 VAR
     y, age, of_age, root_ratings  : integer16;
     tree    : (ficus, palm, poinciana, frangipani, jacaranda);
     grade   : char;

 BEGIN
 write('Enter an integer -- ');
 readln(y);                                              {USAGE 1}
 IF y < 0  THEN
     writeln('Its absolute value equals ', (abs(y)):3);

 write('Enter an age -- ');
 readln(age);                                            {USAGE 2}
 IF age > 18 THEN
     writeln('  An adult')
 ELSE
     begin
     of_age := 18 - age;
     writeln('  A minor for another ', of_age:1,' years.');
     end;

 write('Enter a grade -- ');
 readln(grade);                                          {USAGE 3}
 IF (grade = 'A') OR (grade = 'B') THEN
     writeln('  Good work')
 ELSE IF (grade = 'C') OR (grade = 'D') THEN
     begin
     writeln('  Satisfactory work');
     writeln('  Though improvement is indicated.');
     end
 ELSE IF (grade = 'F') THEN
     writeln('  Failing work');
```

```
write('Enter the name of a tropical tree -- ');
readln(tree);                                              {USAGE 4}
IF (tree = poinciana) OR (tree = jacaranda) THEN
    begin
    writeln('  Blossoms in June and July.');
    root_ratings := 9;
    end
ELSE IF tree = palm THEN
    root_ratings := 8
ELSE
    root_ratings := 2;
END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **if_example**:

```
Enter an integer -- -10
 Its absolute value equals  10
 Enter an age -- 13
   A minor for another 5 years.
 Enter a grade -- B
   Good work
 Enter the name of a tropical tree -- poinciana
   Blossoms in June and July.
```

**In** Evaluates an expression to see if it is a member of a specified set.

## FORMAT

*exp* **in** *setexp* {**in** is a set operator.}

## ARGUMENTS

| | |
|---|---|
| *setexp* | A set expression. |
| *exp* | An expression of the same data type as the elements constituting the base type of *setexp*. |

## OPERATOR RETURNS

The result of an **in** operation is always Boolean.

## DESCRIPTION

Use **in** to determine if **exp** is an element in set **setexp**. **In** returns either true or false.

## EXAMPLE

```
PROGRAM in_example;
 { This program prompts the user for a word, then counts the number of }
 {  ordinary vowels (a, e, i, o, and u) in the word. }

 VAR
     word              : array [1..20] of char := [* of ' '];
     count_of_vowels   : integer16 := 0;
     x                 : integer16;

 BEGIN
 write('Enter a word -- ');
 readln(word);
 for x := 1 to 20 do
    if word[x] IN ['a', 'e', 'i', 'o', 'u']
      then count_of_vowels := count_of_vowels + 1;
 writeln('This word contains ', count_of_vowels:1, ' ordinary vowels.');
 END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **in_example**:

```
Enter a word -- computers
 This word contains 3 ordinary vowels.
```

**In_range** Determines whether or not a specified value is within the defined integer subrange. (Extension)

## FORMAT

**in_range**(*x*) {**in_range** is a function.}

## ARGUMENTS

| | |
|---|---|
| *x* | A variable having a scalar (i.e., integer, Boolean, char, enumerated, or subrange) data type. For most practical purposes, *x* must be an enumerated or a subrange variable. |

## FUNCTION RETURNS

The **in_range** function returns a Boolean value.

## DESCRIPTION

The **in_range** function returns true if the value of *x* is within its defined range; otherwise, it returns false. The following program fragment demonstrates a possible use of **in_range**. We want to use **in_range** in this example, because it generates very efficient code:

```
TYPE
    small_int = -7..7;
 VAR
    x : integer16;
 BEGIN
    readln(x);
    if IN_RANGE(small_int(x))
       then ...
       else ...
```

## EXAMPLE

```
PROGRAM in_range_example;
{This program demonstrates the use of the in_range function }

 TYPE
    possible_temperature_range = 48..97;

 VAR
    air_temp    : possible_temperature_range;
    stop        : boolean;


 BEGIN
    repeat
         write('Enter the current air temp. (in deg. fahrenheit) -- ');
         readln(air_temp);
         if not IN_RANGE(air_temp) then
            begin
            writeln('This temperature is out of the historical range.');
            writeln;
            stop := false;
            end
         else begin
                 writeln('This value is within the historical range.');
                 stop := true;
               end;
    until stop;
 END.
```

## USING THIS EXAMPLE

Following is a sample execution of the program named **in_range_example** :

```
Enter the current air temp. (in deg. fahrenheit) -- 100
This temperature is out of the historical range.

Enter the current air temp. (in deg. fahrenheit) -- 47
This temperature is out of the historical range.

Enter the current air temp. (in deg. fahrenheit) -- 52
```

**Lastof** Returns the last possible value of a type or a variable. (Extension)

## FORMAT

**lastof**(*x*) {**lastof** is a function.}

## ARGUMENTS

| | |
|---|---|
| *x* | Either a variable or the name of a scalar data type. The data type can be a predeclared Domain Pascal data type, or it can be a user-defined data type. *x* cannot be a record, file, set, floating-point, or pointer type. |

## FUNCTION RETURNS

The **lastof** function returns a value having the same data type as *x*.

## DESCRIPTION

The **lastof** function returns the final possible value of *x* according to the following rules:

| Data Type of x | Lastof Returns |
|---|---|
| **integer** or **integer16** | 32767 |
| **integer32** | 2147483647 |
| **char** | A symbol indicating an unprintable character; however, **ord(lastof(char))** returns 255. |
| **boolean** | True. |
| **enumerated** | The last (rightmost) identifier in the data type declaration. |
| **array** | The upper bound of the subrange that defines the array's size. |
| **varying array** | The maximum length of the array. |

The **lastof** function is particularly useful for finding the last value in an enumerated type.

## EXAMPLE

See the example in the **firstof** listing earlier in this encyclopedia.

**Ln** Calculates the natural logarithm of a specified number.

## FORMAT

**ln**(*number*) {**ln** is a function.}

## ARGUMENTS

| | |
|---|---|
| *number* | Any real or integer expression that evaluates to a positive number. |

## FUNCTION RETURNS

The **ln** function always returns a real value (even if **number** is an integer).

## DESCRIPTION

The **ln** function returns the natural logarithm of **number**. Refer to the **exp** listing earlier in this encyclopedia for a practical definition involving **ln**.

## EXAMPLE

```
PROGRAM ln_example;
 { Each radioactive isotope has a unique K constant.}
 { This program uses LN and empirical data to calculate the k constant.}
 VAR
     starting_quantity, ending_quantity : real;
     elapsed_time, k : real;

 BEGIN
     write('Enter the quantity at time zero -- ');
     readln(starting_quantity);
     write('Enter the elapsed time (t) -- ');
     readln(elapsed_time);
     write('Enter the quantity at time (t) -- ');
     readln(ending_quantity);
     k := LN(starting_quantity/ending_quantity) * (1.0 / elapsed_time);
     writeln('The k constant for this radioactive element is ', k);
 END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **ln_example**:

```
Enter the quantity at time zero -- 1230
Enter the elapsed time (t) -- 47
Enter the quantity at time t -- 753
The k constant for this radioactive element is 0.010
```

**Lshft** Shifts the bits in an integer a specified number of bit positions to the left. (Extension)

## FORMAT

**lshft**(*num, sh*) {**lshft** is a function.}

## ARGUMENTS

| | |
|---|---|
| *num, sh* | Integer expressions. |

## FUNCTION RETURNS

The **lshft** function returns an integer value.

## DESCRIPTION

The **lshft** function shifts the bits in **num** to the left **sh** places. **Lshft** does not wrap bits around from the left edge to the right; instead, **lshft** shifts zeros in on the right. For example, consider the following results:

```
VAR
    i, n : INTEGER16;
BEGIN
    i := 2000;     { 2#0000011111010000 = 10#+2000  }
    LSFHT(i, 1);   { 2#0000111110100000 = 10#+4000  }
    LSHFT(i, 3);   { 2#0011111010000000 = 10#+16000 }
    LSHFT(i, 7);   { 2#1110100000000000 = 10#-6144  }
```

Results are unpredictable if **sh** is negative.

Compare **lshft** to **rshft** and **arshft**.

## EXAMPLE

```
PROGRAM lshft_example;
{This program demonstrates the use of the lshft function}
VAR
    unshifted_integer, shifted_integer, shift_left, x : integer16;
    choice     : 0 .. 15;
    drink_info : array[0..6] of integer16 := [* of 0];
```

```
BEGIN

 {In the following subroutine, LSHFT acts as a multiplier according }
 {to the equation LSHFT(num,sh) = num * (2 to the sh power).  Beware}
 { of overflow when you use LSHFT for this purpose.                 }

      unshifted_integer := 15; {15 is 0000000000001111 in binary}
      shift_left := 3;
      shifted_integer := LSHFT(unshifted_integer, shift_left);
      writeln(unshifted_integer:5, ' times 2 to the ', shift_left:1,
                 ' power = ', shifted_integer:1);

 {The result will be 120.}
 {120 is 0000000001111000 in binary.}


 {You can also use LSHFT to pack information more effectively.  For     }
 {example, suppose you asked 24 people to name their favorite soft      }
 {drink from a list of 16 possibilities.  Since we can represent 16     }
 {possibilities in 4 bits, we can store 4 people's responses in one     }
 {16-bit word in the following manner:                                  }
 {                                                                      }
 {Bit #   0          3   4          7   8          11  12          15   }
 {       --------------- ---------------- }
 {       | Response 1  | Response 2  | Response 3  | Response 4  | }
 {       --------------- ---------------- }
 {                                                                      }
 {Therefore, we will only need six 16-bit words to store the data       }
 {rather than 24 16-bit words.  The following code uses the LSHFT       }
 {function to accomplish this data reduction.                           }
      writeln;
      for x := 0 to 23 DO
         BEGIN
            write('Enter the preference (0-15) of client ',
                   x:1, ' -- ');
            readln(choice);
            drink_info[x div 4] := drink_info[x div 4] !
                                   LSHFT(choice, (4 * (x mod 4)));
            writeln(DRINK_INFO[x DIV 4]);
         END;
 {You can also achieve this sort of packing with packed records.        }

 END.
```

## USING THIS EXAMPLE

This program is available online and is named **lshft_example**.

**Max** Returns the larger of two expressions. (Extension)

## FORMAT

**max**(*exp1*,*exp2*) {**max** is a function.}

## ARGUMENTS

| | |
|---|---|
| *exp1, exp2* | Any valid expression. |

## DESCRIPTION

Domain Pascal's **max** function returns the larger of the two input expressions. The arguments *exp1* and *exp2* must be the same type or must be convertible to the same type by Pascal's default conversion rules (for example, integer converted to a real).

If *exp1* and *exp2* are unsigned scalars or pointers, Domain Pascal performs an unsigned comparison. (The unsigned scalar data types are non–negative subranges of integers, Boolean, character, and enumerated.) If they are **real**, **single**, or **double**, a floating–point comparison is done, while if they are signed integers, Domain Pascal performs a signed comparison.

See also **min**.

## EXAMPLE

```
PROGRAM max_example;
{This program demonstrates the use of the max function}
VAR
    small_num, big_num, biggest : REAL;
BEGIN
    big_num := 1000.0;
    small_num := 2.0;
    WHILE small_num <= big_num DO
        BEGIN
            small_num := SQR(small_num);
            big_num := SQRT(big_num);
            biggest := MAX(small_num, big_num);
            WRITELN ('The biggest number now equals ', biggest);
        END;
END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **max_example**:

```
The biggest number now equals  3.162278E+01
The biggest number now equals  1.600000E+01
```

**Min** Returns the smaller of two expressions. (Extension)

## FORMAT

**min**(*exp1*,*exp2*) {**min** is a function.}

## ARGUMENTS

| | |
|---|---|
| *exp1, exp2* | Any valid expression. |

## DESCRIPTION

Domain Pascal's **min** function returns the smaller of the two input operands. The arguments *exp1* and *exp2* must be the same type or must be convertible to the same type by Pascal's default conversion rules (for example, integer converted to a real).

If *exp1* and *exp2* are unsigned scalars or pointers, Domain Pascal performs an unsigned comparison. (The unsigned scalar data types are non-negative subranges of integers, Boolean, character, and enumerated.) If they are **real**, **single**, or **double**, a floating-point comparison is done, while if they are signed integers, Domain Pascal performs a signed comparison.

See also **max**.

## EXAMPLE

```
program min_example;

 var
     storenum : integer;
     lowprice, x, y, newprice1, newprice2 : real;
 begin
 { The program finds the lowest discounted price for an item that two }
 { stores sell. The stores have different regular prices and are      }
 { featuring different discount rates: 18% at the first, and 15% at   }
 { the second.                                                        }

 write ('Enter the regular price at the first store: ') ;
 readln (x);
 write ('Enter the regular price at the second store: ');
 readln (y);

 newprice1 := x - (x * 0.18);
 newprice2 := y - (y * 0.15);

 lowprice := MIN(newprice1,newprice2);
 if lowprice = newprice1 then
     storenum := 1
 else
     storenum := 2;

 write ('The best discounted price is at store number ', storenum:1);
 writeln (' and it is ', lowprice:6:2);
 end.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **min_example**:

```
Enter the regular price at the first store: 1599.99
Enter the regular price at the second store: 1515.15
The best discounted price is at store number 2 and it is 1287.88
```

**Mod** Calculates the remainder upon division of two integers.

## FORMAT

*d1* **mod** *d2* {**mod** is an operator.}

## ARGUMENTS

| | |
|---|---|
| *d1, d2* | Any integer expressions. |

## OPERATOR RETURNS

The **mod** operator returns an integer value.

## DESCRIPTION

Domain Pascal's **mod** operator works like standard Pascal's **mod** operator when *d1* is positive. When *d1* is negative and you compile without the **–iso** switch, Domain Pascal's **mod** operator works in a nonstandard manner.

### When d1 Is Positive

The expression **(d1 MOD d2)** produces the remainder of **d1** divided by **d2**. Therefore, the expression **(d1 MOD d2)** always evaluates to an integer from 0 up to, but not including, |d2|. For example, consider the following results:

```
 9 MOD  3 is equal to 0
10 MOD  3 is equal to 1
11 MOD  3 is equal to 2
12 MOD  3 is equal to 0
13 MOD  3 is equal to 1
13 MOD -3 is equal to 1
```

(To find the quotient (i.e., nonfractional) portion of the division, use the **div** operator described earlier in this encyclopedia.)

### When d1 Is Negative

If **d1** is negative, then **(d1 MOD d2)** equals

```
-1 * remainder of |d1| divided by |d2|
```

For example, consider the following results:

```
 -9 MOD -3 is equal to  0        -9 MOD 3 is equal to  0
-10 MOD -3 is equal to -1       -10 MOD 3 is equal to -1
-11 MOD -3 is equal to -2       -11 MOD 3 is equal to -2
-12 MOD -3 is equal to  0       -12 MOD 3 is equal to  0
-13 MOD -3 is equal to -1       -13 MOD 3 is equal to -1
```

### Compiling with the -iso Switch

If you compile with the **-iso** switch (described in Chapter 6), **mod** follows the standard Pascal rules. That is, **mod** returns a value **result** such that:

```
result := d1 - (d1 DIV d2) * d2
```

Since a negative modulus is illegal under standard Pascal rules, if **result** is negative, then:

```
result := result + d2
```

## EXAMPLE

```
PROGRAM mod_example;
 {This program uses the MOD function to find the coming leap years.}
CONST
     cycle = 4;
VAR
     remainder, year : integer16;
BEGIN
    for year := 1985 to 1999 do
       begin
           remainder := year MOD cycle;
           if remainder = 0
              then writeln(year:4, ' is a leap year');
       end;
 END.
```

## USING THIS EXAMPLE

If you execute the sample program named **mod_example**, you get the following output:

```
1988 is a leap year
1992 is a leap year
1996 is a leap year
```

## New Allocates space for storing a dynamic variable.

### FORMAT

| | |
|---|---|
| **new**(*p*) | {Short form. **New** is a procedure.} |
| **new**(*p, tag1, . . . tagN*) | {Long form.} |

### ARGUMENTS

| | |
|---|---|
| *tag* | An input argument that names one or more constants. *Tag* is valid only if *p*^ is a record. The maximum number of *tag*s is the number of tag fields in the record to which *p* points. |
| *p* | A pointer variable used for input and output. Pascal creates a dynamic variable of the type to which *p* points. After allocating this variable, Pascal returns the address of the newly allocated dynamic variable into p. The contents of the address pointed to is undefined. If there was insufficient address space or disk space remaining to satisfy the request for dynamic memory, then Domain Pascal returns the value **nil** in *p*. |

### DESCRIPTION

**New** causes Pascal to allocate enough space for storing one occurrence of a dynamic variable. You use **new** to create dynamic space and **dispose** (described earlier in this encyclopedia) to deallocate the dynamic space.

You can use the short form of **new** to allocate any kind of dynamic variable. The long form of **new** is only useful for allocating dynamic variant records.

### The Short Form

Consider the following record declaration:

```
TYPE
    employeepointer = ^employee;
    employee = record
       first_name : array[1..10] of char;
       last_name  : array[1..14] of char;
       next_emp   : employeepointer;
    end;
VAR
    current_employee : employeepointer;
```

If you want to store employee records dynamically, then you must call **NEW(current_employee)** for every occurrence of an employee. To allocate space for 100 employees, call **NEW(current_employee)** 100 times. You can assign values to an employee record only after Pascal has allocated space for an occurrence.

## The Long Form

Pascal uses **tag1**..*tagN* to help determine the amount of space to allocate for a variant record. **Tag1**..*tagN* corresponds to the tag fields of the variant record. For example, consider the type declaration for the following variant record:

```
TYPE
    emp_stat      = (exempt, nonexempt);
    workerpointer = ^worker;
    worker = record
       first_name : array[1..10] of char;
       last_name  : array[1..14] of char;
       next_emp   : workerpointer;
       CASE emp_stat OF
            exempt    : (salary  : integer16);
            nonexempt : (wages   : single;
                         plant   : array[1..20] of char);
    end;

 VAR
    current_worker : workerpointer;
```

Because **worker** contains a tag field, you have the option of passing the value of a constant to **new**; for example:

```
NEW(current_worker, exempt)
```

Since **tag1** is **exempt**, when Domain Pascal allocates space for one **worker** record, it allocates two bytes for the variant portion (since **integer16** takes up only two bytes). If **tag1** had been **nonexempt**, Domain Pascal would have allocated the space necessary (24 bytes) to hold it.

Note that the number of constants you pass to **new** must be less than or equal to the number of tag fields in the record declaration.

For machines with a larger address space (DNx60 machines, DN3000, etc.), you can access that larger amount of space by performing the following two steps:

1. Use **new** to allocate a large amount of memory (such as a megabyte) at the beginning of the program's execution.
2. Immediately after allocating the memory, use **dispose** to deallocate it.

These steps cause the operating system to increase the number of memory pages it allocates to your program. You should only use this technique if it is possible that your program may run out of address space.

## EXAMPLE

```
Program build_a_linked_list;

{The following example uses NEW and DISPOSE to create and disassemble}
{a linked list.  For a description of the theory of linked lists,    }
{consult a Pascal tutorial.                                           }

TYPE
   studentpointer = ^ student;
   student = record
      name : array[1..30] of char;
      age  : integer16;
      next_student : studentpointer;
   end;
VAR
   base     : studentpointer;
   a_name   : array[1..30] of char;
   an_age   : integer16;
   option   : char;
   done     : boolean;

Procedure error_message; {gives a message if student not on list}
begin
writeln;
writeln('That person is not on the list.  Let''s try again.');
writeln
end;
```

```
Procedure print_list; { Print the linked list in order. }
VAR
   ns : studentpointer;
BEGIN
    ns := base;
    writeln;
    while ns <> NIL do
      with NS^ do
         begin
            writeln(name, ' - ', age);
            ns := next_student;
         end;
END;


Procedure Enter_data;
VAR
   ns, previous : studentpointer;
BEGIN
   base := nil;
   repeat
       write('Enter the name of a student (or end to stop) -- ');
       readln(a_name);
       if a_name <> 'end' then
          begin
             NEW(ns);
                  { allocate space for a new occurrence of a student.}
             write('Enter his or her age -- ');  readln(an_age);
             if base = nil
                then base := ns           {set base to first record. }
                else previous^.next_student := ns;
                                          {add record to end of list.}
             ns^.name := a_name;  { Initialize fields of new record. }
             ns^.age  := an_age;
             ns^.next_student := nil;
             previous := ns;     { Save pointer to this new student }
          end
   until a_name = 'end';
END;
```

```
Procedure delete_a_student;
VAR
   ns, previous : studentpointer;
BEGIN
   previous  := base;
   ns := base;
   write('What is the name of the student you want to delete? -- ');
   readln(a_name);

   while ns <> nil do
      begin
         if ns ^.name = a_name then {delete record}
            begin
               if ns = base then
                  base := ns^.next_student
               else
                  previous^.next_student := ns^.next_student;
               DISPOSE(ns);
               exit;
            end
         else
            begin
               previous := ns;
               ns := ns^.next_student;
               if ns = nil then error_message;
            end; {if}
      end; {while}
END; {delete_a_student}

BEGIN {main}
   base := nil;
   enter_data;
   print_list;

   repeat
   if base = nil then return;
   write('Do you want to delete a student from the list? (y or n)--');
   readln(option);
   done := not (option in ['y', 'Y']);
   if not done then
      begin
         delete_a_student;
         print_list;
      end
   until done;
END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **build_a_linked_list**:

```
Enter the name of a student (or end to stop) -- Kerry
Enter his or her age -- 28
Enter the name of a student (or end to stop) -- Jan
Enter his or her age -- 27
Enter the name of a student (or end to stop) -- Lance
Enter his or her age -- 29
Enter the name of a student (or end to stop) -- end

Kerry                        -        28
Jan                          -        27
Lance                        -        29
Do you want to delete a student from the list? (y or n) -- y
What is the name of the student you want to delete? -- Jan

Kerry                        -        28
Lance                        -        29
Do you want to delete a student from the list? (y or n) -- n
```

**Next** Jump to the next iteration of a **for, while,** or **repeat** loop. (Extension)

## FORMAT

**Next** is a statement that neither takes arguments nor returns values.

## DESCRIPTION

You use **next** to skip over the *current iteration* of a loop. You can only use **next** within a **for, while,** or **repeat** loop. If **next** appears elsewhere in a program, Domain Pascal issues an error. **Next** tells Domain Pascal to ignore the remainder of the statements within the body of the loop for one iteration. For instance, consider the following example:

```
FOR x := 5 to 35 do
    begin
         . . .
         if (x MOD 10) = 0 then NEXT;
         . . .
    end;
```

When **x** is 10, 20, or 30, Domain Pascal ignores the statements following the **next**. You can use a **goto** statement instead of a **next** statement. For example, you can rewrite the preceding example to look like the following:

```
FOR x := 5 to 35 do
    begin
         . . .
         if (x MOD 10) = 0 then GOTO 100;
         . . .
 100:    end;
```

## EXAMPLE

```
PROGRAM next_example;
 { This program counts the occurrences of the digits 0 through 9 in }
 {  a line of integers and real numbers.                             }

 CONST
    blank = ' ';
    decimal_point = '.';
 VAR
    dig : char;
    count_digits : array[48..57] of integer16 := [* of 0];
    x  : integer16;
 BEGIN
  writeln('Enter a line of integers and real numbers:');
  repeat
     read(dig);
     if ((dig = blank) or (dig = decimal_point)) then NEXT;
     write(dig, ' ');
     count_digits[ord(dig)] := count_digits[ord(dig)] + 1;
  until eoln;
  writeln;
  for x := 48 to 57 do
  writeln( count_digits[x]:2, ' of the digits were ',  chr(x):1, 's');
 END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **next_example**:

```
Enter a line of integers and/or real numbers.
 1741374.13 33 821
 1 7 4 1 3 7 4 1 3 3 8
  0 of the digits were 0s
  4 of the digits were 1s
  1 of the digits were 2s
  4 of the digits were 3s
  3 of the digits were 4s
  0 of the digits were 5s
  0 of the digits were 6s
  2 of the digits were 7s
  1 of the digits were 8s
  0 of the digits were 9s
```

**Nil** A special pointer value that points to nothing.

## FORMAT

**Nil** is a predeclared constant, except when the code has been compiled with the **-iso** option. When the **-iso** option has been used, **nil** is a reserved word. You can only use **nil** in expressions. **Nil** is a pointer value; therefore, you must assign it to or compare it to a pointer variable. **Nil** never points to an object.

## DESCRIPTION

Use **nil** when you must assign a value to a pointer, but you don't know what that value should be. For example, when creating a linked list, you can set the last record in the list to point to **nil**. Then, when walking through the list, you can easily find the end of the list by checking for **nil**.

## EXAMPLE

For a sample program that uses **nil**, refer to the listing for **new** earlier in this chapter.

**Not** Returns true if an expression evaluates to false.

## FORMAT

**not** *b* {**not** is a unary operator.}

## ARGUMENTS

| | |
|---|---|
| *b* | Any Boolean expression. |

## OPERATOR RETURNS

The result of a **not** operation is always a Boolean value.

## DESCRIPTION

If *b* evaluates to false, then **not** *b* evaluates to true. If *b* evaluates to true, then **not** *b* evaluates to false.

Note that you can put **and** or **or** immediately before **not**. Note the order of precedence in an expression like the following:

**a AND NOT b** actually means **a AND (NOT b)**

Another potentially confusing expression is the following:

**NOT a AND b** actually means **(NOT a) AND b**

Please refer to the order of precedence rules in Table 4–3.

## EXAMPLE

```
PROGRAM not_example;
{This program demonstrates the use of the not operator}
 VAR
    pet_lover, timid : boolean;

 BEGIN
    writeln('Career aptitude test.', chr(10));
    write('You like pets (true or false) -- '); readln(pet_lover);
    write('You are timid (true or false) -- '); readln(timid);

    if pet_lover and NOT timid
       then writeln('Have you considered becoming a lion tamer?')
       else writeln('Plastics...there''s a great future in plastics.');
 END.
```

## USING THIS EXAMPLE

This program is available online and is named **not_example**.

## Odd

**Odd** Tests whether the specified integer is an odd number.

### FORMAT

**odd**(*i*) {**odd** is a function.}

### ARGUMENT

| | |
|---|---|
| *i* | Any integer expression. |

### FUNCTION RETURNS

The **odd** function returns a Boolean value.

### DESCRIPTION

**Odd** returns true if *i* is an odd integer and false if *i* is an even integer.

### EXAMPLE

```
PROGRAM odd_example;
{This program demonstrates the use of the odd function}

 VAR
    i : integer;
 BEGIN
    write('Enter an integer -- ');
    readln(i);
    if ODD(i)
       then writeln(i:1, ' is an odd number.')
       else writeln(i:1, ' is an even number.');
 END.
```

### USING THIS EXAMPLE

Following is a sample run of the program named **odd_example**:

```
Enter an integer -- 14
 14 is an even number.
```

**Of** Refer to **Case** earlier in this encyclopedia.

---

**Open** Opens a file so that you can eventually read from or write to it. (Extension)

---

## FORMAT

**open**(*file_variable, pathname, file_history, error_status, buffer_size*); {**open** is a procedure.}

## ARGUMENTS

*file_variable* A variable having the **text** or **file** data type.

*pathname* The name of the file that you want to open. *Pathname* is a string constant or string variable, that you specify in any of the following five ways:

- Enter a Domain pathname as defined in *Getting Started with Domain/OS*.

- Enter a string in the form '^*n*', where *n* is an integer from 1 to 9. *n* corresponds to the ordinal value of the arguments that the user passes to the program when he or she executes or debugs the program. For example, suppose you compile Domain Pascal source code to create executable object file **sample.bin**. You can pass the two arguments xxx and yyy by executing the program as follows:

  ```
  $ sample.bin xxx yyy
  ```

  The preceding command line causes Domain Pascal to assign xxx to '^1' and yyy to '^2'.

- Enter a string in the form '*prompt-string'. At run time, Domain Pascal prints the prompt-string at standard output, and then reads the user's response from standard input. (The response should be the name of the file to be opened.) The prompt-string can contain any printable character except blanks; Domain Pascal stops printing at the first blank it encounters. An asterisk by itself tells Domain Pascal to read the response from standard input without printing a prompt at standard output.

- A string in the form '-STDIN' or '-STDOUT'. These strings correspond to the streams that the operating system opens automatically. However, specifying one of these strings does not cause an error. (See Chapter 8 for an explanation of streams.)

- A variable or constant containing any of the preceding items.

*file_history* A variable or string that tells the **open** procedure how to open the file. The variable or string must have one of the following three values:

- 'NEW' — If the file exists, Domain Pascal reports an error. If the file does not exist, Domain Pascal creates the file and then opens it.
- 'OLD' — If the file exists, Domain Pascal opens it. If the file does not exist, Domain Pascal reports an error.
- 'UNKNOWN' — If the file exists, Domain Pascal opens it. If the file does not exist, Domain Pascal creates the file and then opens it.

Remember to enclose the *file_history* within single quotes (for example, 'NEW').

*error_status* An optional argument. If you specify an error_status, it must have an **integer32** data type. At run time, Domain Pascal returns a hexadecimal number into *error_status* which has the following meaning:

```
0 - no error or warning occurred.

greater than 0 - an error occurred.

less than 0 - a warning occurred.
```

Your program is responsible for handling any errors. We detail error handling in Chapter 9.

*buffer_size* An optional argument that may only be specified for files of type **text.** *Buffer_size* must be at least as long as the longest line in the file being read; if it is shorter, the excess characters in a line are truncated. If the file is open for writing only, you don't need to specify a large *buffer_size*. No data is lost even if a line being written is longer than *buffer_size*. The default size is 256 bytes.

## DESCRIPTION

Before you can read from or write to a file, you must first open it for I/O operations. To open a permanent file, you must use the **open** procedure. To open a temporary file, use the **rewrite** procedure without using an **open** procedure.

After you've opened a file, you then specify whether it is available for reading (by calling **reset**) or for writing (by calling **rewrite**). Note that you do not need to open the standard input (**input**) and standard output (**output**) files before attempting to read from or write to them. They are always open.

When your program terminates, the operating system automatically closes all opened files; however, please refer to the description of the **close** procedure earlier in this chapter.

For a complete overview of Domain I/O, see Chapter 8.

## EXAMPLE

```
Program open_example;

{NOTE: Before running this program, you must obtain file "annabel_lee" }
{      and store it in the same directory as the program.               }
{ This program uses a variety of techniques to open three files.        }
{ In order for it to work properly, you must pass the pathname of a     }
{ file as the first argument on the execution or debug command line.    }
{ For example, if you compile this program to create open_example.bin,  }
{ then you could invoke the program with the following command:         }
{     $ open_example.bin //arnie/nouveau/comps                          }

%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%LIST;

CONST
    name_of_file = 'annabel_lee';
    file3        = '*enter_a_filename--';
VAR

    poem, paragraph, stanza  : text;
    statrec                  : status_$t;

BEGIN
{ Open an existing file.}
     OPEN(poem, name_of_file, 'OLD', statrec.all);

{ If there was no error on open, then specify that the file be     }
{ open for reading.                                                }
     if statrec.all = 0
        then begin
               writeln('Opened ', name_of_file, ' for reading');
               reset(poem)
             end
        else writeln('Difficulty opening ', name_of_file);

{ Open a new file.  The pathname of the new file will be the first }
{ argument that you pass on the execution or debug command line.   }
    OPEN(paragraph, '^1', 'NEW', statrec.all);
```

```
{ If there was no error on open, then specify that the file be open}
{ for writing.  If there was an error, print the error code.       }
    if statrec.all = status_$ok
        then rewrite(paragraph)
        else begin
               writeln(chr(10), 'Did you remember to pass an argument');
               writeln('to the command line?');
               writeln('error code = ', statrec.all, chr(10));
             end;

{ Open a file that may or may not exist. Prompt user for name of       }
{ file at runtime.                                                     }
    OPEN(stanza, file3, 'UNKNOWN', statrec.all);

{ A slightly more sophisticated method of error reporting is to use    }
{ the system call ERROR_$PRINT to print the error message.             }
{ NOTE: In order to call ERROR_$PRINT, you must specify both           }
{   /sys/ins/base.ins.pas and /sys/ins/error.ins.pas as %INCLUDE files.}
    if statrec.all = status_$ok
       then rewrite(stanza)
       else ERROR_$PRINT(statrec);

END.
```

## USING THIS EXAMPLE

This program is available online and is named **open_example**.

**Or, Or Else** Calculate the logical **or** of two Boolean arguments.

## FORMAT

| | |
|---|---|
| *x* **or** *y* | {**or** is an operator.} |
| *x* **or else** *y* | {**or else** is an operator.} |

## ARGUMENTS

| | |
|---|---|
| *x, y* | Any Boolean expressions. |

## OPERATOR RETURNS

The result of an **or** or an **or else** operation is always a Boolean value.

## DESCRIPTION

Use **or** to find the logical **or** of expressions *x* and *y*. Here is the truth table for **or**:

*Table 4-12. Truth Table for Logical* **or** *Operator*

| x | y | Result |
|---|---|---|
| true | true | true |
| true | false | true |
| false | true | true |
| false | false | false |

Compare **or** to **and** and **not** (which are described elsewhere in this encyclopedia).

**NOTE:** Some programmers confuse **or** with the exclamation point (!) operator. ! is a bit operator; it causes Domain Pascal to perform a logical **or** on all the bits in its two arguments. For example, compare the following results:

```
(true OR false) is equal to true
(75 ! 15) is equal to 79
```

Refer to the "Bit Operators" listing earlier in this encyclopedia.

The Boolean operator **or else** is a Domain extension to standard Pascal. You can use **or else** in any statement where you use **or** in standard Pascal. The choice between **or** and **or else**, however, affects the run-time evaluation of a statement.

When **or else** appears between two Boolean operands, the system begins evaluating the operands in the order in which they appear. If the first operand is true, the system does not evaluate the second. If one operand is true, then the entire expression is true.

Hence, **or else** guarantees "short-circuit" evaluation. That is, at run time, the system evaluates an operand only if necessary.

For example, in the statement

```
IF boolean_1 OR ELSE boolean_2
    THEN ...
```

the system first evaluates **boolean_1**. If **boolean_1** is true, the system does not evaluate **boolean_2**. In this statement, **boolean_1** and **boolean_2** can be any valid Pascal Boolean expressions. The operator **or else** can be more efficient than **or**. For example, in the statement

```
IF boolean_1 OR boolean_2
    THEN ...
```

the system may have to evaluate both **boolean_1** and **boolean_2** to test if the statement is true. Also, there is no guarantee that the system will evaluate the two expressions in the order in which they appear.

The **or else** operator helps you avoid nested constructions. For example, compare the standard Pascal code on the left with the equivalent Domain Pascal code on the right:

**Standard Pascal**

```
IF c1 THEN
    s1
ELSE
    IF c2 THEN
        s1:
```

**Domain Pascal**

```
IF c1 OR ELSE c2 THEN
    s1;
```

In this example, the standard Pascal code contains an **if/then/else** statement with another **if** statement nested inside it. The Domain Pascal code, however, contains only one **if** statement.

The following example illustrates how to avoid taking the log of 0.0:

```
REPEAT
    i := i + 1;
    f := a[i];
UNTIL f = 0.0 OR ELSE ln(f) < 2.0;
```

## EXAMPLE

```
PROGRAM or_example;
{This program demonstrates the use of the or operator}

 VAR
    tall, good_jumper, good_athlete  : boolean;

 BEGIN
  writeln('Career aptitude test.', chr(10));
  write('You are taller than 1.95 meters (true or false) -- ');
  readln(tall);
  write('You can jump high (true or false) -- ');
  readln(good_jumper);
  write('You are athletic (true or false) -- ');
  readln(good_athlete);

  if (tall OR good_jumper) AND good_athlete
   then writeln(chr(10),'Have you considered playing pro basketball.')
   else writeln(chr(10), 'Computers are a stable field.');
 END.
```

## USING THIS EXAMPLE

This program is available online and is named **or_example**.

**Ord** Returns the ordinal value of a specified integer, Boolean, char, or enumerated expression.

## FORMAT

**ord**(*x*) {**ord** is a function.}

## ARGUMENT

| | |
|---|---|
| *x* | Any scalar (i.e., integer, Boolean, char, or enumerated) expression. |

## FUNCTION RETURNS

The **ord** function returns an integer value.

## DESCRIPTION

The **ord** function returns the ordinal value of *x* according to the following rules:

| Data Type of x | Ord Returns |
|---|---|
| **Integer** | Numerical value of *x*. |
| **Boolean** | 0 if *x* is false and 1 if *x* is true. |
| **Char** | *x*'s ASCII value. Appendix B contains an ISO Latin–1 table that includes ASCII values. |
| **Enumerated** | An integer representing *x*'s position within the enumeration declaration. For instance, in program **ord_example**, **ORD(rice)** returns 0, **ORD(tofu)** returns 1, and so on up until **ORD(tamari)**, which returns 4. |

Note that the **chr** function is the inverse of **ord**.

## EXAMPLE

```
PROGRAM ord_example;
{This program demonstrates the use of the ord function}

 TYPE
    macro = (rice, tofu, seaweed, miso, tamari);

 VAR
    c : char;
    e : macro;

 BEGIN
    c := ´d´;
    WRITELN(´The ordinal value of ´, c, ´ is ´, ORD(c):3);

    e := seaweed;
    WRITELN(´The ordinal value of ´, e:7, ´ is ´, ORD(e):1);
 END.
```

## USING THIS EXAMPLE

If you execute the sample program named **ord_example**, you get the following output:

```
The ordinal value of d is 100
The ordinal value of SEAWEED is 2
```

**Pack** Copies an unpacked array to a packed array.

## FORMAT

**pack**(*unpacked_array, index, packed_array*) {**pack** is a procedure.}

## ARGUMENTS

| | |
|---|---|
| *unpacked_array* | An array that has been defined without the keyword **packed**. |
| *index* | A variable that is the same type as the array bounds (integer, boolean, char, or enumerated) of *unpacked_array*. *Index* designates the array element in *unpacked_array* from which **pack** should begin copying. |
| *packed_array* | An array that has been defined using the keyword **packed**. |

## DESCRIPTION

**Pack** copies an unpacked array to a packed one. However, data access with unpacked arrays generally is faster since data elements are always aligned on word boundaries.

*Unpacked_array* and *packed_array* must be of the same type, and for every element in *packed_array*, there must be an element in *unpacked_array*. That is, if you have the following **type** definitions

```
TYPE
     x : array[i..j] of single;
     y : packed array[m..n] of single;
```

the subscripts must meet these requirements:

```
j - index >= n - m          {"index" as set in the call to pack}
```

For example, it is legal to use **pack** on two arrays defined like this:

```
TYPE
    big_array   : array[1..100] of integer;
    small_array : packed array[1..10] of integer;
 VAR
    grande : big_array;
    petite : small_array;
```

You use *index* to indicate the array element in *unpacked_array* from which **pack** should begin copying. For instance, given the previous variable declarations and assuming variable **i** is an **integer**, this fragment

```
i := 1;
 pack(grande, i, petite);
```

tells **pack** to begin copying at **grande[1]**. **Pack** keeps copying until it reaches the highest index value that **petite** can take—which in this case is 10. The remaining elements in **grande** are not copied.

*Index* can take a value outside of *packed_array*'s defined subscripts. That is, if in the example above, **i** equals 50, **pack** copies these values this way:

```
petite[1] := grande[50];
 petite[2] := grande[51];
          .
          .
          .
 petite[10] := grande[59];
```

See the listing for **unpack** later in this encyclopedia.

## EXAMPLE

```
PROGRAM pack_example;
 {This program demonstrates the pack procedure}

 TYPE
     uarray = array[1..50] of integer16;
     parray = packed array[1..10] of integer16;
 VAR
     full_range : uarray;
     sub_range  : parray;
     i, j       : integer16;

 BEGIN
 for i := 1 to 50 do
     full_range[i] := i;
 j := 20;
 PACK(full_range, j, sub_range);
 writeln ('The packed array now contains: ');
 for i := 1 to 10 do
     writeln ('sub_range[', i:2, '] = ', sub_range[i]:2);

 END.
```

## USING THIS EXAMPLE

If you execute the sample program named **pack_example**, you get the following output:

```
The packed array now contains:
sub_range[ 1] = 20
sub_range[ 2] = 21
sub_range[ 3] = 22
sub_range[ 4] = 23
sub_range[ 5] = 24
sub_range[ 6] = 25
sub_range[ 7] = 26
sub_range[ 8] = 27
sub_range[ 9] = 28
sub_range[10] = 29
```

**Page** Inserts a formfeed (page advance) into a file.

## FORMAT

**page**(*f*) {**page** is a procedure.}

## ARGUMENT

| | |
|---|---|
| *f* | A **text** variable. *f* is optional. If you do not specify *f*, **page** assumes that the file is standard output (**output**). |

## DESCRIPTION

Use the **page** procedure to insert a formfeed (ISO Latin-1 character 12) into the file specified by *f*. **Page** is useful for formatting text that will be printed or for text that meets fixed-length window dimensions. If you print the file on a line printer, the printer advances to the next page when it encounters the formfeed.

Before calling **page**, you must open the file named in *f* for writing. See Chapter 8 for a description of opening files.

## EXAMPLE

```
PROGRAM page_example;
 {This program demonstrates the PAGE procedure.}

 CONST
     lines_in_a_page  =  54;  {Our printer prints 54 lines to a page.}


 VAR
     information    : text;
     statint        : integer32;
     x              : integer16;

 BEGIN

 { Create a file and open it for writing; exit on error. }
     open(information, 'square_root_table', 'NEW', statint);
     if statint = 0 then
         rewrite(information)
     else
         begin
         writeln('Pascal reports error', statint, ' on OPEN.');
         return;
         end;

 {Print the square roots from 1 to 200, inserting }
 {formfeeds where needed.                         }

    for x := 1 to 200 do
    begin
    writeln(information, 'The square root of ', x:3, ' is ', sqrt(x));
    if ((x mod lines_in_a_page) = 0) then
            PAGE(information);
    end;

 END.
```

## USING THIS EXAMPLE

This program is available online and is named **page_example**.

## Pointer Operations

Chapter 3 explains how to declare pointer types. Here, we describe how to use pointers in the action part of your program.

### DESCRIPTION

You can do the following things with a pointer variable:

- Use the **addr** function to assign the virtual address of a variable to the pointer variable.
- Compare or assign the value of one pointer variable to another compatible pointer variable.
- De-reference a pointer variable. De-referencing means that you find the contents of the variable to which the pointer variable was pointing.

The following program fragment does all three things:

```
Program test;

 TYPE
    pi = ^integer16;

 VAR
    p1quart, p2quart : pi;
    quart1,   quart2 : integer16 := 5;

 BEGIN
    p1quart := addr(quart1);
    p2quart := p1quart;
    quart2  := p2quart^;
 END.
```

#### Manipulating Virtual Addresses—Extension

Domain Pascal supports type transfer functions that are quite useful in manipulating virtual addresses. For example, you cannot directly write a pointer value to a **text** file; however, you can use a type transfer function to transfer the address to an **integer32** value (which can be written).

For example, compare the right and wrong ways to write the virtual address of **quart1** to **output**:

```
writeln(p1quart);             {wrong}
writeln(integer32(p1quart));  {right}
```

### Invoking Procedure and Function Pointers—Extension

You de-reference a procedure or function pointer like any other pointer; that is, with the caret (^). In this way, functions can return pointers to other functions; for example:

```
TYPE
     retbool = ^function : boolean;
     retfunptr = ^function : retbool;

 VAR
     xp : retbool;
     rf : retfunptr;
     flag : boolean;

 FUNCTION myfunc; retbool;
         .
         .
         .

     rf := ADDR(myfunc);
     xp := rf^;
     flag := xp^;
```

The expression **rf^** invokes the **myfunc** function, which returns a pointer to a function that returns a Boolean value. You cannot use the following assignment

```
flag := rf^^;
```

because you cannot de-reference the return value of a function call.

### Addressing Procedure and Function Pointers—Extension

To obtain procedure and function addresses, use the predeclared function **addr**. Thus, there is no ambiguity about a function reference, especially one with no parameters. It is either invoked by name only, or its address is taken by the **addr** function.

Although the **addr** function has been declared to return a **univ_ptr**, the compiler adds extra type checking whenever you try to pass a procedure or function address to a specific procedure or function pointer. In the assignment **pptr := addr(proc2)** from the following program fragment, the declaration for **proc2** must exactly match the template for the procedure type of **pptr**. If not, the compiler reports an error. If, however, the assignment is to a **univ_ptr**, like **xxx := addr(func1)**, the compiler cannot do this extra type checking.

```
VAR
     xxx : UNIV_PTR;
    pptr : ^Procedure(IN i, j : integer;
                      OUT   a : char;
                      VAR   r : real); EXTERN;
 BEGIN
    . . .
    xxx := func1;          {This is a call.}
    . . .
    xxx := addr(func1);    {This takes the address.}
    . . .
    pptr := addr(proc2);   {This takes the address and checks.}
```

**Pred** Returns the predecessor of a specified ordinal value.

## FORMAT

**pred**(*x*) {**pred** is a function.}

## ARGUMENT

| | |
|---|---|
| *x* | An integer, Boolean, char, or enumerated expression. |

## FUNCTION RETURNS

The **pred** function returns a value having the same data type as *x*.

## DESCRIPTION

**Pred** returns the predecessor of *x* according to the following rules:

| Data Type of x | Pred Returns |
|---|---|
| **Integer** | The numerical value equal to *x–1*. |
| **Boolean** | False—even if *x* already equals false. |
| **Char** | The character with the ISO Latin–1 value one less than the ISO Latin–1 value of *x*. If this character (*x–1*) does not exist, Domain Pascal cannot detect the error. |
| **Enumerated** | The identifier to the left of *x* in the type declaration. If *x* is the leftmost identifier, **pred**'s return value is undefined. |

**pred(firstof**(*x*)) generally is undefined; however, Domain Pascal does not report an error. Domain Pascal also doesn't report an error if you specify an integer value that is outside the range of the specified integer type. Therefore, your program should test for an out–of–bounds condition.

Compare the **pred** function to the **succ** function.

## EXAMPLE

```
PROGRAM pred_example;
 TYPE
    jours = (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche);

 VAR
    i        : integer;
    c1, c2   : char;
    semaine  : jours;

 BEGIN
    i := 53;  c1 := 'n';  semaine := vendredi;
    writeln('The predecessor of ', i:2, ' is ', pred(i):2);
    c2 := pred(c1);
    writeln('The predecessor of ', c1:1, ' is ', c2);
    writeln('The predecessor of ', semaine:8, ' is ', pred(semaine):8);
 END.
```

## USING THIS EXAMPLE

If you execute the sample program named **pred_example**, you get the following output:

```
The predecessor of 53 is 52
The predecessor of n is m
The predecessor of VENDREDI is    JEUDI
```

**Ptoc** Appends a null byte to a variable-length string. (Extension)

## FORMAT

**ptoc**(*string*); {**ptoc** is a procedure.}

## ARGUMENTS

*string* A variable-length string.

## DESCRIPTION

**ptoc** appends a null byte to the body of a variable-length string. The address of the string's body can then be passed to routines that expect a C-style, null-terminated string. The length field is not changed, so that the null character will not be visible in Pascal expressions that access the array.

When variable-length strings are allocated by the Pascal compiler, they are padded with at least one byte in order to guarantee that **ptoc** will not write beyond the bounds of the string even if its current run-time size is the maximum size of the string.

See the description of **ctop** for information about converting a null-terminated string into a variable-length string.

## EXAMPLE

In the following example, the Pascal program calls a C function called **strip_extra_spaces** that converts substrings of two or more space characters into a single space character. Note that we pass the body field rather than the variable-length string because the C function expects a pointer to a char, not a pointer to a structure.

```
PROGRAM ptoc_and_ctop_example;
{This program demonstrates the use of the ptoc and ctop }
{procedures.  It calls an external C module, which is   }
{named strip_extra_spaces.  You must compile the C      }
{module and bind it with ptoc_and_ctop_example.bin to   }
{create an  executable program.                         }
TYPE
     str = ARRAY[1..100] of CHAR;
VAR
     var_string  :   VARYING[100] of CHAR;
```

```
PROCEDURE strip_extra_spaces (IN OUT s : str); OPTIONS(EXTERN);

BEGIN
  var_string := 'This     string   has     too many  spaces';
  strip_extra_spaces(var_string.body);

 { The length has not been changed, so the following writeln }
 { procedure outputs extra characters.                       }
     writeln(var_string);

 { Calling CTOP changes the current length by searching for   }
 { the terminating null. }
     CTOP(var_string);
     writeln(var_string);
     var_string := 'This     string   has     too many  spaces';
     var_string.length := 30;  { Change the length explicitly }

 { PTOC places a null char at the 31st position. }
     PTOC(var_string);
     strip_extra_spaces(var_string.body);
     CTOP(var_string);
     writeln(var_string);
END.

/*This module is called by the Pascal program named */
/*ptoc_and_ctop_example.                            */
#define SPACE 32
#define NUL 0
 void strip_extra_spaces( s )
 char *s;
 {
   char *start = s;
   while (*s)
   {
    if ((*start++ = *s++) == SPACE)
       while (*s == SPACE)
         s++;
   }
   *start = NUL;
 }
```

## USING THIS EXAMPLE

Executing this program, named **ptoc_and_ctop_example**, produces the following output:

```
This string has too many spacesny  spaces
This string has too many spaces
This string has too
```

The first **writeln** procedure outputs the string before we have changed the current length. Even though the string was shortened by **strip_extra_spaces**, Pascal has no knowledge of the new length, so it outputs the number of characters indicated by the old length.

Before the second **writeln** procedure, we call **ctop**, which adjusts the current length by searching for the null character implanted by the C function. This is the proper usage of **ctop**.

The last **writeln** call illustrates what happens when you explicitly change the length field. In this case, we change the length to 30, then we call **ptoc** which inserts a null character at the 31st position. This makes the new length known to the C function. The **strip_extra_spaces** function, therefore, looks only at the first 30 characters. Note that we need to call **ctop** again to change the length field after the C function has shortened the string.

**Put** Writes to a file.

## FORMAT

**put**(*f*) {**put** is a procedure.}

## ARGUMENT

| | |
|---|---|
| *f* | A variable having the **file** or **text** data type. |

## DESCRIPTION

If *f* is a **file** variable, then **put(f)** appends one record to the file symbolized by *f*. If *f* is a **text** variable, then **put(f)** appends one character to the file symbolized by *f*.

Before calling **put(f)**, you must assign a record or character to **f^**. So the sequence for writing out data looks like the following:

```
f^ := record_or_character;
 PUT(f);
```

For example, the following program fragment demonstrates output via the **put** procedure:

```
VAR
     primes   : file of integer16;
     poem     : text;
     a_number : integer16;
     a_letter : char;
BEGIN
     . . .
     a_number := 17;
     primes^  := a_number;
     PUT(primes);    { Append 17 to the file symbolized by primes. }
     . . .
     a_letter := 'Q';
     poem^    := a_letter;
     PUT(poem);      { Append 'Q' to the file symbolized by poem. }
     . . .
```

Note that the three statements

```
 a_letter := 'Q';
 poem^    := a_letter;
 PUT(poem);       { Append 'Q' to the file symbolized by poem. }
```

are identical to the two statements

```
a_letter := 'Q';
 write(poem, a_letter);
```

You must open *f* for writing (with **rewrite**) before calling **put**.

> **NOTE:** When you want to close the **text** file on which you were performing **puts**, your program should issue a **writeln** to the file just before closing it. This is in order to flush the file's internal output buffer. If you don't include the **writeln**, the last line of the file may not be written.

## EXAMPLE

```
PROGRAM put_example;

 { This program builds a file of student records in file 'his101'.  }

 CONST
     file_to_write_to = 'his101';
 TYPE
     student =
        record
             name   : array[1..12] of char;
             age    : integer16;
        end;
 VAR
     class           : FILE OF student;
     a_student       : student;
     iostat          : integer32;
```

```
BEGIN

{ Opens file his101 for writing. }
    open(class, file_to_write_to, 'NEW', iostat);
    if iostat = 0
        then rewrite(class)
        else return;

{ Prompt users for input.                           }
    repeat
       writeln;
       write('Enter the name of a student -- ');
       readln(a_student.name);
       if a_student.name = 'end' then exit;
       write('Enter the age of this student -- ');
       readln(a_student.age);

{ Append each record to the end of the rec file.    }
       class^ := a_student;
       PUT(class);

    until false;

END.
```

## USING THIS EXAMPLE

This program is available online and is named **put_example**.

**Read, Readln** Read information from the specified file (or from the keyboard) into the specified variable(s).

## FORMAT

**read**(*f, var1, ..., varN*); {**read** is a procedure.}

and

**readln**(*f, var1, ..., varN*); {**readln** is a procedure.}

## ARGUMENTS

| | |
|---|---|
| *f* | A variable having a file data type. For **read,** *f* can be a **text** or a **file** variable. However, for **readln** *f* must be a **text** variable. *F* is optional. If you do not specify *f,* Domain Pascal reads from standard input (**input**), which is usually the keyboard. |
| *var* | One or more variables separated by commas. *Var* can be any real, integer, char, Boolean, subrange, or enumerated variable. (Boolean and enumerated are extensions to the standard.) *Var* can also be an array variable or variable-length string (see "Array Operations" earlier in this encyclopedia), an element of an array, or a field of a record variable. |

## DESCRIPTION

**Read** and **readln** perform input operations. (Refer to the **get** listing earlier in this encyclopedia.) You use **read** or **readln** to gather one or more pieces of data from *f* and store them into *var1* through *varN*. **Read** and **readln** store the first piece of gathered data into *var1*, the second piece into *var2*, and so on until *varN*.

If *var* is an array of char, **read** and **readln** attempt to read as many characters as the array can hold. If there are not enough characters in the file to fill the array, the remaining elements of *var* are padded with spaces. If *var* is a **varying** array, **read** and **readln** attempt to read the number of characters specified as the maximum length of the string. If there are fewer characters, no padding occurs. The current length of the string is adjusted to reflect the number of characters actually read.

Before calling **read** or **readln**, you must open the file symbolized by *f* for reading. Chapter 8 explains how to do that.

There is a subtle, but important, difference between **read** and **readln**. After a **read**, the stream marker points to the character or component after the last character or component it read from the file. In contrast, after a **readln**, the stream marker points to the character or record after the next end-of-line character in the file. In other words, after getting the data for *varN*, **readln** skips over the remainder of the current line in the input file. (Note that *var1* through *varN* may themselves cover several lines of data in the file.) If you call **readln** and *var* is a record variable, the compiler reports an error; however, it is not an error to call **read** with the same variable—as long as the record variable is the base type of *f*.

If you call **read** or **readln** when **eof**(*f*) is true, the operating system reports an error.

## EXAMPLE

```
PROGRAM read_example;
{ NOTE: Before running this program, you must obtain file          }
{ "annabel_lee" and store it in the same directory as the program.  }
{ This program demonstrates READLN by reading from the poem stored in}
{ pathname 'annabel_lee'.                                            }

 CONST
     pathname = 'annabel_lee';
 VAR
     a_line           : string;
     poem             : text;
     title            : array[1..60] of char;
     st               : integer32;
     count, n         : integer16;

 BEGIN

 { Open the file for reading.                                      }
     open(poem, pathname, 'OLD', st);
     if st = 0
        then reset(poem)
        else begin
                writeln('Cannot open ', pathname);
                return;
             end;
     readln(poem, title);
     writeln('Which line of ', title);
     write('do you want to retrieve? -- ');
     readln(n);
     for count := 1 to (n + 1) do
            readln(poem, a_line);
     writeln(output, a_line);

 END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **read_example**:

```
Which line of                         Annabel Lee
do you want to retrieve? -- 3
That a maiden there lived whom you may know
```

## Record Operations

In Chapter 3 you learned how to declare record types and variables. This section explains how to refer to records in the action part of your program.

### Referring to Fixed Records

In the action part of your program, you specify a field of a fixed record in the following way:

*record_name.field_name*

For example, consider the following declaration of a fixed record variable:

```
VAR
    student : record
       n    : array[1..26] of char;
       id   : integer16;
    end;
```

You can assign values to the two fields with the following statements:

```
student.n   := 'Herman Melville';
student.id  := 37;
```

If the data type of the field is itself a record, you must specify the ultimate field in the following way:

*record_name.field_name.field_name*

For example, consider the following record within a record declaration:

```
TYPE
     name = record
        first  : array[1..10] of char;
        middle : array[1..10] of char;
        last   : array[1..16] of char;
     end;
 VAR
     student : record
          n   : name;
          id  : integer16;
     end;
```

You can assign values to all four fields with the following statements:

```
student.n.first  := 'Kerry';
student.n.middle := 'Bruce';
student.n.last   := 'Raduns';
student.id       := 134;
```

## Variant Records

In the "Variant Records" section of Chapter 3, the variant records worker and my_code were declared as follows:

```
TYPE
    worker_groups = (exempt, non_exempt);   {enumerated type}
    worker = record    {record type}
               employee : array[1..30] of char; {field in fixed part}
               id_number : integer16;           {field in fixed part}
               CASE wo : worker_groups OF       {variant part}
                  exempt : (yearly_salary   : integer32);
                  non_exempt : (hourly_wage : real);
             end;
    my_code = record
                CASE integer OF          {variant part}
                       1 : (all : array[1..4] of char);
                       2 : (first_half : array[1..2] of char;
                            second_half : array[1..2] of char);
                       3 : (x1  : integer16;
                            x2  : boolean;
                            x3  : char);
                       4 : (rall : single);
              end;
 VAR
     w : worker;
    mc : my_code;
```

The following fragment assigns values to w:

```
write('Enter the person''s name -- ');      readln(w.employee);
write('Enter the person''s id number -- '); readln(w.id_number);
write('Enter pay status (exempt or non_exempt) -- '); readln(w.wo);
if w.wo = exempt
    then begin
            write('Enter yearly salary -- '); readln(w.yearly_salary);
         end
    else begin
            write('Enter hourly wage -- '); readln(w.hourly_wage);
         end;
```

**NOTE:** Suppose you execute the preceding fragment and load values into **w.employee**, **w.id_number**, **w.wo**, and **w.hourly_wage**. Note that the compiler won't protect you from mistakenly trying to access **w.yearly_salary** rather than **w.hourly_wage**.

The following fragment assigns values to **mc**. (Notice that we do not use the constants 1, 2, 3, and 4 to specify these fields.)

```
write('Enter two characters -- ');  readln(mc.first_half);
write('Enter two more characters -- '); readln(mc.second_half);
writeln('Together, the four characters are ', mc.all);
```

### Arrays of Records

A common way to store records is as an array of records. You must use the following format to specify a field in an array of records:

*array_name*[*component*].*field_name*

For example, given the following declaration for **school**:

```
TYPE
    student = record
       age : 11..20;
       class : 7..12;
       name : array[1..20] of char;
    end;

 VAR
    school : array[1..1000] of student;
```

you can specify the 500th record as:

```
school[500].age := 15;
school[500].class := 10;
school[500].name := 'John Donne';
```

**Repeat/Until** Executes the statements within a loop until a specified condition is satisfied.

## FORMAT

**repeat** {**repeat** is a statement.}
  *stmnt*; . . .
**until** *cond*;

## ARGUMENTS

| | |
|---|---|
| *stmnt* | An optional argument. For *stmnt*, specify a simple statement or a compound statement. (Ordinarily, you must indicate a compound statement with a **begin/end** pair; however, the **begin/end** pair is optional within a **repeat** statement.) |
| *cond* | Any Boolean expression. |

## DESCRIPTION

**Repeat** marks the start of a loop; **until** marks the end of that loop. At run time, Pascal executes *stmnt* within that loop until *cond* is true. As long as *cond* is false, Pascal continues to execute the statements within the loop.

The following list describes two methods of jumping out of a **repeat** loop prematurely (i.e., before the condition is true):

- Use **exit** to transfer control to the first statement following the **repeat** loop.
- Use **goto** to transfer control outside the loop.

In addition to these measures, you can also execute a **next** statement to skip the remainder of the statements in the loop for one iteration.

## EXAMPLE

```
PROGRAM repeat_example;
 { This program demonstrates two different REPEAT loops. }
 { Compare it to while_example.                          }
 VAR
   num             : integer16;
   test_completed  : boolean;
   i               : integer32;
```

## Repeat/Until

```
BEGIN
     write('Enter an integer -- ');
     readln(num);

     REPEAT
         num := num + 10;
         writeln(num, sqr(num));
     UNTIL (num > 101);

     writeln;
     test_completed := false;
     REPEAT
         write('Enter another integer (or 0 to stop the program) -- ');
         readln(i);
         if i = 0 then
             test_completed := true
         else
             writeln('The absolute value of ', i:1, ' is ', abs(i):1);
     UNTIL test_completed;
 END.
```

### USING THIS EXAMPLE

Following is a sample run of the program named **repeat_example**:

```
Enter an integer -- 70
        80      6400
        90      8100
       100     10000
       110     12100

 Enter another integer (or 0 to stop the program) -- 4
 The absolute value of 4 is 4
 Enter another integer (or 0 to stop the program) -- -5
 The absolute value of -5 is 5
 Enter another integer (or 0 to stop the program) -- 0
```

Now, consider a second run of **repeat_example**. This time, the user enters an integer greater than 101. In contrast to **while_example**, the program still executes the loop once:

```
Enter an integer -- 102
       112     12544

 Enter another integer (or 0 to stop the program) -- 0
```

**Replace** Substitutes a new record for an existing record in a file. (Extension)

## FORMAT

**replace**(*file_variable*) {**replace** is a procedure.}

## ARGUMENT

| | |
|---|---|
| *file_variable* | A **file** variable. |

## DESCRIPTION

Use the **replace** procedure to replace an element in the file specified by *file_variable*. You can use **replace** only on files with **file** type; you cannot use it to replace an element in a **text** file.

Before calling **replace** you must do the following:

1. Open the file for reading. (Chapter 8 explains how to do this.)
2. Specify the record that you wish to replace. To do this, you can use the **find** procedure (described earlier in this chapter).
3. Store the replacement record by entering a statement of the format

   *file_variable^* := *replacement_record*;

The **replace** procedure permits a program to rewrite file components—for example, to correct errors—while the file is open for read access. The program need not close the file and reopen it.

**NOTE:** The term "record", as it applies to a file of **file** type, refers to an object of the file's base type. This may or may not be a Domain Pascal record type. (For example, the object could be an integer type.)

## EXAMPLE

For a full example of **replace**, see the example for the **find** procedure that appears earlier in this encyclopedia.

**Reset** Makes an open file available for reading.

## FORMAT

**reset**(*filename*) {**reset** is a procedure.}

## ARGUMENT

| | |
|---|---|
| *filename* | A variable having the **text** or **file** data type. |

## DESCRIPTION

Before you can read data from a file other than standard input, you must reset the file. If the file is a temporary file and does not already exist, **reset** creates an empty file. If the file is not a temporary file, it must already exist, and you must have previously opened it using **open**. The **open** procedure tells the system to open a file for some type of I/O operation; **reset** tells the system to allow you to read from the file, but prevents you from modifying the file. (See the description of the **find** procedure for one exception to this rule.)

*Filename* must symbolize an open file.

Calling **reset** sets the stream marker to point to the beginning of the file. Therefore, *filename^* will contain the first character or component of the file. You can change the stream marker by reading from the file (with **read**, **readln**, or **get**) or by calling the **find** procedure.

If the file is empty when you call **reset**, then *filename^* is totally undefined. That is, there is no way to predict what the value of *filename^* will be.

To open the file for write access you must call **rewrite** instead of **reset**.

## EXAMPLE

```
PROGRAM reset_example;
{NOTE: Before running this program, you must obtain file "annabel_lee" }
{      and store it in the same directory as the program.              }
{ Demonstrates reset.  After opening a file (with OPEN), the program   }
{ reads the first line of the file and writes it to standard output.   }

{ We need the two include files in order to use status_$t and          }
{ error_$print.                                                        }
 %NOLIST;
 %INCLUDE '/sys/ins/base.ins.pas';
 %INCLUDE '/sys/ins/error.ins.pas';
 %LIST;

 CONST
     pathname_of_file = 'annabel_lee';

 VAR
     assignment        : text;
     openstatus        : status_$t;
     a_line            : string;

 BEGIN

 {Open the file for reading.                                          }
     open(assignment, pathname_of_file, 'OLD', openstatus.all);
     if openstatus.all = status_$ok
        then RESET(assignment)
        else begin
               error_$print(openstatus);  {print any error            }
               return;
             end;
 {See Chapter 9 for a discussion of error handling.                   }

 {Read the first line of the file.                                    }
     readln(assignment, a_line);

 {Write this line to standard output (usually the transcript pad).    }
     writeln(output, a_line);

 END.
```

## USING THIS EXAMPLE

This program is available online and is named **reset_example**.

## Return

**Return** Causes program control to jump back to the calling procedure or function. (Extension)

### FORMAT

**Return** is a statement that takes no arguments and returns no values.

### DESCRIPTION

Ordinarily, after Domain Pascal executes the last statement in a routine, it returns control to the calling routine. However, you can use **return** to jump back prematurely (i.e., before the last statement) to the calling procedure or function. You can use **return** anywhere in the body of a procedure or function.

Using **return** in the main procedure causes the program to terminate.

### EXAMPLE

```
PROGRAM return_example;
 {This program demonstrates the RETURN statement. }
 VAR
     Ph : single;

 Procedure check_Ph;
 BEGIN
     if (Ph < 2.0) or (Ph > 14.0) then
         begin
         writeln('You have entered an invalid result.');
         RETURN;
         end
     else if (Ph <= 4.5) then
         writeln('You have entered a valid (but suspicious) result.')
     else     {Ph > 4.5}
         writeln('You have entered a valid result.');
     writeln('Thank you for your cooperation.');
 END;

 BEGIN  {main procedure}
     write('Enter the Ph of the test sample -- ');
     readln(Ph);
     check_Ph;
 END.
```

## USING THIS EXAMPLE

This program is available online and is named **return_example**.

## Rewrite

**Rewrite** Makes an open file available for writing only.

### FORMAT

**rewrite**(*filename*) {**rewrite** is a procedure.}

### ARGUMENT

| | |
|---|---|
| *filename* | A variable having the **text** or **file** data type. |

### DESCRIPTION

Before you can write data to a permanent file other than standard output, you must do two things. First, open the file with the **open** procedure. Second, call the **rewrite** procedure. **Open** tells the system to open a file for some type of I/O operations; **rewrite** tells the system to allow you to modify the open file. *Filename* must symbolize an open file.

To open a temporary file for writing, you merely have to call the **rewrite** procedure (that is, don't call the **open** procedure).

**NOTE:** **Rewrite** clears an existing file of its entire contents. To avoid inadvertently erasing an important file, you might consider using the **file_history** value 'NEW' when you call **open**.

**Rewrite** sets the stream marker to the beginning of the file. Each call to **write**, **writeln**, or **put** advances the stream marker.

After calling **rewrite**, the file is empty. Therefore, the value of *filename*^ is totally undefined. That is, there is no way to predict what its value will be.

To open the file for read access you must call **reset** instead of **rewrite**.

## EXAMPLE

```
PROGRAM rewrite_example;
{ This program demonstrates rewrite.  Opens a file for writing.     }
{ The program will prompt you for the name of the file to open.     }
{ After opening the file, you can write a sentence to it.           }
{ We need the two include files in order to use status_$t and       }
{ error_$print.                                                     }

 %NOLIST;
 %INCLUDE '/sys/ins/base.ins.pas';
 %INCLUDE '/sys/ins/error.ins.pas';
 %LIST;

 VAR
     name_of_file      : array[1..50] of char;
     profound          : text;
     openstatus        : status_$t;
     a_line            : string;

 BEGIN

 { Prompt the user for the name of a file to open. }
     write('What is the pathname of the file you want to write to -- ');
     readln(name_of_file);
 { Open the file for writing. }
     open(profound, name_of_file, 'NEW', openstatus.all);
     if openstatus.all = status_$ok
        then REWRITE(profound)
        else begin
               error_$print(openstatus);  { Print an error message. }
               return;
             end;
 { Prompt the user. }
     writeln('Now enter a line of text.');
     readln(a_line);
 { Write the line out to the open file. }
     writeln(profound, a_line);

 END.
```

## USING THIS EXAMPLE

This program is available online and is named **rewrite_example**.

**Round** Converts a real number to the closest integer.

## FORMAT

**round**(*n*) {**round** is a function.}

## ARGUMENT

| | |
|---|---|
| *n* | Any real expression. |

## FUNCTION RETURNS

The **round** function returns an integer value.

## DESCRIPTION

The **round** function rounds (up or down) *n* to the closest integer. If the decimal part of *n* is equal to or greater than .5, the **round** function rounds up. (Compare **round** to **trunc.**)

## EXAMPLE

```
PROGRAM round_example;
{This program demonstrates the round function}
VAR
   x : REAL;
   y : INTEGER;
BEGIN
   x := 54.2; y := ROUND(x);   WRITELN('If you round ',x,' you get ',y);
   x := 54.5; y := ROUND(x);   WRITELN('If you round ',x,' you get ',y);
   x := 54.8;   := ROUND(x);   WRITELN('If you round ',x,' you get ',y);
   x := -54.8; y := ROUND(x);  WRITELN('If you round ',x,' you get ',y);
END.
```

## USING THIS EXAMPLE

If you execute the sample program named **round_example**, you get the following output:

```
If you round  5.420000E+01 you get        54
If you round  5.450000E+01 you get        55
If you round  5.480000E+01 you get        55
If you round -5.480000E+01 you get       -55
```

**Rshft** Shifts the bits in an integer a specified number of spaces to the right. (Extension)

## FORMAT

**rshft**(*num, sh*) {**rshft** is a function.}

## ARGUMENTS

| | |
|---|---|
| *num, sh* | Integer expressions. |

## FUNCTION RETURNS

The **rshft** function returns an integer value.

## DESCRIPTION

The **rshft** function shifts the bits in *num* to the right *sh* places. The expression *num* can be any integer expression smaller than 32 bits. **Rshft** does not wrap bits around from the right edge to the left; instead, **rshft** shifts zeros in from the left end.

**Rshft** does not preserve the sign bit. The sign bit moves to the right just like every other bit. This means that if **num** is negative and is a 32-bit integer, the result of an **rshft** is always positive. Of course, if **num** already is positive, the result of an **rshft** will still be positive.

Say, for example, *num* is a 16-bit signed integer and the result of the function is to be stored in a 16-bit integer variable. In this case, **rshft** expands *num* to a 32-bit integer, performs the shift, and converts it back to a 16-bit integer. Note that, because of the expansion and contraction, **rshft** always returns a negative number when *num* is a 16-bit negative expression and *sh* is less than or equal to 16.

Consider this example. Suppose *num* is 16 bits and equals -9. You perform an **rshft** with *sh* equaling 3 and put the result back in a 16-bit integer. Here's what happens at each step:

```
Before the rshft                            1111111111110111 = -9
Convert to 32-bit integer   11111111111111111111111111110111
Rshft 3 bits                00011111111111111111111111111110
Convert to 16-bit integer                   1111111111111110 = -2
```

If you print the **rshft** result *before* it is converted back to 16 bits, you get the number represented in the third step above which, of course, is a different number than the final result. Write your code like this to get that 32–bit result

```
writeln(rshft(num,3));
```

instead of like this

```
answer := rshft(num,3);      {Assume answer is a 16-bit integer.}
writeln(answer);
```

Results are unpredictable if *sh* is negative.

Compare **rshft** to **lshft** and **arshft**.

## EXAMPLE

See the example shown in the **arshft** listing earlier in this encyclopedia.

## Set Operations

In Chapter 3 we described how to declare set variables. This section explains how to use set variables in the code portion of your program.

### ASSIGNMENT

To assign value(s) to a set variable, use one of the following formats:

*set_variable* := [];
*set_variable* := [*el, el, ... el*];
*set_variable* := [*el* .. *el*];
*set_variable* := *set_expression set_operator set_expression*;

The brackets are mandatory. *El* must be an expression with a value having the same data type as the base type of the set variable.

(The **set_operators** are detailed later in this listing.)

The following program fragment shows seven possible set assignments for the **paint** set:

```
TYPE
    colors = (white, beige, black, red, blue, yellow, green);

 VAR
    c : colors;  {enumerated variable}
    paint1, paint2, paint3, paint4, paint5, paint6, paint7 : SET OF col-
ors;

 BEGIN
    c := blue;
    paint1 := [];                {Null set.}
    paint2 := [rojo];            {Illegal assignment.}
    paint3 := [red];
    paint4 := [beige, green, black];
    paint5 := [white .. green];  {All seven elements.}
    paint6 := [beige .. blue];   {beige, black, red, and blue.}
    paint7 := [c];               {blue.}
```

## Set Operations

### SET OPERATORS

Table 4-13 shows the eight set operators Domain Pascal supports. The following subsections describe these operators individually.

*Table 4-13. Set Operators*

| Set Operator | Operation |
|---|---|
| + | Union of two sets |
| * | Intersection of two sets |
| - | Set exclusion |
| = | Set equality |
| <> | Set inequality |
| <= | Subset |
| >= | Superset |
| **in** | Inclusion |

#### Union

The union of two sets is a set containing all members of both sets. In the following example, **paint3** contains the union of sets **paint1** and **paint2**:

```
TYPE
    colors = (white, beige, black, red, blue, yellow, green);

 VAR
    c : colors;
    paint1, paint2, paint3 : SET OF colors := [];

 BEGIN
    paint1 := [white, black, red];
    paint2 := [black, yellow];
    paint3 := paint1 + paint2;
 {paint3 will contain white, black, red, and yellow.}
```

If there are duplicates (for example, **black**), the resulting set does not store the duplicate value twice. Thus, **paint3** contains **black** only once.

### Intersection

The intersection of two sets is a set containing only the duplicate elements. In the following example, **paint3** contains the intersection of sets **paint1** and **paint2**:

```
TYPE
     colors = (white, beige, black, red, blue, yellow, green);

 VAR
     c : colors;
     paint1, paint2, paint3 : SET OF colors := [];

 BEGIN
     paint1 := [white, black, red];
     paint2 := [black, yellow];
     paint3 := paint1 * paint2;
 {paint3 will contain black.}
```

### Set Exclusion

Pascal finds the result of a set exclusion operation by starting with all the elements in the left operand and crossing out any of the elements that are duplicated in the right operand. The following program fragment demonstrates two set exclusion operations:

```
TYPE
     colors = (white, beige, black, red, blue, yellow, green);

 VAR
     c : colors;
     paint1, paint2, paint3 : SET OF colors := [];

 BEGIN
     paint1 := [white, black, red];
     paint2 := [black, yellow];

     paint3 := paint1 - paint2;
 {paint3 will contain white and red.}

     paint3 := paint2 - paint1;
 {paint3 will contain yellow.}
```

## Set Operations

### Set Equality and Inequality

The result of a set equality (=) or inequality (<>) operation is a Boolean value. If two set variables contain exactly the same elements (or are both null sets), then = is true and <> is false. The following program fragment demonstrates set equality:

```
TYPE
     colors = (white, beige, black, red, blue, yellow, green);

 VAR
     c : colors;
     paintl, paint2 : SET OF colors := [];

 BEGIN
     paintl := [white, black, red];
     paint2 := [black, yellow];
     if paintl = paint2
         then writeln('The two sets contain the same elements.');
         else writeln('The two sets do not contain the same elements.');
```

### Subset

The result of a subset operation (<=) is a Boolean value. If the first operand is a subset of the second operand, then the result is true; otherwise, the result is false.

```
TYPE
     colors = (white, beige, black, red, blue, yellow, green);

 VAR
     c : colors;
     paintl, paint2 : SET OF colors := [];

 BEGIN
     paintl := [white, black, red];
     paint2 := [white, red];
     if paint2 <= paintl     {this is true}
         then writeln ('Paint2 is a subset of paintl');
```

### Superset

The result of a superset operation (>=) is a Boolean value. If the first operand is a superset of the second operand, then the result is true; otherwise, the result is false.

```
TYPE
     colors = (white, beige, black, red, blue, yellow, green);
VAR
     c : colors;
     paint1, paint2 : SET OF colors := [];
BEGIN
     paint1 := [white, black, red];
     paint2 := [white, red];
     if paint1 >= paint2  {this is true.}
         then writeln('Paint1 is a superset of paint2.');
```

## Inclusion

See the separate **in** listing earlier in this encyclopedia.

## EXAMPLE

```
PROGRAM set_example;
 {This program demonstrates I/O with set variables. You cannot  }
 {use a set variable as an argument to any of the predeclared   }
 {Pascal I/O procedures, so you must use a somewhat roundabout  }
 {method involving the base type of the set.                    }

 TYPE
     possible_ingredients =
         (sugar, nuts, chips, milk, flour, carob, salt, bkg_soda);
 VAR
     pi      : possible_ingredients;
     cookies : set of possible_ingredients := [];
     answer  : char;
 BEGIN
 {Read the proper cookie ingredients and store }
 {them in the cookies variable.                }
     for pi := sugar to bkg_soda do
         begin
         write('Should the recipe contain ', pi:4, '? (y or n) -- ');
         readln(answer);
         if (answer = 'y') or (answer = 'Y') then
             cookies := cookies + [pi];
         end; {for}
 {Write the list of ingredients.               }
     writeln(chr(10), 'The ingredients are: ');
     for pi := sugar to bkg_soda do
         if pi IN cookies then
             writeln(pi);
 END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **set_example**:

```
Should the recipe contain SUGAR? (y or n) -- y
Should the recipe contain NUTS? (y or n) -- y
Should the recipe contain CHIPS? (y or n) -- y
Should the recipe contain MILK? (y or n) - n
Should the recipe contain FLOUR? (y or n) -- y
Should the recipe contain CAROB? (y or n) -- n
Should the recipe contain SALT? (y or n) -- y
Should the recipe contain BKG_SODA? (y or n) -- y

The ingredients are:
          SUGAR
           NUTS
          CHIPS
          FLOUR
           SALT
       BKG_SODA
```

**Sin** Calculates the sine of the specified number.

## FORMAT

**sin**(*number*) {**sin** is a function.}

## ARGUMENT

| | |
|---|---|
| *number* | Any real or integer expression. |

## FUNCTION RETURNS

The **sin** function returns a real value (even if **number** is an integer).

## DESCRIPTION

The **sin** function calculates the sine of a number. This function assumes that the argument (*number*) is a radian measure (as opposed to a degree measure). (Refer also to the **cos** listing earlier in this encyclopedia.)

## EXAMPLE

```
PROGRAM sin_example;
 {This program demonstrates the SIN function.}

 CONST
 pi = 3.1415926535;

 VAR
 angle_in_radians, c1, converted_to_radians, c2, angle_in_degrees: REAL;

 BEGIN
  write('Enter an angle in radians -- ');
  readln(angle_in_radians);
  c1 := SIN(angle_in_radians);
  writeln('The sine of ', angle_in_radians:5:3, ' radians is ', c1:5:3);

 {The following statements show how to convert from degrees to radians.}
  write('Enter another angle (in degrees) -- ');
  readln(angle_in_degrees);
  converted_to_radians := ((angle_in_degrees * pi) / 180.0);
  c2 := SIN(converted_to_radians);
  writeln('The sine of ', angle_in_degrees:5:3, ' is ', c2:5:3);

END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **sin_example**:

```
 Enter an angle in radians -- 1.0
 The sine of 1.000 radians is 0.841
 Enter another angle (in degrees) -- 14.2
 The sine of 14.200 is 0.245
```

**Sizeof** Returns the size (in bytes) of the specified data object. (Extension)

## FORMAT

The **sizeof** function has two formats:

| | |
|---|---|
| **sizeof**(*x*) | {first form} |
| **sizeof**(*x*, *tag1*, . . . *tagN*) | {second form} |

## ARGUMENTS

| | |
|---|---|
| *x* | The name of a type (standard or user-defined), a variable, a constant, or a string. |
| *tag* | One or more constants corresponding to the fields in a variant record. You specify these *tag*s only if you want to find the size of a variant record. The number of *tag*s can be no greater than the number of tag fields in the variant record. |

## FUNCTION RETURNS

The function returns an integer value.

## DESCRIPTION

**Sizeof** returns an integer equal to the number of bytes that the program uses to store *x*. If the argument to **sizeof** is a variable-length string, **sizeof** returns the size of the entire record, including the current length field and any padding bytes.

You must often supply a string and its length as input arguments when calling a procedure or function. You can use **sizeof** to calculate the string's length, although the way you call **sizeof** affects the answer you get. For example, if your code includes the following:

```
VAR
    length : integer;
    animal : string;
          . . .
 animal := 'wildebeest';
 length := SIZEOF(animal);
```

**sizeof** returns 80 because **animal** is declared to be a **string**, and **string** is defined as being an array of 80 **chars**. However, if you call the function this way:

```
length := SIZEOF('wildebeest');
```

**sizeof** returns a value of 10.

To find the size of a specified variant record, you must pass both the name of the variant record type (or variable), and tag fields. For example, consider the following record declaration:

```
TYPE
    worker_stat  = (exempt, nonexempt);
    worker = record
                name : array[1..24] of char;
                case worker_stat of
                  exempt    : (salary  : integer16);
                  nonexempt : (wages   : single;
                               plant   : array[1..20] of char);
                end;
    end;
```

To find the size of a record if **worker_stat** equals **exempt**, call **sizeof** as follows:

```
SIZEOF(worker, exempt)
```

To find the size of a record if **worker_stat** equals **nonexempt**, call **sizeof** as follows:

```
SIZEOF(worker, nonexempt)
```

## EXAMPLE

```
PROGRAM sizeof_example;
 { This program demonstrates the SIZEOF function. }

 CONST
     tree  = 'ficus';
 TYPE
     student = RECORD
         name : array[1..19] of char;
         age  : integer16;
         id   : integer32;
     end;
 VAR
     t2    : integer16;

 BEGIN

  writeln('The size of constant tree is ', SIZEOF(tree):1);
  writeln('The size of variable t2 is ', SIZEOF(t2):1);
  writeln('The size of an integer32 variable is ', SIZEOF(integer32):1);
  writeln('The size of a student record is ', SIZEOF(student):1);

 END.
```

## USING THIS EXAMPLE

If you execute the sample program named **sizeof_example**, you get the following output:

```
The size of constant tree is 5
The size of variable t2 is 2
The size of an integer32 variable is 4
The size of the student record type is 26
```

**Sqr** Calculates the square of a specified number.

## FORMAT

**sqr**(*n*) {sqr is a function.}

## ARGUMENT

| | |
|---|---|
| *n* | Any integer or real expression. |

## FUNCTION RETURNS

The **sqr** function returns an integer if *n* is an integer, and returns a real number if *n* is real.

## DESCRIPTION

The **sqr** function calculates *n* * *n*. A potential problem for users is that the square of a large **integer16** value often exceeds the maximum value (32,767) for **integer16** variables. If this error is possible in your program, assign the square of an **integer16** variable to an **integer32** variable.

## EXAMPLE

```
PROGRAM sqr_example;
{This program demonstrates the use of the sqr function}
 VAR
    i_short  : integer16;
    i_long   : integer32;
    r1, r2  : real;

 BEGIN
    write('Enter an integer -- '); readln(i_short);
    i_long := SQR(i_short);
    writeln('The square of ', i_short:1, ' is ', i_long:1);

    write('Enter a real number -- '); readln(r1);
    r2 := SQR(r1);
    writeln('The square of ', r1:1, ' is ', r2:1);
 END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **sqr_example**:

```
Enter an integer -- 1100
The square of 1100 is 1210000
Enter a real number -- -5.23
The square of -5.230000E+00 is 2.735290E+01
```

**Sqrt** Calculates the square root of a specified number.

## FORMAT

**sqrt**(*n*)  {**sqrt** is a function.}

## ARGUMENT

| | |
|---|---|
| *n* | Any integer or real expression that evaluates to a number greater than zero. |

## FUNCTION RETURNS

The **sqrt** function returns a real value (even if *n* is an integer).

## DESCRIPTION

The **sqrt** function calculates the square root of *n*.

## EXAMPLE

```
PROGRAM sqrt_example;
{This program demonstrates the use of the sqrt function}

 VAR
    i_long   : integer32;
    r_single : single;
    r_double : double;

 BEGIN
    write('Enter an integer -- '); readln(i_long);
    r_double := SQRT(i_long);
    writeln('The square root of ', i_long:1, ' is ', r_double);

    write('Enter a real number -- '); readln(r_single);
    r_double := SQRT(r_single);
    writeln('The square root of ', r_single, ' is ', r_double);
 END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **sqrt_example**:

```
Enter an integer -- 24
The square root of 24 is    4.898979663848877E+00
Enter a real number -- 24.0
The square root of     2.400000E+01 is    4.898979663848877E+00
```

## Statements

Throughout this encyclopedia, we refer to statements, both simple and compound. Here, we define statement.

### Statement

When a format requires a statement, you must enter one of the following:

- An assignment statement like **x := 5** or **x := y + z**. An assignment statement can also be a call to a function like **x := ORD('a')**.
- A procedure call like **writeln('hi')**.
- A **goto**, **if/then**, **if/then/else**, **case**, **for**, **repeat**, **while**, **with**, **exit**, **next**, or **return** statement.
- A compound statement.
- An empty statement. An empty statement causes no action except an advance to the next statement. It can be represented by two consecutive semicolons (;;).

### Simple and Compound Statements

When the format part of a listing in this chapter says that a command requires a simple statement, it just means that we require one statement (see above). A compound statement is a group of zero or more statements bracketed by the keywords **begin** and **end**. In other words, a compound statement has the following format:

```
begin
      statement1;
          .
          .
          .
      statementN
end;
```

The action part of a routine is itself a compound statement. A statement can be preceded by a label (but not every label is accessible; see the **goto** listing earlier in this chapter).

**Substr** Extracts a substring from a string. (Extension)

## FORMAT

**substr**(*src_string, start, length*); {**substr** is a function.}

## ARGUMENTS

| | |
|---|---|
| *src_string* | A variable-length string, character array, or character-string constant. |
| *start* | The position of the first character in the desired substring of *src_string*. |
| *length* | The number of characters in the desired substring (*length* is a positive integer). |

## FUNCTION RETURNS

The **substr** function returns a variable-length string.

## DESCRIPTION

The **substr** function returns a substring of the source string. The *start* parameter indicates the position of the first character of the desired substring. The *length* parameter indicates the number of characters to return.

If the values of either *start* or *length* specify a character position outside of the bounds of the run-time size of the source string, an error trap occurs.

## EXAMPLE

```
PROGRAM substr_example;
{This program demonstrates the use of the substr function.}
{It displays a substring of a given string.              }

 VAR
     str1    :   array[1..30] of char;
     str2    :   varying[30] of char;
     str3    :   varying[30] of char;

 BEGIN
     str1 := 'Foolish consistency is ';
     str2 := 'the hobgoblin of ';
     str3 := substr( 'little minds', 8, 5 );

     writeln( substr(str1, 1, 8));
     writeln( substr(str2, 5, 10));
     writeln( str3 );
 END.
```

## USING THIS EXAMPLE

Executing this program, named **substr_example**, produces the following output:

```
Foolish
hobgoblin
minds
```

**Succ** Returns the successor of a specified ordinal value.

## FORMAT

**succ**(*x*) {**succ** is a function.}

## ARGUMENT

*x* Must be an integer, Boolean, char, or enumerated expression.

## FUNCTION RETURNS

The **succ** function returns a value having the same data type as *x*.

## DESCRIPTION

The **succ** function returns the successor of *x* according to the following rules:

| Data Type of x | Succ Returns |
|---|---|
| **Integer** | The numerical value equal to *x* + 1. |
| **Boolean** | True—even if *x* already equals true. |
| **Char** | The character with the ISO Latin–1 value one greater than the ISO Latin–1 value of *x*. |
| **Enumerated** | The identifier to the right of *x* in the type declaration. |

**succ**(**lastof**(*x*)) generally is undefined; however, Domain Pascal does not report an error. Domain Pascal also doesn't report an error if you specify an integer value that is outside the range of the specified integer type. Therefore, your program should test for an out-of-bounds condition.

Compare the **succ** function to the **pred** function.

## EXAMPLE

```
PROGRAM succ_example;
{This program demonstrates the use of the succ function}

 TYPE
    jours = (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche);

 VAR
    int  : integer;
    ch   : char;
    semaine : jours;

 BEGIN
    int := succ(53);   writeln('The successor to 53 is ', int:1);
    ch := succ('q');   writeln('The successor to q is ', ch);
    semaine := succ(jeudi);
    writeln('The successor to jeudi is ', semaine:8);
 END.
```

## USING THIS EXAMPLE

If you execute the sample program named **succ_example**, you get the following output:

```
The successor to 53 is 54
The successor to q is r
The successor to jeudi is VENDREDI
```

**Then** Refer to **if** earlier in this encyclopedia.

**To** Refer to **for** earlier in this encyclopedia.

Trunc Truncates a real number to an integer.

## FORMAT

**trunc**(*n*) {**trunc** is a function.}

## ARGUMENT

| | |
|---|---|
| *n* | Any real value. |

## FUNCTION RETURNS

The **trunc** function returns an integer.

## DESCRIPTION

The **trunc** function removes the fractional part of *n* to create an integer. (Compare **trunc** to **round**.)

## EXAMPLE

```
PROGRAM trunc_example;
{ This program demonstrates the use of the trunc function. }
{ Compare to round function.                               }
VAR
   x : REAL;
   y : INTEGER;
BEGIN
  x := 54.2; y := TRUNC(x); WRITELN('If you truncate ',x,' you get ',y);
  x := 54.5; y := TRUNC(x); WRITELN('If you truncate ',x,' you get ',y);
  x := 54.8; y := TRUNC(x); WRITELN('If you truncate ',x,' you get ',y);
  x :=-54.8; y := TRUNC(x); WRITELN('If you truncate ',x,' you get ',y)
END.
```

## USING THIS EXAMPLE

If you execute the sample program named **trunc_example**, you get the following output:

```
54
54
54
-54
```

Compare these results to the results of executing program **round_example**.

**Type Transfer Functions** Permit you to change the data type of a variable or expression in the code portion of your program. (Extension)

## FORMAT

*transfer_function*(*x*) {Type transfer functions are functions.}

## ARGUMENTS

*transfer_function*
: The name of any predeclared Domain Pascal data type or any user-defined data type that has been declared in the program.

*x*
: An expression.

## DESCRIPTION

Domain Pascal type transfer functions enable you to change the type of a variable or expression within a statement. To perform a type transfer function, use any user-created or standard type name as if it were a function name in order to "map" the value of its argument into that type.

With one exception, the size of the argument must be the same as the size of the destination type. (Chapter 3 describes the sizes of each data type.) This size equality is required because the type transfer function does not change any bits in the argument. Domain Pascal just "sees" the argument as a value of the new type. The one exception is that integer subranges are always compatible regardless of their sizes.

It is important to remember that type transfer functions do not convert any value. Consider the following data type declarations:

```
VAR
    i : INTEGER32;
    r : REAL;
```

The following assignment *converts* the value of variable **i** to a floating-point number:

```
r := i;
```

However, in the following assignment, Domain Pascal "sees" the bits in **i** as if they were actually representing a floating-point number. In this case, there is a transfer, but no conversion:

```
r := real(i);
```

Note that there are restrictions on the data types to which you can convert a given Domain Pascal data type. For example, you get an error if you try the following:

```
i := r;
```

In such a case, there's no way for Domain Pascal to know what you want to do with the portion of **r** after the decimal point. Use the **trunc** or **round** functions (described earlier in this encyclopedia) instead.

A practical application of type transfer functions is in controlling the bit precision of a computation. For example, consider the following program fragment:

```
VAR
    x, y : integer16;

 BEGIN
    if x + y > 5
      then . . .
```

By default, the compiler expands operands **x** and **y** to 32-bit integers and performs 32-bit addition before making the comparison to 5. However, by using the following type transfer function, we can produce more efficient code:

```
VAR
    x, y : integer16;

 BEGIN
    if INTEGER16(x + y) > 5
      then . . .
```

The disadvantage to using the type transfer function in the preceding fragment is that it ignores the possibility of integer overflow.

## EXAMPLE

```
PROGRAM type_transfer_functions_example;
{ This program demonstrates two uses of type transfer functions.  }
TYPE
   car_manufacturers = (ford, chevy, nissan, dodge, caddy, honda);
   pointer_to_word   = ^word;
   word              = array[1..10] of char;
VAR
   ordinal_value_of_car : integer16;
   car, actual_value_of_car : car_manufacturers;
   name, rename  : word  := [* of ' '];
   namepointer   : pointer_to_word;
BEGIN
       car := dodge;
       ordinal_value_of_car := ord(car);
       actual_value_of_car := CAR_MANUFACTURERS(ordinal_value_of_car);
              { The above statement transfers an integer value into }
              { an enumerated value. }
       write('The actual value of ordinal value ');
       writeln(ordinal_value_of_car:1,' is', actual_value_of_car:7);
{ It is illegal to perform mathematical operations on a pointer      }
{ variable. However, by using type transfer functions you can        }
{ temporarily make a pointer variable into an integer variable so    }
{ that you can perform mathematical operations on it.  Then, after   }
{ using the integer in a math calculation, you can transfer the      }
{ integer back to a pointer type by using a second type transfer     }
{ function.  This routine prints the final eight characters in the   }
{ specified name.                                                    }
       write('Enter a name that is 10 characters long -- ');
       readln(name);
       namepointer := addr(name);  {get starting address of name array}
       namepointer := UNIV_PTR(INTEGER32(namepointer) + 2);
       rename := namepointer^;
       writeln('The last eight characters of the name are ', rename:8);
END.
```

## USING THIS EXAMPLE

If you execute the sample program named **ttf_example**, you get the following output:

```
The actual value of ordinal value 3 is   DODGE
Enter a name that is 10 characters long -- CALIFORNIA
The last eight characters of the name are LIFORNIA
```

## Unpack

**Unpack** Copies a packed array to an unpacked array.

### FORMAT

**unpack**(*packed_array*, *unpacked_array*, *index*) {**unpack** is a procedure.}

### ARGUMENTS

| | |
|---|---|
| *unpacked_array* | An array that has been defined without the keyword **packed**. |
| *packed_array* | An array that has been defined using the keyword **packed**. |
| *index* | A variable that is the same type as the array bounds (**integer**, **boolean**, **char**, or enumerated) of *unpacked_array*. *Index* designates the array element in *unpacked_array* to which **unpack** should begin copying. |

### DESCRIPTION

**Unpack** copies the elements in a packed array to an unpacked one. Data access with unpacked arrays generally is faster since data elements are always aligned on word boundaries.

*Unpacked_array* and *packed_array* must be of the same type, and for every element in *packed_array*, there must be an element in *unpacked_array*. That is, if you have the following **type** definitions

```
TYPE
     x : array[i..j] of single;
     y : packed array[m..n] of single;
```

the subscripts must meet these requirements:

```
j - index >= n - m          {"index" as set in the call to unpack}
```

For example, it is legal to use **unpack** on two arrays defined like this:

```
TYPE
    big_array   : array[1..100] of integer;
    small_array : packed array[1..10] of integer;
 VAR
    grande : big_array;
    petite : small_array;
```

You use **index** to indicate the array element in *unpacked_array* to which **unpack** should begin copying. For instance, given the previous variable declarations and assuming variable **i** is an **integer**, this fragment

```
i := 1;
unpack(petite, grande, i);
```

tells **unpack** to begin copying into **grande[1]**. **Unpack** keeps copying until it has exhausted all of **petite**'s elements — in this case 10. **Unpack** always copies all of its *packed_array*'s elements, regardless of how many elements are defined for *unpacked_array*.

*Index* can take a value outside of *packed_array*'s defined subscripts. That is, if in the example above, **i** equals 50, **unpack** copies these values this way:

```
grande[50]:= petite[1];
grande[51]:= petite[2];
          .
          .
          .
grande[59]:= petite[10];
```

See the listing for **pack** earlier in this encyclopedia.

## EXAMPLE

```
PROGRAM unpack_example;
{This program demonstratest the use of the unpack procedure}
 TYPE
     uarray = array[1..50] of integer16;
     parray = packed array[1..10] of integer16;
 VAR
     full_range : uarray;
     sub_range  : parray;
     i, j       : integer16;

 BEGIN
 for i := 1 to 10 do
     sub_range[i] := i;
 j := 30;

 UNPACK(sub_range, full_range, j);
 writeln ('The unpacked array now contains: ');
 for i := 30 to 39 do
     writeln ('full_range[', i:2, '] = ', full_range[i]:2);

 END.
```

## USING THIS EXAMPLE

If you execute the sample program named **unpack_example**, you get the following output:

```
The unpacked array now contains:
full_range[30] =  1
full_range[31] =  2
full_range[32] =  3
full_range[33] =  4
full_range[34] =  5
full_range[35] =  6
full_range[36] =  7
full_range[37] =  8
full_range[38] =  9
full_range[39] = 10
```

**Until** Refer to **Repeat** earlier in this encyclopedia.

## Variable-Length String Operations

This section describes how to use variable-length strings in the code portion of your program. See Chapter 3 for information about declaring variable-length strings.

### USING VARIABLE-LENGTH STRINGS

You may assign a character constant, string constant, or character array to a variable-length string. You make assignments to variable-length strings in the same manner that you make assignments to **arrays of char**. For example:

```
VAR
     var_string    :    varying[20] of char;
     ar_of_char    :    array[1..20] of char;

 BEGIN
     var_string := 'test';
     ar_of_char := 'test';
           .
           .
           .
```

When you assign to a variable-length string, however, the unassigned elements of the array are not padded with spaces as they are for arrays of char. Unassigned elements maintain their old values. When you output a variable-length string with **write** or **writeln**, the procedure writes only the characters included in the string's current length. The following example illustrates this essential difference between variable-length and fixed-length arrays:

```
PROGRAM fixed_vs_varying_strings;

 VAR
     var_string   : VARYING[100] of CHAR;
     fixed_string : PACKED ARRAY[1...100] of CHAR;

 BEGIN
     fixed_string := 'string';
                              {length is always 100 - unassigned }
                              { chars are padded with spaces     }
     var_string := 'string';  { current length 6 }
     writeln(var_string);     { outputs only six characters }
     writeln(fixed_string);   { outputs 100 characters }
     var_string := 'longer string'; {current length is 13 }
 END.
```

## Trailing Null Character

The Pascal compiler appends a null character to a variable–length string under the following conditions:

- A string constant is assigned to a variable–length string.
- A character constant is assigned to a variable–length string.
- An array is assigned to a variable–length string.
- A string is assigned to a variable–length string via a **read** call.

The trailing null byte facilitates communication with routines written in other languages, such as C, that expect strings to end with a terminating null character. Note, however, that the current length of the string does *not* include the terminating null character, so the null character is invisible to Pascal routines that do not access the *body* field directly. For example:

```
PROGRAM null_char;

 VAR
     var_string  :   VARYING[80] of CHAR;

 BEGIN
     var_string := 'string';
         { null char is appended, but current length is 6 }
     writeln(var_string);      { Does not output null char }
     writeln(var_string.body[7]); { Outputs null character }
 END.
```

Note that you can inadvertently overwrite the null character by accessing the *length* or *body* fields directly:

```
 var_string := 'string';
 var_string.length := 3;  { terminating character is 'i', not null }
 var_string.body := 'longer string';  { null character is overwritten
                                        and body is padded with spaces }
```

For routines that communicate with C functions, you should be careful about directly changing *length* and *body* fields. To be on the safe side you should use the **ptoc** function, which explicitly appends a null character to a variable–length string. (See the description of **ptoc** in Chapter 4 for an example of passing a variable–length string to a C function.)

## Variable–Length String Operations

### ASSIGNING TO AND FROM VARIABLE–LENGTH STRINGS

When you make an assignment to a variable–length string, the compiler automatically promotes the assigned object to a variable–length string. Promotion of string constants occurs at compile time so it does not affect execution speed. Promotion of characters to variable–length strings occurs at run time, but is relatively efficient. Promotion of character arrays to variable–length strings, however, can have a noticeable impact on execution speed. If execution speed is a serious concern, therefore, you should avoid situations in which character arrays must be promoted to variable–length strings.

The compiler never demotes variable–length strings to character arrays. To treat a variable–length string as a normal array of characters you should reference its *body* field.

The following sections contain more detailed information about legal assignments involving variable–length strings.

#### varying := varying

Only those characters included in the current length of the source string (plus the trailing null character) are copied to the destination string. Note that the compiler does not check to see whether the trailing byte is in fact a null character. It simply copies a string equal to one greater than the current length. The length of the destination string is set equal to the current length of the source string (the trailing null character is not included in the length).

If the compiler option **–subchk** is off (default), the compiler copies the source string to the destination array without checking bounds. If **–subchk** is on, a run–time check verifies that the current length of the source string is not larger than the maximum length of the destination variable. If the string is too large, a run–time error occurs.

#### varying := array of char

The compiler promotes the character array to a variable–length string and assigns it to the destination string variable. It then appends a null character to the destination string. The length of the resulting string is the number of elements in the source character array (i.e., the length does not include the null character). A compile–time error occurs if the character array is larger than the maximum size of the destination string.

**varying := char**

The compiler assigns the character to the body field of the variable–length string and sets the *length* field equal to 1. A trailing null character is also assigned to the destination.

**Pointer Assignment**

Pointers to variable–length strings obey standard Pascal type compatibility rules — two pointers are compatible only if they share the same type name.

**Passing Variable–Length Parameters**

The type compatibility rules for variable–length strings that are procedure/function parameters are very similar to the rules for normal assignment. Characters, character arrays, character–string constants, and variable–length strings may be passed to **varying** parameters that are passed by value or as **in** parameters. The rules for compatibility are the same as if the argument were being assigned to a variable of the parameter's type. Arguments passed to variable–length string parameters by reference (**var, out** or **in out**) must be variable–length strings with exactly the same maximum size as the parameter.

**Comparing Variable–Length Strings**

You can compare two variable–length strings for equality, inequality, and lexicographic order. The lengths of strings being compared depend solely on the strings' current *length* values. The lengths are independent of the strings' maximum sizes, as declared at compile time.

If two strings being compared have different lengths, the comparison is performed as if the shorter string were padded with spaces to make its length equal to the longer string.

**While** Executes the statements within a loop as long as the specified condition is true.

## FORMAT

**while** *condition* **do** {**while** is a statement.}
*stmnt*;

## ARGUMENTS

| | |
|---|---|
| *condition* | Any Boolean expression. |
| *stmnt* | A simple statement or a compound statement. (Refer to "Statements" earlier in this encyclopedia.) |

## DESCRIPTION

**For, repeat,** and **while** are the three looping statements of Pascal. With **while,** you specify the *condition* under which Pascal continues looping.

**While** marks the beginning of a loop. As long as *condition* evaluates to true, Pascal executes *stmnt*. When *condition* becomes false, Pascal transfers control to the first statement following the loop.

To jump out of a **while** loop prematurely (i.e., before the condition is true), do one of the following things:

- Use **exit** to transfer control to the first statement following the **while** loop.
- Use **goto** to transfer control outside the loop.

In addition to these measures, you can also call the **next** statement to skip the remainder of the statements in the loop for one iteration.

## EXAMPLE

```
PROGRAM while_example;
 { This program contains two while loops. }
 { Compare it to repeat_example.          }
 VAR
     num            : integer16;
     test_completed : boolean;
     i              : integer32;

 BEGIN

     write('Enter an integer -- '); readln(num);
     WHILE (num < 101) DO
         BEGIN
         num := num + 10;
         writeln(num, sqr(num));
         END;

     writeln;
     test_completed := false;
     WHILE test_completed = false DO
         BEGIN
         write('Enter an integer (enter a 0 to stop the program) -- ');
         readln(i);
         if i = 0 then
             test_completed := true
         else
             writeln('The absolute value of ', i:1, ' is ', abs(i):1);
         END;
 END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **while_example**:

```
Enter an integer -- 70
         80       6400
         90       8100
        100      10000
        110      12100

 Enter an integer (enter a 0 to stop the program) -- 4
 The absolute value of 4 is 4
 Enter an integer (enter a 0 to stop the program) -- -5
 The absolute value of -5 is 5
 Enter an integer (enter a 0 to stop the program) -- 0
```

Now, consider a second run of **while_example**. In contrast to **repeat_example**, **while_example** does not execute the loop even once when x > 101.

```
 Enter an integer -- 102

 Enter an integer (enter a 0 to stop the program) -- 0
```

**With** Lets you abbreviate the name of a record.

## FORMAT

**With** is a statement that takes the following format:

**with** *v1, v2, ... vN* **do**
        *stmnt*;

This format is equivalent to:

```
with v1 do
        with v2 do
                .
                .
                .
                with vN do
                        stmnt;
```

## ARGUMENTS

| | |
|---|---|
| *v1* | A record expression; that is, *v1* must evaluate to a record. For example, it might be the name of a record, a pointer to a record, or a particular component in an array of records. |
| *v2, ... vN* | Optional record references or references that are qualified by *v1, ... v(N–1)*. |
| *stmnt* | A simple or compound statement. (Refer to the "Statements" listing earlier in this chapter.) |

## DESCRIPTION

Use **with** to abbreviate a reference to a field in a record. **With** works in the following manner. Suppose that **X** is a field within record **MATHVALUES**. Ordinarily, you must specify the full name **MATHVALUES.X** whenever you want to refer to the contents of field **X**. However, by using the statement

```
WITH MATHVALUES
```

you can simply specify **X** to refer to the contents of the field. Moreover, suppose that record **MATHVALUES** contains a field called **TRIGONOMETRY** which itself contains a field named **Y**. By specifying

```
WITH MATHVALUES, TRIGONOMETRY
```

you can refer to **Y** as **Y** rather than as **MATHVALUES.TRIGONOMETRY.Y**. Note that Domain Pascal evaluates the expression *v1* only once, and this evaluated expression is implied within the body of the **with** statement.

Now consider a fragment demonstrating **with**:

```
TYPE
    p = ^basketball_team;
 VAR
    bb = basketball_team;
 BEGIN
    WITH p^ DO
        BEGIN
             mascot    := 'tiger';
             p         := nil;
             height    := 198.2;
             bb.mascot := 'lion';
        END;
 .
 .
 .
```

Note two things about the preceding example. First, changing **p** does not affect access to the record identified by the **with** statement. Second, you can reference other records of the same type by completely qualifying the reference.

## Extension to Standard Pascal

Domain Pascal supports the standard format of **with** and also supports the following alternative format:

**with** *v1*:*identifier1*, *v2*:*identifier2*, ... *vN*:*identifierN* **do**
      *stmnt*;

This is very similar to the standard format for **with**. In this extension, the *identifier* is a pseudonym for the record reference *v*. To specify a record, use the *identifier* instead of the record reference *v*. Furthermore, to specify a field in a record, use *identifier.field_name* rather than merely *field_ name*.

For example, given the following record declaration:

```
VAR
    basketball_team : record
       mascot    : array[1..15] of char;
       height    : single;
    end;
```

consider the following three methods of assigning values:

```
readln(basketball_team.mascot);      {Not using WITH.}
readln(basketball_team.height);      {Not using WITH.}

WITH basketball_team DO         {Using standard WITH.}
  begin
       readln(mascot);
       readln(height);
  end;

WITH basketball_team : B DO     {Using extended WITH.}
   begin
       readln(B.mascot);
       readln(B.height);
   end;
```

This feature is useful for working with long record names when two records contain fields that have the same names. (See the example at the end of this listing.)

## EXAMPLE

```
PROGRAM with_example;

 { This program demonstrates the WITH statement. }

 TYPE
     name = record
         first : array[1..10] of char;
         last : array[1..14] of char;
     end;
     documentation_department = record
         their_name         : name;
         current_project    : string;
     end;

 VAR
     our_technical_writers, our_editors : documentation_department;
```

```
BEGIN

  writeln('In this routine, you enter data about Apollo documentors.');

{ First, we demonstrate the standard use of WITH. }
  WITH our_technical_writers, their_name DO
  BEGIN
    write('Enter the first name of the writer -- ');
    readln(first);

    write('Enter the last name of the writer -- ');
    readln(last);

    write('Enter a brief description of his or her current project--');
    readln(current_project);
  END; {with}

    writeln;
{ Next, we demonstrate the Domain Pascal extensions to WITH.     }
{ Use of the identifiers W and E permits a distinction  between }
{ the records inside the scope of the WITH statement.           }
    WITH our_technical_writers : W, our_editors : E  DO
       BEGIN
          write('Enter the first name of the editor -- ');
          readln(E.their_name.first);

          write('Enter the last name of the editor -- ');
          readln(E.their_name.last);

          E.current_project := W.current_project;
       END;
END.
```

## USING THIS EXAMPLE

This program is available online and is named **with_example**.

**Write, Writeln** Writes the specified information to the specified file (or to the screen).

## FORMAT

**write**(*f, exp1:field_width, ..., expN : field_width*)
{**write** and **writeln** are procedures.}

and

**writeln**(*f, exp1:field_width, ..., expN : field_width*)

## ARGUMENTS

*f* A variable having either the **text** or **file** data type. *F* is optional. If you do not specify *f*, Domain Pascal writes to standard output (**output**) which is usually the transcript pad. (Note that **output** has a **text** data type.)

*exp* One or more expressions separated by commas. An expression can be any of the following:

- A **string** constant
- An **integer, real, double, char, Boolean,** or **enumerated** expression
- An element of an array (assuming the element is one of the previous types listed)
- The name of an array variable whose base type is **char**

Note that *exp* cannot be a set variable.

*field_width* An integer expression that specifies the number of characters that **write** or **writeln** uses to output the value of this arg. *Field_width* is optional. Its effect depends on the data type of the *exp* to which it applies. (We detail these effects in the next section.) Note that you can specify *field_width* only if the *f* has the **text** data type.

## DESCRIPTION

**Write** and **writeln** are output procedures. (**Put** is also an output procedure; see the **put** listing earlier in this encyclopedia.) **Write** and **writeln** both write the values of arguments

**exp1** through *expN* to the file specified by *f*. At run time, Pascal writes the value of *exp1* first, the value of *exp2* second, and so on until *expN*.

**Write** and **writeln** are identical in syntax and effect except that **writeln** appends a newline character after writing the **exps** but **write** does not. In addition, when using **write**, *f* can have a **file** or **text** data type; however, when using **writeln**, *f* must have a **text** data type only.

Before calling **write** or **writeln** to write to an external file, you must open the file for writing. Chapter 8 details this process. Note that you do not need to open the standard output (**output**) file before writing to it.

Following the call to **write**, *f*^ is totally undefined.

The following paragraphs explain the output rules that Domain Pascal uses to print the value of an exp.

### Char Variables, Array of Char Variables, Variable-Length Strings, and String Constants

The following list shows the default *field_width*s for **char** variables, array of **char** variables, variable-length strings, and string constants:

- If **exp** is a **char** variable, the default *field_width* is 1.
- If **exp** is an array of **char** variable, the default *field_width* is the declared length of the array. For example, if you declare an array named **Oslo_array** as

  ```
  Oslo_array : array[1..10] of char;
  ```

  then the default *field_width* is 10.
- If **exp** is a variable-length string, the default *field_width* is the current length of the string.
- If **exp** is a string constant, the default *field_width* is the number of characters in the string.

If you do specify a *field_width*, here's how **write** and **writeln** interpret it:

| field_width | What Domain Pascal Does |
|---|---|
| = default | Writes a value with no leading or trailing blanks. |
| > default | Adds leading blanks. |
| < default | Truncates the excess characters at the end of the array or string. |
| = -1 | Truncates any trailing blanks in the array. Standard Pascal issues an error if you specify a negative *field_width*. (Negative values are legal only for types **char** and **string**.) |

For example, notice how the *field_width*s in the following **writeln** statements affect output. (The first two lines of output form a column ruler to help you notice columns.)

```
Domain Pascal Statements                                 Output

 VAR
 name      : array[1..20] of char;
 name_var  : varying[20] of char;
 grade     : char;


 BEGIN
    name := 'Zonker Harris';
    name_var := 'Zonker Harris';
    grade := 'F';
                                                  1         2         3
                                         123456789012345678901234567890
    WRITELN(name, grade);                Zonker Harris       F
    WRITELN(name_var, grade);            Zonker HarrisF
    WRITELN(name:-1, grade);             Zonker HarrisF
    WRITELN(name_var:-1, grade);         Zonker HarrisF
    WRITELN(name:4, grade);              ZonkF
    WRITELN(name_var:4, grade);          ZonkF
    WRITELN(name:25, grade);                  Zonker Harris       F
    WRITELN(name_var:25, grade);                     Zonker HarrisF
```

## Integer Values

The default *field_width* for an integer value is 10 spaces. This default applies to **integer**, **integer16**, **integer32**, and subrange variables, and to elements of an array that have one of these types as a base type. It also applies to record fields that have one of the aforementioned types.

If you specify a *field_width* greater than the number of digits in the integer value, Domain Pascal prints the value with leading blanks.

If you specify a *field_width* less than or equal to the number of digits in the integer value, Domain Pascal writes the value without leading or trailing blanks. Note that specifying a *field_width* never causes Domain Pascal to truncate the written value.

For example, consider an **integer16** variable named **small_int** with a value of 452 and an **integer32** variable named **large_int** with a value of 70,600,100. Notice how the *field_widths* in the following **writeln** statements affect output. (The first two lines of output form a column ruler to help you notice columns.)

```
Domain Pascal Statements                          Output

                                                           1         2         3
                                                  123456789012345678901234567890
 WRITELN(small_int);                                     452
 WRITELN(large_int);                                70600100
 WRITELN(small_int:5);                              452
 WRITELN(small_int:1);                            452
```

**Real Values**

For real **exp** you can supply no *field_width*, a one-part *field_width*, or a two-part *field_width* with a colon separating the two parts. Here are the rules:

- If you don't supply a *field_width*, Domain Pascal uses 13 spaces to write a single-precision value and 22 spaces to write a double-precision value.
- If you supply a one-part *field_width*, Domain Pascal adds or removes digits from the fractional part.
- If you supply a two-part *field_width*, Domain Pascal interprets the first part of the *field_width* as the total number of characters to print and the second part of the *field_width* as the number of digits to print following the decimal point. Note that the second part of the *field_width* has priority over the first part. For instance, suppose that you request a total width (the first part) of 5 characters and a fractional width (the second part) of 7 characters. Since Domain Pascal cannot satisfy both parts, it will satisfy only the second part.

If you don't supply a two-part *field_width*, Domain Pascal always leaves one leading space for positive numbers; none for negative numbers.

If there is not enough room for all the digits in the number, Domain Pascal rounds the value rather than truncating it.

For example, suppose that a single-precision real variable named **velocity** has a value of 43.54893. The following table shows how various **writeln** statements affect output. (The first two lines of output form a column ruler to help you notice columns.)

```
Domain Pascal Statements                     Output

                                             1         2         3
                                    123456789012345678901234567890

 WRITELN(velocity);                  4.354893E+01
 WRITELN(velocity:20);               4.3548930000000E+01
 WRITELN(velocity:1);                4.4E+01
 WRITELN(velocity:15:4);                    43.5489
 WRITELN(velocity:7:4);             43.5489
 WRITELN(velocity:7:2);               43.55
 WRITELN(velocity:3:0);             44.
 WRITELN(velocity:1:5);             43.54893
```

### Enumerated and Boolean Values

Domain Pascal keeps the same rules for writing enumerated and Boolean values. For both types, the default *field_width* is 15. Here's what happens if you specify your own *field_width*:

- If you specify a *field_width* less than 15, Domain Pascal subtracts a suitable number of leading blanks.
- If you specify a *field_width* greater than 15, Domain Pascal adds a suitable number of leading blanks.

Note that Domain Pascal never truncates any of the characters in the value (even if the *field_width* is less than the number of characters).

## Write, Writeln

The following example shows how various *field_widths* affect output. (The first two lines of output form a column ruler to help you notice columns.)

```
Domain Pascal Statements                       Output

 VAR
    colors : (red, brown, magenta);
    evil   : boolean;
                                               1         2         3
                                      123456789012345678901234567890
 BEGIN
    colors := brown;
    WRITELN(colors);                            BROWN
    WRITELN(colors:8);                   BROWN
    WRITELN(colors:1);                BROWN

    evil := true;
    WRITELN(evil);                               TRUE
    WRITELN(evil:2);                  TRUE
```

### EXAMPLE

```
PROGRAM write_example;
{ This example reads one input line from the keyboard and writes it to }
{ filename 'truth'.  Then it writes a message to the display.          }
CONST
   pathname = 'truth';
VAR
   a_line   : string;
   wisdom   : text;
   statint  : integer32;
BEGIN
   open(wisdom, pathname, 'NEW', statint);
   if statint <> 0
      then return
      else rewrite(wisdom);
   WRITE('Enter a sentence of truth -- ');
   readln(a_line);
   WRITELN(wisdom, a_line);
   close(wisdom);
   WRITELN(chr(10), chr(10), chr(9), 'Thank You', chr(7));
   { ASCII 10 is a line feed.  ASCII 9 is a tab.  ASCII 7 is the bell. }
END.
```

### USING THIS EXAMPLE

This program is available online and is named **write_example**.

**Xor** Returns the exclusive **or** of two integers. (Extension)

## FORMAT

**xor**(*int1, int2*) {**xor** is a function.}

## ARGUMENTS

| | |
|---|---|
| *int1, int2* | Integer expressions. |

## FUNCTION RETURNS

The **xor** function returns an integer value.

## DESCRIPTION

Use the **xor** function to take the bitwise exclusive **or** of *int1* and *int2*. The **xor** function belongs to the bitwise class consisting of **&**, **!**, and **~**, and *does not* belong to the Boolean operator class consisting of **and**, **or**, and **not**. When matching bits for an **xor** function, Domain Pascal uses the following truth table:

*Table 4-14. Truth Table for* xor *Function*

| bit x of op1 | bit x of op2 | bit x of result |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

## EXAMPLE

```
PROGRAM xor_example;

 {This program finds the exclusive or of two integers by using XOR. }

 VAR
    i1, i2, result : integer16;

 BEGIN
    write('Enter an integer -- '); readln(i1);
    write('Enter another integer -- '); readln(i2);
    result := XOR(i1, i2);
    writeln('The exclusive or of ', i1:1, ' and ', i2:1, ' is ',
             result:1);
 END.
```

## USING THIS EXAMPLE

Following is a sample run of the program named **xor_example**:

```
Enter an integer -- 6
Enter another integer -- 20
The exclusive or of 6 and 20 is 18
```

# Chapter 5

# Procedures and Functions

This chapter explains how to declare and call procedures and functions. The term **routine**, which appears throughout this chapter, means either procedure or function. The terms **parameter** and **argument** also appear throughout this chapter. In this case, **argument** means the data passed to a routine, while **parameter** means the templates for the data that the called routine receives. (Another term for argument is **actual parameter**; another term for parameter is **formal parameter**.)

Chapter 2 mentioned that routine headings take the following format:

[*attribute_list*] **procedure** *name* [*(parameter_list)*]; [*routine_options*;]

or

[*attribute_list*] **function** *name* [*(parameter_list)*] : *typename*; [ *routine_options*;]

This chapter details the *parameter_list, routine_options*, and *attribute_list*.

## 5.1 Parameter List

You can declare a routine with or without parameters. If you declare it without parameters, you cannot pass any arguments to the routine. If you declare it with parameters, you must specify the data type of each argument that can be passed to the routine.

You specify parameters within a parameter list. You can specify a maximum of 65 parameters within the list. A parameter list has the following format:

```
([param_type1 ] par_list1 : typename1;
 [   . . .;
     . . .;
     . . .;
 [param_typeN ] par_listN : typenameN]);
```

The *param_type* (or **mode**) is optional; for information, see the "Parameter Types" section later in this chapter.

A *par_list* consists of one or more parameters that have the same data type. Thus, the combination

*par_list* : *typename*

is similar to a variable declaration. Like a variable declaration, each parameter in *par_list* must be a valid identifier. Also, each data type must be a predeclared Domain Pascal or user-defined data type. That is, you cannot specify an anonymous data type. (See the "Var Declaration Part" section in Chapter 2 for more information on anonymous data types.)

You can use a parameter in the action part of the routine just as you would use any variable. Consider the following sample routine declarations:

```
{ Declare a procedure with no parameter list. }
     Procedure simple;


 { Declare a procedure with a parameter list that has two parameters. }
     Procedure con(a : integer;
                   b : real);


 { Declare a function with a parameter list that has two parameters }
 { sharing the same data type. }
     Function anger(x,y : boolean) : integer16;


 { Declare a function with a parameter list that has three parameters. }
     Function big(quart  : integer16;
                  volume : real;
                  cost   : single) : single;
```

The following routine declaration is wrong because it uses an anonymous data type:

```
Procedure range(small_range : 0..10);          {WRONG!}
```

To call a routine, simply specify its name. If the procedure has a parameter list, you must also specify arguments. The data type of each argument must match the data type of its corresponding parameter. For example, if the second parameter is declared as **integer16**, the second argument must be an **integer16** value. The following examples call the routines that were previously declared:

```
{ Call simple with no arguments. }
      simple;


 { Call con with two arguments. The first argument must be an integer, }
 { and the second argument must be a real number. }
      con(14, 5.2);


 { Call anger with two Boolean arguments. Variable answer must have    }
 { been declared as an integer.                                         }
      answer := anger(true, false);


 { Call big with three arguments. Assume that pints is an integer and  }
 { price is a real (single) number.                                     }
     if big(pints * 2, 4.23E3, price) > 1.40
        then ...
```

Domain Pascal supports many features that let you specify precisely how a routine is to be called. The remainder of this chapter describes these features in detail after first describing argument passing conventions.

## 5.2 Argument Passing Conventions

A program or procedure can pass arguments to another procedure by **value** (the actual value of the argument) or by **reference** (the address of the argument). By default, Pascal passes all arguments of externally called routines by reference regardless of the parameter type. However, you can pass arguments by value rather than by reference if you use the **val_param** option in the procedure or function heading. In addition, the **c_param** routine option tells the compiler to return function results in register D0 (on 680x0 workstations), and to pass all record data types by value rather than by reference. This facilitates cross-language calling with C and FORTRAN. (For more information on **val_param**, see Section 5.5.7. For more information on **c_param**, see Section 5.5.12.)

In an external call, even arguments with call-by-value semantics (which is the default) are actually passed by reference. For example, consider the following code fragment:

```
PROCEDURE dump (size: integer);
    .
    .
    .
    IF size > maxsize THEN size := maxsize;
    .
    .
    .
```

Pascal semantics require that the change to **size** not affect the value of the caller's argument. The code that the Pascal compiler generates for this call takes care of this, by making a local copy of the argument that can be modified.

Internal routines can be called only within the same compilation unit in which they are defined. Since the compiler knows all the calls to the routine, it can optimize argument passing without regard to the argument passing conventions. Currently, however, internal routines are treated the same as **val_param** routines.

Table 5-1 summarizes Domain Pascal argument passing conventions.

> **NOTE:** All character arrays are treated like arguments with fewer than four bytes, regardless of their size.
>
> Double-precision reals on Series 10000 workstations are treated like arguments with four bytes or fewer.

Table 5-1. Argument Passing Conventions

| Mode | Default Caller | Default Callee | Modifiers: Arguments > 4 bytes | Modifiers: Arguments <= 4 bytes |
|---|---|---|---|---|
| IN | Passes addresses of arguments. | Makes indirect references to arguments. | **val_param**: No effect<br>**internal**: No effect<br>**UNIV**: No effect | **val_param** or **internal**: Causes the caller to pass the value of the argument and the callee to make direct references to it.<br>**UNIV**: Cancels the effect of **var_param** and **internal**. |
| IN OUT<br>VAR | Passes addresses of arguments. | Makes indirect references to arguments. | **val_param**: No effect<br>**internal**: No effect<br>**UNIV**: No effect | **val_param**: No effect<br>**internal**: No effect<br>**UNIV**: No effect |
| No mode | Passes addresses of arguments. | Creates local copies of arguments and makes direct references to them. | **val_param**: No effect<br>**internal**: No effect<br>**UNIV**: No effect, but considered bad programming practice. Since the callee does not know the type of the incoming argument, it makes a local copy as described by the parameter. | **val_param** or **internal**: Causes the caller to pass the value of the argument and the callee to make direct references to it.<br>**UNIV**: Cancels the effect of **val_param** and **internal**. Considered bad programming practice. Since the callee has no idea of the incoming argument's type, it makes a local copy as described by the parameter. |

## 5.3 Parameter Types

A *param_type* (or mode) is optional. If you do not include one, you are in effect passing a value parameter. Value parameters are discussed in this section. If you want to specify a *param_type*, it must be one of the following:

- **var** (a variable parameter)
- **in**

- **out**
- **in out**

The following subsections describe each of these.

## 5.3.1 Variable Parameters and Value Parameters

In standard Pascal, you pass arguments to and from routines as variable parameters or value parameters. Domain Pascal supports both methods plus certain extensions described later in this subsection. The following examples illustrate the distinction between variable parameters and value parameters.

Pascal regards variable parameters as synonyms for the variable you pass to them. In Figure 5-1, a program named **var_parameter_example**, variable parameter **y** becomes a synonym for argument **x**; that is, whatever happens to **y** in **addc** happens also to **x** (and vice versa). You must pass a variable as an argument to a variable parameter. You cannot pass a value.

Pascal does not regard value parameters as synonyms to the arguments you pass to them. In Figure 5-2, a program named **value_parameter_example**, value parameter **y** takes on a copy of the value of **x** within **addc**; therefore, whatever happens to **y** in **addc** has no effect on the value of **x**. Note that you can pass variables, values, or expressions as arguments to a routine with value parameters.

Both sample programs are available online.

```
{**********************************************************************}
PROGRAM var_parameter_example;
{ Compare this program to value_parameter_example. }
VAR
    x : integer16;
PROCEDURE addc(VAR y : integer16); { y is a variable parameter. }
BEGIN
   y := y + 100;
   writeln('In addc, y=',y:4);
END;
BEGIN
   x := +10;
   addc(x);
   writeln('In main, x=',x:4);
END.
{**********************************************************************}
```

*Figure 5-1. Program Illustrating Variable Parameters:* **var_parameter_example**

```
{*************************************************************************}
PROGRAM value_parameter_example;
{ Compare this program to var_parameter_example. }
VAR
    x : integer16;
PROCEDURE addc(y : integer16); { y is a value parameter. }
BEGIN
   y := y + 100;
   writeln('In addc, y=',y:4);
END;
BEGIN
   x := +10;
   addc(x);
   writeln('In main, x=',x:4);
END.
{*************************************************************************}
```

*Figure 5-2. Program Illustrating Value Parameters:* **value_parameter_example**

The results of the sample programs in Figure 5-1 and Figure 5-2 are shown below.

```
Execution of                          Execution of
var_parameter_example                 value_parameter_example

In addc, y= 110                        In addc, y= 110
In main, x= 110                        In main, x=  10
```

The only difference between the two programs is the keyword **var** in the procedure declaration statement of **var_parameter_example**. This keyword identifies **y** as a variable parameter; the absence of **var** identifies **y** as a value parameter.

## 5.3.2 In, Out, and In Out—Extension

In standard Pascal, you cannot specify the direction of parameter passing. However, Domain Pascal supports extensions to overcome this problem. You can use the following keywords in your routine declaration:

- **In**—This keyword tells the compiler that you are going to pass a value to this parameter, and that the routine is not allowed to alter its value. If the called routine does attempt to change its value (that is, use the parameter on the left side of an assignment statement), the compiler issues an "Assignment to IN argument" error.

- **Out**—This keyword tells the compiler that you are not going to pass a value to the parameter, but that you expect the routine to assign a value to the parameter. It is incorrect to try to use the parameter before the routine has assigned a value to it, although the compiler does not issue a warning or error in this case.

If the called routine does not attempt to assign a value to the parameter, the compiler may issue a "Variable was not initialized before this use" warning. This could occur if your routine assigns a value to the parameter only under certain conditions. If that is the case, you should designate the parameter as **var** instead of **out.**

In some cases, the compiler cannot determine whether or not all paths leading to an **out** parameter assign a value to it. If that happens, the compiler does not issue a warning message.

- **In out**—This keyword tells the compiler that you are going to pass a value to the parameter, and that the called routine is permitted to modify this value. It is incorrect to call the routine before assigning a value to the parameter, although the compiler does not issue a warning or error in this case. The compiler also doesn't complain if the called routine does not attempt to modify this value.

For example, consider the program shown in Figure 5-3, which is also available online.

```
{***********************************************************************}
PROGRAM in_out_example;
 {This program demonstrates the IN, OUT, and IN OUT parameters.}
 VAR
     leg1, leg2     : integer16;
     hypotenuse : single;
     temp   : real;
     unit   : char;


     PROCEDURE pythagoras(IN         leg1 : integer16;
                          IN         leg2 : integer16;
                          OUT  hypotenuse : single);
     BEGIN
        hypotenuse := sqrt((leg1 * leg1) + (leg2 * leg2));
     END;


     FUNCTION boiling(IN OUT  temp : real;
                      IN      unit : char) : boolean;
     BEGIN
         if unit = 'F'
            then temp := (temp - 32) * 0.55555;
         if temp >= 100
            then boiling := true
         else boiling := false;
     END;


 BEGIN
    write('Enter the length of a leg of a right triangle --');
    readln(leg1);
    write('Enter the length of the other leg --'); readln(leg2);
    pythagoras(leg1, leg2, hypotenuse);
    writeln('Hypotenuse of the triangle is ', hypotenuse);

    writeln(chr(10), chr(10), 'Assume 1 Atm. pressure');
    write('Enter the water temperature --'); readln(temp);
    write('Is this temp. in Fahrenheit or Celsius (F or C) -- ');
    readln(unit);

    if boiling(temp, unit)
        then writeln(temp:5:1, ' degrees C water will boil!')
        else writeln(temp:5:1, ' degrees C water will not boil.');
 END.
{***********************************************************************}
```

*Figure 5-3. Program Illustrating* **in**, **out**, *and* **in out** *Value Passing:* **in_out_example**

**NOTE:** The compiler checks for misuses of **in**, **out**, and **in out** at compile time, but the system does not check for such errors at run time.

### 5.3.3 Univ—Universal Parameter Specification—Extension

**Univ** is a special parameter type that you specify immediately prior to the *typename* (rather than prior to the *par_list*). You use **univ** to pass an argument that has a different data type than its corresponding parameter.

By default, Domain Pascal checks that the argument you pass to a routine has the same data type as the parameter you defined for the routine. However, you can tell Domain Pascal to suppress this type checking by using the keyword **univ** prior to a type name in a parameter list.

**Univ** is especially useful for passing arrays. For example, the following program would be incorrect without the keyword **univ**. That's because **little_array** and **big_array** have different data types:

```
TYPE
     big_array     = array[1..50] of integer32;

 VAR
     large_array  : array[1..50] of integer32;
     medium_array : array[1..25] of integer32;
     little_array : array[1..10] of integer32;
     sum          : integer32;

     Procedure sum_elements(in          b : UNIV big_array;
                            in array_size :      integer16;
                            out       sum :      integer32);
     BEGIN
        . . .
     END;

 BEGIN  {main}
    sum_elements(little_array, 10, sum);
 END.
```

In addition to the procedure call listed above, you could also make either of the following calls to procedure **sum_elements**:

```
sum_elements(medium_array, 25, sum);
```

or

```
sum_elements(large_array, 50, sum);
```

Use **univ** carefully! It can cause problems if improperly used. The most frequent source of trouble is a difference in size between the argument and parameter data types. The data type of the parameter determines how the called routine treats the data passed to it. Typically, routines that use **univ** parameters have another parameter that supplies additional information about the size or type of the argument. In the preceding example, the **array_size** parameter gives the size of the array parameter passed. In addition, you should not pass an argument that is larger than the parameter. If you do, your program may produce unexpected results. The following example shows this misuse of **univ**:

```
Program univ_example;                        {POOR USE OF UNIV!!!}

 {This example demonstrates poor use of UNIV.}
 {The program uses UNIV to pass two double-precision arguments to two}
 {single-precision parameters.  The calculation of 'mean' will not be}
 {correct because single- and double-precision real numbers have     }
 {different bit patterns for exponent and mantissa.  Furthermore,    }
 {the compiler will not warn you about this problem.                 }

 VAR
    first_value, second_value : double;

      Procedure average(s,t : UNIV single);
      VAR
         mean : double;
      BEGIN
         mean := (s + t) / 2.0;
         writeln('The average is ', mean);
      END;

 BEGIN {main}
     write('Enter the first  value --'); readln(first_value);
     write('Enter the second value --'); readln(second_value);
     average(first_value, second_value);
 END.
```

**NOTE:** To prevent some problems that result from suppressing type checking, explicitly declare **univ** parameters as **in**, **out**, **in out**, or **var**.

When you pass an expression argument (as opposed to a variable argument) to a **univ** parameter, Domain Pascal extends the expression to be the same size as the **univ** parameter. 0 In addition, the compiler issues the following message:

```
Expression passed to UNIV formal NAME was converted to NEWTYPE.
```

## 5.3.4 Pointers to Routines—Extension

As noted in Chapter 3, Domain Pascal supports a special pointer data type that points to a procedure or a function. You can use these routine pointers in combination with the **addr** function to pass addresses of routines as parameters.

For example, the sample program in Figure 5-4, which is also available online, returns the square of the number provided by the user. It is a simple illustration of the use of a routine pointer as a parameter. Note that the **square** function must be in an external module. You cannot obtain the addresses of internal routines. See the "Procedure and Function Pointer Data Types—Extension" section of Chapter 3 for more details about declaring routine pointers.

```
{***********************************************************************}
PROGRAM pass_routine_ptrs;
{ This program prompts the user for a number and returns the square    }
{ of the number.  The procedure write_square uses a routine pointer    }
{ to call the external function square.  To run the pass_routine_ptrs }
{ program you must compile the square.pas module and then bind         }
{ together square.bin and pass_routine_ptrs.bin.                       }

TYPE
func_ptr = ^FUNCTION (x:integer) :integer;
                                        {declaration of routine pointer}
VAR
number: integer;
FUNCTION square (x:integer):integer; EXTERN;
PROCEDURE write_square(a_ptr:func_ptr; y:integer);

  BEGIN
  writeln('The square of the number is ',a_ptr^(y));
  END;

BEGIN
write('What number would you like to square?  ');
readln(number);
write_square (addr(square), number);
END.


MODULE square;
{This function is called by the pass_routine_ptrs program}

FUNCTION square (x:integer):integer;
BEGIN
square := x*x;
END;
{***********************************************************************}
```

*Figure 5-4. Program and Module Illustrating Pointers to Routines:* **pass_routine_ptrs** *and* **square**

Here is a sample run of the compiled and bound **pass_routine_ptrs** program:

```
What number would you like to square?  25
The square of the number is        625
```

#### 5.3.4.1 Data Type Checking

The compiler checks pointers to routines for data type name compatibility when addresses are assigned to a pointer or procedure. For example, assume you use the statements shown below to declare a data type, a pointer to a routine, and a procedure named **foo**:

```
TYPE
    pinteger = 0...65535;

VAR
    pptr: ^PROCEDURE (r: pinteger);

PROCEDURE foo (r: integer);
```

The type **pinteger** is defined as a subrange of base type **integer**. The pointer **pptr** points to a procedure that takes one parameter of type **pinteger**. The procedure **foo**, however, takes one parameter of type **integer**. Even though **pinteger** and **integer** are both 16-bit integers, the following assignment statement causes an error because **integer** and **pinteger** are not name compatible (that is, they do not have the same name):

```
pptr := addr(foo);
```

## 5.4 Procedures and Functions as Parameters

Any procedure or function can be a parameter for any other procedure or function. Procedure and function parameters must appear in the parameter list. If the function or procedure being passed has parameters, these parameters must also appear in the declaration. For example:

```
PROCEDURE caller (PROCEDURE callee (A, B : integer) );
```

Function parameters must specify the type they return, for example:

```
PROCEDURE caller (FUNCTION callee : integer);
```

A function or procedure that is a parameter may also contain functions or procedures as parameters; for example:

```
PROCEDURE caller (FUNCTION A (PROCEDURE B) : integer);
```

You pass the name of a procedure or function just as you would any other parameter. Only the name of the procedure or function is specified. Its parameter list, if any, does not appear here. For example:

```
caller (callee);
```

The following example uses a procedure as a parameter.

```
{***********************************************************************}
PROGRAM procedure_as_parameter;
{This program invokes a procedure that has another procedure as a
 parameter.}

VAR
    I : integer;

Procedure square;
    BEGIN
        I := I * I;
    END;

Procedure callproc (procedure A);
    BEGIN
        A;      {Invokes procedure A}
    END;

BEGIN  {main program}
    I := 3;
    callproc(square);       {Procedure square is the parameter.}
    writeln(I)              {I = 9}
END.
{***********************************************************************}
```

The following example uses a function as a parameter.

```
{***********************************************************************}
PROGRAM function_as_parameter;
{This program invokes a function that has another function as a
 parameter.}

VAR
    I, val : integer;

Function value : integer;
    BEGIN
        value := 4;             {Function value is 4.}
    END;

Function cube(Function a : integer) : integer;
    BEGIN
        cube := a * a * a       {Cubes the function value.}
    END;

BEGIN
    I := cube(value) ;          {Argument is Function value.}
    writeln(value);
    writeln(I);                 {I = 64}
END.
{***********************************************************************}
```

## 5.5 Routine Options

As mentioned in the beginning of this chapter, you can optionally specify *routine_options* at the end of the routine declaration. Domain Pascal supports the following routine options:

- **forward**
- **extern**
- **internal**
- **variable**
- **abnormal**
- **val_param**
- **nosave**
- **noreturn**
- **d0_return**
- **a0_return**
- **c_param**

### 5.5.1 Routine Option Syntax

Use the following format to specify any of the routine options:

**options**(*routine_option1*, . . . *routine_optionN*);

Here are some examples of this format:

```
FUNCTION  eggs_and_ham(letter : char) : char;  INTERNAL;

PROCEDURE sam_i_am(x, y : real);   OPTIONS(EXTERN, ABNORMAL);
```

You can use the shorter format shown below only for the **forward**, **extern**, **internal**, and **val_param** routine options:

*routine_option1*; . . . *routine_optionN*;

Here are some examples of the shorter format:

```
FUNCTION  eggs_and_ham(letter : char) : char;  INTERNAL;

PROCEDURE sam_i_am(x, y : real);   EXTERN; val_param;
```

The remainder of this section explains the routine options.

### 5.5.2 forward

The **forward** option is a feature of standard Pascal and Domain Pascal. By default, you can call only a routine that was previously declared in the program. The **forward** option identifies the procedure prior to both its use (call) and its definition.

Figure 5-5 is a program named **forward_example**, which is available online. In this program, the procedure **convert_degrees_to_radians** is declared as **forward**. This allows procedure **find_tangent** to call procedure **convert_degrees_to_radians** even though **find_tangent** precedes it in the file.

```
{***************************************************************************}
PROGRAM forward_example;
 {This program demonstrates the FORWARD option}

 Function convert_degrees_to_radians(d : real) : real; FORWARD;

 VAR
    degrees, tangent : real;

 Procedure find_tangent(IN  degrees : real;
                        out tangent : real);
 VAR
    radians : real;
 BEGIN
    radians := convert_degrees_to_radians(degrees);
    tangent :=  sin(radians)  /  cos(radians);
 END;

 Function convert_degrees_to_radians(d : real) : real;
 CONST
     degrees_per_radian = 57.2958;
 BEGIN
     convert_degrees_to_radians := (d / degrees_per_radian);
 END;

 BEGIN
    write('Enter a value in degrees -- ');
    readln(degrees);
    find_tangent(degrees, tangent);
    writeln('The tangent of ', degrees:6:3, ' is ', tangent:6:3);
 END.
{***************************************************************************}
```

*Figure 5-5. Program Illustrating the* **forward** *Option:* **forward_example**

If it were not for the **forward** option, the compiler would issue the following error:

```
CONVERT_DEGREES_TO_RADIANS has not been declared in routine FIND_TANGENT
```

Note that the program in Figure 5-5 declares function **convert_degrees_to_radians** and its parameters in the declaration part of the main program *and* in the routine heading.

Domain Pascal allows you to choose whether to include parameters in the routine heading. Thus, you could substitute the following routine heading for the one shown above:

```
Function convert_degrees_to_radians;  {No parameters included here.}
```

In any case, you must include the **forward** option in the declaration part of the main program.

## 5.5.3 extern—Extension

**Extern** is an extension to standard Pascal. It tells the compiler that the routine is possibly defined outside of this source code file. (The "Accessing a Variable Stored in Another Pascal Module" and "Accessing a Routine Stored in Another Pascal Module" sections of Chapter 7 detail **extern**. See also **define**, which is also detailed in the same sections of Chapter 7.)

## 5.5.4 internal—Extension

**Internal** is an extension to standard Pascal. By default, all top-level routines defined in a module become global symbols. But if you declare the routine with the **internal** option, the compiler makes the routine a local symbol. (The "Accessing a Variable Stored in Another Pascal Module" and "Accessing a Routine Stored in Another Pascal Module" sections of Chapter 7 detail **internal**.) Declaring routines "internal" results in slightly more efficient code and faster linking/loading.

## 5.5.5 variable—Extension

**Variable** is an extension to standard Pascal. By default, you must pass the same number of arguments to a routine each time you call the routine. However, the **variable** option allows you to pass a variable number of arguments to the routine. You may want to specify an argument count as the first parameter. There is one case in which you cannot use the **variable** extension. You must supply an argument if the called routine makes a local copy of any parameter. See Section 5.2 for more information on argument passing conventions.

Figure 5-6 is a sample program named **variable_attribute_example** that is available online. It illustrates the use of the **variable** option.

```
{**********************************************************************}
PROGRAM variable_attribute_example;

 {This program demonstrates the routine attribute called VARIABLE which}
 { allows you to pass a variable number of arguments to a routine.     }

 VAR
    first_value, second_value : real;
    precision : real;
    answer  : char;

 Procedure average(arg_count : integer16;
                  d1, d2    : real;
                  p : real);
                  options(VARIABLE);
                                 {We can pass up to four arguments.}
 VAR
    mean : real;
 BEGIN
    mean := (d1 + d2) / 2.0;
    if arg_count = 3
       then writeln('The mean is ', mean:4:1, ' to a precision of 0.1')
    else if (arg_count = 4) and (p = 0.01)
       then writeln('The mean is ', mean:4:2, ' to a precision of .01')
    else if (arg_count = 4) and (p = 0.001)
       then writeln('The mean is ', mean:4:3, ' to a precision of .001')
    else
       writeln('Improper argument count or precision');
 END;


 BEGIN {main}
     writeln('This program calculates the mean of two real numbers.');
     write('Enter the first  value --'); readln(first_value);
     write('Enter the second value --'); readln(second_value);
     write('Do you want to specify a precision (enter y or n) --');
     readln(answer);
     if answer = 'y' then
        begin
             write('Please enter the precision (.01 or .001) --');
             readln(precision);
             average(4, first_value, second_value, precision);
        end
     else    average(3, first_value, second_value);
 END.
{**********************************************************************}
```

*Figure 5-6. Program Illustrating the* **variable** *Option:* **variable_attribute_example**

## 5.5.6 abnormal—Extension

**Abnormal** is an extension to standard Pascal. It warns the compiler that a routine can cause an abnormal transfer of control. This option affects the way the compiler optimizes the calling routine, but does not affect the way the compiler optimizes the called routine (that is, the routine that is declared **abnormal**).

For example, the following use of **abnormal** causes the compiler to be careful about optimizing around any cleanup handler:

```
FUNCTION pfm_$cleanup (current_record: clean_up_record) :
                                       status_$T; OPTIONS(ABNORMAL);
```

## 5.5.7 val_param—Extension

**Val_param** is an extension to standard Pascal. By default, Pascal passes arguments by reference. This means that Domain Pascal passes the address of the argument rather than the value of the argument, regardless of the declared parameter type (**in**, **out**, **in out**, or **var**). When you use the **val_param** option in a procedure or function heading, you tell Domain Pascal to pass arguments by value when possible. Under the **val_param** option, all arguments four bytes or less in size (on 680x0 workstations) and 8-byte double-precision reals on Series 10000 workstations (except for character arrays) are passed by value, provided that they are declared as value parameters or as **in** parameters. All other arguments are passed by reference. See Table 5-1 for a summary of the use of **val_param** with different qualifiers.

This option produces more efficient calling sequences for Domain Pascal routines. However, when you are writing a routine that calls a Domain/C routine, you should use the **c_param** option. (See Section 5.5.12 for details about **c_param**.)

The following example illustrates a **val_param** option in a function declaration. The first declaration uses the standard syntax. The second uses the shorter syntax:

```
FUNCTION pass_value (letter : char) : char;  OPTIONS(EXTERN, VAL_PARAM);

FUNCTION pass_value (letter : char) : char;  EXTERN; VAL_PARAM;
```

## 5.5.8 nosave—Extension

**Nosave** is an extension to standard Pascal. You should use it with a Pascal program call to an assembly language routine that doesn't follow the usual conventions for preserving these registers:

- Data registers D2 through D7
- Address registers A2 through A4
- Floating-point registers FP2 through FP7

**Nosave** indicates that the contents of these registers will not be saved when the assembly language routine finishes executing and returns to the Pascal program. However, the assembly language routine must *always* preserve register A5, which holds the pointer to the current stack area. It also must *always* preserve A6, which holds the address of the current data area. That is, the called routine must preserve A5 and A6 even if you use **nosave**.

### 5.5.9 noreturn—Extension

**Noreturn** is an extension to standard Pascal. This routine specifies an unconditional transfer of control; once a procedure or function with **noreturn** is called, control can never return to the caller. The routine marked **noreturn** is executed, and the program terminates.

When you specify this keyword, the compiler may optimize the code it generates so that any return sequence or stack adjustments after the call to the routine marked **noreturn** are eliminated as being unreachable code.

### 5.5.10 d0_return—Extension

The **d0_return** option is an extension to standard Pascal. By default, a Pascal function returning the value of a pointer puts that value in address register A0. When you use the **d0_return** option, the compiler does the following:

- Puts the value of the returned pointer in A0 *and also in* data register D0.
- Causes routines that call a function marked **d0_return** to expect the value of the returned pointer variable to be in D0.

You should use **d0_return** in the heading for Pascal functions that call external C or FORTRAN routines because C and FORTRAN return function results in D0.

For example, consider the following routine heading:

```
function string_c : string_ptr; options (extern,d0_return);
```

The preceding declaration tells the compiler to expect the **string_ptr** type value returned by **string_c** to be in register D0.

**NOTE:** The second character in this option is a zero, not a capital O.

### 5.5.11 a0_return—Extension

The **a0_return** option is an extension to standard Pascal. It is allowed on any function signature. If you specify **a0_return** for a Pascal routine, the compiler does the following:

- Puts into A0 any values returned from that routine that should go into a register.
- Looks in A0 for any values returned to a caller of that routine.

**NOTE:** The second character in this option is a zero, not a capital O.

### 5.5.12 c_param—Extension

The **c_param** option is an extension to standard Pascal. Specifying **c_param** is equivalent to specifying **d0_return** and **val_param**. In addition, the **c_param** option tells the compiler to pass all record data types by value rather than by reference. For an example of this option, see Section 7.9.5.

Any Pascal procedure that calls or is called by a C program **must** use the **c_param** option to pass by value any single-precision or double-precision floating-point, record or **struct**, simple datum, or pointer.

The following example illustrates the use of a **c_param** option in a function declaration:

```
FUNCTION c_function (letter : char) : char;  OPTIONS(EXTERN, C_PARAM);
```

> NOTE: Using the **c_param** option does *not* guarantee that the size of arguments passed by value from a Domain Pascal routine will match the size of arguments in the Domain/C called routine. For example, when you pass Domain Pascal **integer, integer16, char** values to Domain/C, the compiler does *not* widen them to 32 bits as Domain/C does. Similarly, when you pass Domain Pascal **real** values, the compiler does *not* widen them to 64 bits. Thus, if you use **c_param**, you must make sure that you declare the parameters that you pass from Domain Pascal to Domain/C to be the correct size.

## 5.6 Defining Your Own Routine Options

In addition to the predeclared routine options, Domain Pascal supports a **routine_option** declaration part that allows you to define your own names for groups of routine options. Furthermore, Domain Pascal provides a special name—**default_routine_options**—that allows you to define the default routine options for every routine in a module. We describe both of these features in this section.

### 5.6.1 Syntax of the routine_option Declaration Part

The syntax for the **routine_option** declaration part is as follows:

**routine_option**
    *identifier1 = routine_option_name1*
        [, *routine_option_name2, . . ., routine_option_nameN*];
    [ *identifierN = routine_option_name1*
        [, *routine_option_name2, . . ., routine_option_nameN*];]

An *identifier* is any valid Domain Pascal identifier. A *routine_option_name* is the identifier of a routine option that you created earlier in the **routine option** declaration part or any of the following predeclared Domain Pascal routine options:

| | | | |
|---|---|---|---|
| **a0_return** | **abnormal** | **c_param** | **d0_return** |
| **noreturn** | **val_param** | **variable** | |

The following routine options *cannot* appear in a **routine_option** declaration part:

| | | |
|---|---|---|
| **extern** | **forward** | **internal** |

## 5.6.2 Examples of the routine_option Declaration Part

Here's a sample **routine_option** declaration part along with a corresponding routine heading:

```
ROUTINE_OPTION
         my_C_routine_options = a0_return, d0_return, c_param;
...

PROCEDURE calling_a_C_module ; OPTIONS (my_C_routine_options);
...
```

The above declaration tells the compiler to handle the procedure **calling_a_C_module** as if it specified **a0_return, d0_return,** and **c_param** in its routine heading.

## 5.6.3 Rules for Using the routine_option Declaration Part

You can use routine options that you define in a **routine_option** declaration part in any context that you can use the predeclared routine options included in the definition.

## 5.6.4 Using default_routine_options

You can use the special name **default_routine_options** for the list of options that you declare in a **routine_option** declaration part. If you do so, then the options that you define in your list are used as the default routine options for every subsequent routine in that module *that does not specify a routine option*. The rules for the scope of **default_routine_options** are the same rules that Domain Pascal follows for the scope of variables.

For example, consider the following fragment:

```
ROUTINE_OPTION
        default_routine_options = a0_return, d0_return, val_param;
...

PROCEDURE fee;
...

PROCEDURE fie;
...

PROCEDURE foe;
...

PROCEDURE fum;
...
```

The preceding declarations tell the compiler to handle the procedures **fee, fie, foe,** and **fum** as if each one specified **a0_return, d0_return,** and **val_param** as routine options in its routine heading. In other words, they have the same effect as:

```
PROCEDURE fee; OPTIONS (a0_return, d0_return, val_param);
...

PROCEDURE fie; OPTIONS (a0_return, d0_return, val_param);
...

PROCEDURE foe; OPTIONS (a0_return, d0_return, val_param);
...

PROCEDURE fum; OPTIONS (a0_return, d0_return, val_param);
...
```

You can override the **default_routine_options** for any routine by specifying different routine options for that particular routine. The override does not affect the other routines.

Continuing the previous example, assume you have defined **default_routine_options** as listed previously for a module that contains the **fee, fie, foe,** and **fum** procedures.

However, you do not want to apply all of the **default_routine_options** to the **foe** procedure. You can change the routine heading for **foe** as follows:

```
ROUTINE_OPTION
          default_routine_options = a0_return, d0_return, val_param;
. . .

PROCEDURE fee ;
. . .

PROCEDURE fie;
. . .
PROCEDURE foe; OPTIONS (abnormal) ;
. . .

PROCEDURE fum;
. . .
```

The above set of declarations tells the compiler to handle the **foe** procedure as if it specified just one routine option—namely, the **abnormal** option.

## 5.7 Attribute List—Extension

As noted in the beginning of this chapter, you can declare an optional routine *attribute_list* at the beginning of a routine heading. With this list, you can specify a nondefault section name for the code and data of a routine. The *attribute_list* affects a routine body, while the *routine_options* affect the routine interface.

The *attribute_list* consists of one or more routine attributes enclosed by brackets. Domain Pascal currently supports the **section** routine attribute.

### 5.7.1 Section—Extension

By using the routine attribute **section**, you can specify a nondefault section name for the code and data in a routine. A "section" is a named contiguous area of an executing object. (See the *Domain/OS Programming Environment Reference* for details on sections.)

By default, the compiler assigns code to the **.text** section and data to the **.data** section. Thus, by default, all code from every routine in the program is assigned to **.text**, and all static data from every routine in the program is assigned to **.data**. However, Domain Pascal permits you to override the default of **.text** and **.data** on a routine-by-routine basis. (You can also override the defaults on a variable-by-variable or module-by-module basis.)

You can use the **section** routine attribute to organize the run-time placement of routines so that logically related routines can share the same page of main memory and thus reduce page faults. Likewise, you can declare a rarely called routine as being in a separate section from the frequently called routines.

To override the default sections, preface your routine heading with a phrase of the following format:

**[section(** *codesect*, *datasect* **)] procedure** . . .

or

**[section(** *codesect*, *datasect* **)] function** . . .

To specify an alternate data section while keeping the default **.text** section, use the following syntax:

**[section(** , *datasect* **)] procedure** . . .

or

**[section(** , *datasect* **)] function** . . .

If you omit either the *codesect* or the *datasect*, the present default continues to take effect.

For example, consider the following fragment:

```
Program example;
    VAR stat: integer;          { In .data }

 PROCEDURE top;                 { In .text }
    VAR dat1 : static integer;  { In .data }
    BEGIN
         ...                    { In .text }
    END;

 [SECTION(npc, npd)] FUNCTION foobar1 : REAL;  { In "npc" }
    VAR seed : static real;                    { In "npd" }
    BEGIN
         ...                                   { In "npc" }
    END;

 [SECTION(error_rout_c, error_rout_d)] PROCEDURE xxx; {In error_rout_c }
    VAR check : static integer32;                     {In error_rout_d }
    BEGIN
         ...                                          {In error_rout_c }
    END;

 FUNCTION regular : integer;  { In .text }
    VAR dat2 : static real;   { In .data }
    BEGIN
         ...                  { In .text }
    END;
```

```
[SECTION(npc, npd)] PROCEDURE foobar2;          { In "npc" }
   VAR seedling : static real;                  { In "npd" }
   BEGIN
         ...                                    { In "npc" }
   END;

 BEGIN {main}
         ...                 { In .text }
 END;
```

Nested routines inherit the section definitions of their outer routine unless they specify their own section definitions. For example, if the **foobar1** function contained nested routines, Domain Pascal defaults to placing their code and static data into the "**npc**" and "**npd**" sections respectively.

## 5.8 Recursion

A recursive routine is a routine that calls itself. Domain Pascal, like standard Pascal, supports recursive routines. The following example demonstrates a recursive method for calculating factorials:

```
PROGRAM recursive_example;
 { Demonstrates recursion by calculating a factorial. }

 VAR
    x, y : integer32;

 Function factorial(n : integer32) : integer32;
 BEGIN
    if n = 0
      then factorial := 1
      else factorial := n * factorial(n-1);   {factorial calls itself.}
 END;

 BEGIN {main}
    writeln('This program finds the factorial of a specified integer.');
    write('Enter a positive integer (from 0 to 16) --');  readln(x);
    y := factorial(x);
    writeln('The factorial of ', x:1, ' is ', y:1);
 END.
```

# Chapter 6

# Program Development

This chapter describes how to produce an executable object file (that is, a finished program) from Domain Pascal 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.

## 6.1 Program Development in a Domain Environment

Briefly, you create an executable object file using the following steps:

1. Compile each file of source code that constitutes the program. The compiler creates one object file for each file of source code.

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 so that they can communicate with one another. Before linking, you may wish to package related object files into a library file with the UNIX archiver utility or the Aegis librarian.

3. Debug or execute the program.

Figure 6–1 illustrates the general program development process. As described in later sections, the details differ somewhat depending on whether you are developing programs in an Aegis or UNIX environment.

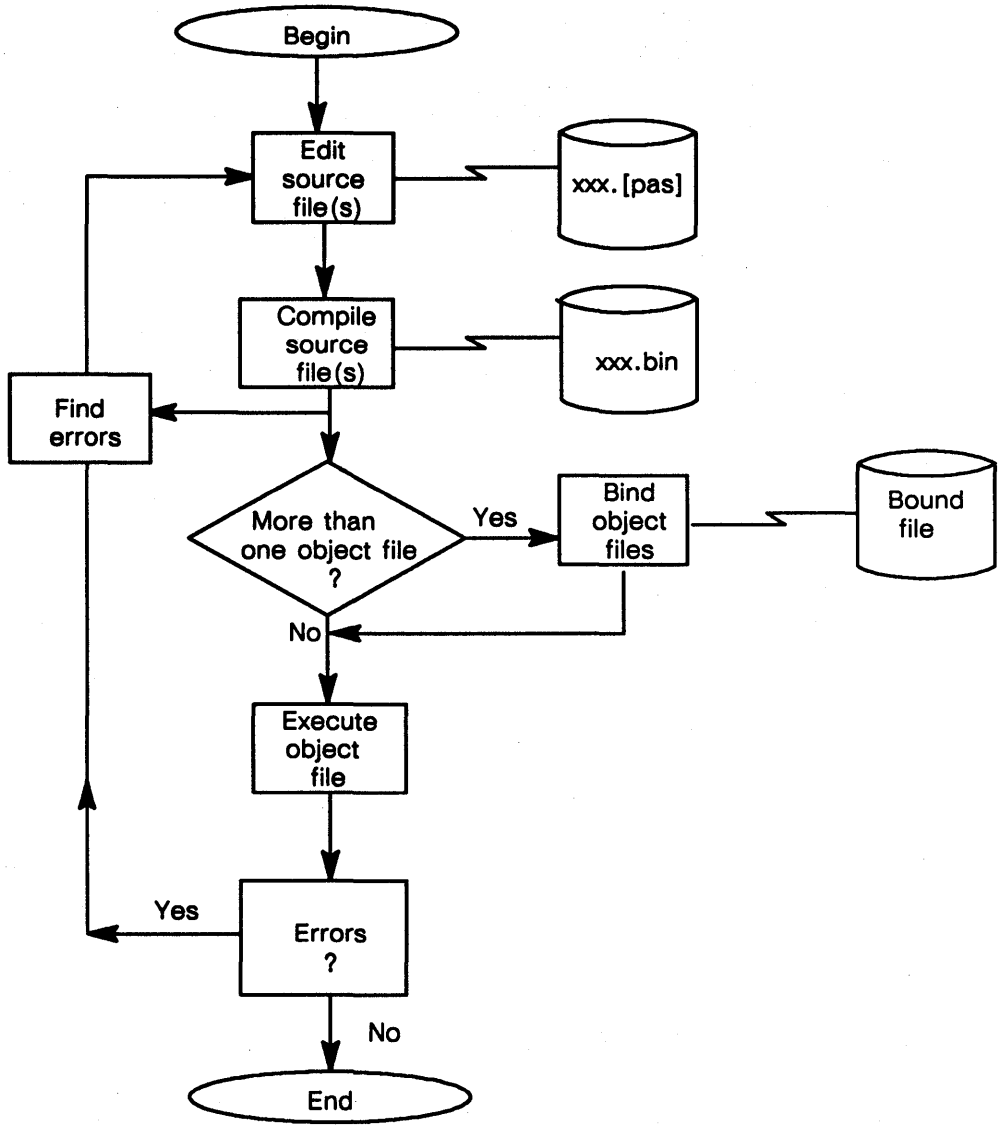

*Figure 6-1. Program Development in a Domain System*

This chapter details the compiler and provides brief overviews of the linker (binder), and debugger utilities.

In addition to the traditional programming development scheme shown in Figure 6-1, you can also use the Domain Software Engineering Environment (DSEE) system to develop Pascal programs. This chapter also contains a brief description of Domain/Dialogue, a product that simplifies the writing of user interfaces.

Use the **pas** command to preprocess and compile a single source file (plus **%included** files), and produce a single object file. If your program contains more than one object file, 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 the UNIX link editor. You can use either command to link object modules together. Use the **-b** binder option to tell the binder to create one executable object file.

If your program accesses routines in a user-supplied library, you need to link your program with the user-supplied library.

## 6.2 Compiler Variants

We have four different variants of the Domain Pascal compiler. The four variants differ in the kind of machine they run on and the kind of machine for which they generate code. The variants are

- MC680x0 native compiler
- Series 10000 (*PRISM*) native compiler
- MC680x0-to-*PRISM* cross compiler
- *PRISM*-to-MC680x0 cross compiler

You probably have two of these four variants installed on your system, but you may be able to use the other two by means of links to other systems. To find out which variant of the compiler you are using, use the **-version** option of the **pas** command. The following codes indicate the variants:

| | |
|---|---|
| **68K** | MC680x0 native compiler |
| **PRISM** | Series 10000 (*PRISM*) native compiler |
| **68K=>PRISM** | MC680x0-to-*PRISM* cross compiler |
| **PRISM=>68K** | *PRISM*-to-MC680x0 cross compiler |

Where the compiler differs from one variant to another, we describe each variant separately; otherwise, you can assume that all four variants of the compiler behave the same.

## 6.3 Compiling

You compile a Domain Pascal source code file in any Domain/OS environment by entering the following command:

```
$ pas source_pathname [option1 ... optionN]
```

*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 Pascal source programs, we recommend that *source_pathname* end with a **.pas** suffix. If you use the suffix, you need not specify it in the compile command line.

The compile command line can contain one or more of the options listed in Table 6-2. Note that you cannot abbreviate these options.

For example, consider the following three sample compile command lines, all of which compile source code file **circles.pas**:

```
$ pas circles

$ pas circles -l

$ pas circles -map -exp -cond -cpu 3000
```

### 6.3.1 Compiler Output

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 files it creates according to the following rules:

- If the source pathname ends with **.pas**, the compiler *replaces* that suffix with **.bin**.
- If the source pathname does *not* end with **.pas**, then by default Domain Pascal 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 non-default name, use the **-b** *pathname* option.

Table 6-1 shows examples for each of these rules.

*Table 6-1. Sample Object File Names*

| | Source Code | Command | Object File Name |
|---|---|---|---|
| **Source code file named with .pas suffix** | extratext.pas | $ pas extratest *or* $ pas extratest.pas | extratest.bin |
| **Source code file named with other suffix** | test.first | $ pas test.first | test.first.bin |
| **Source code file named with no suffix** | test | $ pas test -b //good/compilers/newtest | //good/compilers/newtest.bin |

## 6.4 Compiler Options

Domain Pascal supports a variety of compiler options. Table 6-2 summarizes the options, and the following sections describe all the options in detail.

*Table 6-2. Domain Pascal Compiler Options*

| Option | What It Causes the Compiler to Do |
|---|---|
| **-ac** | Produce absolute code. The compiler sets the load address at compile time, and therefore your program runs faster. Default for MC680x0 compiler variants, invalid for Series 10000 compiler variants. |
| **-pic** | Produce position-independent code). The compiler uses relative addressing for your data and sets the address at run time. Default for Series 10000 compiler variants. |
| ☆ **-alnchk** | Display messages about alignment of data structures. Default for Series 10000 compiler variants. |
| ☆ **-b** *pathname* | Generate a binary file in the current directory at *source_file_name*.**bin**, or in another file at *pathname*.**bin**. |
| **-nb** | Suppress creation of binary file. |
| ☆ **-bounds_violation** | Allow referencing beyond the end of an array. Optimization of code is less efficient. |
| **-no_bounds_violation** | Do not allow referencing beyond the end of an array. This option allows superior optimization of the code. |
| **-comchk** | Issue a warning if comments are not paired correctly. |
| ☆ **-ncomchk** | Suppress checking for paired comments. |
| ☆ **-compress** | Store the object file in compressed form. |
| **-ncompress** | Store the object file in noncompressed form. |
| **-cond** | Compile lines prefixed with the **%debug** compiler directive. |
| ☆ **-ncond** | Ignore lines prefixed with the **%debug** compiler directive. |
| **-config** *var1... varN* | Set special conditional compilation variables to true. |
| **-cpu** *id* | Generate code for a particular workstation type. 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 Series 10000 compilers is **a88k.** |
| ☆ denotes a default option | |

*(Continued)*

*Table 6-2. Domain Pascal Compiler Options (Cont.)*

| Option | What It Causes the Compiler to Do |
|---|---|
| -ndb | Suppress creation of debugging information. The debugger cannot debug such a program. |
| ☆ -db | Generate minimal debugging information. When you debug this program, you can set breakpoints, but you can't examine variables. |
| -dbs | Generate full debugging information and optimize the code in the executable object file. (Implies **-opt 3**.) |
| -dba | Generate full debugging information but don't optimize the code in the executable object file. |
| -exp | Generate assembly language listing (implies **-l**). |
| ☆ -nexp | Suppress creating assembly language listing. |
| -frnd | Round floating-point numbers at key points during program execution. |
| ☆ -nfrnd | Optimize execution by computing floating-point expressions in greater precision than that specified by the program, when the compiler detects an opportunity to do so. |
| -idir *dir1... dirN* | Search for an include file in alternate directories. |
| -imap | Generate symbol table maps for **%included** files (implies **-map**). |
| ☆ -nimap | Suppress creation of symbol table maps for **%included** files (implies **-nmap**). |
| -indexl | Produce a 32-bit index for all array references. |
| ☆ -nindexl | Refer to source code for array reference index information. |
| -info *n* | Display information messages to the *n*th level. *n* is an optional specifier that must be between 0 and 4. If *n* is omitted, or the entire switch is omitted, display information messages to level 2. |
| -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. |
| -iso | Compile the program using ISO/ANSI Standard Pascal rules for certain Domain Pascal features that deviate from the standard. |
| ☆ -niso | Compile the program using Domain Pascal features. |
| -l *pathname* | Generate a listing file at *program_name*.**lst** or *pathname*.**lst**. |
| ☆ -nl | Suppress creation of listing file. |
| ☆ denotes a default option | |

*(Continued)*

*Table 6–2. Domain Pascal Compiler Options (Cont.)*

| Option | What It Causes the Compiler to Do |
|---|---|
| **–map**<br>☆ **–nmap** | Generate symbol table map (implies **–l**).<br>Suppress creation of symbol table map. |
| **–msgs**<br>☆ **–nmsgs** | Generate final error and warning count message.<br>Suppress creating final error and warning count message. |
| **–natural**<br>**–nnatural** | Set environment to natural alignment. Has the same effect as the **%natural_alignment** compiler directive.<br>Suppress setting the environment to natural alignment. |
| **–nclines** | Suppress the generation of COFF line number tables so as to save space in the object file. Valid option for MC680x0 compiler variants only. |
| ☆ **–opt** *n* | Cause the compiler to perform global program optimizations to the *n*th level, where *n* is between 0 and 4. If *n* is omitted, or if the option is omitted, the default optimization level is 3. |
| ☆ **–prasm**<br>**–nprasm** | Create an expanded listing in Series 10000 assembly–code format, if **–exp** is specified and if Series 10000 code is being generated.<br>Create an expanded listing in alternate assembly–code format. |
| **–slib** *pathname* | Treat the input as an include file and produce a precompiled library of include files at *program_name*.**plb** or *pathname*.**plb**. |
| **–std**<br>☆ **–nstd** | Issue warning messages when nonstandard language elements are encountered.<br>Suppress warning messages for nonstandard elements. |
| **–subchk**<br>☆ **–nsubchk** | 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.<br>Suppress subscript checking. |
| **–version** | Display version number of compiler. Note that you do not include a filename when you use this option. The following command line shows the usage of this option: **pas –version** |
| ☆ **–warn**<br>**–nwarn** | Display warning messages.<br>Suppress warning messages. |
| ☆ **–xrs**<br>**–nxrs** | Save registers across a call to an external procedure or function.<br>Do not assume that calls to external routines have saved the registers. |
| ☆ denotes a default option. | |

### 6.4.1 -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.

If you use the **-ac** option, Domain Pascal uses absolute code to compile your program. This means that the compiler sets the load address for your data at compile time. As a result, your program runs faster.

If you use the **-pic** option, then the compiler generates PIC (Position Independent Code). This means that the compiler uses *relative* addressing and that the addresses are set at *run time*, rather than at *compile time*. As a result, your program may be less efficient to run than if you use the default **-ac** option.

Use the **-pic** option for compiling library files that you load with the **-inlib** option. Since these files are used by many different programs, their addresses must be set at run time. You should also use **-pic** for extensible streams and GPIO drivers.

### 6.4.2 -alnchk: Displaying Messages about Alignment

The **-alnchk** option is always set for the compiler variants that generate Series 10000 code.

When you use the **-alnchk** option, the compiler displays messages telling you whether your data is naturally aligned. Naturally aligned data increases efficiency at least slightly on any workstation, but the increase in efficiency is very significant on Series 10000 workstations.

See the "Internal Representation of Unpacked Records" and "Alignment—Extension" sections of Chapter 3 for more details about alignment.

### 6.4.3 -b and -nb: Binary Output

The **-b** option is the default.

If you use the **-b** option, and if your source code compiles with no errors, Domain Pascal creates an object file with the source pathname and the **.bin** suffix. If you specify a pathname as an argument to **-b**, then Domain Pascal creates an object file at **pathname.bin**.

If you use the **-nb** option, Domain Pascal suppresses creating 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.

Given that error-free Domain Pascal source code is stored in file **test.pas**, here are some sample command lines:

```
$ pas test
 {Domain Pascal creates test.bin}

% pas test -b
 {Domain Pascal creates test.bin}

$  pas test -b jest
 {Domain Pascal creates jest.bin}

$  pas test -b jest.bin
 {Domain Pascal creates jest.bin}

% pas test -nb
 {Domain Pascal doesn't create an object file}
```

## 6.4.4 -bounds_violation and -no_bounds_violation: Array Bounds Checking

The **-bounds_violation** option is the default.

If you use the **-bounds_violation** option, you are permitted to reference an element beyond the bounds of an array. For example, if you declared an array as:

```
a : ARRAY [0..15] OF integer32;
```

you could assign a value to **a[16]**. However, you may destroy the values stored in other variables by writing beyond the bounds of an array.

If you use the **-no_bounds_violation** option, you are not permitted to reference an element beyond the bounds of an array. This option provides better optimization than the default.

If the following example is compiled with the **-bounds_violation** option, it will print the value 10. If it is compiled with **-no_bounds_violation**, it will print 0, but the program will be better optimized.

```
PROGRAM bounds_violation;
    TYPE
        boundless = RECORD
            a           : ARRAY [ 0..9 ] OF INTEGER32;
            i           : integer32;
        END;

    VAR
        b   : boundless;
        j   : integer;

    BEGIN
        b.i := 0;
        FOR j := 0 TO 10 DO
            b.a[j] := j;
        writeln(b.i);
    END.
```

### 6.4.5 -comchk and -ncomchk: Comment Checking

The **-ncomchk** option is the default.

If you compile with **-ncomchk**, the compiler does not check to see if you've balanced your comment delimiters. Consequently, if you forget to close an open comment, the compiler will probably misinterpret a piece of code as part of a comment. If you compile with the **-ncomchk** option, you get the default results.

If you compile with the **-comchk** option, the compiler checks to see that comment pairs are balanced; that is, that there are no extra left comment delimiters before a right comment delimiter. The left comment delimiters are {, (*****, and "; the right comment delimiters are }, *****), and ". If you compile with **-comchk**, the compiler returns a warning for every additional left comment delimiter.

For example, the following fragment produces weird results because of an unclosed comment:

```
{This comment should be closed, but I forgot to do it!

 x := 0;        {We need this statement.}
```

However, if you compile with **-comchk**, the compiler returns the following warning message:

```
Warning:  Unbalanced comment; another comment start found
 before end.
```

Note that **-comchk** causes the compiler to look only for the same kind of left comment delimiter. For example, if you start the comment with (*****, the compiler does not flag any extra left brace { that occurs before the next *****).

### 6.4.6 –compress and –ncompress: Object File Storage

The **–compress** option is the default.

The **–compress** option stores object file data in compressed form. The **–ncompress** option stores the data in uncompressed form. Uncompressed data is an exact image of the data as it will be loaded into memory. Compressed data contains instructions to the loader that describe how to load the data into memory.

For example, consider an array of 100 bytes, all initialized to 0. In uncompressed form, this array occupies 100 bytes in the object file. In compressed form, it occupies only enough space for a single **.rwdi** instruction: load 100 bytes with value 0. (**.rwdi** stands for "read-write data initialization.")

You may find the **–ncompress** option especially useful if you are linking modules with uncompressed data to modules with compressed data. In this case, the linker will expand the compressed data. The linker output file then contains the uncompressed data from the two modules and, in addition, the **.rwdi** instructions from the compressed module, creating an output file that is larger than if both modules had stored the data in uncompressed form.

### 6.4.7 –cond and –ncond: Conditional Compilation

The **–ncond** option is the default.

The **–cond** option invokes conditional compilation. If you compile with **–cond**, Domain Pascal compiles the lines of source code marked with the **%debug** directive. (Refer to the "Compiler Directives" listing in Chapter 4 for details on **%debug**.)

If you compile with **–ncond**, Domain Pascal treats the lines of source code marked with **%debug** as comments.

You can simulate the action of this switch with the **–config** compiler option. For new program development, you should use the **–config** syntax, since the **–cond** option is considered obsolete.

### 6.4.8 –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 the "Compiler Directives" listing) or with the **–config** option. The format of the **–config** option is

**–config** *var1* [... *varN*]

where **var** must be a conditional variable declared with **%var**.

For example, consider the program shown in Figure 6-2. The program is available online and is named **config_example**.

```
{**********************************************************************}
PROGRAM config_example;

 { You can use this program to experiment with the }
 { -CONFIG compiler option. }

 VAR
     x, y, z : integer16 := 0;

 BEGIN
    writeln('The start of the program.');
  %VAR first, second, third

 %IF first %THEN
    x := 5;
    writeln(x,y,z);
 %ENDIF

 %IF second %THEN
    y := 10;
    writeln(x,y,z);
 %ENDIF

 %IF third %THEN
    z := 15;
    writeln(x,y,z);
 %ENDIF

    writeln('The end of the program');
 END.
{**********************************************************************}
```

*Figure 6-2. A Program Illustrating Conditional Variables:* **config_example**

First, notice what happens when you compile without **-config**.

```
$ pas config_example
 No errors, no warnings, Pascal Rev 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:

```
$ pas config_example -config first third
 No errors, no warnings, Pascal Rev 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 switch, enclose the section of code you want conditionally compiled in an **%if** *config_variable* **%then** structure. Then use **-config** to set *config_variable* to true when you want to compile that section of code.

### 6.4.9 -cpu: Target Workstation Selection

The **-cpu mathlib_sr10** option is the default if you are using a compiler variant that generates MC680x0 code. The **-cpu a88k** option is the default if you are using a compiler variant that generates Series 10000 code. For information about compiler variants, see Section 6.2.

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. The format for the **-cpu** option is

**-cpu** *id*

You select the code generation mode through the argument that you specify immediately after **-cpu**. Table 6-3 shows the possible arguments and the code generation mode that each argument selects. For example, to compile **prog.pas** with the **mathchip** argument, use the following command line:

```
pas prog.pas -cpu mathchip
```

The advantage of the processor-specific code generation modes is that the compiler generates code optimized for that particular processor, which makes the programs so compiled run faster. The advantage is seen mostly in programs that make frequent use of floating-point operations. Programs that make heavy use of multiplication and division with 32-bit integers may also show significant improvement. To find out how to obtain the best floating-point performance on the new Domain MC68040-based workstations, refer to Appendix F.

The **-cpu mathchip** option generates the best possible code for the following Apollo workstations, each of which has an MC68020 or MC68030 microprocessor and an MC68881 or MC68882 floating-point coprocessor:

HP Apollo 9000 Series 400 Model 400dl
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 **mathlib_sr10** and **any** arguments allow your code to run on a wider variety of platforms with some loss of performance. The default option for MC680x0-based workstations, **-cpu mathlib_sr10**, generates code that runs very well on all of the above workstations and on MC68040-based workstations. This option is the default if you are using a compiler variant that generates MC680x0 code.

Note that there are many possible arguments to **-cpu**; however, many of them generate identical code. For example, **-cpu mathchip** produces exactly the same code as **-cpu 570**, **-cpu 580**, and **-cpu 3000**.

*Table 6-3. Arguments to the* **-cpu** *Option*

| Argument | What the Argument Causes the Compiler To Do |
|---|---|
| ☆ **mathlib_sr10** | Generates code for workstations with an MC68040 microprocessor, or with an MC68020 or MC68030 microprocessor and an MC68881 or MC68882 floating-point coprocessor. Code compiled with this argument runs on SR10.0 and later versions of Domain/OS. The **-cpu mathlib_sr10** option is the default if you are using a compiler variant that generates MC680x0 code. |
| ☆ **a88k** | Generates code for a Series 10000 workstation. The **-cpu a88k** option is the default if you are using a compiler variant that generates Series 10000 code. |
| **mathlib** | Produces optimal code for workstations with an MC68040 microprocessor (including the HP Apollo 9000 Series 400 Model 425t and 433s). Also generates code for workstations with an MC68020 or MC68030 microprocessor and an MC68881 or MC68882 floating-point coprocessor. Code compiled with this argument runs only on SR10.3 and later versions of Domain/OS. Use **mathlib_sr10** if your code must also run on SR10.0, SR10.1, or SR10.2. |
| **mathchip**<br>**3000**<br>**580**<br>**570**<br>**560**<br>**330**<br>**90** | Generates code for workstations with an MC68020 or MC68030 microprocessor and an MC68881 or MC68882 floating-point coprocessor. These seven arguments generate identical code. We recommend that you use the **mathchip** argument; the other six arguments will become obsolete at a future compiler release. (Code compiled with **mathchip** will also run on the MC68040, but not very well.) |
| **160**<br>**460**<br>**660** | Generates code for a DSP160, DN460, or DN660 workstation. These three arguments generate identical code. |
| **fpa1** | Generates code for DN3000, DN4000, or DN4500 workstations with an FPA1 floating-point accelerator unit. |
| **fpx** | Generates code for DN5xx workstations with an FPX floating-point accelerator unit. |
| **peb** | Generates code for workstations with a Performance Enhancement Board (PEB) (includes the DN100, DN320, DN400, and DN600, when equipped with an optional PEB). |
| **any** | Generates Series 10000 code if you are using a compiler variant that generates Series 10000 code; generates generic MC680x0 code if you are using a compiler variant that generates MC680x0 code. |
| **m68k** | Generates code for any MC680x0-based workstation (same as **any** on all workstations other than the Series 10000). |
| ☆ denotes a default option | |

Table 6–4 shows the relative performance of the MC680x0 code generated with different arguments to the **–cpu** option.

*Table 6–4. Relative Performance with Different* **–cpu** *Arguments*

| | **Machine Type** | | | | | | |
|---|---|---|---|---|---|---|---|
| **Argument** | **100, 400** | **PEB** | **160, 460, 660** | **FPX** | **FPA1** | **MC68020/030, 68881/82** | **MC68040** |
| **mathlib_sr10** | -- | -- | -- | fair | * | fair | good |
| **mathlib** | -- | -- | -- | fair | * | good | best |
| **mathchip** | -- | -- | -- | fair | * | best | fair |
| **peb** | -- | best | -- | -- | -- | -- | -- |
| **160, 460, 660** | -- | -- | best | -- | -- | -- | -- |
| **fpx** | -- | -- | -- | best | -- | -- | -- |
| **fpa1** | -- | -- | -- | -- | best | -- | -- |
| **any** | best | fair | poor | poor | poor | poor | fair |

**Legend:**

| | |
|---|---|
| -- | Code generated with this argument will not run on this machine type. |
| * | The compiler selects instructions that do not use the FPA1 accelerator. In this case, the code runs exactly the same as code generated for an MC68020-based or MC68030-based machine. |
| **best, good, fair, poor** | These four terms 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 MC6820-based or MC68030-based machine. |

### 6.4.9.1 Choosing an Appropriate –cpu Argument: The cpuhelp Utility

The **cpuhelp** utility provides quick online information about which **–cpu** argument is appropriate for a particular machine or about which machines a particular **–cpu** argument will work on. For information about this utility, type **help cpuhelp** in an Aegis environment or **man cpuhelp** in a UNIX environment.

## 6.4.10 -db, -ndb, -dba, -dbs: Debugger Preparation

The **-db** option is the default.

Use these switches to prepare the compiled file for debugging by the Domain Distributed Debugging Environment. Domain Pascal stores the debugger preparation information within the executable object file, so in general, the more debugger information you request, the longer your executable object file.

If you use the **-ndb** option, the compiler puts no debugger preparation information into the **.bin** file. If you try to debug such a **.bin** file, the system reports the following error message:

```
?(debug) 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 (e.g., 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. When you use the **-dbs** option, the compiler sets the **-opt** option. (You can override this with the **-nopt** or **-opt 0** option.)

The **-dba** option is identical to the **-dbs** option except that when you use the **-dba** option, the compiler sets the **-nopt** option (even if you specify **-opt**).

NOTE: The **-dba** option overrides anything you specify for the **-opt** option. When you specify **-dba**, the **-opt** option is set to **-opt 0**, regardless of what you specified for **-opt** on the command line for the compilation. See the "-opt: Optimized Code" section of this chapter for more details about the optimizations that are set with **-dba**.

For more information on these four options, see the *Domain Distributed Debugging Environment Reference*.

## 6.4.11 -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. 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 **-map** or **-l** options).

### 6.4.12 -frnd and -nfrnd: Floating-Point Rounding

The **-nfrnd** option is the default.

If you compile with **-nfrnd**, the compiler generates 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 the arithmetic in greater precision, it does so.

The **-frnd** option causes floating-point numbers to be rounded to the programmer-specified precision (either 32-bit single precision or 64-bit double precision) at key points in the execution of the program, so that programs executing on Domain systems will give results similar to those obtained on machines that use different floating-point representation. Programs compiled with **-frnd** execute more slowly and with less floating-point precision (that is, less mathematical exactness) than those compiled with **-nfrnd**.

With **-nfrnd**, floating-point operands may be kept in registers that support more accuracy than memory does. Consequently, when a register operand is compared with a memory operand, the result may not be what is expected. This is particularly true of equality comparisons. Consider the following Pascal program:

```
PROGRAM main;

VAR
    x : double;

FUNCTION fetch : double;
BEGIN
    fetch := 1.1;
END;

BEGIN
    x := fetch;
    IF (x - 0.1) = 1.0 THEN
         writeln('Pass')
    ELSE
         writeln('Fail');
END.
```

If you compile with **-nfrnd**, this program fails because the values 0.1 and 1.1 cannot be represented exactly in base 2 floating-point. Thus, the quantity **(x - 0.1)** can only be approximated. This value is calculated in an 80-bit register, and then a compare is generated to see if this value is *exactly* equal to 1.0, which is stored in memory. Since the register has more precision than memory has, the comparison fails.

If you compile with **-frnd**, the 80-bit register is stored (and rounded) in a double-precision 64-bit temporary memory location. Now when it is compared with 1.0, which is also stored in memory, the two values compare as equal.

Floating-point calculations on Apollo workstations run considerably faster if you do not use the **-frnd** option. Moreover, using **-frnd** makes floating-point calculations less precise. If you find that your results with **-frnd** differ significantly from your results without it, your code is exposing the inherent imprecision of floating-point arithmetic. In this case, you should investigate whether you can rewrite your program so that it produces the same results with and without **-frnd**. Try to eliminate practices like the following:

- Equality comparisons of floating-point values.
- Subtraction of two nearly equal floating-point values. The imprecision in the low bits of the two numbers can dominate the value of the difference.
- Expressions that evaluate a function near a singularity. For example:

```
tan(pi/2.0 - small_delta)
sin(1/small_epsilon)
```

**NOTE:** The **-frnd** option gives a precision that is closer to that required by the IEEE-754 floating-point standard than that provided by **-nfrnd**. However, using **-frnd** does not bring a program into compliance with the IEEE standard. If you want exact conformance to the IEEE standard, use the Domain/OS system call **fpp_$set_rounding_mode**. This routine puts the floating-point processor in a round-every-computation mode, which gives exact IEEE-754 conformance but, like **-frnd**, reduces precision and increases execution time.

## 6.4.13 -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 Pascal 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.pas'
```

Domain Pascal searches for that relative pathname at **//nord/minn/mytypes.ins.pas**. 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.

For example, consider the following compile command line:

```
$ pas test -idir //ouest/hawaii
```

This command line causes the compiler to search for **mytypes.ins.pas** at **//ouest/hawaii/mytypes.ins.pas** if it can't find **//nord/minn/mytypes.ins.pas.**

You can put up to 63 pathnames following an **-idir** option. Separate each pathname with a space.

## 6.4.14 -imap and -nimap: Generate Symbol Table Maps for Include Files

The **-nimap** option is the default.

The **-imap** option tells the compiler to generate map files for files that are **%included** by the files that you specify on the command line. The generated map files are the same as those generated when you use the **-map** option. **-imap** implies **-map.**

See Section 6.4.20 for further details about the **-map** option and about the symbol table maps that are generated.

## 6.4.15 -indexl and -nindexl: Array Reference Index

The **-nindexl** option is the default.

The **-indexl** option disables some optimizations and forces the compiler to use 32-bit indexing in subscript calculations. The **-nindexl** option causes the compiler to use the source code's array dimension information to determine whether to use 16-bit or 32-bit indexing.

## 6.4.16 -info n 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. The purpose of **information messages** is to alert you to ways in which you could improve the efficiency of your code. 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.

**-ninfo** 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 3,** the compiler will display all the messages specified by **-info 3** *plus* all the messages specified by **-info 1** and **-info 2.**

Domain Pascal provides the following information message levels:

**-info 0** This level is equivalent to **-ninfo**. At this level, the compiler displays no informational messages.

**-info 1** Messages at this level are about the size and alignment of variables and types.

**-info 2** Messages at this level describe optimizations performed by the compiler. All **-info 2** messages were warning messages prior to SR10.

**-info 3** If you specify this level, the compiler issues a message in addition to level 2 messages *only* if *all* of the following conditions occur:

- You do *not* specify the alignment for a variable.
- The variable is *not* naturally aligned.
- The compiler loads or stores from the variable.

Use this message level to discover variables for which you could specify the alignment as natural alignment and thereby make your program run more efficiently.

**-info 4** If you specify this level, you get the same message as you get with **-info 3**. However, you get this message even if you do not specify natural alignment for a variable.

Use this level to discover any variables for which you may want to change the alignment.

Level 4 messages also indicate if a routine has been expanded inline. For example, if you compile a program containing a function named **test_pos** with **-info 4**, you get the following message at compile time:

```
******** Line 38: [Information  266]  routine expanded INLINE
           at call site: "test_pos".
```

This message indicates that the compiler substituted an expansion of **test_pos** at line 38 of your source file, where your source code contained a function call to the **test_pos** function. For more information about inline expansion, see the discussion of the **%begin_inline**, **%end_inline**, **%begin_noinline**, and **%end_noinline** directives in the "Compiler Directives" listing of Chapter 4.

Note that using the **-info** option does not affect the display of warning and error messages. See Chapter 9 for more details about information, warning, and error messages.

### 6.4.17 -inlib: Library Files

Use **-inlib** to tell the compiler to load library files at run time. The **-inlib** option makes code available to an executing program without actually binding the code into the output object file. If your program needs to access code in a library file that has 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 created with the **-pic** option. (For information about **-pic**, see Section 6.4.1.)

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 absolute procedure references.
3. At run time, the loader loads the library file, and the externals identified in Step 2 are dynamically linked.

## 6.4.18 -iso and -niso: Standard Pascal

The **-niso** option is the default.

Domain Pascal implements a few features differently than ISO standard Pascal. The **-iso** switch lets you tell the compiler to use standard Pascal rules for some of these features. It also tells the compiler to turn on the **-std** switch, which issues error messages for nonstandard language elements.

The **-iso** option tells the compiler to use ISO rules with the **mod** operator. Domain Pascal implements **mod** using the Jensen and Wirth semantics; see the **mod** listing in Chapter 4 for details.

In addition, if you compile with **-iso**, comment delimiters no longer are required to match up, which means that the following becomes valid:

```
{This comment starts with one type of delimiter and ends with another.*)
```

Finally, if you use the **-iso** option, the compiler flags as an error a **goto** statement that jumps into an **if/then/else** statement.

For example, the following is incorrect under the **-iso** switch:

```
label
     bad_jump;
 .  .  .
 if num > 0 then
     {statement}
 else
 bad_jump:
     {next statement};
 .  .  .
 goto bad_jump;             {WRONG}
```

The preceding structure is permitted under Domain Pascal.

## 6.4.19 –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, including 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 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 file name includes the **.pas** suffix, **.lst** replaces it. If the source file name does not include **.pas**, **.lst** is appended to the end of the name.

The **–nl** option suppresses the creation of the listing file. (See also **–map** and **–exp**.)

## 6.4.20 –map and –nmap: Symbol Table Map

The **–nmap** option is the default.

If you use the **–map** option, Domain Pascal creates a map file. A map file contains everything in the listing file (**–l**) plus a special symbolic map. The special symbolic map consists of two sections.

The first section describes all the routines in the compiled file. For example, here is a sample first section:

```
001 EXAMPLE Program(Proc = 00005A,Ecb = 000030,Stack Size = 0)
002 DO_NOTHING Procedure(Proc = 000000,Ecb = 000020,Stack Size = 8)
003 DO_SOMETHING Function(Proc = 00003E,Ecb = 00000C,Stack Size = 8)
```

The preceding data tells you that the compiled file contains a main program (called **example**), a procedure (called **do_nothing**), and a function (called **do_something**). There are three pieces of data inside each pair of parentheses. The first piece tells you the start address in hexadecimal bytes from the start of a section. The second piece is the offset (in bytes) of the routine entry point. The third piece is the size (in hexadecimal bytes) of the stack. The second and third pieces of data probably are of interest only to systems programmers.

For example, in the sample first section, the starting address of the main program was offset 16#5A bytes from the beginning of the **.text** section. The offset relative to the beginning of the **.data** section of the main program's routine entry point is 16#30 bytes. Its stack size is 0 bytes.

The second section lists all the variables, types, and constants in the compiled program. For example, here is a sample second section:

```
002 A       Var(-000008/S): CHAR
002 A2      Var(-000006/S): CHAR
001 BI      Type= ARRAY[1..2] OF INTEGER16
001 BILBOA  Const='The rain'
001 Q       Var(+000002/MICROS): CHAR
001 R5      Var(+000004/MICROS): DOUBLE
001 S       Var(+00000C/MICROS): BI
001 X       Var(/MICROS): INTEGER16
001 Y       Var(/D): INTEGER16
003 Z       Var(+000010/S): INTEGER16
```

The map tells you, for example, that R5 is a Var (variable) that is stored +000004 bytes from the beginning of the MICROS section. It also tells you that R5 has the data type **double**. Also, note that /D means the **.data** section, and /S means the stack.

If you specify **-nmap**, Domain Pascal does not create the special symbol map.

## 6.4.21 -msgs and -nmsgs: Messages

The **-msgs** option is the default.

If you use the **-msgs** option, the compiler produces a final compilation report having the following format:

*xx* errors, *yy* warnings, *zz* info msgs, Pascal compiler *variant* Rev *n.n*

where *xx*, *yy*, and *zz* are either "no" or a number, *n.n* is the version number, and *variant* is one of the following:

```
68K
PRISM
68K=>PRISM
PRISM=>68K
```

If you use **-nmsgs**, the compiler suppresses this final report.

## 6.4.22 -natural and -nnatural: Setting the Environment to Natural Alignment

The **-nnatural** option is the default.

The **-natural** option tells the compiler to use natural alignment for laying out data structures that do not have alignment attributes in their declarations and that are not controlled by compiler directives.

The hardware is designed to transfer data most efficiently if the data is naturally aligned. Thus, if your data is naturally aligned, you can increase the processing speed of your program, even though you may sacrifice some efficiency in memory usage.

> NOTE: **-natural** has the same effect as the **%natural_alignment** compiler directive.

See the "Internal Representation of Unpacked Records" and the "Alignment—Extension" sections of Chapter 3 for more details about natural alignment.

### 6.4.23 -nclines: COFF Line Number Tables

By default, the compiler variants that generate code for the MC680x0 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 might require a lot of disk space, you should use the **-nclines** option if you do not need the COFF line number tables and you wish to save space.

> NOTE: This option has no effect on the compiler variants that generate code for the Series 10000 workstations.

### 6.4.24 -opt: Optimized Code

The **-opt 3** option is the default.

The **-opt** option allows you to specify the kinds of optimization performed on your source program, by means of an **optimization level.**

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 the **-opt** option completely, the default option, **-opt 3**, is assumed. The obsolete option **-nopt** is equivalent to **-opt 0**.

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 compilation 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 implies **-opt 0**. 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 **-opt** option is **-opt 0**.

  The **-dba** option tells the compiler to store variables in memory after every statement instead of allowing them to remain in registers. Even with the **-dba** option the compiler still does some optimizations. Specifically, the compiler

  - 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
  - Computes multiple occurrences of the same expression within a statement only one time rather than many times

  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.

  Furthermore, **-dba** represents a level of optimization even lower than **-opt 0**. 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**. See the section on **-dba** and **-dbs** for details about using these 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)
  - Merges identical sequences of instructions in generated code by eliminating all but one of them and branching just to that one sequence

- **-opt 1** performs the following additional optimizations:

  - Eliminates some 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 transformations for speed
- Merges assignment statements where possible

- **-opt 2** performs the following additional optimizations:

  - 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 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**. So 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 already computed value 6 to **c**.

- **-opt 3** is the default optimization level.

  At this level, the compiler performs the following additional optimizations:

  - Redundant assignment statement elimination

    Redundant assignment elimination performed at this optimization level may result in warning messages such as the following:

    ```
    ******** Line 14: [Warning 279]  Value assigned to SMALL_RANGE
    is never used; assignment is eliminated by optimizer.
    ```

    Consider the following example:

    ```
    program B;

    var
       i, j :integer;

    begin
       readln ( i,j );
       if ( i = 0 ) then
         j := 3;
       writeln ( 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, in this particular program, 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, the assignment is no longer eliminated:

```
program B;

var
   i, j :integer;

begin
   readln ( i,j );
   if ( i = 0 ) then
     j := 3;

   writeln ( i, j );
end.
```

— Global register allocation

Global register allocation allows local variables 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 compiler never uses or updates the copy of the variable in the computer's main memory. 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.

— Removal of **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:

```
for i := 1 to 10 do
     begin
       j := k * m;
       j := i + j;
     end;
```

The expression **k * m** is invariant in the above example. The compiler can safely transform this loop as follows:

```
temp := k * m;
for i := 1 to 10 do
     begin
       j := temp;
       j := i + j;
     end;
```

After the invariant expression is removed from the loop, the example does only one multiply instead of 10 to make the assignment to **j**.

— 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 more expensive ones.

For example, consider the following loop:

```
for i := 1 to 10 do
    j := i * 5;
```

In the loop above, **i** is a counter or induction variable; its value is incremented by a constant each trip through the loop. The compiler can replace multiplication expressions involving induction variables with cheaper 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 }
for i:= 1 to 10 do
    begin
      j := T$00001
      T$00001:= T$00001 + 5
    end;
```

Strength reduction has replaced the multiplication with an equivalent addition. Notice that a new variable has been introduced, **T$00001**. This variable is a **strength reduction temporary variable**. The optimizer uses this variable 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, the optimizer may eliminate all assignments to **i**. Since the assignment elimination is a side effect of strength reduction, the compiler does not issue a warning message when it does this. As a result, you may find it difficult to examine induction variables when you are debugging your program. However, the optimizer 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 your code refers to an element in an array, say **A[i]**, it must calculate an address in order to fetch the correct element of A. The formula for this address calculation is as follows:

(base address of A) + (i – lower bound) * (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:

```
for i := 1 to 10 do
    A[i] := 0.0;
```

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. The optimizer can transform this loop as follows:

```
T$00001 := (base address of A) + (1 - 1) * (4)
for i := 1 to 10 do
    begin
      T$00001^ := 0.0;
      T$00001 := T$00001 + 4;
    end;
```

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 = 0 ) then
    begin
      i := 2;
      j := 3 * j;
    end
else
    begin
      k :=  i * 4;
      writeln( k );
    end;
```

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. Therefore, the compiler considers **j** to be *dead* from the statement following the **else** to the end of the routine.

Within the **then** clause, there *is* a use of **j**. Therefore, the compiler considers **j** to be *live* within the **then** clause. If there were other uses of **j** that could be reached from the **then** clause, **j** would be considered live *along the paths* that lead to those uses.

Live analysis is important because it allows the compiler to allocate a local variable to a machine register 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 you use registers efficiently, your programs will run faster.

— Extensive searches through each routine 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.

— Inline expansion of routines specified by **%begin_inline** and **%end_inline** directives.

  **Inline expansion** means that the compiler will attempt to generate code for the function everywhere the function is referenced from within the same source file. Inline expansion may create a larger object file, since the code for the function is replicated at each call site. However, inline expansion may result in an increase in execution speed. Usually, good candidates for inline expansion are small functions that are frequently executed.

  If you use **-opt 3** in conjunction with the **%begin_inline** and **%end_inline** directives, you can specify exactly which routines are to be expanded inline. These directives have no effect with **-opt 0**, **-opt 1**, or **-opt 2**. For more information about these directives, see "Compiler Directives" in Chapter 4.

- **-opt 4** performs the following additional optimization:

  In addition to performing the same inline expansions as **-opt 3**, the compiler selects routines for inline expansion whether or not you specify any routine with the **%begin_inline** and **%end_inline** directives. To specify that certain routines are not to be expanded inline, use the **%begin_noinline** and **%end_noinline** directives, which are described in the "Compiler Directives" listing in Chapter 4.

#### 6.4.24.1 Using the Debugger at Higher Optimization Levels

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 a machine register rather than the computer's 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.

## 6.4.25 -prasm and -nprasm: Expanded Listing Format

The **-prasm** option is the default if you are using a compiler variant that generates Series 10000 code and if you are compiling with the **-exp** option.

The **-prasm** and **-nprasm** options give you control over the format of expanded assembler listings when you use the **-exp** option. If you use them without the **-exp** option, they have no effect.

The **-prasm** option tells the compiler to use the Series 10000 assembly language format for the expanded listing it generates. The **-nprasm** option tells the compiler to use an alternate format for the expanded listing. Most programmers find this format easier to use when debugging a program.

If you use these options with the compiler variants that generate MC680x0 code, they have no effect.

## 6.4.26 -slib: Precompilation of Include Files

Because there usually are a number of common tasks most programs must perform, Domain Pascal programs often contain include files. Frequently used files include **/sys/ins/base.ins.pas** and **/sys/ins/error.ins.pas**. But it would be time-consuming to compile such files every time a program in which they were **%included** was compiled. The **-slib** option allows you to precompile an include file. The **%slibrary** compiler directive (described in Chapter 4) allows you to insert that file in your program. Then, the compiler knows that the file already has been compiled and doesn't bother to parse it again.

The syntax for -**slib** is:

**-slib** *pathname*

If you specify a *pathname* following **-slib**, the compiler creates the precompiled file at *pathname*.**plb**. When no *pathname* is present, and the name of the input program file ends in **.pas**, **-slib** replaces that ending with **.plb** and creates the file at **program_name.plb**. If the input filename does not end in **.pas**, **-slib** appends **.plb** to the name.

For example, suppose you want to precompile the file **mystuff.ins.pas**. This command precompiles it and puts the result in **mystuff.ins.plb**.

```
$ pas mystuff.ins -slib
```

The following restrictions apply to the contents of files that are going to be precompiled:

- They can contain only declarations.
- They must not contain routine bodies.
- They must not declare variables that would result in the allocation of storage in the default data section, **.data**.

  This means the declarations must either put variables into a named section, or must use the **extern** variable allocation clause. See Chapter 3 for more information about named sections, and Chapter 7 for details on **extern**.

Conditional compilation directives in the files you **slib** are executed during precompilation.

If you have several files that you want to combine into a single precompiled library, you can create a **container** file with a series of **%include** directives. For example, to combine some frequently used include files into the single precompiled library **/sys/ins/domain.plb**, you can create a file **systemstuff.pas** which contains the following:

```
{ Files we often use together. }
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
```

Then use **-slib** as follows to create the precompiled, combined library.

```
$ pas systemstuff -slib /sys/ins/domain
```

You can include the new precompiled library in any program. For example:

```
PROGRAM errortest;
 %SLIBRARY '/sys/ins/domain.plb';

 BEGIN
     .
     .
     .
 END.
```

### 6.4.27 -std and -nstd: Nonstandard References

**-nstd** is the default.

The **-std** option tells the compiler to compile in the usual way but to issue error messages for nonstandard language elements (i.e, extensions to the ISO standard).

Note that using **-std** does not otherwise change the way Domain Pascal compiles your program. To tell the compiler to use standard Pascal rules for compiling rather than its usual rules, you should use the **-iso** option.

The **-nstd** option suppresses reporting of nonstandard elements.

### 6.4.28 -subchk and -nsubchk: Subscript Checking

**-nsubchk** is the default.

If you use **-subchk,** the compiler generates additional code at every subscript to check that the subscript is within the declared range of the array. This extra code slows your program's execution speed.

If you use **-nsubchk,** the compiler does not generate this extra code.

### 6.4.29 -version: Version of Compiler

If you use the **-version** option, the compiler reports its version number.

> **NOTE:** When you use the **-version** option, *do not include the filename on the command line.* If you do, you will get an error message from the compiler.
>
> The following command line shows the correct use of the **-version** option:
>
> ```
> $ pas -version
> ```

### 6.4.30 -warn and -nwarn: Warning Messages

**-warn** 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 on the total number of warnings that would have been issued).

We strongly recommend that you avoid using **-nwarn**. Warnings are issued when the compiler believes it knows what the program meant to say, and so thinks it can still generate the right code. But the compiler isn't always right, and if you use **-nwarn**, you won't see the messages that could indicate where the compiler got confused.

### 6.4.31 -xrs and -nxrs: Register Saving

**-xrs** is the default.

This option controls whether the compiler believes that the contents of registers are saved across a call to an external routine or a call through a *procedure-ptr* or *function-ptr* variable. If you use **-xrs**, the compiler assumes registers are saved, while if you use **-nxrs**, it does not assume 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.

The primary use for this option is when your program contains calls back to subprograms 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 Linking Programs

There are two commands that enable you to link (bind) object modules to form an executable image. The **ld** utility is the standard UNIX link editor with some Domain enhancements. The **bind** command invokes the **ld** utility but offers a somewhat different command syntax.

NOTE: We support the **bind** command primarily for backward compatibility with scripts written to run on older versions of the Domain system. We recommend you use the **ld** command for new programs.

### 6.5.1 The ld Utility

Use the UNIX 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** (the UNIX archiver)
- Libraries created by **lbr** (the Aegis librarian)
- Object modules created by the Domain/C, Domain Pascal, or Domain FORTRAN compilers

- Object modules previously created by **ld**
- Object modules created by **bind** (the Aegis binder)

The **ld** utility resolves external references and reports external references it can't resolve. An external reference is a symbol (variable, constant, or routine) that you refer to in one object file and define in another. You can also use the UNIX utility **nm** to perform a check of resolved and unresolved global symbols.

You can execute the output from **ld** (if there is a start address) or use it as input for a further **ld** run. For syntax details on **ld** and its options, see the *SysV Command Reference* and the *BSD Command Reference*.

### 6.5.2 The bind Command

The **bind** command is similar in function to the **ld** link editor. Use the binder utility to combine an object file with other object files to which it refers. The main purpose of the binder is to resolve external references.

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). See Section 6.6 for more information about creating library files.

The **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. Refer to the *Domain/OS Programming Environment Reference* manual for a complete discussion of the binder and its options.

For example, suppose you write a program consisting of three source code files—**test_main.pas, mod1.pas,** and **mod2.pas**. To compile the source code, you issue the following three commands:

```
$ pas test_main
$ pas mod1
$ pas mod2
```

The Domain Pascal 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 T3
```

This command creates an executable object file in filename **T3**.

NOTE: At SR10 the compiler generates an object file format which 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 pre-COFF and COFF modules together. Be aware that COFF object files will not run on pre-SR10 nodes.

Refer to *Domain/OS Programming Environment Reference* for a complete discussion of the binder and its options.

## 6.6 Archiving and Using Libraries

Use the UNIX 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 user-defined installed libraries, system-defined installed libraries, system-defined global libraries, and the user-defined global library. All 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.

## 6.7 Executing a Program

To execute a program, simply enter its pathname. For example, to execute program **T3**, just enter

```
$ T3
```

The operating system searches for a file named **T3** according to its usual search rules, then calls the **loader** utility. The loader utility is user transparent. It binds unresolved 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. You can redirect these files by using the shell's redirection notation. For example, to redirect standard input when you invoke T3, type:

```
$ T3 <TRADING_DATA
```

The < character redirects standard input to the file **TRADING_DATA**. You can redirect standard output in a similar fashion, for example:

```
$ T3 >results
```

This command uses the > character to redirect standard output for **T3** to the file **results**.

## 6.8 Debugging Programs in a Domain Environment

The Domain systems support two source level debuggers. The following sections describe them briefly. For more information refer to the debugging manual and the *Domain/OS Programming Environment Reference* manual.

### 6.8.1 The Domain Distributed Debugging Environment Utility

The Domain Distributed Debugging Environment is a powerful screen-oriented debugger that provides all the features of other high-level language debuggers. To prepare a file for debugging, you do not have to do anything special at bind time but you do have to compile with the **-db**, **-dba**, or **-dbs** compiler options. **-db** provides minimal debugger preparation; **-dba** and **-dbs** provide full debugger preparation.

For complete details on the debugger, refer to the *Domain Distributed Debugging Environment Reference*.

### 6.8.2 The dbx Utility

**dbx** is the traditional Berkeley UNIX source language debugger. Although it is usually available only on BSD systems, the Domain version is available regardless of what environment you are running. However, it is not available on Series 10000 workstations. 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. If you specify a *coredump* filename, or if a file named core exists in the working directory, you can use **dbx** to examine the state of a program that has aborted prematurely.

For complete details about the **dbx** utility, refer to the *BSD UNIX Programmer's Manual* or the *SysV Programmer's Guide, Volume II.*

## 6.9 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)
- Domain/Dialogue
- Domain/PAK (Domain Program Analysis Kit)

### 6.9.1 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

```
$ tb
```

immediately after a faulty execution of the program.

For example, suppose you run a program named **test_tb** that asks for numerical input. The source code for **test_tb** is as follows:

```
program test_tb;
var
     n:integer;
begin
     write('please enter a number ');
     readln(n);
     writeln('your number is ',n)
end.
```

However, when you run **test_tb**, you give it a character string instead of a number, 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 a number alta
?(sh) "test_tb.bin" - invalid read data (library/Pascal)
$ tb
Process         8835 (parent 8761, group 8761)
Time            90/10/25.10:07(EDT)
Program         //my_node/my_name/pascal_programs/test_tb.bin
Status          050D000A: invalid read data (library/Pascal)
In routine      "pfm_$error_trap" line 185
Called from     "error" line 430
Called from     "get_integer" line 1742
Called from     "pas_$read" line 1915
Called from     "test_tb" line 6
Called from     "PM_$CALL" line 176
Called from     "pgm_$load_run" line 903
Called from     "pgm_$invoke_uid_pn" line 1124
$
```

After listing identifying information about the process, time and program, the **tb** utility reports the error status, which in this case is:

```
Status          050D000A: invalid read data (library/Pascal)
```

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 **error** routine. The **error** routine was called by the **get_integer** routine. The **get_integer** routine was called by line 1915 of the **pas_$read** routine, which was called by line 6 of the **test_tb** routine. The **test_tb** routine was called by **PM_$CALL** which was called by **pgm_$load_run**. 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 *BSD Command Reference*, the *SysV Command Reference*, and the *Aegis Command Reference* for details about the **tb** utility.

## 6.9.2 The DSEE Product

The DSEE (Domain Software Engineering Environment) environment 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

This chapter described a traditional program development cycle (i.e., compiling, building libraries, binding, debugging); the DSEE product provides some sophisticated enhancements to this cycle. For information on the optional DSEE product, see *Engineering in the DSEE Environment*.

## 6.9.3 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 the UNIX system 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 user interface 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 a 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

- *Open Dialogue Reference*
- *Creating User Interfaces with Open Dialogue*
- *MOTIF Style Guide*
- *Customizing Open Dialogue*

### 6.9.4 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 allows 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.

## 6.10 Program Development Using the Network File System (NFS)

Domain Pascal is fully compatible with the Network File System (NFS). You can redirect the binary output of the Pascal 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 Pascal, 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:

```
$ /etc/mount -o soft faraway:/ /other_node
```

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**.

For instance, to compile the program **test.pas** and place it in the directory **/tmp** on the remote node, issue the command

```
$ /com/pas test -b /other_node/tmp/test
```

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:

```
$ /other_node/tmp/test.bin
```

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 **-l** option to generate a listing file, as shown in the following sequence of commands (the example assumes you are working in a BSD or SysV shell):

```
% /etc/mount -o soft faraway:/ /other_node
% cp ~/test.pas  /other_node/test.pas
% cd /other_node
% /com/pas test -l
.
.
.
% ls
test.lst
.
.
.
```

For more information about Domain NFS, see *Using NFS on the Domain Network.*

# Chapter 7

# External Routines and Cross–Language Communication

This chapter describes how to create and call Pascal modules and how to call FORTRAN and C routines from a Domain Pascal program. Briefly, this chapter covers the following topics:

- Creating Pascal modules
- Accessing a Pascal variable or routine stored in a separately compiled module
- Accessing FORTRAN routines from a Pascal program
- Accessing C routines from a Pascal program

## 7.1 Modules

It is usually a good idea to break a large Pascal program into several separately compiled modules. After you compile each module, you can bind the resulting object files into one executable object file. (See Chapter 6 for details about binding.)

Every program must consist of one main program. It may also contain one or more modules. Chapter 2 contains a description of the main program's format. A main program must begin with the keyword **program**. A module begins with the keyword **module**. It takes the format shown in Figure 7–1.

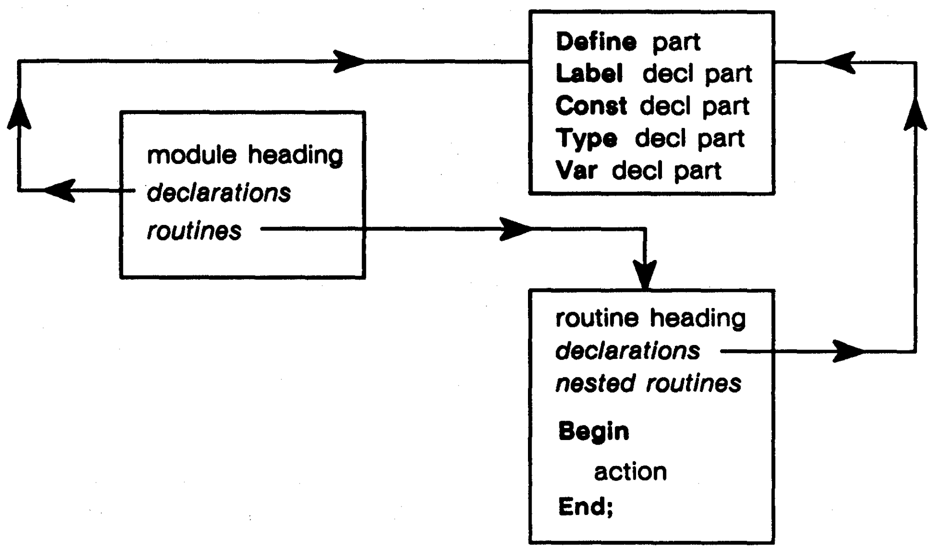

*Figure 7-1. Format of a Module*

At run time, the start address of the program is the first statement in the main routine of the main program.

The format of a module is very similar to the format of the main program shown in Figure 7-1. The differences between the formats are:

- A module takes a module heading rather than a program heading. (See the next section for a description of the module heading.)
- A module can contain zero or more routines; each routine must have a name. That is, the main program always contains one main (unnamed) routine, but a module must consist of named routines only.
- The declarations part of a module may contain a **define** part. The "Method 2" section of this chapter describes the **define** part.

## 7.1.1 Module Heading

The module heading is similar to a program heading except that it starts with the keyword **module** instead of **program**, and that it cannot take a **file_list**. Therefore, the module heading takes the following format:

**module** *name* [, *code_section_name*] [, *data_section_name*];

*Name* must be an identifier. The name you pick has no effect on the module.

*Code_section_name* and *data_section_name* are optional elements of the module heading. Use them to specify nondefault section names for the code and data in the module. A section is a named contiguous area of memory that shares the same attributes. (See the *Domain/OS Programming Environment Reference* for details on sections and attributes.) By default, Domain Pascal assigns all the code in your module to the **.text** section and all the data in your module to the **.data** section. To assign your code and data to nondefault sections, specify a *code_section_name* and a *data_section_name*.

Chapter 2 described the format of a main program, this chapter describes the format of a module, and Chapter 6 detailed the method for compiling and binding modules and main programs. The following sections explain how to write your source code so that the separately compiled units can communicate with one another.

## 7.2 Accessing a Variable Stored in Another Pascal Module

Domain Pascal provides four methods for accessing the value of a variable stored in a separately compiled file. The trick to the first three methods is using the correct *variable_allocation_clause* when declaring variables. The optional *variable_allocation_clause* precedes the data type in a **var** declaration part as shown below:

**var** (*section_name*)

*identifier_list1* : [*variable_allocation_clause*] *typename1*;

[

.

.

.

*identifier_listN* : [*variable_allocation_clauseN*] *typenameN*;]

For *variable_allocation_clause*, enter one of the following three identifiers:

- **Define**—tells Pascal to allocate the variable in a static data area and make its name externally accessible.
- **Extern**—tells Pascal to not allocate the variable because it is possibly allocated in a separately compiled module or program.
- **Static**—tells Pascal to allocate the variable in a static data area and keep its name local to this module or program. Use the **static** clause when a variable needs to retain its value from one execution of a routine to the next.

  For example, the following fragment declares **x** as a statically allocated variable:

```
VAR
        x : static integer16;
```

  After the execution of the routine in which **x** is declared, **x** still retains its value.

The following sections demonstrate three of the four methods for accessing a variable stored in another Domain Pascal module.

### 7.2.1 Method 1

The following fragments demonstrate the first method for accessing an externally stored variable:

```
program Math;                       module Sub2;
 var                                 var
    x : EXTERN integer16;               x : DEFINE integer16 := 8;

                                     Procedure twoa;
 BEGIN                               BEGIN
    writeln(x); {access x}              writeln(x); {access x}
 END.                                END;
```

This method uses the variable allocation clauses **extern** and **define** to make the value of **x** available to both **Math** and **Sub2**. The **extern** clause in **Math** tells the compiler that variable **x** is probably defined somewhere outside of **Math**. The **define** clause in **Sub2** tells the compiler that **x** is defined within **Sub2**; the ":= 8" tells the compiler to initialize **x** to 8. If **x** is not initialized, the **writeln(x)** statement generates undefined output.

When you bind, the binder matches the external reference to x in program **Math** with the global symbol **x** defined in module **Sub2**. Note that you can specify **x** as an **extern** in as many modules as you want; however, you should only **define** **x** in one module. If you **define** **x** in more than one module, the binder reports an error.

### 7.2.2 Method 2

The following fragments demonstrate the second method for accessing an externally stored variable:

```
program Math;                       module Sub2;
 var                                 var
    x : EXTERN integer16;               x : EXTERN integer16;
                                     DEFINE
                                        x := 8;

                                     Procedure twoa;
 BEGIN                               BEGIN
    writeln(x); {access x}              writeln(x); {access x}
 END.                                END;
```

Method 2 introduces the **define** statement. The **define** statement is similar to the **define** variable allocation clause. The **define** statement in **Sub2** serves two purposes. First, it tells the compiler that **x** is defined in module **Sub2**. Second, it tells the compiler that the initial value of **x** is 8.

The **define** statement takes the following syntax:

**define**

$\quad$ *variable1* $\left[\ := \textit{initial\_value1}\ \right]$;

$\qquad$ ...

$\left[\textit{variableN}\ \left[:= \textit{initial\_valueN}\right];\right]$

Notice that the *:= initial_value* is optional. If you do not provide an *initial_value,* the value of **variable** is undefined; that is, there is no way to predict what the variable's initial value will be.

**NOTE:** Order is crucial. If you use Method 2, the **var** declaration part must precede the **define** statement. (However, see Method 3 later in this chapter.)

Allowing a **define** statement in module Sub2 along with a corresponding **extern** clause may appear contradictory at first. After all, a **define** statement means "it's defined here" and an **extern** clause implies "it's defined somewhere else." However, allowing this practice can be useful. For example, when you write an **%include** file, there is no way for you to know where the **%include** file is going to end up. If the **extern** clause is stored in an **%include** file that happens to end up in a file with a matching **define** statement no harm is done.

### 7.2.2.1 Initializing Extern Variable Arrays

The "Initializing Array Variables" section in Chapter 3 contained a lengthy section on initializing variable arrays. The following paragraphs provide additional details about initializing an array variable that has the **extern** variable allocation clause.

Ordinarily the **-subchk** compiler option causes the compiler to generate code to check that subscripts are within the defined range of an array. However, the compiler does not generate this extra code if you do not specify initialization data for an **extern** array variable. The compiler cannot check subscripts because the upper bound is not known. If there is data initialization with the **extern** declaration, the data is ignored until the variable is defined in a file. However, since the upper bound is known, the compiler can perform subscript checking.

When you let the compiler determine the upper bound of the array, make sure that variable declarations do not share the same type declaration. If they do, the first data initialization for that variable sets the size for all other variables in the type declaration. All initialized variables are then checked against that size, just as if you had specified a constant upper bound.

For example, consider the following fragment:

```
VAR
    table1, table2  : EXTERN array[1..*] of char;
    table3          : EXTERN array[1..*] of char;
    table4          : EXTERN array[1..*] of char;

 DEFINE
    table1 := 'This table sets the size,';
    table2 := 'so this table will be truncated';
    table3 := 'But separate variable declarations';
    table4 := 'solve the problem.';
```

In the preceding example, variables **table1** and **table2** share the same anonymous type declaration. Since **table1** precedes **table2**, Domain Pascal uses the defined size of **table1** to set the size for **table2**. In this case, **table1** is a 25-character string. Therefore, the compiler truncates the string **"so this table will be truncated"** to its first 25 characters. Variables **table3** and **table4** are declared separately, so their initializations do not affect each other.

Domain Pascal issues a warning message when it truncates arrays with a base type of **char**, as for **table2**. However, if the array has a base type other than **char**, the compiler issues an error message when it truncates it.

## 7.2.3 Method 3

The following fragments demonstrate the third method for accessing an externally stored variable:

```
program Math;                       module Sub2;
 VAR                                 DEFINE
    x : EXTERN integer16;               x;
                                     VAR
                                        x : EXTERN integer16 := 8;

                                     Procedure twoa;
 BEGIN                               BEGIN
    writeln(x); {access x}              writeln(x); {access x}
 END.                                END;
```

Method 3 is similar to Method 2 except that the initialization (:= 8) is done in the **var** declaration part rather than in the **define** statement.

**NOTE:** Order is crucial. If you use Method 3, the **define** statement must precede the **var** declaration part.

### 7.2.4 Method4

The following fragment demonstrates this fourth method for accessing an externally stored variable:

```
Program Math;                               Module Sub2;
 VAR (my_sec)                                VAR (my_sec)
    x : integer16 := 5;                         x : integer16;
    y : real := 6.2;                            y : real;
    q : array[1..3] of char := 'cat';           q : array[1..3] of char;

                                             Procedure twoa;
 BEGIN                                       BEGIN
    writeln(x, y, z);                           writeln(x, y, z);
 END.                                        END;
```

Notice that the variables in each module occupy the same nondefault section name, **my_sec**. This means that at run time, the variables **x**, **y**, and **q** from **Math** occupy the same memory locations as x, y, and **q** from **Sub2**. Therefore, whatever is assigned to x in **Math**, can be retrieved in **Sub2**, and vice versa.

There's no requirement that the variables in **Sub2** have the same names as the variables in **Math**. However, it does make your program far easier to understand if you do give them the same names. Taking this philosophy one step further, you could greatly simplify variable declaration by putting a named **var** declaration part into a separate file and using **%include** to put it into every module.

You should preserve the order of declarations from one module to the next. For example, suppose the variable declarations look like this:

```
Program Math;                               Module Sub2;
 VAR (my_sec)                                VAR (my_sec)
    x : integer16 := 5;                         y : real;
    y : real := 6.2;                            x : integer16;
    q : array[1..3] of char := 'cat';           q : array[1..3] of char;
```

If you try to access x or y in **Sub2**, you get garbage results at run time. That's because when compiling **Sub2**, the compiler sees y as being the first four bytes in **my_sec**. However, when compiling **Math**, the compiler sees y as being bytes 2 through 5 in **my_sec**.

## 7.3 Accessing a Routine Stored in Another Pascal Module

Domain Pascal provides two methods for accessing a procedure or function stored in a different module or program. This section explains both methods, but first describes the **extern** and **internal** routine options, which are both critical to that description.

### 7.3.1 Extern

The routine option **extern** serves exactly the same purpose for routines that the variable allocation clause **extern** serves for variables. Namely, it specifies that a routine is possibly defined in a separately compiled file. By specifying a routine as **extern**, you can call it even if it is defined in another file.

Chapter 5 describes the format of routine options. **Extern** is like any other routine option, except that when you use **extern**, you put the routine heading at the end of the main declaration part of the program or module.

### 7.3.2 Internal

By default, all routine names in modules are global symbols, and all routine names in main programs are local symbols. A routine name from the main program can never become a global symbol. However, you can make a routine name from a module into a local symbol by specifying the routine option **internal**. See the "Routine Options" section in Chapter 5 for details on the syntax rules of routine options.

### 7.3.3 Method 1

Figure 7-2 demonstrates the first method for accessing an externally stored routine. In this example, program **math** refers to function **exponent**; however, function **exponent** is stored in module **math2**. Therefore, function **exponent** is declared as an **extern** in program **math**. There is no need to do anything special to module **math2** because the compiler automatically makes **exponent** a global symbol in **math2**. At bind time, the binder resolves external symbol **exponent** with global symbol **exponent**. To keep **exponent** local to module **math2**, you could change the following line in the source code:

```
FUNCTION exponent(number, power : INTEGER16) : REAL;
```

to the line shown below:

```
FUNCTION exponent(number, power : INTEGER16) : REAL; INTERNAL;
```

The sample program follows.

```
{***********************************************************************}
PROGRAM math;
 VAR
      n, p : integer16;
      answer : real;

 FUNCTION exponent(n,p : integer16) : real; EXTERN; {exponent is defined
                                                     outside this file.}
 BEGIN
     writeln('The main program calls a separately-compiled routine in');
     writeln('order to calculate the value of an integer raised to an');
     writeln('integer power.');
     writeln;
     write('Enter an integer -- ');     readln(n);
     write('Raised to what power -- '); readln(p);
     answer := exponent(n,p);
     writeln(n:1, ' raised to the ', p:1, ' power is ', answer:1);
 END.


MODULE math2;                        {You must bind this module with
                                     program math.}
 VAR
     number, power, count : INTEGER16;
     run : INTEGER32;

 FUNCTION exponent(number, power : INTEGER16) : REAL;
 BEGIN
     writeln('This is module math2');
     if power = 0 then exponent := 1;
     run := 1;
     for count := 1 to abs(power) do
           run := run * number;
     if power > 0
         then exponent := run
         else exponent := (1 / run);
 END;
{***********************************************************************}
```

*Figure 7-2. Method 1 for Accessing an External Routine*

### 7.3.4 Method 2

Figure 7-3 demonstrates the second method for accessing an externally-stored routine.

```
{**********************************************************************}
PROGRAM math;
 VAR
      n, p : integer16;
      answer : real;

 FUNCTION exponent(n,p : integer16) : real; EXTERN;
 {exponent is defined outside this file.}

 BEGIN
     writeln('The main program calls a separately-compiled routine in');
     writeln('order to calculate the value of an integer raised to an');
     writeln('integer power.');
     writeln;
     write('Enter an integer -- ');     readln(n);
     write('Raised to what power -- '); readln(p);
     answer := exponent(n,p);
     writeln(n:1, ' raised to the ', p:1, ' power is ', answer:1);
 END.


MODULE math2;                {You must bind this module with program
                             math}

 DEFINE
     exponent;

 FUNCTION exponent(number, power : INTEGER16) : REAL; EXTERN;
 VAR
     number, power, count : INTEGER16;
     run : INTEGER32;

 FUNCTION exponent;
 BEGIN
     writeln('This is module math2');
     if power = 0 then exponent := 1;
     run := 1;
     for count := 1 to abs(power) do
           run := run * number;
     if power > 0
         then exponent := run
         else exponent := (1 / run);
 END;
{**********************************************************************}
```

*Figure 7-3. Method 2 for Accessing an External Routine*

Under Method 2 you call external routines exactly as you do in Method 1. Therefore, program **math** is unchanged from Method 1. However, you must make the following three changes in module **math2**:

- **define exponent;** — Use a **define** statement to tell the compiler that **exponent** is a global symbol defined in module **math2**. With one exception, **define** takes the syntax described in the "Accessing a Variable Stored in Another Pascal Module" section earlier in this chapter. The one exception is that you cannot assign an initial value to a procedure or function symbol.

- **function exponent (number, power : INTEGER16) : REAL; EXTERN;** — Copy the function declaration from program **math**. Since this line should be an exact copy of the function declaration in the calling module, you should put this line into a separate file and use the **%include** directive to include it in both the module and the program. That way, you only keep one version of the function declaration.

- **function exponent;** — Notice that there are no arguments here; only the name of the function.

Figure 7-4 consists of one main program and two modules. The main program calls two external routines stored in the two modules. The two modules access some variables that the main program defines.

```
{*************************************************************************}
Program relativity1;

 CONST
    speed_of_light = 2.997925e8;

 { Both of the following routines are defined in another file: }
 Procedure convert_mph_to_light(IN speed_in_mph   : real;
                                OUT fraction_of_c : real); EXTERN;
 Function contracted_length(IN rest_length  : real;
                            IN fraction_of_c: real) : real; EXTERN;
 VAR
    speed_in_mph, fraction_of_c, rest_length : real;
    c : DEFINE real := speed_of_light;
                                  {Make the value of c globally known.}
 BEGIN
    write('Enter the speed of the object (in mph) -- ');
    readln(speed_in_mph);
    convert_mph_to_light(speed_in_mph, fraction_of_c);
    write('What is the rest length of the object (in meters) -- ');
    readln(rest_length);
    write('It will be perceived in the rest frame of reference as ');
    writeln(contracted_length
                         (rest_length, fraction_of_c):8:6, ' meters');
 END.

Module relativity2;
 DEFINE
     convert_mph_to_light;

 Procedure convert_mph_to_light(IN speed_in_mph   : real;
                                OUT fraction_of_c : real); EXTERN;
 VAR
     speed_in_mph, speed_in_mps, fraction_of_c, percentage : real;
     c : EXTERN real;

 { The following procedure converts a speed (given in miles per hour) }
 { into a percentage of the speed of light.                           }

 Procedure convert_mph_to_light;
 BEGIN
    speed_in_mps := (0.44704 * speed_in_mph);
    fraction_of_c := speed_in_mps / c;
    percentage := (100 * fraction_of_c);
    writeln('This speed is ', percentage:8:6, ' percent of c.');
 END;
                                                          {continued}
{*************************************************************************}
```

*Figure 7-4. Another Example of Calling External Routines*

```
{***********************************************************************}
Module relativity3;

 DEFINE
    contracted_length;

 Function contracted_length(IN rest_length   : real;
                            IN fraction_of_c : real) : real; EXTERN;

 VAR
    rest_length : real;
    speed_in_mps : real;

 { This function calculates the relativistic length contraction. }
 Function contracted_length;
 BEGIN
    contracted_length := rest_length * sqrt(1 - sqr(fraction_of_c) );
 END;
{***********************************************************************}
```

*Figure 7-4. Another Example of Calling External Routines (Cont.)*

Suppose you store program **relativity1** in file **relativity1.pas**, module **relativity2** in file **relativity2.pas**, and module **relativity3** in file **relativity3.pas**. To compile the three files, enter the following three commands:

```
$ pas relativity1
$ pas relativity2
$ pas relativity3
```

You must now bind them together by entering a command like the following:

```
$ bind relativity.bin relativity2.bin relativity3.bin -b einstein
```

Here is a sample execution of the program:

```
$ einstein
Enter the speed of the object (in mph) -- 4500000
This speed is 0.667121 percent of c.
What is the rest length of the object (in meters) -- 10
It will be perceived in the rest frame of reference as 9.999778 meters
```

## 7.4 Calling a FORTRAN Routine from Pascal

Domain Pascal permits you to call routines written in Domain FORTRAN source code. To accomplish this, perform the following steps:

1. Write source code in Domain Pascal that refers to an external routine. Compile with the Domain Pascal compiler. Domain Pascal creates an object file.

2. Write source code in Domain FORTRAN. Compile with the Domain FORTRAN compiler. Domain FORTRAN creates an object file.

3. Bind the object file(s) the Pascal compiler created with the object file(s) the FORTRAN compiler created, using the **-b** binder option to tell the binder to create one executable object file.

4. Execute the object file as you would execute any other object file.

The following sections describes steps 1 and 2. For information on steps 3 and 4, see Chapter 6.

**NOTE:** The following sections explain how to call Domain FORTRAN from Domain Pascal. If you want to learn how to call Pascal from FORTRAN, see the *Domain FORTRAN Language Reference*.

## 7.5 Data Type Correspondence for Pascal and FORTRAN

There is no great difference between making a call to a FORTRAN function or subroutine and making a call to an **extern** Pascal routine. However, before passing data between Domain Pascal and Domain FORTRAN, you must understand how Pascal data types correspond to FORTRAN data types. Table 7-1 lists these correspondences.

*Table 7-1. Domain Pascal and Domain FORTRAN Data Types*

| Domain Pascal | Domain FORTRAN |
|---|---|
| **Integer, Integer16** | **Integer*2** |
| **Integer32** | **Integer, Integer*4** |
| **Real, Single** | **Real, Real*4** |
| **Double** | **Double Precision, Real*8** |
| **Char** | **Character*1** |
| **Boolean** | **Logical*1** |
| **Set** | set emulation calls |
| User-defined record as shown in Section 7.5.2 | **Complex, Complex*16** |
| **Array** | array (with restrictions) |
| **Pointer** | **Pointer** statement |

The integer, real, and character data types in both languages correspond very well to each other. For example, Pascal's **integer16** data type is identical to FORTRAN's **integer*2** 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. **Integer** in Domain Pascal is a 2-byte entity, while in Domain FORTRAN, **integer** is four bytes by default. To avoid any confusion, you should use the specific integer data types (**integer16**, **integer32**, **integer*2**, and **integer*4**) rather than the generic **integer**.

Unlike Pascal, Domain FORTRAN doesn't actually support a set data type. However, you can make special set emulation calls from within a Domain FORTRAN program. Therefore, you can pass a set variable as an argument from a Pascal program and use the set emulation calls within your FORTRAN program.

## 7.5.1 Boolean and Logical Correspondence

Pascal's **boolean** data type and FORTRAN's **logical*1** data type are identical 1-byte objects. They may be freely passed between programs written in the two languages.

In some instances, however, you may need to use FORTRAN's 4-byte **logical** (the default if you omit a size specification in the declaration) or **logical*4** (which is equivalent to **logical**) to communicate with a Pascal routine. FORTRAN's 4-byte **logical** takes up four bytes of memory. The true or false value is set in all four bytes. By default, a Pascal **boolean** object consumes only one byte. To make the data types match, therefore, you should use the **long** size attribute to create a 4-byte **boolean** type, as shown in the following fragment:

```
TYPE
    bool4 = [long] boolean;
```

See the "Size—Extension" section of Chapter 3 for details about the **long** attribute.

## 7.5.2 Simulating FORTRAN's Complex Data Type

Unlike FORTRAN, Domain Pascal doesn't support a predeclared **complex** data type. However, you can easily declare a Pascal record type that emulates **complex**. In FORTRAN, **complex** consists of two single-precision real numbers. Therefore, you could define a **complex** Pascal record as follows:

```
TYPE
    complex = record
                r         : single;
                imaginary : single;
              end;
```

Similarly, you could define a **complex*16** Pascal record as follows:

```
TYPE
    complex16 = record
                r         : double;
                imaginary : double;
              end;
```

## 7.5.3 Array Correspondence

Single–dimensional arrays (including **boolean/logical** arrays when you make the adjustments described above) of the two languages correspond perfectly; for example:

| *In Domain Pascal* | *In Domain FORTRAN* |
|---|---|
| `x : Array[1..10] of CHAR` | `character*10   x` |
| `x : Array[1..50] of INTEGER16` | `integer*2  x(50)` |
| `x : Array[1..20] of DOUBLE` | `real*8     x(20)` |

The one exception is as follows: you cannot call from Pascal a Domain FORTRAN routine that has as a parameter a **character** array of unspecified length. For example, do not specify an array like the following as a parameter in the Domain FORTRAN routine:

```
CHARACTER*(*)    x
```

Multidimensional arrays in the two languages do not correspond very well. The tricky part is that Pascal represents multidimensional arrays differently than FORTRAN. To represent arrays in Domain Pascal, the rightmost index varies fastest. For example, Domain Pascal represents the six elements of an **array[1..2, 1..3]** in the following order:

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

However, the leftmost element varies fastest in Domain FORTRAN arrays. Therefore, Domain FORTRAN represents the six elements of an **array(2,3)** in the following order:

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

Obviously this can lead to confusion if you pass a multidimensional Pascal array as an argument to a Domain FORTRAN parameter. However, there is a way to avoid this confusion: declare the array dimensions of the Domain FORTRAN parameter in reverse order. For example, instead of declaring **integer*4 array(2,3)**, declare **integer*4 array(3,2)**. Following are two more examples:

| **Argument in Pascal** | **Parameter in FORTRAN** |
|---|---|
| `x : array[1..5, 1..10] of real` | `real*4  x(10,5)` |
| `x : array[1..2, 1..3, 1..4] of real` | `real*4  x(4,3,2)` |

## 7.6 Passing Data Between FORTRAN and Pascal

There are two ways to pass data between a Domain Pascal program and a Domain FORTRAN function or subroutine. You can either establish a common section of memory for sharing data, or you can pass the data as an argument to a routine. The next section demonstrates the second method, while the following paragraphs demonstrate the first method.

Earlier in this chapter you learned how to use a named section to pass data between two separately compiled Pascal modules. A named section in Domain Pascal is identical to the named **common** area of Domain FORTRAN. If you give a named section the same name as a named **common** area, the binder establishes a section of memory for sharing data.

For example, suppose that you want both a Pascal program and a FORTRAN function or subroutine to access two variables—a 16-bit integer and an 8-byte (double-precision) real number. In the Domain Pascal program, you can declare the two variables as follows:

```
VAR (my_sec)
     x : INTEGER16;
     d : DOUBLE;
```

If you want the value of these two variables to be accessible from the Domain FORTRAN program, declare them as follows in the FORTRAN program:

```
INTEGER*2  x
REAL*8     d
COMMON /my_sec/ x,d
```

Remember to preserve the same order of variable declaration in the **common** statement that you did in the **var** declaration part. For example, you will get peculiar run-time results if you declare your **common** statement as

```
COMMON /my_sec/ d,x
```

## 7.7 Calling FORTRAN Functions and Subroutines

This section demonstrates how to call a Domain FORTRAN function or subroutine from a Domain Pascal program. Calling Domain FORTRAN from Domain Pascal is straightforward; the only possible complication is that the data types of the arguments and parameters may not correspond perfectly. (See the "Data Type Correspondence" section earlier in this chapter for ways to remedy the data type mismatches.)

Domain Pascal supports a variety of parameter types including value parameters, variable parameters, **in**, **out**, and **in out** parameters. (See Section 5.3.) Domain FORTRAN supports only one kind of parameter type, and it is equivalent to the **in out** parameter type in Domain Pascal. That is, Domain FORTRAN accepts whatever value(s) you pass to it, and in turn, always passes a value back.

### 7.7.1 Calling a Function

Figure 7-5 and Figure 7-6 show a Domain Pascal program that calls a Domain FORTRAN function. The call is trivial since Domain Pascal's **single** data type corresponds perfectly to the **real*4** data type of Domain FORTRAN.

```
{*******************************************************************}
program pas_to_ftn_hypo_func;
{ NOTE: You must also obtain the FORTRAN program named "hypotenuse."  }
{       After compiling pas_to_ftn_hypo_func and hypotenuse, you must }
{       bind them together.                                           }
{ This Pascal program calls an external function named HYPOTENUSE.    }
{ Compare to pas_to_ftn_hypo_sub, a program that  calls a subroutine  }

 VAR
     leg1, leg2 : single;

 function hypotenuse (in out leg1, leg2 : single) : single; extern;

 BEGIN

 writeln ('This program calculates the hypotenuse of a right');
 writeln ('triangle given the length of its two legs.');
 write ('Enter the length of the first leg -- ');
 readln (leg1);
 write ('Enter the length of the second leg -- ');
 readln (leg2);

 writeln ('The length of the hypotenuse is: '
               , hypotenuse(leg1,leg2):5:2);

 END.
{*******************************************************************}
```

*Figure 7-5. An Example of Calling FORTRAN:* **pas_to_ftn_hypo_func**

```
********************************************************************
 * This is a FORTRAN function for calculating the hypotenuse of a
 * right triangle. You don't have to do anything special to this file
 * to make it callable by Pascal. In fact, this function could just
 * as easily be called by a FORTRAN program.

       real*4 function hypotenuse(l1, l2)
       real*4 l1, l2
                 {real*4 corresponds to the Pascal data type single.}
       hypotenuse = sqrt((l1 * l1) + (l2 * l2))
       end
********************************************************************
```

*Figure 7-6. An Example of Calling FORTRAN:* **hypotenuse**

These programs are available online and are named **pas_to_ftn_hypo_func** and **hypotenuse**.

## 7.7.2 Calling a Subroutine

A function in Pascal corresponds to a function in FORTRAN. A procedure in Pascal corresponds to a subroutine in FORTRAN. In the example in Figure 7-7 and Figure 7-8, **hypotenuse** changes from a function to a subroutine, and the Pascal program changes to reflect that it is expecting an external procedure. Note that the Pascal program could actually be calling a Pascal procedure. There's nothing in the program that designates the language in which the called procedure is written.

```
{*********************************************************************}
program pas_to_ftn_hypo_sub;
{ NOTE: You must also obtain the FORTRAN program named "hypot_sub."  }
{       After compiling pas_to_ftn_hypo_sub and hypot_sub, you must  }
{       bind them together.                                          }
{ This Pascal program calls an external subroutine named HYPOTENUSE. }
{ Although program pas_to_ftn_hypo_sub is bound to hypot_sub, which  }
{ is a FORTRAN program, we could also bind it to a Pascal module     }
{ containing a procedure named HYPOTENUSE.                           }

 VAR
     leg1, leg2, result : real;

 procedure hypotenuse (in out leg1, leg2, result : real) ; extern;

 BEGIN

 writeln ('This program calculates the hypotenuse of a right');
 writeln ('triangle given the length of its two legs.');
 write ('Enter the length of the first leg -- ');
 readln (leg1);
 write ('Enter the length of the second leg -- ');
 readln (leg2);

 hypotenuse(leg1,leg2,result);
 writeln ('The length of the hypotenuse is: ', result:5:2);

 END.
{*********************************************************************}
```

*Figure 7-7. An Example of Calling FORTRAN:* **pas_to_ftn_hypo_sub**

```
*********************************************************************
 * This is a FORTRAN subroutine for calculating the hypotenuse of a
 * right triangle.

       subroutine hypotenuse(l1, l2, result)
       real*4 l1, l2, result

       result = sqrt((l1 * l1) + (l2 * l2))
       end
*********************************************************************
```

*Figure 7-8. An Example of Calling FORTRAN:* **hypot_sub**

These programs are available online and are named **pas_to_ftn_hypo_sub** and **hypot_sub**.

## 7.7.3 Passing Character Arguments

Passing arguments when two languages' data types match exactly is relatively easy, but passing them when they don't often means you need to do extra work.

If you pass a string of **chars** from Domain Pascal to Domain FORTRAN, you should add an extra parameter to the Pascal routine heading. This is because FORTRAN adds an implicit string length argument whenever it passes a **character** string back to a calling routine.

Suppose your Pascal program includes the following:

```
TYPE
     name = array[1..10] of char;
VAR
     first_name : name;
     len        : integer16;

procedure change_name(in out first_name : name;
                      in out        len : integer16); extern;

change_name(first_name,len);
                     {Assume "change_name" is a FORTRAN subroutine.}
```

This Pascal routine heading includes an "extra" parameter for the length of **first_name** that FORTRAN will add when Pascal calls the routine. The length argument must be of type **integer16** because FORTRAN's implicit length argument is an **integer*2**.

The FORTRAN routine heading does not explicitly include the length argument. For this example, it would look like this:

```
subroutine change_name(first_name)
character*10 first_name
```

If you send multiple strings to Domain FORTRAN and you include FORTRAN's implicit length arguments in the Pascal parameter list, the length parameters must always appear at the end of the routine heading. That is, it is not correct to list them as **string1, len1, string2, len2,** etc. For instance:

```
{ Pascal program fragment. }
TYPE
    fn = array[1..10] of char;
    ln = array[1..20] of char;
        . . .
procedure process_name(in out       first_name : fn;
                       in out   middle_initial : char;
                       in out        last_name : ln;
                       in out len1, len2, len3 : integer16); extern;
        . . .
process_name(first_name, middle_initial, last_name, len1, len2, len3);
        .
        .
        .

*  FORTRAN subroutine fragment.
      subroutine process_name(first_name, middle_initial, last_name)

      character*10 first_name
      character    middle_initial
      character*20 last_name
```

## 7.7.4 Passing a Mixture of Data Types

The Domain Pascal program in Figure 7-9 and the Domain FORTRAN subroutine in Figure 7-10 demonstrate passing a variety of data types.

```
{***********************************************************************}
program pas_to_ftn_mixed;
{ NOTE: You must also obtain the FORTRAN program named "mixed_types."  }
{       After compiling pas_to_ftn_mixed and mixed_types, you must bind}
{       them together.                                                  }
{ This program demonstrates passing arguments of several different data}
{ types to a FORTRAN subroutine.                                        }

TYPE
    last_names = array[1..10] of char;
    two_by_four = array[1..2, 1..4] of integer16;
    complex = record
                r         : real;
                imaginary : real;
              end;
    boolrec = record
                bool_var : boolean;
                a,b,c    : boolean;
              end;
VAR
    age   : integer32 := 1000000;
    lying : boolrec;
    name  : last_names := 'Tucker';
    multi : two_by_four := [ [5,8,11,14]
                             [100, 103, 106, 109] ];
    c     : complex := [4.53, 0.98];
    count1, count2, len : integer16;

procedure print_vals (in age : integer32; in lying : boolrec;
                      in name : last_names; in multi : two_by_four;
                      in c : complex);
{ This procedure prints the values of the variables. }

BEGIN
writeln ('Age = ', age:5);
writeln ('Lying = ', lying.bool_var:5);
writeln ('Name = ', name);

writeln ('Multi = ');
for count1 := 1 to 2 do
    begin
    for count2 := 1 to 4 do
        write(multi[count1,count2]:5);
    writeln;
    end; {for}

writeln ('Complex = ', c.r:4:2, ',', c.imaginary:5:2);
END;        {end procedure print_vals}
                                                            {continued}
{***********************************************************************}
```

*Figure 7-9. An Example of Calling FORTRAN:* **pas_to_ftn_mixed**

```
{*************************************************************************}
{ Note "extra" len argument in the parameter list of mixed_types.}
procedure mixed_types (in out age : integer32;
                       in out lying : boolrec;
                       in out name : last_names;
                       in out multi : two_by_four;
                       in out c : complex;
                       in out len : integer16);
                       extern;

BEGIN  {main program}
lying.bool_var := true;
writeln (chr(10), 'Before calling FORTRAN', chr(10));
print_vals(age, lying, name, multi, c);

mixed_types(age, lying, name, multi, c, len);
writeln (chr(10), 'After calling FORTRAN', chr(10));
print_vals(age, lying, name, multi, c);
END.
{*************************************************************************}
```

*Figure 7-9. An Example of Calling FORTRAN:* **pas_to_ftn_mixed** *(Cont.)*

```
*********************************************************************
 * This is a FORTRAN subroutine for assigning new values to arguments
 * passed in from a Pascal program. It demonstrates how to pass a
 * variety of data types.

       subroutine mixed_types (a,l,n,m,c)
       integer*4    a              { Declare variables.}
       logical      l
       character*10 n
       integer*2    m(4,2), count
       complex      c

 * Make reassignments.
       a = a * 2
       l  = .false.
       n = 'Carter'

       do count = 1,4
           m(count,l) = m(count,l) + 1000
       enddo

       c = (2.0, -2.0)
       end
*********************************************************************
```

*Figure 7-10. An Example of Calling FORTRAN:* **mixed_types**

These programs are available online and are named **pas_to_ftn_mixed** and **mixed_types**. If you compile, bind, and execute these programs, you get the following output:

```
Before calling FORTRAN

 Age = 1000000
 Lying =  TRUE
 Name = Tucker
 Multi =
     5    8   11   14
   100  103  106  109
 Complex = 4.53, 0.98


 After calling FORTRAN

 Age = 2000000
 Lying = FALSE
 Name = Carter
 Multi =
  1005 1008 1011 1014
   100  103  106  109
 Complex = 2.00,-2.00
```

### 7.7.5 Passing Procedures and Functions

Passing a Pascal routine to a FORTRAN subprogram is complicated by the fact that FORTRAN expects all arguments to be passed by reference. For this reason, you must pass a pointer to the Pascal routine, not the routine itself. To do this, you declare the type of the parameter as a procedure or function pointer, and pass the address of the routine as the actual argument.

The Pascal program shown in Figure 7-11 passes a function, an unsorted array, and the size of the array to a FORTRAN subroutine. 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 FORTRAN subroutine **sort_array** calls one of the two Pascal functions when it sorts the array.

```
{***************************************************************************}
program pass_func_to_fortran_p(input,output);

{  NOTE:  To execute this program, you must also obtain the      }
{  Pascal functions in the module "funcs_for_fortran_p" and the }
{  FORTRAN program named "sort_array_f".  After compiling       }
{  pass_func_to_fortran_p, funcs_for_fortran_p, and             }
{  sort_array_f, you must link them together.                   }

type
  arrtype = array [1..10] of integer32;
  proc_ptr_type = ^function(var x, y :integer32) : integer32;

var
  direction, i : integer32;
  intarr : arrtype := [ 27, 19, 34, 65, 7, 9, 2, 12, 75, 1 ];
  SIZE : integer32 := 10;

procedure sort_array(in compare : proc_ptr_type;
                     var size : integer32;
                     var list : arrtype); extern;

function ascend(var a, b : integer32) : integer32; extern;
function descend(var a, b : integer32) : integer32; extern;

begin
    writeln('Enter 1 to sort in ascending order, ',
            '2 to sort in descending order: ');
    readln(direction);
    if direction = 1 then
         sort_array(addr(ascend), SIZE, intarr)
    else
         sort_array(addr(descend), SIZE, intarr);
    for i := 1 to SIZE do
         writeln(intarr[i]);
end.
{***************************************************************************}
```

*Figure 7-11. An Example of Calling FORTRAN:* **pass_func_to_fortran_p**

Pascal can take the address of an external routine, but not of an internal routine. Because FORTRAN requires us to pass the address of the Pascal routine, we have to declare the **ascend** and **descend** functions **extern** in the Pascal program and define them in the separately compiled module shown in Figure 7-12.

```
{*********************************************************************}
MODULE funcs_for_fortran_p;

{  NOTE:  To execute this program, you must also obtain the }
{  Pascal program "pass_func_to_fortran_p" and the FORTRAN  }
{  program named "sort_array_f".  After compiling           }
{  pass_func_to_fortran_p, funcs_for_fortran_p, and         }
{  sort_array_f, you must link them together.               }

FUNCTION ascend(var a, b : integer32) : integer32;
BEGIN
  if a > b then
    ascend := 1
  else
    ascend := 0;
END;

FUNCTION descend(var a, b : integer32) : integer32;
BEGIN
  if b > a then
    descend := 1
  else
    descend := 0;
END;
{*********************************************************************}
```

*Figure 7-12. An Example of Calling FORTRAN:* **funcs_for_fortran_p**

The FORTRAN subroutine is shown in Figure 7-13.

```
C************************************************************************
C  Program name is "sort_array_f".
C  If you use f77, compile with the -WO,-nuc option.

C NOTE: You must also obtain the Pascal program named
C       "pass_func_to_fortran_p".  After compiling
C       pass_func_to_fortran_p and sort_array_f, you must
C       link them together.
       SUBROUTINE sort_array (compare, size, list)
       INTEGER*4 compare, size, list(size)
       INTEGER*4 i, temp
       LOGICAL out_of_order

       out_of_order = .TRUE.
       DO WHILE (out_of_order)
         out_of_order = .FALSE.
         DO 10 i = 1, size-1
              IF (COMPARE(list(i), list(i+1)).EQ.1) THEN
                   out_of_order = .TRUE.
                   temp = list(i)
                   list(i) = list(i+1)
                   list(i+1) = temp
              END IF
10       CONTINUE
       END DO
       RETURN
       END
C************************************************************************
```

*Figure 7-13. An Example of Calling FORTRAN:* **sort_array_f**

You can compile, link, and execute this program as follows:

```
$ pas pass_func_to_fortran_p.pas
No errors, no warnings, no info msgs, Pascal compiler 68K ...
$ pas funcs_for_fortran_p.pas
No errors, no warnings, no info msgs, Pascal compiler 68K ...
$ /usr/bin/f77 -c -W0,-nuc sort_array_f.f
sort_array_f.f:
$ bind pass_func_to_fortran_p.bin funcs_for_fortran_p.bin sort_array_f.o -b pass
All Globals are resolved.
$ pass
Enter 1 to sort in ascending order, 2 to sort in descending order:
1
          1
          2
          7
          9
         12
         19
         27
         34
         65
         75
```

## 7.8 Calling a C Routine from Pascal

In addition to allowing you to call FORTRAN routines, Domain Pascal permits you to call routines written in Domain/C source code. To accomplish this, perform the following steps:

1. Write source code in Domain Pascal that calls a routine. Compile it with the Domain Pascal compiler. Domain Pascal 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 Pascal compiler created with the object file(s) the C compiler created, using the **-b** binder option to create one executable object file.

4. Execute the object file as you would execute any other object file.

The remainder of this chapter describes steps 1 and 2. For information on steps 3 and 4, see Chapter 6.

NOTE: The following sections explain how to call Domain/C from Domain Pascal. If you want to learn how to call Pascal from C, see the *Domain/C Language Reference*.

## 7.8.1 Reconciling Differences in Argument Passing

Pascal usually passes arguments by reference, while C usually passes them by value. In order to pass arguments by reference correctly, you must declare your parameters in C to be pointers so that they can take the addresses Pascal passes in. The examples in the following sections demonstrate how to do this.

If you want to pass your Pascal arguments by value, you must use the **val_param** or **c_param** routine option in your procedure or function heading. See the "val_param—Extension" and "c_param—Extension" sections of Chapter 5 for more details about these routine options.

## 7.8.2 Case-Sensitivity Issues

When the Domain Pascal compiler parses a program, it makes all identifier names lowercase, regardless of the way you type the names in your source code. In contrast, the Domain/C compiler is case sensitive.

It is important to understand this when you are calling C from Pascal. 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 always-lowercased identifier names in the Pascal program.

## 7.8.3 Using Registers

By default, a Pascal function returning the value of a pointer type variable puts that value in address register A0, whereas, by default, C routines return values in data register D0. If you want to pass variables through registers, you should use the **d0_return** routine option for Pascal routines that call C routines. The **d0_return** option tells the compiler that any routines called by the routine with the **d0_return** option should return values to the D0 register. See the "d0_return—Extension" section of Chapter 5 for details about this routine option.

## 7.9 Data Type Correspondence for Pascal and C

Before you try to pass data between Domain Pascal and Domain/C, you must understand how Pascal data types correspond to C data types. Table 7–2 lists these correspondences.

*Table 7–2. Domain Pascal and Domain/C Data Types*

| Domain Pascal | Domain/C |
|---|---|
| **char** | **char** |
| **integer, integer16** | **short** |
| **integer32** | **int, long** |
| **real, single** | **float** |
| **double** | **double** |
| enumerated types | **enum** |
| record | **struct** |
| variant record | **union** |
| pointer(^) | pointer(*) |
| **boolean** | none |
| **set** | none |
| [byte] 0..255 | **unsigned char** |
| 0..65335 | **unsigned short** |
| 0..4295967295 | **unsigned long** |

As the table shows, the integer, real, and character data types in both languages correspond very well. For example, Pascal's **integer16** data type is identical to C's **short** data type, and a **double** variable in Pascal is the same as a **double** in C.

However, there are some important differences. Domain/C has no equivalent types for Pascal's **boolean** or **set** types, although you can simulate these types.

### 7.9.1 Passing Integers and Real Numbers

Since the Pascal 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.

Figure 7–14 shows a Pascal program that solicits the values for two sides of a right triangle. It then sends those values into the C function shown in Figure 7–15, which computes and returns the length of the hypotenuse. The arguments for the triangle's sides are 32 bits each, while the result is 64 bits.

```
{*********************************************************************}
PROGRAM pas_to_c_hypo;
{ NOTE: You must also obtain the C program named "hypot_c".   }
{       After compiling pas_to_c_hypo and hypot_c, you must   }
{       bind them together.                                   }
{ This program passes two real arguments to a C function.     }

 VAR
     leg1, leg2 : single;

 function hypot_c(in out leg1, leg2 : single) : double; extern;

 BEGIN

 writeln('This program calculates the hypotenuse of a right triangle ');
 writeln ('given the length of its two sides.');
 write ('Enter the lengths of the two sides: ');
 readln (leg1,leg2);

 writeln ('The triangle''s hypotenuse is: ', hypot_c(leg1,leg2):5:2);

 END.
{*********************************************************************}
```

*Figure 7-14. An Example of Calling C:* **pas_to_c_hypo**

```
/***************************************************************/
 /* This is a C function for finding the hypotenuse of a       */
 /* right triangle. The arguments must be declared as pointers.*/
 #include <math.h>
 double hypot_c(a,b)
        float *a,*b;
 {
 double result;
 result = sqrt((*a * *a) + (*b * *b));
 return(result);
 }
/***************************************************************/
```

*Figure 7-15. An Example of Calling C:* **hypot_c**

These programs are available online and are named **pas_to_c_hypo** and **hypot_c**. Following is a sample execution of the bound program.

```
 This program calculates the hypotenuse of a right triangle
 given the length of its two sides.
 Enter the lengths of the two sides: 3 4
 The triangle's hypotenuse is:  5.00
```

## 7.9.2 Passing Strings

In C, the end of a string is marked by a null character. Therefore, when a Pascal routine receives a string from C, the string probably includes a terminating null character that needs to be stripped. Conversely, when a Pascal routine sends a string to a C function, it should append a null character so that the C function can handle the string properly.

The simplest way to pass strings between Pascal and C routines is to use variable-length strings on the Pascal side. You can then use the **ctop** procedure to strip the null character from a C string, and the **ptoc** procedure to add a null character to a Pascal string.

The Pascal program in Figure 7-16 passes a variable-length string to the C function **capitalize,** shown in Figure 7-17, which converts all characters in the string to uppercase.

```
{**********************************************************************}
PROGRAM pas_to_c_strings;
{This program demonstrates how to pass variable-length strings to }
{and from C routines.                                              }
TYPE
     name = varying[15] of char;
     arg = array[1..15] of char;
VAR
     first_name  : name;
     last_name   : name;
PROCEDURE capitalize (in out first_name :arg) ; extern;
BEGIN
     first_name := 'sherlock';
     last_name := 'holmes';
     write('Before calling C, this is the name: ', first_name);
     writeln(' ',last_name);
     ptoc(first_name);
     ptoc(last_name);
     capitalize (first_name.body);
     capitalize (last_name.body);
     ctop(first_name);
     ctop(last_name);
     write('After calling C, this is the name: ', first_name);
     writeln(' ',last_name);
END.
{**********************************************************************}
```

*Figure 7-16. An Example of Calling C:* **pas_to_c_strings1**

```
/******************************************************************/
/* This function converts all characters in the string */
/* argument to uppercase */
#include <ctype.h>
void capitalize (s1)
char *s1;
{
     while (*s1)
    {
          *s1 = toupper(*s1);
          s1++;
    }
}
/******************************************************************/
```

*Figure 7-17. An Example of Calling C:* **capitalize**

Following is a sample execution of the bound program.

```
Before calling C, this is the name: sherlock holmes
After calling C, this is the name: SHERLOCK HOLMES
```

If you use fixed-length arrays rather than **varying** arrays, you need to handle the null character explicitly. The Pascal program in Figure 7-18 sends two strings to a C routine that prompts a user for new values and then sends the new string values back. The C function in Figure 7-19 strips the trailing null character from the strings.

```
{*********************************************************************}
PROGRAM pas_to_c_strings2;

{ NOTE: You must also obtain the C program named "pass_char".        }
{       After compiling pas_to_c_strings2 and pass_char, you must    }
{       bind them together.                                          }
{ This program shows how to pass character variables to from a C     }
{ routine.}

TYPE
    fn = array[1..10] of char;
    ln = array[1..15] of char;
VAR
    first_name : fn;
    last_name  : ln;

procedure pass_char(in out first_name : fn;
                    in out last_name  : ln); extern;
BEGIN
first_name := 'Sherlock';
last_name := 'Holmes';
write('Before calling C, the name is ', first_name:-1, ' ');
writeln(last_name:-1);
pass_char(first_name, last_name);
write('After calling C, the name is ', first_name:-1, ' ');
writeln(last_name:-1);
END.
{*********************************************************************}
```

*Figure 7-18. An Example of Calling C:* **pas_to_c_strings2**

```
/*********************************************************************/
/* NOTE: You must also get the Pascal program named pas_to_strings2.  */
/*       After compiling pas_to_c_strings2 and pass_char, you must    */
/*       bind the two together.                                        */
#include <stdio.h>

/* This C function takes two strings, prompts the user for new values,*/
/* and strips off the null characters before sending the strings back.*/

pass_char(first_name, last_name)
    char *first_name, *last_name;
{
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 previously used 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] = ' ';
}
/*********************************************************************/
```

*Figure 7-19. An Example of Calling C:* **pass_char**

These programs are available online and are named **pas_to_c_strings2** and **pass_char**. Following is a sample execution of the bound program.

```
 Before calling C, the name is Sherlock Holmes
Enter the first name and last name of a detective: Jane Marple
After calling C, the name is Jane Marple
```

## 7.9.3 Passing Arrays

Single-dimensional arrays (except for **boolean** arrays) of the two languages correspond fairly well. The major difference is that in C, array subscripts always begin at zero, while Pascal allows the programmer to determine the subscript at which the array begins. In order to make arrays match up, you should define your Pascal subscripts to begin at zero. For example:

| *In Domain Pascal* | *In Domain/C* |
|---|---|
| `x = array[0..9] of char` | `char x[10]` |
| `x = array[0..49] of integer` | `short x[50]` |
| `x = array[0..19] of single` | `float x[20]` |

With such declarations, the following code fragments access the identical elements in an array:

*In Domain Pascal* *In Domain/C*

```
for i := 0 to 9 do              for (i = 0; i < 10; i++)
  my_array[i] := i;               my_array[i] = i;
```

As described earlier, Pascal by default passes arguments by reference, so when it sends an array argument to a routine, it actually is sending the address of the first element in the array. C gets that address when you declare the array variable in C to be a pointer. This means you don't have to specify the size of a single-dimensional array that your C subprogram receives from Pascal.

That is, if your Pascal program includes the following—

```
type
     x = array[0..9] of integer32;
var
     my_array : x;
          . . .
 pass_array(my_array);
```

your C routine heading can look like this:

```
pass_array(my_array)
     long *my_array; /*Notice that there's no indication of the
                                  array's dimensions.             */
```

The example in Figure 7-20 and Figure 7-21 shows a Pascal program that loads five user-entered scores into a single-dimensional array and then sends that array to a C procedure to compute the average. Notice that the argument **size** determines the dimension of the array; the C declaration of the array contains no dimensioning information.

```
{****************************************************************************}
PROGRAM pas_to_c_arrays;
{ NOTE: You must also obtain the C program named "single_dim".  }
{       After compiling pas_to_c_arrays and single_dim, you must}
{       bind them together.                                     }
{ This program passes a one-dimensional array to a C routine.   }
TYPE
    scores = array[0..4] of integer;
VAR
    grades : scores;
    i, j   : integer;
    result : single;
    size   : integer := 5;
procedure single_dim(out result : single;
                     in    size : integer;
                     in  grades : scores); extern;
BEGIN
writeln ('Enter ', size:1, ' integer test scores separated by spaces.');
for i := 0 to 4 do
    read(grades[i]);
readln;
single_dim(result, size, grades);
writeln ('C computed the average of the test scores, and it is: ',
         result:5:2);
END.
{****************************************************************************}
```

*Figure 7-20. An Example of Calling C:* **pas_to_c_arrays**

```
/****************************************************************************/
/*NOTE: You must also get the Pascal program named pas_to_c_arrays    */
/*      After compiling pas_to_c_arrays and single_dim, you must bind */
/*      them together.                                                 */
/* This function computes the average of an array of integers.         */

single_dim(result,size,grades)
    float *result;
    short *size, *grades;
{
short i,total;
total = 0;
for (i = 0; i < *size; i++)     /* Add up array values */
    total += grades[i];         /* and then compute    */
*result = total / 5.0;          /* average.            */
}
/****************************************************************************/
```

*Figure 7-21. An Example of Calling C:* **single_dim**

These programs are available online and are named **pas_to_c_arrays** and **single_dim**. Following is a sample execution of the bound program.

```
Enter 5 integer test scores separated by spaces.
 85 92 100 79 96
 C computed the average of the test scores, and it is: 90.40
```

Multidimensional arrays in the two languages also correspond fairly well. Both languages store such arrays in the same order; that is, the rightmost subscript varies fastest. So these two arrays would be stored identically:

| *In Domain Pascal* | *In Domain/C* |
|---|---|
| `x = array[0..1,0..2] of integer32;` | `long *x[2][3];` |

## 7.9.4 Passing Pointers

Passing pointers between Pascal and C is fairly straightforward. In both cases, pointers are 4–byte entities. The example in Figure 7–22 and Figure 7–23 shows a simple linked–list application. The Pascal program creates the first element of the list and then calls the C routine **append** to add new elements to the list. The routine **printlist** is a Pascal routine that prints the entire list. In addition to illustrating how to pass pointers, this example also shows the correspondence of Pascal records to C structures.

```
{***********************************************************************}
PROGRAM pas_to_c_ptrs;

{This program calls an external C function that appends an item}
{to a linked list.  The program then prints the contents of the}
{list. You must also compile the C module append and bind it   }
{with pas_to_c_ptrs in order to obtain an executable file.     }
  TYPE
     link = ^list;
     list = record
            data : char;
            p : link;
            end;
  VAR
     last_let     : char := 'z';
     val          : char;
     base, first  : link;

  PROCEDURE append (in base: link;
                    in  val: char); EXTERN;

  PROCEDURE printlist;
  {printlist prints the data in each member of the linked list }
  BEGIN
  while base <> nil do
     begin
     writeln(base^.data);
     base:=base^.p;
     end;
  END;

  BEGIN  {main program}
  base:=nil;
  new(first);
  first^.data := 'a';   {Assign to first element of the list      }
  first^.p := base;     {The first element is also the last so    }
                        {set the pointer to nil                   }
  base := first;        {Base points to the beginning of the list }
  val := 'b';
  append (base,val);
  append(base, last_let);
  printlist;            {Call procedure to print contents of list.}
  END.  {main program}
{***********************************************************************}
```

*Figure 7–22. An Example of Calling C:* **pas_to_c_ptrs**

```
/***************************************************************************/
 /* C function that appends items to a linked list.                         */

 #module pass_pointers_c
 #include <stdio.h>

 typedef struct
     {
     char data;
     struct list *next;
     } list;

 void append (base, val)
     list **base;                  /* Note extra level of indirection */
     char *val;                    /* because arguments are passed by */
                                   /* reference                       */
 {
 list *newdata, *last_rec;

 last_rec = *base;                 /* Point temp variable last_rec at  */
                                   /* the beginning of the list.       */
 newdata = (list*)malloc(sizeof(list));
                                   /* Allocate memory for new element. */

 while (last_rec->next != NULL)    /* Walk to the end of the list.     */
     last_rec = last_rec->next;

 last_rec->next = newdata;         /* Add new data.                    */
 newdata->data = *val;
 newdata->next = NULL;

 }
/***************************************************************************/
```

*Figure 7-23. An Example of Calling C:* **append**

These programs are available online and are named **pas_to_c_ptrs** and **append**. If you compile and bind the programs, and execute the result, you get this output:

```
a
b
z
```

### 7.9.5 Passing Procedures and Functions

You may pass a Pascal procedure or function as an argument to a C function. The Pascal program shown in Figure 7-25 passes a function, an unsorted array, and the size of the array to a C function. The function being passed 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 Pascal functions when it sorts the array.

C expects function arguments to be passed by value. Therefore, the Pascal program declares the first parameter of the C function **sort_array** to be a **function**, and declares the C function with the **c_param** option. This use of **c_param** makes Pascal pass the other parameters by value as well, so that the C function does not declare them as pointers. (See Section 5.5.12 for information about **c_param**.)

Figure 7-24 shows the C function.

```
/**********************************************************************/
/* NOTE: Program name is "sort_array_c".  You must also
 *       obtain the C program named  "pass_func_to_c_p".
 *       After compiling pass_func_to_c_p and sort_array_c,
 *       you must link them together.
 */

void sort_array(int (*compare)(), int size, int list[])
{
    int i, temp, out_of_order;

    do {
         out_of_order = 0;
         for (i = 0; i < size-1; i++)
              if (compare(list[i], list[i+1])) {
                   out_of_order = 1;
                   temp = list[i];
                   list[i] = list[i+1];
                   list[i+1] = temp;
              }
    } while (out_of_order);
}
/**********************************************************************/
```

*Figure 7-24. An Example of Calling C:* **sort_array_c**

```
{*********************************************************************}
program pass_func_to_c_p(input,output);

{  Program name is "pass_func_to_c_p".  To execute             }
{  this program, you must also obtain the C program named      }
{  "sort_array_c".  After compiling pass_func_to_c_p and       }
{  sort_array_c, you must link them together.                  }

const
  SIZE = 10;
type
  arrtype = array [1..SIZE] of integer32;
var
  direction, i : integer32;
  intarr : arrtype := [ 27, 19, 34, 65, 7, 9, 2, 12, 75, 1 ];

procedure sort_array(function compare (a, b : integer32) :
                         integer32;
                     size : integer32;
                     in out list : arrtype);
                         options(c_param, extern);

function ascend(a, b : integer32) : integer32;
begin
  if a > b then
    ascend := 1
  else
    ascend := 0;
end;

function descend(a, b : integer32) : integer32;
begin
  if b > a then
    descend := 1
  else
    descend := 0;
end;

begin
    writeln('Enter 1 to sort in ascending order, ',
            '2 to sort in descending order: ');
    readln(direction);
    if direction = 1 then
         sort_array(ascend, SIZE, intarr)
    else
         sort_array(descend, SIZE, intarr);
    for i := 1 to SIZE do
         writeln(intarr[i]);
end.
{*********************************************************************}
```

*Figure 7-25. An Example of Calling C:* **pass_func_to_c_p**

You can compile, link, and execute the program as follows:

```
$ pas pass_func_to_c_p.pas
No errors, no warnings, no info msgs, Pascal compiler 68K ...
$ /bin/cc -c sort_array_c.c
$ bind pass_func_to_c_p.bin sort_array_c.o -b pass
All Globals are resolved.
$ pass
Enter 1 to sort in ascending order, 2 to sort in descending order:
2
        75
        65
        34
        27
        19
        12
         9
         7
         2
         1
```

## 7.9.6 Data Sharing Between C and Pascal

There are two ways to declare global variables in Pascal and C so that the linker can resolve references:

- Declare the variables so that they are placed in the **.data** or **.bss** sections.
- Declare the variables so that they are placed in named overlay sections.

We describe these two methods in the following sections.

### 7.9.6.1 Declaring .data and .bss Global Variables

When you use **define** to define an external variable, the compiler places that variable in the **.data** section of the the object file. When you declare a variable as **extern**, that variable is listed as an unresolved reference in the symbol table. If you want these global variables to be compatible with global variables in C programs, you must compile with **/bin/cc** or use the **-bss** switch with **/com/cc**.

There are several scenarios for declaring and defining variables in Pascal and C. The three most common are described below:

- **Define a variable in Pascal and allude to it in C.** For example, the Pascal source file might contain the following:

  ```
  VAR    x: DEFINE INTEGER32 := 0;
  ```

  and the C file would contain:

  ```
  extern int x;
  ```

In this case, the definition in the Pascal module causes the compiler to allocate space for x in the **.data** section. The C declaration produces an undefined reference to x in the symbol table, which is resolved by the linker.

- **Define a variable in C (initialized) and allude to it in Pascal.** For example, the C file would contain:

  ```
  int x = 10;
  ```

  and the Pascal source file would declare **x** as:

  ```
  VAR x: EXTERN INTEGER32;
  ```

  In this case, the definition of **x** in the C module forces the C compiler to allocate space for **x** in the **.data** section. The declaration of **x** in the Pascal file causes the compiler to produce an undefined reference to **x** in the symbol table, which is resolved by the linker.

- **Define a variable in C (uninitialized) and allude to it in Pascal.** For example, the C file would contain:

  ```
  int x;
  ```

  and the Pascal source file would declare **x** as:

  ```
  VAR x: EXTERN INTEGER32;
  ```

  In this case, the uninitialized definition of **x** in the C module causes the C compiler to make a "weakly defined" entry in the symbol table. The declaration of **x** in the Pascal file causes the compiler to produce an undefined reference to **x** in the symbol table. The linker then places **x** in the **.bss** section, initialized to zero, and resolves the Pascal reference.

It is also possible to define the same variable in C and in Pascal, as long as only one or neither of the definitions contain initializers. If both definitions contain initializers, the linker will report an error.

In the following example, we define the global variable **xx** at the top of the C source file; the function **main()** prints the initial value of xx and then calls the C routine **add_three()** which adds 3 to **xx**; finally, **add_three** calls the Pascal procedure **sub_two** which subtracts 2 from **xx**.

The Pascal routine is:

```
MODULE global_var_p;

PROCEDURE sub_two;
    VAR
      xx : EXTERN INTEGER32;

    BEGIN
      xx := xx - 2;
      WRITELN('Value of xx after sub_two():',xx);
    END;
```

The C routines are:

```
#module global_var_c
#include <stdio.h>

int xx = 1;                 /* Definition of xx */
int main( void )
{
    extern void add_three( void );
    printf( "Initial value of xx: %d\n", xx );
    add_three();
}

void add_three( void )
{
    extern void sub_two( void );
    xx += 3;
    printf( "Value of xx after add_three(): %d\n", xx );
    sub_two();
}
```

The result of executing the program is:

```
Initial value of xx: 1
Value of xx after add_three(): 4
Value of XX after sub_two():         2
```

### 7.9.6.2 Creating Overlay Data Sections

Both C and Pascal have syntaxes that enable you to produce named overlay sections for global data. Since the binder ensures that overlay sections with the same name refer to the same memory locations, this mechanism enables you to share data across procedures.

In Pascal, you create an overlay section with the syntax:

**VAR** ( *section_name* )
  *declaration*
  *declaration*
    .
    .
    .

For instance, the following statements define an overlay section called **example** with two variables:

```
VAR (example)
    x : INTEGER16;
    y : DOUBLE;
```

In C, there are two ways to create overlay sections. If you use the **/com/cc** compiler, you can create an overlay section simply by defining an external variable. All external variables are automatically stored in their own named sections. For instance, if compiled with **/com/cc**, the declarations shown below create three overlay sections called **first_sec**, **second_sec**, and **example**.

```
int first_sec=0;
float second_sec=1.0;
struct {
       short x;
       double y;
       } example;
main()
{
        .
        .
        .
```

Note that **example** contains two variables: x and y.

If you compile your program with **/bin/cc**, you need to use a special **__attribute((__section))** syntax to create a named overlay section:

```
int first_sec __attribute(( __section(first_sec) )) = 0;
float second_sec __attribute(( __section(second_sec) )) = 1.0;
struct {
       short x;
       double y;
       } example  __attribute(( __section(example) ));
main()
{
        .
        .
        .
```

See the *Domain/C Language Reference* for details about this attribute.

# Chapter 8

# Input and Output

Domain Pascal supports the following three methods of performing I/O:

- Input/Output Stream (IOS) calls
- Variable format (VFMT) calls
- Predeclared Domain Pascal I/O procedures

In general, you can perform all your I/O with the predeclared Domain Pascal I/O procedures. However, the other two methods can be very useful in certain circumstances.

This chapter provides a brief overview of all three methods, along with some background information that may aid you in whatever method you choose.

## 8.1 Some Background on Domain I/O

This section describes some information that may be helpful in understanding how I/O works on the Domain system. It's only a brief sketch; the full details are published in *Programming with Domain/OS Calls*. Before we describe the mechanics of Domain I/O, we provide 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 Pascal 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, type UID, accessibility, etc.)
- Access magnetic tape files or serial lines

IOS calls are more primitive than the predeclared Domain Pascal I/O procedures. 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 predeclared Domain Pascal I/O procedures. If you want to do something out of the ordinary, then you will most likely need to use IOS calls.

See *Programming with Domain/OS Calls* for details about IOS.

### 8.1.2 VFMT Calls

VFMT (variable format) calls are special system calls that format input and output. Since the Pascal language does not support elaborate formatting features, you may find it useful to make a VFMT call in situations such as the following:

- A variable contains a hexadecimal value and you wish to prompt your user with its ASCII equivalent.
- You want to tabulate results in fixed columns using scientific notation.
- You need to parse an input line without worrying about whether the user separates the arguments with spaces or semicolons.

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 text strings 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 text (typically typed by the user), interpreting it in a way that you specify, and storing the apparent data values in program–defined variables.

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- and double-precision reals in floating-point and scientific notations

This includes the following Domain Pascal data types: **char**, **integer16**, **integer32**, **single**, and **double**.

See *Programming with Domain/OS Calls* for details about 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 predeclared Domain Pascal I/O procedures take a file variable as an argument. The file variable is a synonym for a temporary or permanent pathname to the file. If you are using IOS calls rather than the Domain Pascal I/O procedures, you refer to a pathname by its IOS ID rather than by its file variable. A stream ID is a number assigned by the operating system when you open a file or device. Since Domain Pascal I/O procedures in your source code ultimately translate to IOS calls at run time, a file variable in your source code becomes a 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 Pascal deals in file variables, not IOS IDs, so the table also shows the names of the file variables corresponding to these streams. You need not explicitly declare these; the system opens these streams automatically as described in the next subsection.

*Table 8-1. The Default Streams*

| **Stream Name** | **File Variable Name** | **Description** |
|---|---|---|
| **ios_$stdin** | **Input** | If you do not explicitly specify a file variable for a Domain Pascal input procedure, the compiler reads from input. By default, **input** is the process input pad, but you can redirect this stream with the < character.* |
| **ios_$stdout** | **Output** | If you do not explicitly specify a file variable for a Domain Pascal output procedure, the compiler assumes **output**. By default, the system associates this stream with the process transcript pad, but you can redirect this stream with the > character.* |
| **ios_$errout** | | Domain Pascal 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 Interactive I/O

Domain Pascal uses the following system for interactive processing of the standard input (**input**) and standard output (**output**) files. Domain Pascal does not actually open **input** and **output** until the program first refers to them. When Domain Pascal finds the first reference to **input** in your program, it calls **reset(input)**. **Reset** expects to fill the file buffer variable with the first **char** of a text file, which means that **reset(input)** can cause a request for input.

For example, consider the following program:

```
PROGRAM lazy;                {This program is WRONG!}
 VAR
    c : char;
 BEGIN
 while not eof(input) do
    begin
    write('enter a letter or an EOF --');
    readln(c);
    end;
 END.
```

If you run this program, you might expect results like the following:

```
enter a letter or an EOF -- A
enter a letter or an EOF -- Z
enter a letter of an EOF -- <EOF>
```

However, in reality, the program does not produce those expected results. That's because the first reference to **input** is in the **eof** function. This causes the system to perform a **reset(input)** prior to the test for **eof**. **Reset** expects to fill the file buffer, so the test for **eof** actually results in a request for input, for which, unfortunately, the user is not prompted. Therefore, running the program results in the following:

```
A           {here the user is required to enter input for which he/she is not prompted}
 enter a letter or an EOF -- Z
 enter a letter or an EOF -- <EOF>
```

To eliminate this problem, you must take advantage of a feature of **reset** known as **delayed access**. Delayed access means that data will not be supplied to fill the input buffer at the **reset**, but rather at the next reference to the file. Since **reset** initiates delayed access, and since **eof** and **eoln** cause the file buffer to be filled, you must place the first prompt for input *before* any tests for **eof** or **eoln**. The data you enter in response to the prompt is retained until you make another reference to the input file.

The following shows how to use delayed access to make the previous example work correctly:

```
PROGRAM interactive;
 VAR
     c : char;
 BEGIN
 write ('enter a letter or an EOF -- ');
 while not eof(input) do
     begin
     readln(c);
     write ('enter a letter or an EOF -- ');
     end;
 END.
```

## 8.1.6 Stream Markers

When you open a file, the operating system assigns a stream marker to the file. 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 (or record) that the system can next access. When you open the file with the Domain Pascal **open** procedure, 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 Pascal I/O procedures, you control the stream marker indirectly through the procedures you call.

### 8.1.7 File Organization

Although Domain/OS supports a variety of file types, the standard I/O library and UNIX functions allow you to access only ASCII files. These are files that consist of a string of ASCII characters. (The Domain Pascal ISO Latin-1 character set includes these characters.) You can create your own records within such a file by entering a delimiting character, but there is no predefined record structure. Also, you can read and write bytes in numeric rather than string formats, but it is your responsibility to keep track of how data is represented.

The default file type created by Domain/OS is **unstruct**. An **unstruct** file is an ordinary text file that is fully compatible with text files produced by programs compiled under a UNIX system. The system stores a text file consisting of ASCII characters. The operating system makes no attempt to organize or structure the data in a text file. That is, '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, the maximum length of each line of a text file that Domain Pascal opens is 256 characters. You can change this to a larger length by using the optional **buffer_size** parameter with the **open** statement. (See the listing for "Open" in Chapter 4 for details about the **buffer_size** parameter.) By "line", we mean all the characters between two end-of-line characters. No text file can have more than 32767 lines.

## 8.2 Predeclared Domain Pascal I/O Procedures

The encyclopedia of Domain Pascal code in Chapter 4 details the syntax of each of the predeclared Domain Pascal I/O procedures. This section provides a global view of these procedures.

### 8.2.1 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.

To create a permanent file, you call the **open** procedure and specify 'NEW' as the **file_history**. This not only creates the file, but opens it for future access as well.

To create a temporary file, you call the **rewrite** procedure.

Both **open** and **rewrite** take a **file** or **text** variable as an argument. In either case, Domain Pascal creates an **unstruct** file. If the file variable has the **file** data type, however, you can only read and write objects that have the file's base type. For text type files, you can access each byte one at a time.

> **NOTE:** Pre-SR10 versions of Domain Pascal created special record structured Domain files (called **rec** files) when you opened a file. For compatibility with older versions, you can use the current version of Domain Pascal to manipulate **rec** files, but you can no longer *create* **rec** files. When you open an existing file, Domain Pascal checks whether it is a **rec** or **unstruct** file and accesses it appropriately. Whenever you open a new file, however, Domain Pascal creates an **unstruct** file.

## 8.2.2 Opening an Existing File

To open an existing file for future access, you call the **open** procedure and specify either 'OLD' or 'UNKNOWN' as the **file_history**.

Note that you do not have to explicitly open the shell transcript pad. It is already open. (See the "Default Input/Output Streams" section earlier in this chapter for details.)

## 8.2.3 Reading from a File

In order to read from an open file, you must call the **reset** procedure. **Reset** tells the system to treat the open file as a read-only file. You can change the open file to a write-only file with the **rewrite** procedure.

After calling **reset**, you are free to call any or all of the three input procedures that Domain Pascal supports, namely, **read**, **readln**, and **get**. All three procedures read information from the specified file and assign it the specified variable(s).

The following list describes the differences among the three procedures:

- **Get** can access both **file** type files and **text** type files. Use it to assign the contents of the next record or character in a file to a file buffer variable.

- **Read** can access both **file** type files and **text** type files. Use it to read information from the specified file into the given variables. After reading a record (if a **file** type file) or a character (if a **text** type file), **read** positions the stream marker to point to the next record or character in the file.

- **Readln** can access **text** type files only. It is similar to **read** except that after reading the information, **readln** sets the stream marker to the character immediately after the next end-of-line character.

It is often useful to know when the stream marker has reached the end of the line or the end of the file. You can use **eoln** to test for the end of line, and **eof** to test for the end of file.

Domain Pascal supports the **find** procedure as an extension to standard Pascal. Use it to set the stream marker to point to a particular record in a **file** type file. This procedure permits you to skip randomly through a **file** type file, while the other read procedures imply a sequential path.

### 8.2.4 Writing to a File

In order to write to a file, you must call the **rewrite** procedure. (If you used **rewrite** to open a temporary file, then you don't have to call **rewrite** again.) The **rewrite** procedure tells the system to treat the open file as a write-only file. You can read from this file only if you call **reset.**

Once the file has been opened for writing, you can call any of these four standard Pascal output procedures: **write, writeln, put,** and **page. Write, writeln,** and **put** are the output mirrors to **read, readln,** and **get.** Using **write, writeln,** or **put** causes Domain Pascal to write the specified information from the specified variable(s) to the specified file.

Here are the differences among the four procedures:

- **Put** can access both **file** type files or **text** type files. Use it to assign the contents of the file buffer variable to the next file position, causing the contents to be written to the file.

- **Write** can access both **file** type files and **text** type files. Use it to write information from the specified variables into the specified file. After writing a record (if a **file** type file) or character (if a **text** type file), **write** positions the stream marker to point to the next record or character in the file.

- **Writeln** can access **text** type files only. It is similar to **write** except that after writing the information, **writeln** sets the stream marker to the character immediately after the next end-of-line character.

- **Page** can access **text** type files only. Use it to insert a formfeed (page advance) into the file.

In addition to the standard Pascal output procedures, Domain Pascal also supports the **replace** procedure. Use the **replace** procedure to substitute a new record for an existing record in a **file** type file. The **replace** procedure has the distinction of being an output procedure that you can call only when the file has been open for input. In other words, before you call **replace**, you must first call **open** and **reset** to open the file for reading. The **replace** procedure is usually used with **find**. Use **find** to skip through a **file** type file looking for a particular record, then use **replace** to modify the record in its place.

## 8.2.5 Closing a File

When a program terminates (naturally or as a result of a fatal error), the operating system automatically "closes" all open files. "Closing" means that the operating system unlocks the file. When the operating system closes a **file** type file, it automatically preserves any changes made to the file. However, when the operating system closes a **text** type file that was open for output, there is a possibility that some modifications won't be preserved. To ensure that all modifications are kept, make sure that the last output operation on the file is a **writeln**.

Domain Pascal supports a **close** procedure 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, warnings, and information messages. However, we start this chapter with a discussion of the errors reported by the predeclared procedures **open** and **find**, and we end with a discussion of run-time error messages.

## 9.1 Errors Reported by Open and Find

The **open** and **find** procedures are the only two predeclared Domain Pascal routines that return an **error status parameter**. This parameter tells you whether or not the call was successful. If the call was successful, the operating system returns a value of 0 in the error status parameter. If the call was not successful, the operating system returns a number denoting the error. Your program is responsible for handling the error. You may wish to print the error and terminate execution. Or, you may wish to code your program so that it can take appropriate action when it encounters an error.

This error status parameter is identical to the error status parameter returned by all system calls. This is more than coincidental since **open** and **find** are executed as stream calls at run time. In the next section, we summarize the error status parameter as it relates to the **open** and **find** procedures. For complete details on using the error status parameter, refer to *Programming with Domain/OS Calls*.

## 9.1.1 Printing Error Messages

To print an error message generated by an errant **open** or **find**, you must do the following:

- Put the following two include directives in your program just after the program heading:

  ```
  %INCLUDE '/sys/ins/base.ins.pas';
  %INCLUDE '/sys/ins/error.ins.pas';
  ```

  Make sure you put **base.ins.pas** before **error.ins.pas**.

- Declare the error status parameter with the **status_$t** data type; for example,

  ```
  VAR
      err_stat : status_$t;
            {status_$t is declared in base.ins.pas}
  ```

- Specify the error status parameter as the **all** field of the error status variable; for example,

  ```
  OPEN(f, pathname1, 'NEW', err_stat.all);
  ```

- Call the **error_$print** procedure with the error status parameter as its sole argument; for example,

  ```
  error_$print(err_stat);
   {The error_$print procedure is defined in error.ins.pas}
  ```

The following program puts all the steps together:

```
Program test;

 %INCLUDE '/SYS/INS/BASE.INS.PAS';
 %INCLUDE '/SYS/INS/ERROR.INS.PAS';

 VAR
           f : text;
    err_stat : status_$t;

 BEGIN
     OPEN(f, 'grok', 'OLD', err_stst.all);
     Error_$print(err_stat);
 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)
```

## 9.1.2 Testing for Specific Errors

The previous subsection introduced the **status_$t** data type and its **all** field. This section describes another field in **status_$t**—the **code** field. The **code** field of **status_$t** contains a number that corresponds to a particular error. To test for a specific error, compare this code field against expected errors. Table 9-1 lists the common error codes returned by **open** and Table 9-2 lists the common error codes returned by **find**. The symbolic names come from the **/sys/ins/ios.ins.pas** file. To use these symbolic names, all you have to do is list this file as an **include** file.

*Table 9-1. Common Error Codes Returned by* **open**

| Code | Symbolic Name | Cause of Error |
|---|---|---|
| 1 | **ios_$not_open** | You specified a file_history of 'NEW', but the pathname existed. Alternatively, the type UID of this existing file differed from the type UID of the file you were trying to open. |
| 14 | **ios_$already_exists** | You specified a file_history of 'NEW', but the pathname already exists. |
| 21 | **ios_$name_not_found** | You specified a file_history of 'OLD', but the pathname does not exist. |
| 28 | **ios_$object_not_found** | You specified a file_history of 'OLD', but the operating system cannot locate the disk containing the pathname (indicates network problems.) |
| 45 | **ios_$insufficient_rights** | The ACL of the pathname prohibits you from opening the file. |

*Table 9-2. Common Error Codes Returned by* **find**

| Code | Symbolic Name | Cause of Error |
|---|---|---|
| 1 | **ios_$not_open** | You called **find**, but without the file being open. |
| 9 | **ios_$end_of_file** | You specified a record number greater than the number of records in the file. |

For example, consider the following program fragment, which tries to first open pathname **my_book1**. If this pathname exists, the program then attempts to open pathname **my_book2**.

```
Program test;
 %INCLUDE '/sys/ins/base.ins.pas';
 %INCLUDE '/sys/ins/error.ins.pas';
 %INCLUDE '/sys/ins/ios.ins.pas';

 VAR
      f1, f2 : text;
      st     : status_$t;
 BEGIN
    open(f1, 'my_book1', 'NEW', st.all);
    if st.code = ios_$already_exists then
        open(f2, 'my_book2', 'NEW', st.all);
    . . .
```

## 9.2 Compiler Error, Warning, and Information Messages

When you compile a program, the compiler reports errors, fatal errors, warnings, and information.

A **fatal** error indicates a problem severe enough to stop compiler execution.

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.

**Information messages** do not prevent the compiler from creating an executable object file. The purpose of the information messages is to alert 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

See the "-info n and -ninfo: Information Messages" section of Chapter 6 for more details about information messages.

The following pages list the common Domain Pascal compiler error, fatal error, warning, and information messages, and suggest ways to handle them.

In addition, remember the cardinal rule of Pascal debugging: *look for missing semicolons.* For example, suppose the compiler reports an error at line 50, but line 50 appears to be a correct statement. It's quite likely that you forgot a semicolon at line 49.

### 9.2.1 Error, Fatal Error, Warning, and Information Message Conventions

The error, warning, and information messages listed in the rest of this chapter follow these conventions:

- Keywords in the message text are all capitals, 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;
  VAR
      num  :  integer17;
  .  .  .
  ```

  Because the fragment includes an undefined data type (**integer17**), it triggers Error 23, which reads:

  23 ERROR *Identifier* has not been declared in routine *name_of_routine*.

  When you compile, *identifier* and *name_of_routine* are filled in like this:

  ```
   (0003)     num: integer17;
  ******** Line 3: [Error 023]  INTEGER17 has not been declared in
           routine err_test.
  ```

### 9.2.2 Error, Warning, and Information Messages

Following are Domain Pascal's compiler error, warning, and information messages.

1 ERROR Unterminated comment.

You started a comment, but you did not close it, or you closed it with the wrong delimiter. Comment delimiters must match unless you compile with the **-iso** switch. If you don't compile with that switch and you start a comment with {, you must end it with }. Similarly, if you start a comment with (*, you must end it with *). If you start a comment with ", you must end it with ".

2 ERROR Improper numeric constant.

You specified a base that fell outside the legal range of 2 to 16. For example, you cannot specify a number in base 32. Perhaps you mistakenly specified the integer first and the base second. (See the "Integers" section in Chapter 2 for an explanation of base.)

3 ERROR Unterminated character string.

You started a string with an apostrophe ('), but you forgot to end it with an apostrophe. See Chapter 2 for a definition of string.

4 ERROR Bad syntax *(token)*.

The compiler encountered *token* when it was expecting to find something else.

6 ERROR Period expected at end of program *(symbol)*.

You must finish the program with an **end** statement followed by a period. The final character that the compiler found in your source code is *symbol*. Typically, *symbol* is /EOF/ (an end-of-file character) or a semicolon. The most frequent cause of this error is putting a semicolon rather than a period after the final **end**.

7 ERROR Text following end of program *(token)*.

You have put some text other than a comment after the end of the program. The phrase "END." marks the end of the program. You can put comments after the end of the program, but you cannot put anything else there.

8 ERROR PROGRAM or MODULE statement expected *(token)*.

The first noncomment in your source code must be either a **program** heading or a **module** heading. The compiler found *token* instead of a **program** or **module** heading. Domain Pascal also issues this error when there is some sort of mistake in your **program** or **module** heading. Chapter 2 describes the **program** heading. Chapter 7 describes the **module** heading.

10 ERROR Semicolon expected at end of **program/module** statement *(token)*.

You forgot to end your **program** or **module** heading with a semicolon. Domain Pascal encountered *token* when it was expecting a semicolon.

11 ERROR Improper declarations syntax *(token)*.

Domain Pascal found an unexpected *token* when processing a declaration part.

12 ERROR Improper CONST statement syntax *(token)*.

You made a mistake when declaring a constant. See Chapter 2 for the correct format. *Token* is the invalid token that Domain Pascal encountered. A possible trigger for this error is that you tried to declare two identifiers for the same constant value as in the following example:

```
CONST
    x,y = 5;
```

13 ERROR Improper TYPE statement syntax *(token)*.

You made a mistake in the **type** declaration part of your program. Domain Pascal encountered *token* when it was expecting something else. See Chapter 2 for the correct format of the **type** declaration part. A possible trigger for this error is that you used the wrong symbol to associate the identifier with its type as in the following example:

```
TYPE
    long := integer32;
```

14 WARNING *Datatype* cannot be PACKED.

Domain Pascal only recognizes the packed syntax for records, arrays, sets, and file types. *Datatype* is a data type that the compiler doesn't recognize when used with **packed**. Note that **packed** only economizes space when used before **array** and **record** declarations.

15 ERROR Improper type specification *(token)*.

You made some mistake when specifying a data type in the **var** declaration part. *Token* identifies the unexpected part of the **var** declaration part. For example, the following declaration triggers this error because **r** is a variable, not a data type:

```
VAR
     r : real;
     data : array[1.10]  of r;
```

See Chapter 2 for a complete description of the **var** declaration part.

16 ERROR Improper enumerated constant syntax *(message)*.

All constants in an enumerated type must be valid identifiers. See Chapter 2 for a definition of identifier. *Message* identifies the first character or token that did not conform to the rules for identifiers.

17 ERROR OF expected in SET specification *(token)*.

When you declared a set type or set variable, you forgot to specify the keyword **of** in between the word **set** and the base type. Refer to Chapter 3 for information on declaring set types. Domain Pascal encountered *token* rather than the keyword **of**.

18 ERROR Improper ARRAY specification syntax *(token)*.

You declared an array incorrectly. See Chapter 3 for details on declaring arrays. *Token* is the token that Domain Pascal encountered when it expected to find something else.

19 ERROR Improper RECORD specification syntax *(token)*.

You declared a record incorrectly. See Chapter 3 for details on declaring records. Domain Pascal encountered *token* when it expected to find something else. A common trigger of this error is a type declaration such as the following:

```
TYPE
     student = record
       a : integer32;
       b = boolean;{cause of error. '=' should be ':'}
    end;
```

20 ERROR Improper pointer specification *(token)*.

You declared a pointer incorrectly. See Chapter 3 for details on declaring pointers. Domain Pascal encountered *token* when it expected to find something else. The following type declaration triggers this error, because the up-arrow (^) can only appear before a type name, not a type specification:

```
TYPE
     ptr_to_ptr_to_long = ^^integer32;
```

21 ERROR Improper VAR statement syntax *(token)*.

In a variable declaration, you probably forgot to specify a semicolon after the data type. Perhaps you specified a comma instead of a semicolon, or perhaps you did not specify any punctuation mark at all. This error also occurs if you begin an identifier with a digit (0–9) or dollar sign ($).

22 ERROR Parameter list must only be specified when the procedure or function is declared as FORWARD.

You specified the parameter list for this routine in two places: first when you specified **forward** or **extern** and second in the routine heading itself. To correct this error, eliminate the second parameter list.

23 ERROR *Identifier* has not been declared in routine *name_of_routine*.

Several conditions can trigger this error. You might have used *identifier* in the code portion of *name_of_routine* without *identifier* being accessible to this routine. Study the "Global and Local Variables" and "Nested Routines" sections in Chapter 2 to learn about the scope of declared identifiers.

Another possible trigger for this error is that you tried to make a forward call to a procedure without declaring the procedure with the **forward** attribute. (See the "Routine Attribute List" section in Chapter 5 for details on the **forward** attribute.)

This error also can occur if you specify a data type that either is invalid or that you've forgotten to define in the **type** section of your program.

24 ERROR Multiple declaration of *identifier*, previous declaration was on line *line_number*.

You declared *identifier* (which could be a data type, variable, constant, label, or routine) more than once.

25 ERROR Improper MODULE structure *(token)*.

The compiler was expecting to encounter a procedure or function declaration, but found *token* instead. Remember, that unlike a program, the action part of a module must always be contained inside a named routine.

26 ERROR Number of array subscripts exceeds limit of eight.

Domain Pascal supports arrays of up to eight dimensions.

29 ERROR Subrange bound *(token)* is not scalar.

You were trying to declare an array or subrange, but one of the bounds of the subrange was not a scalar. The scalar types are integer, Boolean, char, enumerated, and subrange. The *token* is the token that Domain Pascal encountered when it was searching for a scalar expression.

30 ERROR Lower bound of subrange *(lower_bound* ) is not of the same type as the upper bound *(upper_bound)*.

You were trying to declare an array or subrange, but the data type of *lower_bound* is not the same data type as that of *upper_bound*. The types must match.

31 ERROR Lower bound of subrange *(left_scalar* ) is greater than the upper bound *(right_scalar)*.

You were trying to declare an array or subrange, but you set the value of the *left_scalar* to a higher value than the value of the *right_scalar*. The *left_scalar* must be lower than the *right_scalar*.

32 ERROR Base type *(type)* of SET is not scalar.

You tried to declare a nonscalar type as the base type of a set variable or type. The scalar types are integer, char, Boolean, enumerated, and subrange. *Type* is the token that Domain Pascal encountered instead of a scalar type.

33 ERROR SET elements must be positive (-).

You tried to declare a set with a base type of integer or subrange, but Domain Pascal discovered a negative number in the base type.

34 ERROR SET exceeds limit of 1024 elements (*(token)*).

You cannot declare a set that exceeds 1024 elements. See the "Internal Representation of Sets" section in Chapter 3 for details.

35 ERROR Improper use of *(identifier)*, only a TYPE defined name is valid here.

The compiler was expecting a data type, but you specified an *identifier* instead. Possibly, you tried to create a pointer variable with improper declarations like the following:

```
VAR
    x : integer;
    y : ^x;
```

See the "Standard Pointer Type" section in Chapter 3 for details on setting up pointer types.

36 ERROR Multiple declaration of *variable* in parameter list.

You declared *variable* more than once in the parameter list of a procedure or function.

37 ERROR PROCEDURE/FUNCTION name required *(token)*.

You used the keyword **procedure** or **function** without specifying a valid identifier immediately after it. See the "Identifiers" section in Chapter 2 for a definition of a valid identifier. *Token* is either the name of the invalid identifier or the null set (if you did not supply any name at all).

38 ERROR Improper PROCEDURE/FUNCTION declaration *(token)*.

You were probably confusing procedure with function. A function has a data type, and a procedure does not; therefore, the following declaration triggers this error:

```
procedure one (r2 : single) : real;
```

39 ERROR Improper parameter declaration ( *token*).

You did not specify the parameter list of your procedure or function in the correct manner. A common cause of this error is using semicolons incorrectly in the parameter list. (See Chapter 5 for details on parameter lists.) Domain Pascal encountered *token* when it was expecting something else (probably a semicolon).

40 ERROR Colon expected in FUNCTION declaration *(token)*.

You forgot to put a colon before the type specification of the function; for example, compare the right and wrong ways to declare a function:

```
FUNCTION pyth_theorem(a : integer16)   real;  {Wrong!}
 FUNCTION pyth_theorem(a : integer16) : real;  {Right }
```

Domain Pascal found *token* instead of a colon.

42 ERROR FUNCTION type specification required.

You forgot to specify a data type for the function itself; for example, compare the right and wrong ways to declare a function:

```
{wrong} FUNCTION pyth_theorem(a : integer16);
 {right} FUNCTION pyth_theorem(a : integer16) : real;
```

43 ERROR CASE type is not scalar.

You specified an expression in a **case** statement that did not have a scalar data type. The scalar data types are integer, Boolean, char, enumerated, and subrange. For example, the following **case** statement triggers this error:

```
VAR
     r : real;

          . . .
 BEGIN
    CASE r OF {error: r is real, but should have been
                      integer or integer subrange.   }
      1 : writeln('One');
      2 : writeln('Two');
    end;
```

44 ERROR *Constant* is not of the correct type for the CASE on line *number*.

You specified a *constant* in a **case** statement that was not the same data type as the expression of the **case** statement. For example, the following case statement triggers this error:

```
VAR
    r : integer16;

        . . .

 CASE r OF
   1.5 : writeln('One');      {error: 1.5 is a real;
                               it should be integer.}
   2   : writeln('Two');
 end;
```

45 ERROR *(Constant)* is outside the subrange of the CASE on line *number*.

In a CASE statement, you specified a *constant* that was not within the declared range of the **case**.

For example, the following **case** statement triggers this error because the constant 5 is outside the declared subrange 0 to 3.

```
VAR
    x : 0.3;
 BEGIN
    CASE x OF
       5 : . . . {error}
```

46 ERROR *Constant* has already occurred as a CASE constant on line *number*.

You specified the same *constant* more than once in the same **case** statement. For example, the following **case** statement triggers this error because constant 6 appears twice:

```
CASE r OF
   4     : writeln('square root is rational');
   5,6,7 : writeln('square root is irrational');
   6     : writeln('even');  {error}
 end;
```

47 ERROR *Token* is not a valid option specifier.

You specified *token* in an OPTIONS clause, but *token* is not a valid option. See the "Routine Options" section in Chapter 5 for details.

48 ERROR Include file name must be quoted *(token)*.

You used the **%include** directive but forgot to put the name of the include file in apostrophes. *Token* is the token that Domain Pascal found when it was expecting to find an apostrophe.

49 ERROR Too many include files.

Note that this error can be triggered by include files nested within include files. To correct this error, you can break the program into separately compiled modules.

50 ERROR *Token* is not a recognized option.

You specified *token* as a compiler directive, but *token* is not a valid compiler directive. Refer to the "Compiler Directives" listing in Chapter 4 for a description. Possibly, you put an extra percent sign (%) in your program. Another possibility is that you specified *token* on your compile command line as a compiler option. If you do this, the operating system returns this error message. See Chapter 6 for a complete list of compiler options. Possibly, you caused this error by trying to compile two files at once, and the compiler interpreted the second file as an (invalid) option.

51 ERROR Include file *pathname* is not available.

You specified a *pathname* for an include file, but either it does not exist or network problems prevent the compiler from accessing it.

52 ERROR Semicolon expected following option specifier *(token)*.

You specified compiler directive **%debug** or **%eject** but forgot to specify a semicolon immediately after the directive. See the "Compiler Directives" listing in Chapter 4.

53 ERROR Multiple declaration of *identifier* in RECORD field list.

You specified the same field twice in a record declaration. For example, the following record declaration triggers this error because x is declared twice:

```
r = record
    x : Boolean;
    x : integer16;  {error}
 end;
```

54 ERROR Array bound type is not scalar *(datatype)*.

You specified *datatype* as the index type of an array. However, *datatype* must be a scalar data type. The scalar data types are integer, char, enumerated, subrange, and Boolean.

55 ERROR Improper LABEL statement syntax *(token)*.

Labels must be unsigned integers or identifiers, but Domain Pascal found *token* instead.

56 ERROR Multiple definition of *element*, previous definition was on line *number*.

You declared the same *element* (variable, data type, constant, label, procedure, or function) twice.

57 ERROR Improper usage of *identifier*, only a LABEL name is valid here.

You used *identifier* as a label, but you had already declared it as a variable, type, or constant. Possibly, you accidentally put a colon (:) immediately following the *identifier* in the action part of your program. The colon could cause the compiler to interpret the *identifier* as an illegal label. Perhaps you meant to specify a statement like the following:

```
x := 8;
```

but you forgot the equal sign and ended up specifying the following instead:

```
x : 8;
```

58 ERROR *Constant* is declared as a CONST name, and cannot be assigned a value.

You mistakenly tried to assign a value to a *constant*. Perhaps you should have declared *constant* as a variable rather than as a constant.

59 ERROR Improper use of *identifier*, only a VAR name is valid here.

You tried to assign a value to an *identifier* that is not a valid variable. Possibly, you tried to assign a value to a type rather than a variable. For example, code like the following causes this error:

```
TYPE
     int = integer32;
 VAR
       q : int;
 BEGIN
     int := 8;  {Wrong!}
       q := 8;  {Right!}
```

60 ERROR Improper use of *identifier*, only a VAR or CONST name is valid here.

You tried to assign the value of a data type or label to a variable. *Identifier* must be a variable or a constant.

61 ERROR *Token* is not an ARRAY.

You specified an expression of the format

```
TOKEN[. . .
```

This format is reserved for specifying a particular element of an array; however, *token* is not an array variable. Possibly, you were trying to call a procedure or function and used brackets rather than parentheses.

62 ERROR *Variable* is not a pointer variable.

You tried to dereference a *variable* that was not declared as a pointer variable. (See the "Pointer Operations" listing of Chapter 4.)

63 ERROR *Token* is not a RECORD.

Domain Pascal was expecting a **record** variable, but it found *token* instead. (See the "Record Operations" listing of Chapter 4.)

64 ERROR *Token* is not a field of *record*.

Domain Pascal was expecting a field of the *record* variable, but it found *token* instead.

65 ERROR Too many subscripts to *array_variable*.

You declared *array_variable* as an *n*–dimensional array, but you have specified more than *n* subscripts for the array at this line.

66 ERROR *Kind_of_declaration* declaration must precede internal PROCEDURE and FUNCTIONS declarations.

You put a routine in the middle of a declaration part. A nested routine must come at the end of a declaration part (not in the beginning or the middle).

67 ERROR Improper use of *identifier*, only a FUNCTION name is valid here.

You probably had no intention of calling a function and are puzzled as to why you got this message. If you used a statement of the form

```
IDENTIFIER(TOKEN)
```

then Domain Pascal assumed that you were trying to call a function. Possibly, you were trying to access an array, but you used parentheses instead of brackets.

68 ERROR The types of *operand1* and *op erand2* are not compatible with the *operator* operator.

You made a mistake such as specifying (21.0 DIV 3.0). (It's a mistake because **div** only accepts integer operands.) See Chapter 4 for a complete list of operators and their valid operands.

69 ERROR The type of *operand* is not compatible with the *operator* operator.

You made a mistake such as specifying an expression like:

```
(NOT 3.0)
```

It's a mistake because **not** only accepts Boolean operands. See the beginning of Chapter 4 for a summary of operators.

70 ERROR Incompatible operands [*operand1*, *operand2*] to the *operator* operator.

See Table 4–1 for a summary of operators.

71 ERROR Subscript *expr* to array *name_of_array* is not of the correct type.

See the "Array Operations" listing in Chapter 4.

73 ERROR *Statement* expression is not Boolean.

You were using a non–Boolean expression in a manner reserved for Boolean expressions. For example, the following program fragment triggers this error because variable **int** is an integer, not a Boolean:

```
VAR
     int : integer;
     b : Boolean;

          . . .

 if int   then ..{error, int is not a Boolean expr. }
 if int=9 then ..{no error, int=9 is a Boolean expr.}
 if b     then ..{no error, b is a Boolean expr.    }
```

74 ERROR FOR statement index variable is not scalar.

The scalar data types are integer, Boolean, char, enumerated, and subrange. If you specify an index variable with a data type other than one of these five types, Domain Pascal issues this error. See the **for** listing in Chapter 4.

75 ERROR FOR statement initial value expression is not compatible with the index variable.

You specified a start_expression of a different data type than the index variable. See the **for** listing in Chapter 4.

77 ERROR FOR statement limit value expression is not compatible with the index variable.

You specified a stop expression of a different data type than the index variable. See the **for** listing in Chapter 4.

79 ERROR Assignment statement expression is not compatible with the assignment variable.

You tried to assign the value of an expression to a variable, but the data type of the value and the variable were not compatible. In general, the data type of the expression must match the data type of the variable; however, there are a few exceptions. For example, you can assign an integer expression to a real variable (though you cannot do the reverse). In most cases, this error is just a simple programming mistake, but if you do intend to assign a value to a variable of a different data type, refer to the "Type Transfer Functions" listing of Chapter 4.

81 ERROR Too many arguments to *routine*.

You attempted to call *routine*, but you tried to pass more arguments to *routine* than it was expecting. The number of arguments cannot exceed the number of parameters declared in the parameter list of the *routine*. See Chapter 5 for details on parameter passing.

82 ERROR Too few arguments to *routine*.

You attempted to call *routine*, but you tried to pass fewer arguments to *routine* than it was expecting. If you want to pass *n* arguments to a routine declaring more than *n* parameters, you must use the **variable** routine attribute (which is described in the "Variable" section in Chapter 5).

83 ERROR Argument *n* to *routine* is not compatible with the declared argument type.

You tried to call *routine*, but the *n*th argument in the call does not have the same data type as the *n*th parameter. You can suppress this error by using the **univ** routine attribute. See Chapter 5 for details on parameter passing.

84 ERROR Argument *n* to *routine* is not within the declared argument subrange.

You tried to pass a subrange expression as the *n*th argument to call *routine*, but the value of the expression was not within the declared range of the subrange.

85 ERROR Improper use of *element*, only a PROCEDURE name is valid here.

Domain Pascal assumed you were trying to call a procedure, but *element* is not a procedure. Any statement having the following format is assumed to be a procedure call:

```
IDENTIFIER(anything);
```

86 ERROR Unrecognized statement *(token)*.

The compiler could not classify a statement into one of the basic categories of Domain Pascal statements (such as assignment, procedure call, function call, **goto**, **repeat**). Possibly, you misspelled a keyword, or perhaps you forgot to close the previous statement with a semicolon.

87 ERROR GOTO label expected *(token)*.

You forgot to specify a label immediately after the keyword **goto**. Domain Pascal expected a declared variable, but found *token* instead. See the **goto** listing of Chapter 4.

89 ERROR The value of *number* is outside the range of valid set elements.

You tried to assign a *number* greater than 1023 to a set. See the "Set Operations" listing in Chapter 4 for details on assigning values to sets, and see the "Sets" section in Chapter 3 for information on declaring set types and variables.

91 ERROR Function type must only be specified when the function is declared FORWARD.

You specified a function as **forward**, but you mistakenly specified the data type of the function twice. You must only specify the data type of the function once. Specify the data type when you specify **forward**. See the "Forward" section in Chapter 5 for an explanation of **forward**.

92 ERROR *(Option)* specifier is not valid when defining a procedure/function previously declared to be FORWARD.

If *option* is **forward**, then you probably declared **forward** twice for the same routine. Possibly, you declared a routine as **forward**, but you also used the routine with both **define** and **extern**. If *option* is **extern**, then you probably declared **extern** twice for the same routine.

93 ERROR Improper use of the DEFINE statement.

See Chapter 7 for a complete description of the **define** statement.

94 ERROR Improper DEFINE statement structure.

See Chapter 7 for a complete description of the **define** statement.

95 ERROR Multiple declaration of *element* i n DEFINE statement.

You used **define** to define the same *element* twice. See Chapter 7 for a complete description of the **define** statement.

96 ERROR Constant value cannot be evaluated at compile time.

You specified an expression in a **const** declaration that the compiler could not reduce to a constant. For example, the following declarations trigger this error because **x** is not a constant:

```
VAR
      x : integer;

 CONST
     ax : addr(x);
```

97 ERROR Label *label* is never defined.

You declared a *label* in the **label** declaration part of a routine, but you never specified this label inside the code portion of the routine. Possibly, you declared the label in the **label** declaration part of a routine, but specified this label inside the code portion of another routine. See Chapter 2 for a description of labels, and see the **goto** listing in Chapter 4 for a description of the **goto** statement.

98 ERROR Improper PROCEDURE/FUNCTION structure *(token)*.

Refer to Chapter 2 for the rules on routine structure. Possibly, you put a period instead of a semicolon at the end of a routine.

99 ERROR BEGIN expected in routine *name_of_routine*; found "*token*".

The code portion of a routine must start with the keyword **begin**. Domain Pascal discovered *token* instead of **begin**. Refer to Chapter 2 for the rules on routine structure. Note that every routine (including the main program) must at least include the keywords **begin** and **end**.

100 ERROR END expected; found "*token*".

You forgot to mark the finish of a routine with an **end** statement. Ignore the line number the error is reported at; the compiler usually does not discover this error until the end of the program. See Chapter 2 for the rules on program structure.

101 ERROR Statement separator expected *(token)* .

Domain Pascal discovered two statements with nothing to separate them. You probably made one of the following three mistakes: you forgot a semicolon; or, you forgot a closing **end** in a compound statement; or, you forgot an **else** in an **if/then/else** statement.

102 ERROR Improper argument list *(token)*.

You forgot to specify a ")" to terminate a type transfer function. See the "Type Transfer Functions" listing in Chapter 4.

103 ERROR THEN expected in IF statement *(token)*.

You forgot the **then** part of an **if/then/else** statement. For details on **then**, see the **if** listing in Chapter 4.

105 ERROR OF expected in CASE statement ((*token*)).

A **case** statement must begin with the format

```
CASE expr OF
```

but you forgot the keyword **of**. See the **case** listing in Chapter 4.

106 ERROR CASE label expected *(token)*.

In a **case** statement, you specified a statement without specifying a constant. Possibly, you forgot to conclude the **case** statement with **end**. See the **case** listing in Chapter 4.

107 ERROR END/OTHERWISE expected in CASE statement *(token)*.

You probably forgot to conclude a simple statement with a semicolon or a compound statement with an **end**.

109 ERROR DO expected in WHILE statement *(token)*.

You forgot to specify the keyword **do** following the condition in a **while** statement.

110 ERROR UNTIL expected in REPEAT statement *(token)*.

Domain Pascal found *token* instead of **until** in a **repeat** statement. Refer to the **repeat** listing in Chapter 4.

111 ERROR := expected in FOR statement *(token)* .

Domain Pascal found *token* instead of := in a **for** statement. Refer to the **for** listing in Chapter 4.

112 ERROR TO or DOWNTO expected in FOR statement *(token)*.

Domain Pascal found *token* instead of **to** or **downto**. Refer to the **for** listing in Chapter 4.

113 ERROR DO expected in FOR statement *(token)* .

Domain Pascal found *token* instead of **do**. Refer to the **for** listing in Chapter 4.

114 ERROR Improper WITH statement *(token)*.

Refer to the **with** listing in Chapter 4.

115 ERROR DO expected in WITH statement *(token)*.

Refer to the **with** listing in Chapter 4.

116 ERROR Improper expression *(expression)*.

A variety of situations could have caused this error. Probably Domain Pascal was expecting a keyword, and you either did not enter a keyword, or you did not enter a keyword that was appropriate to the situation. The inappropriate expression is *expression*. For example, you can trigger this error by using the keyword **if** without using the keyword **then**. Another possibility is that you forgot a semicolon on the line preceding the line that the compiler reported the error. Another possibility is that you began an identifier with a digit or dollar sign ($) rather than a character.

117 ERROR Identifier expected *(token)*.

Domain Pascal was expecting an identifier and found *token* instead. Chapter 2 defines identifiers.

120 ERROR OF expected in FILE declaration *(token)*.

You used the keyword **file** without following it with the keyword OF. See Chapter 3 for details on declaring file types.

121 ERROR Expression/constant cannot be passed as argument *n* to *routine*.

You specified an expression or constant as the *n*th argument to *routine*. However, the *n*th parameter of *routine* is declared as **var** or **in out**, and you can only pass variables as arguments to such a parameter.

122 ERROR Improper use of *identifier*.

You probably tried to call a predeclared procedure as a function or a predeclared function as a procedure. See Chapter 5 for a description of the difference between calling procedures and calling functions.

123 ERROR Attempted assignment to *(variable)*, a FOR-index variable, or formal parameter marked as IN.

You either tried to assign a value to *variable* inside a routine that declared it as **in**, or you tried to modify a FOR loop's index variable inside the loop. If you did the former, you can correct this error by changing the **in** parameter to **in out** or **var**. If your error was attempting to modify a **for** loop's index variable, you can eliminate the code inside the loop that modifies the variable.

124 ERROR *Routine* requires TEXT file parameter.

You specified a **file of** variable as an argument to *routine*, but *routine* requires a **text** variable instead.

125 ERROR *Procedure* requires FILE parameter.

You specified an illegal file parameter for **open** or **close**. Only identifiers are legal file parameters. FORTRAN programmers might have triggered this error by using an integer as a file parameter. Possibly, you forgot to specify any file parameter at all.

126 ERROR *Procedure* cannot be performed on INPUT file.

The standard input file (**input**) cannot be an argument to **rewrite**, **put**, or **page**. See the "Default Input/Output Streams" section in Chapter 8 for details on **input**.

127 ERROR *Procedure* cannot be performed on OUTPUT file.

The standard output file (**output**) cannot be an argument to **reset**, **get**, **eof**, or **eoln**. See the "Default Input/Output Streams" section in Chapter 8 for details on **output**.

128 ERROR Argument to *identifier* is not a pointer reference.

A predeclared procedure (typically **new** or **dispose**) requires an argument of a pointer type.

129 ERROR Operand *operand1* is not compatible with *(routine)*.

You tried to read a value into an expression; for example, consider the following statements:

```
READ(x + 1);                  {wrong}
 READLN(x + 1);               {wrong}
 READLN(x); x := x + 1;       {right}
```

130 ERROR Fraction width specified for operand *(element)* that is not of a REAL type.

In a **write** or **writeln** statement, you specified a two-part field width for a nonreal expression. If the expression is real, you can specify an optional one- or two-part field width, but if the expression is not real, then you can only specify an optional one-part field width. See the **write** listing in Chapter 4 for a complete description of field widths.

131 ERROR Field width specifier is not permitted.

You can only specify a field width for a **write** or **writeln** statement. If you specify a field width for any other statement, Domain Pascal issues this error. When you call a procedure or function, Domain Pascal interprets any colon (:) inside the call as a field width.

132 ERROR Field width specifier *(token)* is not INTEGER.

Domain Pascal was expecting to find an integer field width, but found *token* instead. Remember, you format real numbers with a two-part field (not a decimal). Note that when you call a procedure or function, Domain Pascal interprets any colon (:) inside the call as a field width. So, possibly you triggered this error with an inadvertent colon.

133 ERROR Type of operand *(identifier)* is not compatible with the *routine* operation.

You tried to read or write an aggregate variable (such as an array or record) to or from a text (unstruct) file. You can correct this mistake by specifying a rec file instead of an unstruct file. If you must use a text file, you can correct the error by specifying a field (if a record) or an element (if an array) rather than the full aggregate.

134 ERROR Improper file name in OPEN.

The file name is the second argument to the predeclared **open** procedure. The name must be a string constant or string variable. See the **open** listing in Chapter 4 for details on filenames that you can specify.

135 ERROR Improper file mode in OPEN.

The file mode is the third argument to the predeclared **open** procedure. The file mode must be a character string, a string constant, or a variable whose data type is an array of char. See the **open** listing in Chapter 4 for details on file modes.

136 ERROR Improper status argument in *procedure*.

You specified a status argument to *procedure* that had a data type other than **integer32**. A common mistake is to misuse a **status_$t** variable. For example, compare the right and wrong ways to use such a variable in an **open** procedure:

```
%INCLUDE '/sys/ins/base.ins.pas';
    VAR
        st     : status_$t;
        f1, f2 : text;
    . . .
    OPEN(f1, 'anger1', 'NEW', st);       {wrong}
    OPEN(f2, 'anger2', 'NEW', st.all);   {right}
```

Refer to the **open** and **find** listings in Chapter 4 for details on syntax.

137 ERROR *Procedure* cannot be used on a text file.

The first argument to **find** or **replace** must be a variable of type **file**. You have mistakenly specified a variable of type **text**. Refer to the **find** or **replace** listings in Chapter 4 for details.

138 ERROR Record number is not integer in FIND.

The second argument to the **find** procedure is the record number, and this argument must be an integer expression. Refer to the **find** listing in Chapter 4 for details on syntax.

139 ERROR Improper parameter list in PROGRAM statement ^1.

You made a mistake in your program heading. If you specified a file list in the program heading, then make sure that you specified the files as identifiers (and not as strings). For example, compare the following two file lists:

```
PROGRAM test(input, output);      {right}
PROGRAM test('input', 'output'); {wrong}
```

140 ERROR Compiler failure, unknown tree node.

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

141 ERROR Compiler failure, unknown top node.

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

142 ERROR Compiler failure, no temp space.

Your program requires too much space on the stack. There are several ways you might be able to eliminate this error.

Try changing *dynamic* variables that require a lot of space, such as large records and arrays, to *static* variables. Static variables do not require stack memory. Use the **static** attribute to allocate a variable in a static data area and keep its name local to the module or program in which it is declared.

Another way to reduce your program's stack requirements is to reduce the number of temporary variables that it uses. For example, you could

- Break the subroutines into smaller subroutines that can be separately compiled. Smaller compilation units require fewer temporary variables and will therefore require less space on the stack during compilation.
- Try using the same temporary variables in all your program's loops, so that the compiler does not create additional temporary variables for each loop.
- For functions returning large aggregates, use **var** or **in out** parameters in a **procedure** instead of **function** return values. Thus, the compiler will not have to create temporary variables to hold the return values.

143 ERROR Compiler failure, lost value of node.

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

144 ERROR Compiler failure, registers locked.

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

145 ERROR Compiler failure, no emit inst.

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

146 ERROR Compiler failure, procedure too large.

The routine you are trying to compile may have exceeded compiler implementation limits. Try to break the routine into multiple routines and modules and then recompile. If the problem still persists, please contact either your HP Response Center or your local HP representative.

147 ERROR Compiler failure, inst disp too large.

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

148 ERROR Compiler failure, obj module too large.

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

149 ERROR Compiler failure, no free space.

The compiler ran out of dynamic memory while compiling your program. Try to break the program into multiple routines and modules and then recompile. Also, check the amount of free disk space on your system (for example, with the **/com/lvolfs** command). If you seem to have enough disk space. contact either your HP Response Center or your local HP representative.

150 ERROR Compiler failure, short branch optimization.

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

151 ERROR *Routine* was declared FORWARD on line *number* and never defined.

You specified *routine* as **forward**, but you did not define it in your program. See the "Forward" section in Chapter 5 for details on **forward**.

152 ERROR Section name *(identifier)* conflicts with procedure or data section name.

You specified *identifier* as a section name for a **var** declaration part; however, *identifier* is a reserved section name. Please pick another name instead, or remove the section name completely. See the "Putting Variables Into Sections" section in Chapter 3 for details on naming sections.

153 ERROR Improper section name specification *token*.

You were declaring a section name for a group of variables, but you specified *token* rather than an identifier as the name of the section. See the "Putting Variables into Sections" section in Chapter 3 for details on naming sections.

154 ERROR Conflicting storage allocation specifications.

You declared a variable as **static** and as belonging to a nondefault section name. It cannot be declared as both at the same time.

155 WARNING Constant subscript *(value_of_constant)* to array *name_of_array* is out of range.

The *value_of_constant* was not within the declared range of *name_of_array*. You must either use a different constant or expand the declared range of the array. See the "Arrays" section in Chapter 3 for a description of array declaration.

157 ERROR *Identifier* was declared in a DEFINE statement but never defined.

You used *identifier* in a **define** statement, but forgot to use it as the name of a procedure, function, or variable. See Chapter 7 for an explanation of the **define** statement.

158 ERROR Improper OPTIONS specification *(token)*.

You made a mistake while declaring OPTIONS for a routine. Probably, you specified *token* instead of a valid routine attribute. Possibly, you forgot to mark the end of an OPTIONS clause with a semicolon. The valid routine attributes are listed in Chapter 5.

159 ERROR Duplicate OPTIONS specification *(routine_ attribute)*.

You specified the same routine attribute twice in an OPTIONS clause. You can only specify it once. The "Routine Options" section in Chapter 5 describes the OPTIONS clause.

160 ERROR Conflicting OPTIONS specification *(routine_attribute)*.

You specified several routine attributes in an OPTIONS clause; however, *routine_attribute* cannot appear in the same OPTIONS clause as one of the previous routine attributes. For example, you cannot specify both **forward** and **extern** in the same OPTIONS clause. You can also trigger this error by mistakenly using the same routine attribute twice. The "Routine Options" section in Chapter 5 describes the OPTIONS clause.

161 ERROR Unrecognized OPTIONS specification *(token).*

You specified *token* inside an OPTIONS clause, but *token* is not a valid routine attribute (described in Chapter 5).

162 WARNING Conditional compilation user warning.

You triggered a warning-level problem through misuse of the conditional compiler directives. More specific messages will follow this one. See the "Compiler Directives" listing of Chapter 4 for details on the conditional compilation directives.

163 ERROR Conditional compilation user error.

You triggered an error-level problem through misuse of the conditional compiler directives. More specific messages will follow this one. See the "Compiler Directives" listing of Chapter 4 for details on the conditional compilation directives.

164 ERROR Conditional compilation syntax error; look at prior "(PreProc)" message.

"(PreProc)" is an abbreviation for the Domain Pascal preprocessor. This error is telling you that the preprocessor found an error and passed it along to the compiler. The preprocessor found an error in a conditional compilation directive. (The conditional compilation directives are **%var, %if, %then, %else, %config, %elseif, %elseifdef, %enable, %endif,** and **%ifdef.**) Possibly, you used a conditional compilation variable without having first declared it (with a **%var** directive). Another possibility is that you used an operator other than **and, or,** and **not** in a predicate. See the "Compiler Directives" listing in Chapter 4 for details.

165 ERROR Conditional compilation not balanced.

Probably, you forgot to end an **%if** directive with the **%end** directive. (Making this mistake may trigger several other errors including Error 6: "Period expected at end of program.") See the "Compiler Directives" listing in Chapter 4 for details.

166 ERROR Compiler failure, data frame overflow.

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

168 ERROR Compiler failure, register consistency.

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

169 ERROR Compiler failure, no temp created.

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

170 ERROR Compiler failure, improper forward label at *token*.

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

171 ERROR Compiler failure, pseudo pc consistency.

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

173 ERROR Cannot take the address of internal routine *identifier*.

You cannot pass *identifier* as an argument to the **addr** function, because it is an internal routine. An internal routine is any routine declared in the main program, any routine nested inside another routine, or any routine specified with the routine option **internal**.

174 ERROR Ptr is *keyword1* type but operand is a *keyword2*.

You tried to assign a value that has a data type other than what the pointer is expecting. Remember that in Domain Pascal, unless you use **univ_ptr**, a pointer can only point to a value of the specified data type. See Chapter 3 for a discussion of pointer types.

175 ERROR Incompatible function return types.

A pointer to a function is expecting a value of a particular data type to be returned to it, but you mistakenly tried to return a value of a different data type to it. See Chapter 3 for a discussion of pointer types.

176 ERROR Incompatible VARIABLE arguments options.

Possible, there may be mismatch between a pointer to a procedure or function, and the pointer value you are actually trying to assign to it.

177 ERROR Incompatible number of parameters.

Possibly, there is a mismatch between a pointer to a procedure or function, and the pointer value you are actually trying to assign to it.

178 ERROR Incompatible ECB options.

Possibly, there is a mismatch between a pointer to a procedure or function, and the pointer value you are actually trying to assign to it.

179 ERROR Incompatible parameter passing conventions for parameter *identifier*.

Possibly, there is a mismatch between the parameter–passing conventions declared for a pointer to a procedure or function, and the pointer value you are actually trying to assign to it.

180 ERROR Incompatible types specified for parameter *identifier*.

Possibly, there is a mismatch between the parameter-passing conventions declared for a pointer to a procedure or function, and the pointer value you are actually trying to assign to it.

181 ERROR INTERNAL option is illegal for PROCEDURE^ or FUNCTION^ types.

You mistakenly tried to use the routine option **internal** in a procedure or function pointer.

182 ERROR Incompatible VAL_PARAM options.

You called a routine that used the **val_param** routine option inconsistently. For example, suppose you declare a pointer to a function as follows:

```
TYPE
   func_ptr = ^FUNCTION (x:integer): real;
```

You want **func_ptr** to correspond to an external function that you declare as:

```
FUNCTION bad_call (x:integer) : real; EXTERN; VAL_PARAM;
```

However, calling **bad_call** by passing its address as a **func_ptr** type variable causes an error. The **type** declaration did not specify **val_param**, but the **function** heading did.

183 ERROR "[" expected; "*token*" found.

The compiler was expecting to encounter a left bracket "[", but found *token* instead.

184 ERROR "]" expected; "*token*" found.

The compiler was expecting to encounter a right bracket "]", but found *token* instead.

185 ERROR Illegal type of constant "*token*" for variable "*identifier*".

You were trying to initialize variable *identifier*, but you mistakenly specified a value *token* that did not have the same type as *identifier*. The compiler will not perform automatic type transfers for variable initializations in the **var** declaration part.

188 ERROR Dynamic variable *identifier* cannot be initialized.

By default, all variables declared in routines other than the main program will be allocated dynamically. You cannot initialize dynamic variables in the **var** declaration part. If you want to get around this problem, you can use the variable allocation clause **static** to force the compiler to store a routine variable nondynamically (statically). If you use **static**, the compiler lets you initialize the variable.

190 ERROR Cannot initialize null array *identifier*.

You specified a null array (that is, an array that takes up no space in main memory) which by itself would only cause a warning; however, you mistakenly tried to initialize the null array.

191 WARNING String initializer too long for *name_of _array*; truncated to fit.

You tried to initialize *name_of_array* with a string that had too many characters. The compiler is warning you that you lost one or more characters of the string in the initialization. To avoid this, you should probably use an asterisk in the index expression of the array. The asterisk tells the compiler to figure out how many characters the string requires and declares the array accordingly. See the "Defaulting the Size of an Array" section in Chapter 3 for details.

192 ERROR Variable *name* is not EXTERN; cannot DEFINE it.

You declared *name* in a **var** declaration part, and you tried to define it in a **define** statement. You can only specify *name* in a **define** statement if you also declare it as an **extern** variable. (See Chapter 7 for details.)

194 WARNING Unbalanced comment; another comment start found before end.

You specified two comment start delimiters without specifying a comment end delimiter in between them. You can suppress this warning with the **-ncomchk** compiler option (described in Chapter 6.) Refer to the "Comments" section in Chapter 2 for details on comments.

195 ERROR Lower bound must be an integer value for upper bound of "*".

You used an asterisk (*) to force the compiler to determine the number of elements in the array, but you mistakenly specified a noninteger value as the lower bound of the array. For example, consider the right and the wrong way to use the asterisk:

```
VAR
      x : array['a'..*] of char := 'HELLO';   {Wrong!}
      x : array[ 1..* ] of char := 'HELLO';   {Right!}
```

196 WARNING Size of *array* is zero.

You specified an array whose index makes no sense. For example, you specified an enumerated value for the lower bound and an asterisk for the upper bound. (See Chapter 3 for details on array declaration.)

197 ERROR Illegal repeat count usage; valid for array elements only.

See the "Using Repeat Counts to Initialize Arrays" section in Chapter 3.

198 ERROR Illegal type for repeat count *(token)*; must be integer.

See the "Using Repeat Counts to Initialize Arrays" section in Chapter 3 for details on using repeat counts.

199 ERROR Illegal repeat count value *(token)*; must be greater than zero.

See the "Using Repeat Counts to Initialize Arrays" section in Chapter 3 for details on using repeat counts.

200 ERROR Repeat count too large by *number* for array.

See the "Using Repeat Counts to Initialize Arrays" section in Chapter 3 for details on using repeat counts.

201 ERROR OF expected for repeat count.

See the "Using Repeat Counts to Initialize Arrays" section in Chapter 3 for details on using repeat counts.

202 ERROR Illegal use of "*" repeat count for variable array *identifier*.

See the "Using Repeat Counts to Initialize Arrays" section in Chapter 3 for details on using repeat counts.

203 ERROR ":=" expected in record initialization.

You were trying to initialize a record field with a value, but you forgot the assignment phrase (:=). Perhaps you mistakenly used (=) instead of (:=).

204 ERROR Too many initializers for record init; field list exhausted at constant *value_of_constant*.

If a record has *n* fields, you tried to initialize more than *n* fields. You can only initialize *n* or less than *n* fields. See the "Initializing Data in a Record" section in Chapter 3 for details.

205 ERROR Size of *type_transfer_function* is not the same as the size of *datatype*.

You misused a type transfer function. The size (in bytes) of the *datatype* and the *type_transfer_function* must be equal. Chapter 3 details the sizes of all data types.

206 ERROR := expected in assignment statement (*token*).

Probably, you made a mistake on the left side of an assignment statement that involved a type transfer function.

207 ERROR Constant cannot be passed as argument *n* to *routine*.

You tried to pass a constant as the *n*th argument to *routine;* however, the *n*th parameter in *routine* was declared as **var**, **out**, or **in out**. There are three ways to get around this problem. First, you can change **var**, **out**, or **in out** to **in**. Second, change **var**, **out**, or **in out** to a value parameter. Third, change the constant to a variable. See Chapter 5 for a complete explanation of parameters.

208 ERROR Expression (operator = *token*) cannot be passed as argument *n* to *routine*.

You mistakenly tried to pass an expression as the *n*th argument to *routine*. The problem is that the *n*th parameter of *routine* is a **var**, **out**, or **in out** parameter. You should probably change the parameter to become a value parameter. See Chapter 5 for a complete explanation of parameters.

209 WARNING Large (*number_of_bytes* bytes) copy of argument *name_of_arg* will be done when *routine* is invoked.

You are trying to pass a large data structure (probably an array) as a value parameter. This is going to take up a lot of CPU time at run time. You should change the value parameter to a variable, **in** or **out** parameter. Chapter 5 describes the various kinds of parameters.

210 WARNING Routine *name_of_routine* needs *number* bytes of stack, which approaches the maximum stack size of *max_size* bytes.

You are trying to pass a large data structure (probably an array) as a value parameter. Consequently, your program will probably execute quite slowly. You should change the value parameter to a **var**, **in**, **out**, or **in out** parameter. The "Parameter Types" section in Chapter 5 describes all the parameter types.

211 WARNING Routine *name* needs *number* bytes of stack, which exceeds the maximum stack size of *max_size* bytes.

You probably have a large data structure (usually an array) in your code, and you may be trying to pass the structure as a value parameter. For example, an array like this

```
VAR
    big_array : array[1.100000  of integer32;
```

might exceed the maximum stack size.

If you try to run the program, you will probably get an "access violation" error. If the structure is a value parameter, you should change it to a **var**, **in**, **out**, or **in out** parameter. The "Parameter Types" section in Chapter 5 describes all the parameter types.

This warning can occur when you compile a program on one type of workstation, but not occur when you compile on another type. For example, your program might work fine on a DN460 but when you compile it on a DN330 this warning might occur. This is because of the difference in virtual address space available on different nodes.

212 ERROR Function *name* returns more than 32K bytes.

The data type of the function consumes more than 32K bytes of memory. Probably, the data type of the function is a large array. Instead of passing the information back through the function, you should pass it back through a parameter.

213 ERROR Illegal FOR statement index variable; *identifier* is a record or an array reference.

Domain Pascal does not permit a component of a record or an element of an array to be the index-variable in a **for** statement.

214 ERROR Size of argument *n* to *routine* is not equal to the expected size of *number* bytes.

You tried to pass a string as the *n*th argument to *routine*, but the *n*th parameter of *routine* was expecting a larger or smaller string. You must either change the size of the argument to match the size of the parameter, or you must declare the parameter as **univ**. See the "Univ" section in Chapter 5 for details.

228 ERROR Too many initializers for array init; " " expected, "*token*" found.

You specified more data for the array than the array can hold. (See Chapter 3 for details on declaring arrays.)

234 ERROR Compiler failure, too many nodes.

This program is so large that the compiler cannot optimize it. You can try recompiling with **-opt 0** (see Chapter 6), but we recommend that you reduce the size of the program by breaking it up into modules. Chapter 7 explains modules.

235 WARNING Potential illegal use of FOR index variable *(identifier)* outside of FOR stmt.

Domain Pascal forbids the use of the value of the index-variable after normal termination of a **for** loop. The compiler generates this message if a **for** loop has no premature exits (**exit** or **goto**) and the value of the index-variable is used outside the loop.

236 ERROR Floating-point constant "*number*" conversion problem.

*Number* was so large that the compiler encountered an overflow error when it tried to convert it from a double to a single, from a double to an integer, or from a single to an integer.

237 ERROR Compiler failure, unexpected data init construct: *token*.

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

238 ERROR *Type_of_routine1 identifier* was previously declared as a *type_of_routine2*.

You specified *identifier* as a forward procedure, but in the routine heading, you specified it as a function. Or, you specified *identifier* as a forward function, but in the routine heading, you specified it as a procedure. You must declare it as a procedure in both places or as a function in both places.

240 WARNING Size of constant *(value)* is greater than the number of bits *(number)* in packed field *name* ; constant has been truncated.

*Value* is outside the declared subrange of *name*. You must specify a *value* that falls within the declared range, redeclare *name*, or omit the keyword packed from the record declaration. (See the "Records" section in Chapter 3 for details on space allocation in packed and unpacked records.)

241 ERROR Dividing by zero in a compile-time constant expression.

You tried to divide by zero.

242 ERROR Size of a PROCEDURE or FUNCTION is undeterminable.

You mistakenly specified the name of a routine as an argument to the **sizeof** function. See the **sizeof** listing in Chapter 4 for a list of its legal arguments.

243 WARNING Variable *name* was not initialized before this use.

The compiler is warning you of the possibility of a garbage result when using the value of variable *name*. To solve this problem, you must assign a value to *name*. If *name* was declared as an **out** parameter, then you should probably change it to an **in out** parameter.

244 WARNING UNIV parameter *name* should not be passed as a value-parameter.

You specified a **univ** parameter as a value parameter. You should explicitly declare **univ** parameters as **in**, **out**, **in out**, or **var**. (See the "Univ" section in Chapter 5 for details on **univ**.) At run time, the called routine copies the value parameter. Since the site of the parameter and the argument might differ, using a **univ** value parameter might cause run-time problems.

245 ERROR Compiler Failure, Store Elimination Error

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

246 WARNING Expression passed to UNIV formal *name* was converted to *newtype*.

See the "Univ" section in Chapter 5 for details on this warning message.

247 ERROR Compiler failure, implementation restriction: Identifier-list contains too many names.

This is an implementation restriction. You specified too many identifiers in an enumerated type. (See the "Enumerated Data" section in Chapter 3 for details on declaring enumerated types.)

248 ERROR Compiler failure, limit exceeded; *limitation_message*.

The *limitation_message* explains the problem.

249 ERROR Too many nested pointer references for debug tables.

This is an implementation restriction. The symbol table (used by the debugger) cannot process a record containing pointers to other records in a chain longer than 256 elements.

250 INFO2 *Name_of_statement* statement was constant-folded at compile time.

Several conditions can cause the compiler to constant-fold a statement. One of the more common is because the Domain Pascal compiler (SR9 and later) recognizes an attempt to compare a negative number to an unsigned subrange variable. When it recognizes such an attempt, it optimizes the code and issues a warning message. For example, consider the following fragment:

```
VAR
     x : 0..65535;

 BEGIN
          . . .
     IF x = -1 THEN RETURN;
          . . .
 END;
```

The compiler generates *no* code for the **if/then** statement because it knows that a negative value of **x** is not possible.

When the compiler notes such a contradiction, it issues warning messages. These messages come from the following three groups:

```
(IF|WHILE|CASE) statement was constant-folded at
     compile-time.

 Comparison is false
 Comparison is true

 <= becomes =
 > becomes <>
 >= becomes =
 < becomes <>
```

For example, suppose you compile the following program:

```
Program warning_test;
 VAR
    x : 0.100;
 BEGIN
    write('Enter an integer--'); readln(x);
    if x <= 0 then writeln('hi');
 END.
```

The compiler issues the following two warning messages:

```
<= becomes =
      and
 IF statement was constant-folded at compile-time.
```

The first message tells you that the compiler is going to optimize the **if/then** statement. The second message tells you that the compiler is going to code the <= as an = because a < condition is not possible.

If you write the **if/then** statement in the program as

```
if x < 0 then writeln('hi');
```

the compiler prints a "Comparison is false" warning message because it is apparent to the compiler that there is no way that **x < 0** can ever be true. In such a case, the compiler generates *no* code for the **then** part of the statement.

251 ERROR Conflicting use of section name (*name_of_section*).

You specified *name_of_section* as both a code section name and a data section name. It cannot be both. See the "Section" section in Chapter 5 for details.

252 ERROR Compiler failure, invalid use of multiple sections and non-local goto to label *name_of_label*.

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

253 ERROR Compiler failure, bad address constant.

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

254 ERROR Compiler failure, invalid use of multiple sections and up-level referencing in routine ^1.

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

255 INFO2 <= becomes =.

See the description of INFO2 message number 250.

256 INFO2 > becomes <>.

See the description of INFO2 message number 250.

257 INFO2 Comparison is false.

See the description of INFO2 message number 250.

258 INFO2 Comparison is true.

See the description of INFO2 message number 250.

259 INFO2 >= becomes =.

See the description of INFO2 message number 250.

260 INFO2 < becomes <>.

See the description of INFO2 message number 250.

262 ERROR Value-parameter *name* was not specified in call to FUNCTION or PROCEDURE *identifier* declared OPTIONS(VARIABLE).

You forgot to specify a value for the argument corresponding to parameter *name*. You would get run-time access violations since the called procedure copies value parameters into temporary storage, and the procedure has a variable number of parameters. You can correct this problem in one of two ways. You can assign a value to the argument, or you can change the value parameter to a **var**, **in**, or **in out** parameter.

263 ERROR Only records may have variant tags.

Variant tags give you the capability to create records with variable sizes. For example, consider the following:

```
TYPE
    emp_stat      = (exempt, nonexempt);
    workerpointer = ^worker;
    worker = record
       first_name : array[1.10  of char;
       last_name  : array[1.14  of char;
       next_emp   : workerpointer;
       CASE emp_stat OF
          exempt    : (salary : integer16);
          nonexempt : (wages  : single;
                       plant  : array[1.20  of char);
    end;

 VAR
    current_worker : workerpointer;
```

The **emp_stat** field is a variant tag field because it uses different amounts of storage depending on its value. The function **sizeof** and the procedures **new** and **dispose** can use variant tags—for example, **NEW(current_worker, exempt)**—but only when such tags are part of a **record** variable. This error occurs if you try to use a variant tag that is not part of a **record**.

264 ERROR Too many variant tags specified for record.

You used more variant tags in the routines **new**, **dispose**, and **sizeof** than are present in the record type variable. (See the "Variant Records" section of Chapter 3 for a description of variant tags.)

265 ERROR Type of tag *name* incompatible with variant.

The value you supplied when specifying a variant tag field is not one of the choices listed in the field declaration of the record variable. (See the "Variant Records" section of Chapter 3 for a description of variant tags.)

266 ERROR No variant with value of tag *name* exists.

The value you supplied for a variant tag in the routines **new**, **dispose**, or **sizeof** is not one of the choices listed in the field declaration of the record variable. For example, you would get this error if you used the record declaration listed at ERROR message number 263, and then included this line in your program:

```
NEW(current_worker, salaried)
```

The error would occur because **salaried** is not one of the choices for the variant tag **emp_stat**. (See the "Variant Records" section of Chapter 3 for a description of variant tags.)

267 WARNING *Token1* should not be followed by *token2*; the *token1* will be ignored.

This usually appears when you have a misplaced semicolon. You might have put a semicolon (*token1*) before the reserved word **else** (*token2*).

268 WARNING Missing operator or statement terminator; inserted *token* to continue parsing.

The compiler generates this message when it is attempting to recover from errors and so continue parsing. It acts as if the missing *token* (usually a semicolon) were present, generates this message, and then goes on. To eliminate this message, insert the necessary delimiter(s) in your program and recompile.

269 ERROR Variables in libraries must be external.

The variables declared in a precompiled library file must be accessible to a program that uses the file. However, if your precompiled library contains **static** and/or **define** variables, the calling program cannot access those variables because they are not explicitly external. Such variables are not permitted. To eliminate this error, eliminate the **static** or **define** identifier from the library precompilations.

270 ERROR Compiled library failure, illegal object type.

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

271 ERROR File is not a library *pathname*.

*Pathname*, which is supposed to specify a precompiled source library file, either is not a precompiled library at all, or is a precompiled library file whose data has been corrupted. Verify *pathname*. If it is incorrect, make the appropriate fix to your code. If it is correct, precompile it again to try to get rid of the corrupted data.

272 ERROR *Library* is incompatible because it was generated by a more recent version of compiler.

Library files are not guaranteed to be upward compatible. For example, if you use libraries produced by the newest compiler in a program compiled with an older compiler, they may be incompatible. To make them compatible, you can do either of the following: use the newer compiler to compile your source program, or, use an older compiler to produce the libraries.

273 ERROR Bodies of PROCEDUREs/FUNCTIONs may not be declared in LIBRARY MODULES.

The routines defined in a precompiled library file must be accessible to a program that uses the file. However, such routines are not accessible unless they are marked with the **extern** attribute (described in Chapter 7). A routine in your precompiled library file was not marked **extern**.

274 ERROR FORWARD PROCEDURE/FUNCTION declarations are not allowed in LIBRARY MODULES.

**Forward** declarations of procedures or functions are not allowed in a precompiled library file because such routines are not accessible to a program that uses the file. Rewrite your code to eliminate the **forward** declaration and recompile.

275 ERROR INTERNAL

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

276 ERROR ":" not permitted after OTHERWISE.

You put a colon (:) after the keyword **otherwise** in a Domain Pascal **case** statement. **Otherwise** is a clause, not a label, so it does not take a colon.

277 ERROR Labels not permitted at MODULE level.

Since a module (described in Chapter 7) consists of named routines only, a label can only be declared within the scope of one of those routines. That is, there is no "main program" in a module with which a label at module level can be associated. However, you declared a label at module level. To correct this error, declare the label within the scope of the **program** block, or inside one of the routines in the module.

278 WARNING *Identifier* has already been used in another context in current scope.

Pascal forbids the redeclaration of a name that has already been used within a program block. For example, the following code fragment declares a constant **ten** at program level. Within procedure **bo**, **ten** is used to initialize variable **hold**. **Ten** then is illegally redeclared as a variable of type **real**.

```
PROGRAM illegal;
 CONST ten := 10;
 PROCEDURE bo;
    VAR hold : integer := ten;
        ten  : real;            { This is illegal! }
         .
         .
         .
```

279 INFO2 Value assigned to *identifier* is never used; assignment is eliminated by optimizer.

If *identifier*'s value has side effects, such as in a function call or in a reference to variables with the **device** attribute, the value still is computed, and the optimizer only eliminates the assignment to *identifier*. However, if there are no side effects, the optimizer also eliminates the value's computation.

You can eliminate this warning by eliminating the value assignment to *identifier*. However, there are times when you need to call a function, but are not interested in the value it returns and so don't use that value. In that case, use the **discard** procedure to explicitly eliminate the value assignment. See Chapter 4 for a description of the **discard** procedure and Chapter 3 for information about the **device** attribute.

280 WARNING Current semantics for subrange of CHAR is incompatible with SR9.

Earlier versions of the compiler (SR9 and before) incorrectly use 16 bits to store a subrange of **char**. However, SR9.5 and later versions of the compiler use only eight bits to store the subrange. The difference in the number of bits the compiler versions use can introduce incompatibilities among compilation units and data files.

281 FATAL Too many compilation errors—compilation terminated.

The errors in your program have caused an access violation in the compiler and so compilation cannot continue. Correct the problems already indicated and recompile.

283 ERROR EXTERN PROCEDURES/FUNCTIONS may not be DEFINED in PROGRAM *name*.

A main program (which contains the **program** heading) may reference externally declared routines, but it may not **define** any global entry points. Your program tried to define one or more such points.

284 ERROR FILE parameters may not be passed by value *paramname*.

Pascal forbids passing a file variable as a value parameter. To eliminate this error, change *paramname*'s declaration to **var**. The "Parameter Types" section in Chapter 5 describes all the parameter types.

285 ERROR GOTO transfers control to a structured statement outside of its scope *token*.

Your code includes an erroneous **goto** into a structured statement. Structured statements include **case**, **while**, **repeat**, **for**, and **with**.

286 ERROR Maximum line length (argument 5 to OPEN) must be an INTEGER.

You gave a value that is not an **integer** for the buffer_size argument to the **open** statement. The value must be an **integer**. See the listing for **open** in Chapter 4 for more details.

287 ERROR Maximum line length (argument 5 to OPEN) may only be specified for TEXT files.

An **open** statement may only include the **buffer_size** argument if you are opening a **text** file. However, you included the argument for an **open** of some other file type.

288 ERROR Base types of *token1* and *token2* are incompatible.

This error occurs if you try to use the **pack** or **unpack** built-in procedures on two arrays that have different base data types. Those types must be the same. For example, if one is an array of **integer32**, the other must be an array of **integer32**.

289 WARNING Must overflow bounds of array (*token1*) in order to match PACKED array.

This error can occur if you are using the **pack** or **unpack** built-in procedures. For every element in the packed array there must be a corresponding element in the unpacked array. See the listings for **pack** and **unpack** in Chapter 4 for more details.

290 ERROR Index of VARYING must be a positive integer < 65536.

In a **varying[x] of char**, declaration, you specified a negative value or a value larger than 65535 for *x*.

291 ERROR Data type of VARYING strings must be CHAR.

In a **varying[x] of** *TYPE* declaration, you specified a type other than **char** for *TYPE*.

292 ERROR Improper VARYING specification syntax (EXPRESSION).

This error can be caused by a variety of syntax errors (for example, missing '[').

293 ERROR Argument ACTUAL ARGUMENT to FORMAL ARGUMENT is not a VARYING string.

The formal argument is declared as a variable length string, but you passed it some other type of value.

294 ERROR PROCEDURE may not be called in this context (*token*).

You used a procedure name in a context where a value must be returned, such as the following: A := X(P(A)), where P is a procedure. It is illegal to use a procedure name (*token*) in the argument list of a call to another routine. This is because all of a routine's arguments must have or resolve to values. While a function returns a value, a procedure does not.

295 ERROR Modulus must be >= zero (*token*).

You specified a MOD operation, **A mod B**, where B is less than zero. This error can only occur if you compile with the **-iso** option.

296 ERROR Length of constant string exceeds maximum of 1024 characters.

A string constant may not exceed 1024 characters.

297 ERROR Only an integer constant is valid here (EXPRESSION).

There are a number of instances where only integer constants are allowed. For example, you may use only an integer constant to specify an ASCII character in Domain Pascal's syntax for embedding special characters in string constants.

298 ERROR Argument to *token* attribute conflicts with value already specified for this type.

You specified conflicting attribute types in an attribute list. For example,

```
TYPE
   int = [word, long] -32768..32767;
```

The code in this example specifies two conflicting size attributes for **int**.

299 WARNING Specified *token* attribute conflicts with attributes of base type.

You specified an attribute for an object that is not consistent with the attributes you specified previously. For example,

```
TYPE
    int = [word] -32768..32767
    long_int = [long] int;
```

The **word** attribute for the subrange **-32768..32767** tells the compiler to allocate 2 bytes of space for **int** types, but **long** tells the compiler to allocate 4 bytes for **long_int**, which is an **int** type.

300 FATAL Size of *token*1 bits is invalid for specified type.

You specified an invalid number of bits for an object. (See the "Size—Extension" section of Chapter 3 for details about size attributes.) For example,

```
TYPE
    number = integer32;
 VAR
    bad : [byte] number;
```

The size attribute, **byte**, specifies 8 bits, but an **integer 32**-type variable requires at least 32 bits.

301 WARNING Size of array element rounded up from *token*1 to *token*2 bits.

In packed arrays, if the element size is less than 16 bits, then the element is padded up to the nearest power of 2 bits. Thus, in the following example,

```
array1: PACKED ARRAY [low..high] OF 0..7;
```

4 bits would be allocated for each array element.

304 INFO1 Actual alignment of array elements (*token*1) is less than natural alignment (*token*2).

You probably specified an array of records, and the array's elements are not naturally aligned. See the "Internal Representation of Packed Records" and the "Alignment—Extension" section of Chapter 3 for more information about declaring records so that they will be naturally aligned.

305 INFO1 Actual alignment of *token*1 (*token*2) is less than natural alignment (*token*3).

A value is naturally aligned if it begins at an address that is a multiple of its size in bytes. For example, a 2-byte value is naturally aligned if it starts on a 2-byte address boundary. Similarly, an 8-byte value is naturally aligned if it starts on an 8-byte boundary. This message tells you that *token1*'s alignment is less than natural alignment.

306 INFO1 Size of variable (*token*1) rounded up to *token*2 bytes.

The size of all simple data types must be at least one byte. Therefore, if you specify a size which is a number of bits that is not evenly divisible by 8, the size will be rounded up to the next byte.

307 INFO1 Size of function result rounded up to *token* 2 bytes.

The size of all simple data types must be at least one byte. Suppose you declare a function, **foo**, that returns an integer. If you specify a number of bits that is not evenly divisible by 8 as the size for the result of **foo**, the compiler rounds the result of **foo** up to the next byte. For example, if you specify the result of **foo** as follows:

```
FUNCTION foo: [bit (20)] integer;
```

the compiler rounds the result of **foo** to 24 bits, the next byte.

308 FATAL Compiler failure, no case for object type.

The error is in the compiler, not in your code. Please contact either your HP Response Center or your local HP representative.

309 INFO3 Unnaturally aligned load/store (*token*1) diminishes code quality.

A value is naturally aligned if it begins at an address that is a multiple of its size in bytes. For example, a 2-byte value is naturally aligned if it starts on a 2-byte address boundary. Similarly, an 8-byte value is naturally aligned if it starts on an 8-byte boundary. Most computers are designed to transfer data most efficiently if the data is naturally aligned. This message tells you that your code is inefficient because it causes the compiler to load and store data that is not naturally aligned.

310 WARNING Alignment of argument is less than expected by formal parameter *token*1.

You passed an argument as a formal parameter that was aligned on a boundary lower than the boundary expected for the formal parameter. The default alignment for formal parameters is natural alignment. Therefore, unless you *specify* the alignment for formal parameters to be something else, arguments passed to them should be naturally aligned.

312 ERROR Illegal to redefine *token*1 (*token* 2).

The name of a user–defined **attribute** list may not conflict with a predeclared attribute or option.

316 ERROR *token*1 is inappropriate in this context.

You declared an attribute incorrectly.

319 WARNING Alignment of field (*token*1) is dependent on the current default alignment environment.

The alignment of the fields changes, according to the state of the **%word_alignment** or **%natural_alignment** compiler directives. See the "Compiler Directives" listing for these directives in Chapter 4 for more information.

320 WARNING Alignment of array elements is dependent on the current default alignment environment.

The alignment of array elements changes, according to the state of the **%word_alignment** or **%natural_alignment** compiler directives. See the "Compiler Directives" listing for these directives in Chapter 4 for more information.

321 WARNING Unrecognized option (*%token*1); assuming *%token*2.

You misspelled a compiler directive, and the compiler is substituting *%token*2. For example, if the compiler finds **%insert** which is *not* legal in Domain Pascal, the compiler substitutes **%include** and warns you.

322 ERROR Parameter *token1* conflicts with FORWARD or EXTERN declaration of this routine.

You changed the definition of a parameter when you repeated definitions in a routine declared with **forward** or **extern.** See the "Routine Options" section of Chapter 5 for more details about routine options.

323 ERROR Function return type conflicts with FORWARD or EXTERN declaration of this routine.

You changed the definition of a function return type when you repeated function definitions in a function declared with **forward** or **extern.** See the "Routine Options" section of Chapter 5 for more details about routine options.

324 ERROR OPTIONS declaration conflicts with FORWARD or EXTERN declaration of this routine.

You changed the definition of a routine's routine options when you repeated routine definitions in a routine declared with **forward** or **extern.** See the "Routine Options" section of Chapter 5 for more details about routine options.

325 ERROR Number of parameters in FORWARD or EXTERN declaration is different than in this declaration.

You changed the number of parameters when you repeated routine declarations in a routine declared with **forward** or **extern.** See the "Routine Options" section of Chapter 5 for more details about routine options.

326 ERROR ALIGN() may not be passed to UNIV formal parameter *token*1.

You cannot use the **align** function with a **univ** formal parameter. See the "Align" listing in Chapter 4 for more details about the **align** function.

327 ERROR Null string is illegal.

ISO Pascal does not permit the string null string ('').

**NOTE:** This is an error *only* if you compile with the **-std** option.

328 ERROR Type of FILE may not be or contain a FILE type.

It is illegal to have a FILE OF x, where x is itself a file type (e.g. FILE OF TEXT).

329 ERROR *token*1 contains a FILE type.

It is illegal to do an aggregate assignment of a record or array to another record or array of the same type if it contains a file type.

330 ERROR *token*1 is PACKED.

You cannot pack *token*1 in this context. See the "Packed Arrays" sections of Chapter 3 for details about packed arrays.

331 ERROR *token*1 is not PACKED.

You must pack *token*1 in this context. See the "Packed Arrays" sections of Chapter 3 for details about packed arrays.

332 ERROR Tag field of variant selector may not be passed by reference as parameter *token*1.

**NOTE:** This is an error *only* if you compile with the **–std** option.

333 ERROR No assignment to function variable (*token*1).

ISO requires at least one assignment to the function variable in a function's body.

**NOTE:** This is an error *only* if you compile with the **–std** option.

334 ERROR Return type of FUNCTION (*token*1) must be simple.

ISO does not allow structured types to be the return types of functions.

**NOTE:** This is an error *only* if you compile with the **–std** option.

335 ERROR Function identifier (*token*1) may not be used as a pointer variable.

A function that returns a pointer may not be dereferenced:

```
TYPE
       i_ptr = ^integer;
FUNCTION x : i_ptr;
        BEGIN
            NEW(x);
            x^ := 5;    {THIS IS AN ILLEGAL STATEMENT}
        END;
```

**NOTE:** This is an error *only* if you compile with the **–std** option.

336 ERROR At least one parameter must follow a file variable argument to READ/WRITE.

ISO requires that READ/WRITE have at least one thing to read or write.

**NOTE:** This is an error *only* if you compile with the **–std** option.

337 ERROR All values of tag type (*token*1) must appear as case constants.

**NOTE:** This is an error *only* if you compile with the **–std** option.

338 ERROR FOR index variable (*token*1) must be declared in the routine in which it is used.

**NOTE:** This is an error *only* if you compile with the **–std** option.

344 ERROR Incompatible routine OPTIONS.

The **options** clause in the procedure or function parameter you specified is incompatible with the **options** clause of the formal type signature.

348 WARNING No path exists to this statement.

The program never reaches this statement; therefore, the compiler does not generate any code for it. Sometimes a **goto** statement triggers this warning.

349 ERROR Alignment stack overflow.

You may not have more than 255 **%push_alignment** directives in a source file before the corresponding **%pop_alignment** directives. See the "Compiler Directives" listing in Chapter 4 for information about these directives.

350 ERROR Alignment stack underflow.

You cannot have more **%pop_alignment** directives than **%push_alignment** directives. See the "Compiler Directives" listing in Chapter 4 for information about these directives.

351 ERROR Unmatched %PUSH_ALIGNMENT directive.

This error indicates that you have more **%push_alignment** directives than **%pop_alignment** directives. The compiler ignores the extra directive. See the "Compiler Directives" listing in Chapter 4 for information about these directives.

352 ERROR Type of index constant (*constant*) does not match array declaration.

You tried to initialize an array with *constant*. *constant* is not the same data type as was declared for the array. The types must match.

353 ERROR Index constant (*constant*) is not within array bounds.

*constant* was not within the declared range of the array. You must either use a different constant or expand the declared range of the array.

354 ERROR ":=" is expected after index constant.

An assignment operator (:=) must appear after the index constant(s) and before the value used to initialize the constant.

355 ERROR Array elements may not be initialized more than once.

You cannot initialize an array component more than once in a single array initialization statement.

356 INFO4 Size of record is *number* in word and *number* in natural alignment environment.

You declared a record whose alignment varies with the alignment environment. Porting an application from one machine to another may cause problems if new alignment rules are used.

357 WARNING Span between elements of array (*number* bytes) does not match size of base type *typename* (*number* bytes).

This warning appears if, in a word-alignment environment, you define a record that can have different sizes in different environments, and you then define an array of those records in a natural-alignment environment. If you do not correct this error, your program will produce unpredictable results.

358 ERROR Reference outside bounds of array *number* also falls outside of stack frame.

You have an array reference that falls outside the allocated area for that particular routine. Even though it is usually a poor programming practice, it is sometimes desirable to access beyond an array declaration. But in this case you've made a reference to data that also falls outside the stack frame.

359 ERROR Expression overflows parse stack; please simplify your expression.

The expression stack of the compiler overflowed. You need to break the complex expression into smaller pieces.

360 WARNING Routine not expanded INLINE at call site (indefinite cause): *routine_name*.

This warning appears if you request inline expansion for a routine, but the compiler cannot expand the function inline because of a problem with the function call. See the discussion of **%begin_inline** and **%end_inline** under "Compiler Directives" in Chapter 4 for information about inline expansion.

361 WARNING Routine not expanded INLINE (indefinite cause): *routine_name*.

This warning appears if you request inline expansion for a routine, but the compiler cannot expand the function inline because of a problem with the routine to be expanded. See the **%begin_inline** and **%end_inline** entries under "Compiler Directives" in Chapter 4 for information about inline expansion.

362 INFO4 Routine expanded INLINE at call site: *routine_name*.

This informational message appears if you specify an optimization level of 4 and the compiler selects for inline expansion a function that you have not requested inline expansion for. See the **%begin_inline** and **%end_inline** entries under "Compiler Directives" in Chapter 4 for information about inline expansion.

363 WARNING Routine not expanded INLINE at call site (recursion): *routine_name*.

This warning appears if you request inline expansion for a recursive routine (a routine that calls itself). Recursive routines cannot be expanded inline. See the **%begin_inline** and **end_inline** entries under "Compiler Directives" in Chapter 4 for information about inline expansion.

364 WARNING Routine not expanded INLINE at call site (caller + callee too large): *routine_name*.

This warning appears if you request inline expansion for a routine, but the caller and callee routine together contain too many statements. You also get this warning if the caller and callee together have more than 255 variables, or if the callee's stack frame contains more than 24K bytes. See the **%begin_inline** and **%end_inline** entries under "Compiler Directives" in Chapter 4 for information about inline expansion.

365 WARNING Routine not expanded **inline** at call site (callee contains unexpanded procedure): *routine_name*

This warning appears if you request inline expansion for a routine that has a nested routine that makes uplevel references *and* there is more than one call site. For uplevel references to work correctly, there must be only one call site for a routine that has any number of nested routines that make uplevel references. If all nested routines are expanded in turn into the outermost routine, this problem disappears. See the **%begin_inline** entry under "Compiler Directives" in Chapter 4 for information about inline expansion.

366 WARNING Routine not expanded **inline** at call site (parameter is proc): *routine_name*.

This warning appears if you request inline expansion for a routine that has a routine as a parameter. Routines with routines as parameters cannot be expanded inline. See the **%begin_inline** and **%end_inline** entries under "Compiler Directives" in Chapter 4 for information about inline expansion.

368 WARNING Routine not expanded **inline** at call site (psect mismatch): *routine_name*.

This warning appears if you request inline expansion for a routine, but you have placed the caller and the callee in different procedure sections by means of a **section** routine attribute. The caller and callee must both occupy the same procedure section. See the **%begin_inline** and **%end_inline** entries under "Compiler Directives" in Chapter 4 for information about inline expansion; see Section 5.7.1 for information about **section**.

369 WARNING Routine not expanded INLINE (file variable): *routine_name*.

This warning appears if you request inline expansion for a routine that has a parameter of type **file**. Routines with **file** type parameters cannot be expanded inline. See the **%begin_inline** discussion under "Compiler Directives" in Chapter 4 for information about inline expansion.

370 WARNING Routine not expanded INLINE (record variable): *routine_name*.

This warning appears if you request inline expansion for a routine that has a parameter of type **record** that contains a **file** type. Routines with records that have **file** type fields cannot be expanded inline. See the **%begin_inline** discussion under "Compiler Directives" in Chapter 4 for information about inline expansion.

371 WARNING Routine not referenced, code deleted: *routine_name*.

You declared a routine that is not external and has no callers within the current compilation unit. The compiler has deleted the code for the routine.

374 ERROR Unable to convert floating point constant to integer: *token*.

The compiler encountered an overflow error when it tried to convert the floating-point constant to an integer.

375 ERROR Call passes argument block of *number* bytes (implementation limit is 2K).

On a Series 10000 system, you may not pass an argument under the **c_param** option that contains more than 2K bytes—for example, a very large record. If possible, pass a pointer to the record, or pass it as a reference argument.

376 ERROR References to atomic/volatile objects must be properly aligned.

Under normal circumstances the Series 10000 compiler generates, loads, and stores to and from memory in such a way as to avoid alignment traps. However, if the reference in question is **atomic**, the compiler insists on having the datum at least word aligned.

377 ERROR Unmatched inline directive.

You must match an **%end_inline** to every **%begin_inline** and an **%end_noinline** to every **%begin_noinline**. You cannot nest inline directives. You must close one inline scope before opening another. See the "Compiler Directives" listing in Chapter 4 for information about these directives.

385 ERROR Auto-padding of strings is restricted to *number* bytes.

You are assigning all the elements of a string to a larger array of **char**. Domain Pascal limits you to 4096 bytes. Auto-padding for larger arrays is not allowed.

388 WARNING Loop structure deleted.

The compiler has deleted a loop. This loop has both of the following characteristics:

- The exit condition is not dependent on the code within the loop.
- The loop has no side effects; that is, it contains no calls and does not update a variable that is subsequently used.

You should pay particular attention to this warning, as the compiler may have noticed a subtle bug in your code.

400 WARNING Expression will fault at runtime — divide by zero.

You have a divide-by-zero expression. An evaluation of this expression will cause a run-time fault.

401 WARNING Expression will fault at runtime — argument is outside function domain.

You have an expression whose value is undefined. An example of this is the natural log of a negative number. The compiler is informing you that an evaluation of this expression will cause a run-time fault.

402 WARNING Expression will fault at runtime — floating-point overflow.

You have an expression that causes floating-point arithmetic to overflow. An evaluation of this expression will cause a run-time fault.

403 WARNING Expression will fault at runtime — floating-point underflow.

You have an expression that causes floating-point arithmetic to underflow. An evaluation of this expression will cause a run-time fault.

404 WARNING Expression will fault at runtime — negative to non-integer power.

You have an expression that raises a negative to a real power. This is an undefined operation. An evaluation of this expression will cause a run-time fault.

## 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:

- Operating system errors, described in Section 9.3.3
- Floating–point errors, described in Section 9.3.4

### 9.3.1 Causes of Run–Time Errors

Operating system run–time errors most commonly occur when your program attempts to access a forbidden area of memory. There are several ways in which a program can do this:

- If your program tries to write to an area of memory that has been allocated but is read–only, such as a library or your program text, it causes an access violation.
- If your program tries to access an area of memory that has not been allocated at all, it causes a "reference to illegal address" error.
- If your program writes over a part of the stack frame that contains data the function needs in order to return to the caller, it causes a stack unwind error.
- If your program tries to write to a guard segment—a small area of memory on each side of a stack frame—it causes a guard fault.

Figure 9–1 shows the main areas of memory a program uses and the errors caused by invalid uses of these areas.

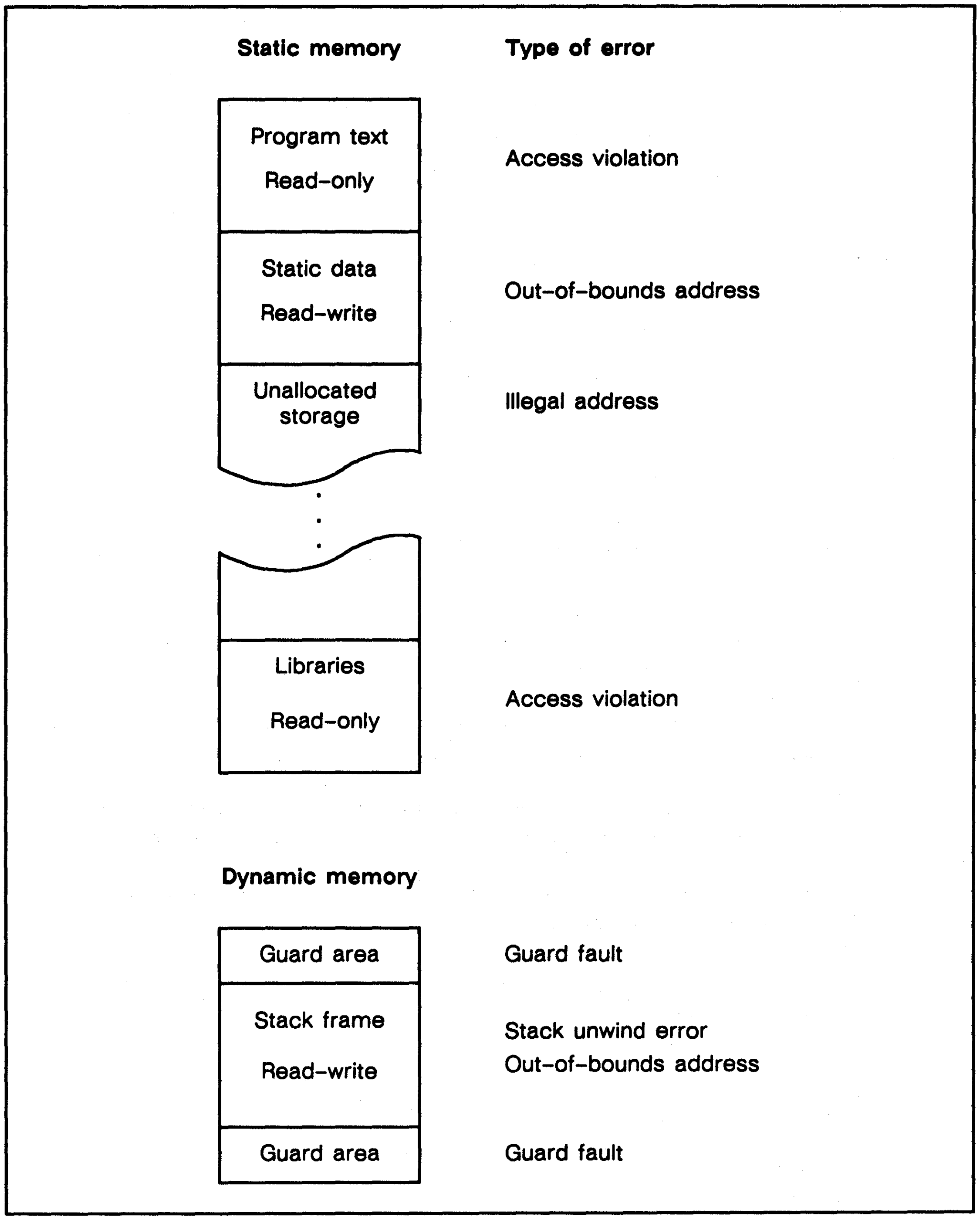

*Figure 9-1. System Memory and Run-Time Errors*

## 9.3.2 Debugging Run-Time Errors

If you get a run-time error after your program compiles with a warning message, the warning message may tell you what and where the problem is. However, if the program compiles with no warnings, you need to determine what line of your program is causing the error before you can determine what the error is and how to fix it. Two Domain/OS tools can provide this information:

- The traceback tool, **tb**
- 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 command **tb** tells you what line of your source code caused the error. The command **tb -full** provides information about the address that caused the error and the contents of the machine registers. For more information about **tb**, see Section 6.9.1.

If the information provided by **tb** is not enough to enable you to find the problem, you need to invoke the debugger. For information about using the debugger, refer to the *Domain Distributed Debugging Environment Reference Manual.*

## 9.3.3 Operating System Error Messages

`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. An invalid address, such as zero or a negative number, also causes this error.

An access violation is most commonly caused by stray pointers. For example, you can cause an access violation by assigning a value to a record field before you have allocated storage for the record.

`Apollo-specific fault (UNIX/signal)`

If you do a full traceback, this error will probably resolve to another error, such as "reference to out-of-bounds address." For information, refer to the discussion of that error.

`Bus error`

If you do a traceback, this BSD and SysV environment run-time error will resolve to another error, such as "odd address error." For information, refer to the discussion of that error.

`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.) This error is usually caused by exceeding the stack size. One solution is to increase the stack size with the following command line:

**bind** *object_files* **-stacksize** *decimal_number*

The **bind** utility is described in Section 6.5.2 and in the *Domain/OS Programming Environment Reference*.

Another solution is to check whether you are walking off the end of an array into the guard region.

This error can also be caused by an infinitely recursive call or by a stray pointer that happened to land on a guard segment.

`Memory fault`

If you do a traceback, this SysV environment run-time error will probably resolve to "access violation," "guard fault," "reference to illegal address," or "reference to out-of-bounds address." For information, refer to the discussions of these errors.

`odd address error (OS/fault handler)`

Your program attempts to access an address that is odd (that is, not divisible by 2). You may not access an odd address.

This error occurs only on Series 10000 systems and on older M680x0 machines such as the DN300.

This problem is most commonly caused by stray pointers.

`reference to illegal address (OS/MST manager)`

Your program attempts to access address space that isn't allocated at all; it is not mapped into memory.

This problem is most commonly caused by a stray pointer or by an array reference that gets into an invalid area of memory.

`reference to out-of-bounds address (OS/MST manager)`

Your program references address space that is allocated but lies beyond the end of the mapped object.

This problem is most commonly caused by running off the end of an array.

`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 discussions of these errors.

```
unable to unwind stack because of invalid stack frame (process manager/process
               fault manager)
```

Somehow, somewhere, the run-time stack has been trashed (most likely by your program going astray) and is unusable.

This problem is commonly caused by an infinitely recursive program, or by an array assignment that runs off the end of the array into the stack frame. It can also be caused by stray pointers and by mismatched arguments.

## 9.3.4 Floating-Point Errors

This section gives brief explanations of some common floating-point errors. We discuss only errors whose meaning is not obvious; we omit other common errors, such as division by zero or floating-point overflow, that are easy to understand. For complete information about floating-point calculations on Domain/OS systems, consult the *Domain Floating-Point Guide*.

`Floating exception`

If you do a traceback, this BSD and SysV environment run-time error will resolve to another error, such as "floating point operand error." For information, refer to the discussion of that error.

`floating point operand error  (OS/fault handler)`

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. An example is to give a negative number as argument to a built-in logarithm function.

`floating point branch/set on unordered condition (OS/fault handler)`

This error means that you got a QNAN signal (Quiet Not-A-Number). The bit pattern for the floating point variable has all exponent bits set (to 1) and also had the MSB (most significant bit) of the fraction set. Such a bit pattern should never occur as the result of an arithmetic operation on legitimate floating-point values. It can result from uninitialized variables, from type transfers, from an operand error, or from operations with NAN (Not-A-Number) inputs.

`floating point signalling not-a-number (OS/fault handler)`

This error, a Signaling NAN (SNAN), is like a QNAN except that the MSB of the fraction is not set (0), but at least one bit in the fraction is set (to 1). An SNAN is never created as the result of an operation. Any arithmetic operation on an SNAN will give this error.

# Appendix A

# Reserved Words and Predeclared Identifiers

This appendix lists the reserved words and predeclared identifiers in Domain Pascal.

Reserved words, listed in Table A-1, are names of statements, data types, and operators. You can use reserved words only with their reserved meanings (and within strings and comments). You cannot use a reserved word as an identifier.

*Table A-1. Reserved Words*

| | | | |
|---|---|---|---|
| **and** | **end** | **not** | **set** |
| **array** | **file** | **of** | **then** |
| **begin** | **for** | **or** | **to** |
| **case** | **function** | **packed** | **type** |
| **const** | **goto** | **procedure** | **until** |
| **div** | **if** | **program** | **var** |
| **do** | **in** | **record** | **while** |
| **downto** | **label** | **repeat** | **with** |
| **else** | **mod** | | |

Table A-2 lists the predeclared identifiers. These identifiers name types, functions, procedures, values, and files. You can redefine predeclared identifiers; however, doing so means that you can no longer use the identifier for its original meaning within the scope of the redefinition.

*Table A-2. Predeclared Identifiers*

| | | | |
|---|---|---|---|
| **abs** | **exp** | next | set_sr |
| addr | extern | **nil** | **sin** |
| align | **false** | **odd** | single |
| append | find | open | sizeof |
| **arctan** | firstof | **ord** | **sqr** |
| arshft | **forward** | otherwise | **sqrt** |
| **boolean** | **get** | out | static |
| **char** | in_range | **output** | string |
| **chr** | **input** | **page** | substr |
| close | **integer** | **pred** | **succ** |
| **cos** | integer16 | ptoc | **text** |
| ctop | integer32 | **put** | **true** |
| define | internal | **read** | **trunc** |
| disable | lastof | **readln** | univ |
| discard | **ln** | **real** | univ_ptr |
| **dispose** | lshft | replace | val_param |
| double | max | **reset** | varying |
| enable | **maxint** | return | **write** |
| **eof** | min | **rewrite** | **writeln** |
| **eoln** | module | **round** | xor |
| exit | **new** | rshft | |

# Appendix B

## ISO Latin–1 Table

Domain Pascal 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 ISO Latin–1 characters.

You can use Latin–1 characters in comments or character strings, but 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 Pascal 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 (Cont.)*

| 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 (Cont.)*

| 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 (Cont.)*

| 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 | Ð | 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

# Extensions to Standard Pascal

This appendix describes Domain Pascal's extensions to ISO standard Pascal.

## C.1 Extensions to Program Organization

Chapter 2 describes the elements that make up a Pascal program. This section describes the extensions to standard Pascal.

### C.1.1 Identifiers

Although an identifier must begin with a letter, you can include underscores (_) or dollar signs ($) in the name. For example, **mailing_$lists** is a legal identifier.

### C.1.2 Integers

You can specify integers in any base from 2 to 16. To do so, use the following syntax:

*base#value*

For *base*, enter an integer from 2 to 16. For *value* enter any integer within that base. If the *base* is greater than 10, use the letters A through F (or a through f) to represent digits with the values 10 through 15.

For example, consider the following integer constant declarations:

```
half_life := 5260;       /* default     (base 10) */
hexagrams := 16#1c6;     /* hexadecimal (base 16) */
luck      := 2#10010;    /* binary      (base 2)  */
wheat     := 8#723;      /* octal       (base 8)  */
```

### C.1.3 Comments

You can specify comments in any of the following three ways:

{ comment }
(* comment *)
"comment"

(The spaces before and after the comment delimiters are for clarity only; you don't have to include these spaces.) For example, here are three comments:

```
{ This is a comment. }
 (* This is a comment. *)
 "This is a comment."
```

Unlike standard Pascal, the comment delimiters of Domain Pascal must match. For example, a comment that starts with a left brace doesn't end until the compiler encounters a right brace. Therefore, you can nest comments, for example:

```
{ You can (*nest*) comments inside other comments. }
```

The Domain Pascal compiler ignores the text of the comment, and interprets the first matching delimiter as the end of the comment.

Standard Pascal does not permit nested comments. If you want to use unmatched comment delimiters, as standard Pascal allows, you must compile with the **-iso** switch. Chapter 6 describes that switch.

Finally, Domain Pascal permits you to put compiler directives inside comment delimiters. However, if you do so, you *cannot* use spaces; see the listing for "Compiler Directives" in Chapter 4 for details.

### C.1.4 Sections

Domain Pascal allows you to assign code and data in your program to a nondefault section. A section is a named contiguous area of main memory.

### C.1.5 Declarations

You can declare the **label**, **const**, **type**, and **var** declaration parts in any order. You can specify the declaration parts an unlimited number of times.

In addition to **label**, **const**, **type**, and **var** declaration parts, you can also declare a **define** part, which is detailed in Chapter 7, an **attribute** part, which is detailed in Chapter 3, and a **routine options** part, which is detailed in Chapter 5.

## C.1.6 Constants

You can set a constant equal to a real, integer, string, char, or set expression. The constant can also be the pointer expression **nil**. The expression can contain the following types of terms:

- A real number, an integer, a character, a string, a set, a Boolean, or **nil**
- A constant that has already been defined in the **const** declaration part (note that you cannot use a variable here)
- Any predefined Domain Pascal function (for example, **chr**, **sqr**, **lshft**, **sizeof**), but only if the argument to the function is a constant, or, in the case of **sizeof**, an array.
- A type transfer function

You can optionally separate these terms with any of the following operators:

| **Operator** | **Data Type of Operand** |
|---|---|
| +, –, * | Integer, real, or set |
| / | Real |
| **mod, div, !, &, ~** | Integer |

For example, the following **const** declaration part defines ten constants:

```
CONST
    x = 10;
    y = 100;
    z = x + y;
    current_year = 1994;
    leap_offset  = (current_year mod 4);
    bell         = chr(7);
    BEL          = 7
    alert        = 'WARNING'(BEL, BEL)
    pathname     = '//et/go_home';
    pathname_len = sizeof(pathname);
```

## C.1.7 Labels

In standard Pascal, only integers can be used as labels. In Domain Pascal, you can use both identifiers and integers as labels.

## C.2 Extensions to Data Types

Chapter 3 describes the data types supported by Domain Pascal. This section describes the extensions that Domain Pascal supports.

### C.2.1 Initializing Variables in the Var Declaration Part

Domain Pascal lets you initialize variables in the var declaration part. You can initialize integer, real, Boolean, char, subrange, set, enumerated, array, record, and pointer variables with an assignment statement following the data type; for example:

```
VAR
     x : integer := 17;
     r : real := 5.3E-14;
     a : array[1..7] of char := 'Wyoming';
```

For arrays in particular, Domain Pascal supports many extensions for simplifying initialization. See Chapter 3 for details.

### C.2.2 Integers

Domain Pascal supports the following two nonstandard predeclared integer data types:

- **Integer16** — Use it to declare a signed 16-bit integer. (**Integer** and **integer16** have identical meanings.)
- **Integer32** — Use it to declare a signed 32-bit integer. A signed 32-bit integer variable can be any value from -2147483648 to +2147483647.

### C.2.3 Reals

Domain Pascal supports the following two nonstandard predeclared real data types:

- **Single** — Same as **real**.
- **Double** — Use it to declare a signed double-precision real variable. Domain Pascal represents a double-precision real number in 64-bits. A double-precision real variable has approximately 16 significant digits.

### C.2.4 Pointer Types

In addition to the standard pointer type, Domain Pascal supports a **univ_ptr** type and a special pointer type that points to procedures and functions.

The predeclared data type **univ_ptr** is a universal pointer type. A variable of type **univ_ptr** can point to a variable of any type. You can use a **univ_ptr** variable in the following contexts only:

- Comparison with a pointer of any type
- Assignment to or from a pointer of any type
- Formal or actual parameter for any pointer type
- Assignment to the result of a function

Domain Pascal supports a special pointer data type that points to a procedure or a function. By using procedure and function data types, you can pass the addresses of routines obtained with the **addr** predeclared function. (See the **addr** listing of Chapter 4 for a description of this function.) You may only obtain the addresses of top-level procedures and functions; you cannot obtain the addresses of nested or explicitly declared **internal** procedures and functions. (See Chapter 5 for details about **internal** procedures.)

## C.2.5 Variable-Length Strings

Domain supports variable-length strings, which are declared with the varying type specifier. Unlike fixed-length arrays, the size of variable-length strings can change dynamically during program execution.

## C.2.6 Named Sections

By default, Domain Pascal stores all variables declared in the **var** declaration part at the program or module level to the **.data** section. However, Domain Pascal enables you to assign variables to sections other than **.data**. Named variable sections are synonymous with named common blocks in FORTRAN.

## C.2.7 Variable and Type Attributes

Domain Pascal supports attributes for variables and types. These attributes supply additional information to the compiler when you declare a variable or a type.

The **volatile, atomic,** and **device** attributes enable you to turn off certain optimizations that would otherwise ruin programs that access device registers or shared memory locations. The **address** attribute associates a variable with a specific virtual address. Alignment and size attributes enable you to enhance your program's performance by specifying methods for storage allocation. Domain Pascal supports the following alignment attributes: **aligned, natural**. Domain Pascal supports the following size attributes: **bit, byte, word, long, quad**.

Domain Pascal supports an **attribute** declaration that allows you to define attributes.

### C.2.8 Aligned Record and Unaligned Record

Domain Pascal supports **aligned record** and **unaligned record** data types that you can use to make sure that records receive the same layout in all alignment environments.

## C.3 Extensions to Code

Chapter 4 describes the action part (the executable block) of a Pascal program. This section describes the extensions.

### C.3.1 Exponentiation Operator

Domain Pascal supports an exponentiation operator in the following form:

*mantissa**exponent.*

### C.3.2 Bit Operators

Domain Pascal supports four bit operators for bitwise **and, not, xor,** and **or** operations. These operators perform Boolean operations by comparing the bits in each bit position of two integer arguments. For details on these operators see the "Bit Operators" listing in Chapter 4.

### C.3.3 Boolean Short-Circuit Operators

Domain Pascal supports the **and then** and **or else** operators. You can use **and then** and **or else** to guarantee that Domain Pascal will evaluate Boolean expressions in the order that you write them. They also guarantee "short-circuit" evaluation; that is, at run time, the system will only evaluate as many expressions as is necessary. For details, see the **and** and the **or** listings in Chapter 4.

### C.3.3 Bit-Shift Functions

Domain Pascal supports the following three bit-shift functions:

- **rshft,** which shifts the bits in an integer a specified number of spaces to the right.
- **arshft,** which shifts the bits in an integer a specified number of spaces to the right and preserves the sign of the integer.
- **lshft,** which shifts the bits in an integer a specified number of spaces to the left.

For syntax details, see the **rshft, arshft,** and **lshft** listings in Chapter 4.

## C.3.4 Compiler Directives

Domain Pascal supports the compiler directives shown in Table 4-11.

You can place a directive anywhere in your program. To use a directive, specify its name in a comment or as a statement. For example, all of the following formats are valid:

```
{%directive}
(*%directive*)
%directive
```

If you specify a directive within a comment, the percent sign must be the first character after the delimiter. (Spaces count as characters.)

Domain Pascal supports a predeclared conditional variable, **_BFMT__COFF**, whose value is set to true whenever the compiler is generating COFF (Common Object File Format) files.

Domain Pascal supports a pair of predeclared conditional variables that you can use to find out whether the compiler is generating code for the 68000 family of workstations or for the Series 10000 workstation. These variables are: **_ISP__M68K** (for 68000 code generation) and **_ISP__A88K** (for Series 10000 code generation).

## C.3.5 Addr Function

Domain Pascal supports an **addr** function that returns the virtual address of the specified variable or routine. For syntax details, see the **addr** listing of Chapter 4.

## C.3.6 Align Function

Domain Pascal supports an **align** function that copies an expression to memory and aligns it to match the specification of a formal parameter of an external routine.

## C.3.7 Max and Min Functions

Domain Pascal supports a **max** and a **min** function for finding the larger and smaller of two operands, respectively.

## C.3.8 Discard Procedure

Domain Pascal supports a **discard** procedure for explicitly discarding the computed value of an expression. It usually is used with a function (which in standard Pascal *must* return a value) for which the value is not needed. The optimizer may eliminate the computation and issue a warning message if the return value isn't used, but **discard** explicitly throws away the computed value and so eliminates the warning message.

## C.3.9 Routines for Variable-Length Strings

Domain Pascal supports four routines for manipulating variable-length strings:

| | |
|---|---|
| **append** | Concatenates two or more strings. The destination string must be a variable-length string. |
| **ctop** | Adjusts the current length of a variable-length string based on the presence of a terminating null character. |
| **ptoc** | Appends a terminating null character to the body of a variable-length string. |
| **substr** | Finds a substring in a variable-length or fixed-length string. |

## C.3.10 I/O Procedures

Domain Pascal supports the I/O procedures of standard Pascal, plus the following four procedures:

- **Open**, which opens permanent files for I/O access.
- **Close**, which explicitly closes an open file.
- **Find**, which locates a specific element in a record-structured file.
- **Replace**, which modifies an existing element in a record-structured file.

As in standard Pascal, you can create a temporary file with the **rewrite** procedure; however, by using the **open** procedure, you can create a permanent file. (Here, "permanent" means a file that exists even after the program terminates.)

When a program terminates, the operating system automatically closes any open files. However, because an open file can clog system resources, Domain Pascal provides a **close** procedure that allows you to close a file from within your program.

The **find** procedure locates records from a record-structured file. Records here refer to the elements in a file whose file variable was declared as a **file of** data type. By using **replace** in combination with **find**, you can replace an existing record.

From a Domain Pascal program, you can easily access Input Output Stream calls (known as IOS calls) and formatting calls (known as VFMT calls).

For syntax details, see the **open**, **close**, **find**, and **replace** listings in Chapter 4. For an overview of I/O (including IOS calls and VFMT calls), see Chapter 8.

## C.3.12 Loops

Domain Pascal supports **for**, **while**, and **repeat**, which are the three looping statements of standard Pascal. Domain Pascal also supplies the following two additional statements for further control within a loop:

- A **next** statement for skipping over the current iteration of a loop
- An **exit** statement for unconditionally jumping out of the current loop

For syntax details, see the **next** and **exit** listings of Chapter 4.

## C.3.13 Range of a Specified Data Type

Domain Pascal supports

- A **firstof** function for returning the first possible value of a specified scalar data type
- A **lastof** function for returning the last possible value of a specified scalar data type

For syntax details, see the **firstof** and **lastof** listings in Chapter 4.

## C.3.14 Integer Subrange Testing

By default, Domain Pascal does not check the value of input data to see that it falls within the defined range of a variable declared as a subrange of integers. To determine whether or not a specified value is within the defined integer subrange, you can use the **in_range** function. For syntax details, see the **in_range** listing in Chapter 4.

## C.3.15 Extensions to Read and Readln

In addition to allowing input into any real, integer, char, or subrange variable, as standard Pascal allows, Domain Pascal's **read** and **readln** also allow input to a Boolean or enumerated variable.

## C.3.16 Premature Return from Routines

As in standard Pascal, Domain Pascal returns control to the calling routine after executing the last line in the called routine. If you want to return to the calling routine before reaching the last line, you can issue a **return** statement. For syntax details, see the **return** listing in Chapter 4.

## C.3.17 Memory Allocation of a Variable

Domain Pascal supports a **sizeof** function. This function returns the size (in bytes) that a data type (predeclared or user-defined), variable, constant, or string inhabits in main memory.

## C.3.18 Extensions to With

Domain Pascal supports the standard format of **with** as well as the following alternative format:

**with** *v1:identifier1, v2:identifier2, ... vN:identifierN* **do**
    *stmnt*;

An *identifier* is a pseudonym for the record variable *v*. To specify a record, use the *identifier* instead of the record variable *v*. Furthermore, to specify a field in a record, use *identifier.field_name* rather than *v.field_name*.

For example, given the following record declaration

```
basketball_team = record
                  mascot    : array[1..15] of char;
                  height    : single;
                  end;
```

consider the following three methods of assigning values:

```
readln(basketball_team.mascot);        {Not using WITH.}
readln(basketball_team.height);

WITH basketball_team DO               {Using standard WITH.}
    begin
        readln(mascot);
        readln(height);
    end;

WITH basketball_team : B  DO   {Using extension to WITH.}
    begin
        readln(B.mascot);
        readln(B.height);
    end;
```

The **with** extension is useful for working with long record names when two records contain fields that have the same names.

### C.3.19 Type Transfer Functions

Domain Pascal supports type transfer functions which enable you to change the type of a variable or expression within a statement. To perform a type transfer function, use any user–created or standard type name as if it were a function name in order to "map" the value of its argument into that type.

With one exception, the size of the argument must be the same as the size of the destination type. (Chapter 3 describes the size of each data type). This size equality is required because the type transfer function does not change any bits in the argument. Domain Pascal just "sees" the argument as a value of the new type. The one exception is that integer subranges are compatible.

### C.3.20 Extensions to Write and Writeln

Domain Pascal allows you to specify a negative field width for **chars**, **strings**, and arrays of **chars**. Also, if you specify a one–part field width for a real number, Domain Pascal adds or removes leading blanks. See the listing for **write** and **writeln** in Chapter 4 for details.

## C.4 Extensions to Routines

Chapter 5 describes procedures and functions. The term "routine" means either procedure or function. Also, the term "argument" refers to the data passed to a routine while "parameter" means the templates for the data to be received.

The following subsections describe Domain Pascal's extensions to routine calling.

### C.4.1 Direction of Data Transfer

In standard Pascal, you cannot specify the direction of parameter passing. However, Domain Pascal supports extensions to overcome this problem. You can use the following keywords in your routine declaration:

- **In** — This keyword tells the compiler that you are going to pass a value to this parameter, and that the routine is not allowed to alter its value. If the called routine does attempt to change its value (that is, use it on the left side of an assignment statement), the compiler issues an "Assignment to IN argument" error.

- **Out** — This keyword tells the compiler that you are not going to pass a value to the parameter, but that you expect the routine to assign a value to the parameter. It is incorrect to try to use the parameter before the routine has assigned a value to it, although the compiler does not issue a warning or error in this case.

If the called routine does not attempt to assign a value to the parameter, the compiler may issue a "Variable was not initialized before this use" warning. This could occur if your routine only assigns a value to the parameter under certain conditions. If that is the case, you should designate the parameter as **var** instead of **out.**

In some cases, the compiler cannot determine whether all paths leading to an **out** parameter assign a value to it. If that happens, the compiler does not issue a warning message.

- **In out** — This keyword tells the compiler that you are going to pass a value to the parameter, and that the called routine is permitted to modify this value. It is incorrect to call the routine before assigning a value to the parameter, although the compiler does not issue a warning or error in this case. The compiler also doesn't complain if the called routine does not attempt to modify this value.

## C.4.2 Universal Parameter Specification

By default, Domain Pascal and standard Pascal check to ensure that the argument you pass to a routine has the same data type as the parameter you defined for the routine. As an extension, you can tell Domain Pascal to suppress this type checking. You do this by using the keyword **univ** prior to a type name in a parameter list. By using **univ**, you can pass an argument that has a different data type than its corresponding parameter.

**Univ** is especially useful for passing arrays.

## C.4.3 Routine Options

Standard Pascal supports a **forward** option. Domain Pascal supports the **forward** option, and also supports the following routine options:

- **Extern** — By default, Pascal expects a called routine to be defined within the source code file where it is called. The **extern** option tells the compiler that the routine is possibly defined outside of this source code file.

- **Internal** — By default, all routines defined in modules become global symbols. But, if you declare the routine with the **internal** option, the compiler makes the routine a local symbol.

- **Variable** — By default, you must pass the same number of arguments to a routine each time you call the routine. However, by using the **variable** option in a routine declaration, you can pass a variable number of arguments to the routine.

- **Abnormal** — This option warns the compiler that a routine can cause an abnormal transfer of control.

- **Val_param** — By default, Domain Pascal passes arguments by reference. However, by using the **val_param** option, you tell Domain Pascal to pass arguments by value.

- **Nosave** — This option indicates that the contents of data registers D2 through D7, address registers A2 through A4, and floating-point registers FP2 through FP7 will not be saved when a called assembly language routine finishes and returns to the Domain Pascal program. However, two registers are preserved: A5, which holds the pointer to the current stack area, and A6, which holds the address of the current stack frame.

- **Noreturn** — This option specifies an unconditional transfer of control; once a routine marked **noreturn** is called, control can never return to the caller.

- **D0_return** — By default, a Pascal function returning the value of a pointer type variable puts that value in address register A0. **D0_return** tells the compiler to put the value in A0 *and* data register D0.

- **A0_return** — Specifying **A0_return** tells the compiler to put any values returned from a called routine into A0 and also to load return values to the calling routine from A0.

- **C_param** — Specifying **C_param** implies **D0_return** and **val_param** and also tells the compiler to pass record data types by value, rather than by reference.

Domain Pascal supports a **routine_option** declaration part that allows you to define your own names for groups of routine options. Furthermore, Domain Pascal provides a special name, **default_routine_options**, that allows you to define the default routine options for every routine in a module.

### C.4.4 Routine Attribute List

You can specify a routine attribute list when you declare a routine. Within the routine attribute list, you can specify a nondefault section name.

A "section" is a named contiguous area of an executing object. (Refer to the *Domain/OS Programming Environment Reference* for full details on sections.) By default, the compiler assigns code to the **.text** section and data to the **.data** section. Thus, by default, all code from every routine in the program is assigned to **.text**, and all data from every routine in the program is assigned to **.data**.

However, Domain Pascal permits you to override the default of **.text** and **.data** on a routine-by-routine basis. (You can also override the defaults on a variable-by-variable or module-by-module basis.) This makes it possible to organize the run-time placement of routines so that logically related routines can share the same page of main memory and thus reduce page faults. Conversely, you can declare a rarely called routine as being in a separate section from the frequently called routines.

## C.5 Modularity

Domain Pascal allows you to break your program into separately compiled source files. After compiling all the source files, you can bind the resulting objects into one executable object file. Chapter 7 documents the details.

## C.6 Other Features of Domain Pascal

Domain Pascal supports many other features, such as the ability to call routines written in other Domain languages. However, the remaining features are all implementation-dependent features, and not actual extensions.

# Appendix D

# Deviations from Standard Pascal

This appendix describes Domain Pascal's deviations from ISO standard Pascal, and documents the sections in the ISO standard document to which Domain Pascal does not completely adhere.

## D.1 Deviations from the Standard

Domain Pascal does not include certain features of standard Pascal, and this list documents the deviations:

- Standard Pascal does not limit the length of identifiers; Domain Pascal limits identifiers to 4096 characters.
- In standard Pascal the file list in the program heading is optional; Domain Pascal ignores the file list in the program heading.
- Standard Pascal does not limit the number of dimensions for arrays; Domain Pascal allows arrays of up to eight dimensions only.
- Standard Pascal supports **packed** arrays, **packed** records, and **packed** sets. Domain Pascal does not support **packed** sets, but you can get around this restriction by putting small sets in **packed** records. See Section 3.9.2 for further information on packing small sets within a **packed** record.
- Standard Pascal requires that certain errors in code generate run–time faults. Examples of such errors include dereferencing NIL pointers and exceeding the bounds of an array. However, if your program contains one of these errors, but the statement with the erroneous code has no effect elsewhere in the program, the optimizer may eliminate the code from your program. In this case, the run–time fault will not occur.

## D.2 Deviations from Specific Sections of the Standard

The following section lists the sections in the ISO standard document (ISO 7185–1982) to which Domain Pascal does not completely adhere and the ways in which Domain Pascal does not adhere.

6.1.2 You may redeclare NIL.

6.1.5 You are not required to include a sequence of digits after a period in a floating-point number.

6.1.8 You are not required to leave a blank between a number and a word-type operator. For example, Domain Pascal accepts the following:

```
result := 10mod 1;
```

6.2.2 The following type of declaration works under Domain Pascal:

```
TYPE
     rec = record
           ptr    : ^my_var;
           my_var : integer
     end;
     my_var = rec;
```

6.4.3.3 There is no requirement that all values of a tag-type in a record appear as **case** constants.

Domain Pascal does not detect a reference to an inactive variant field. Also, it does not mark variant fields as inactive when a new variant tag becomes active.

6.4.5 Subranges of the same type that are defined with different ranges are considered identical.

Domain Pascal considers structurally identical types to be identical. For example, the following are identical under Domain Pascal:

```
TYPE
     first = array[1..20] of integer32;
     second = array[1..20] of integer32;
```

Also, Domain Pascal ignores the keyword **packed**, so structurally identical sets are considered identical.

6.4.6 You may assign structured types containing a file component to each other.

You may make an assignment from an integer expression that includes a value outside one of the variables' declared subranges. Also, you may pass an integer argument that is outside the corresponding parameter's declared subrange. In addition, Domain Pascal considers sets of different subranges from the same enumerated type to be compatible for assignment and as parameters.

6.5.5 Domain Pascal does not detect when a field variable passed as a **var** parameter is modified. It also does not detect a modification to a file variable when a reference to the buffer exists.

6.6.3.3 You may pass the selector of a variant or a component of a **packed** variable as a **var** parameter. Also, Domain Pascal accepts a procedure call like the following where **x** is declared in the procedure heading as being a **var** parameter:

```
proc_name((x));
```

6.6.3.5, 6.6.3.6

Domain Pascal treats **integer** and subrange of **integer** as identical.

6.6.3.6 Domain Pascal considers the following to be identical:

```
VAR                                 VAR
     a : integer;                        a,b : integer;
     b : integer;
```

6.6.5.2 Domain Pascal does not detect a **put** of an undefined buffer variable at compile time. It does not consider a **read** of an enumerated type to be an error, or an assignment from a file variable of an enumerated type followed by a **get** to be an error. You may write an integer expression that includes a value outside one of the variables' declared subranges. Also, you may make an assignment from an integer expression that includes a value outside one of the file variables' declared subranges.

6.6.5.3 You may **dispose** a pointer that is active because it has been dereferenced as a parameter or in a **with** block. In addition, Domain Pascal does not report an error if you use a pointer variable after you have **disposed** of it or if you **dispose** of a dangling pointer (that is, a pointer with an address assigned to another pointer).

You also may use a record allocated with a long form of **new** as an operand in an expression or as a variable in an assignment statement. You may pass as an argument a variable that was allocated with **new** and that uses variant tags.

Domain Pascal does not report an error if you use different tags for a variable in **new** and **dispose**. It also does not detect the activation of a variant on a variable that **new** allocated with a different tag, or if your program includes illegal variant tags in a **dispose**.

6.6.5.4 The **pack** procedure accepts a normal array where **packed** is expected, and a **packed** array where a normal array is expected.

In **pack**, Domain Pascal does not detect an uninitialized component in the unpacked array. Similarly, in **unpack**, Domain Pascal does not detect an uninitialized component in the packed array.

6.6.6.2 On some workstations, the **sqr** function does not detect overflow.

6.6.6.3 On some workstations, **trunc** and **round** do not detect overflow.

6.6.6.4 **Succ**, **pred**, and **chr** do not detect overflow.

6.7.2.2 Domain Pascal does not detect an overflow or underflow on integer arithmetic. Also, it allows you to supply a negative value for **j** in an expression like the following:

```
i mod j
```

6.7.2.4 Domain Pascal does not detect operations on overlapping sets with incompatible elements.

6.8.1 Domain Pascal permits jumps between branches of a **case** statement and from one structured statement into the middle of another.

6.8.3.5 Domain Pascal does not detect the lack of a **case** statement constant corresponding to a run-time **case** value.

6.8.3.9 Domain Pascal does not detect an underflow of an assignment from **pred** to a **for** statement index variable. Also, it does not issue an error if there is an overflow in the final value of a **for** statement index variable.

Domain Pascal does not detect the possibility that an inner block will change the value of a **for** statement's index variable. Also, Domain Pascal allows a non-local variable, a formal parameter, or a value parameter to be used as a **for** statement's index variable. You also can use a program level global variable as the index variable for a **for** statement that resides in an inner block.

6.9.1 Domain Pascal does not detect an overflow of a subrange boundary for a **read** statement.

6.9.3.1 You may supply a nonpositive field width or a nonpositive fractional-digits field width to a **write** or **writeln**.

6.10 You don't have to declare **input** and **output** in a **program** heading to use them in the program. You can, however, repeat parameters in a **program** heading such as

**program** *testing (output, output);*

or redeclare program parameters as some type other than a file. Also, you don't have to declare program parameters in the var declaration part of your program.

# Appendix E
# Systems Programming Routines

Domain Pascal includes several routines designed specifically for systems programmers' use. Systems programmers are those who need to do very low-level work in their programs and who need direct access to specific registers and bits within those registers. They frequently write some programs in Pascal and some in assembly language.

If you are a systems programmer, you might use these routines when writing device drivers, or when doing other low-level manipulations of the hardware status register.

## E.1 Overview

Table E-1 briefly describes the available systems programming routines, all of which are extensions to standard Pascal. A more complete explanation follows the table.

*Table E-1. Systems Programming Routines*

| Routine | Action |
|---|---|
| **disable** | Turns off the interrupt enable in the hardware status register. |
| **enable** | Turns on the interrupt enable in the hardware status register. |
| **set_sr** | Saves the current value of the hardware status register and then inserts a new value. |

## E.2 Restrictions for Use

All the routines described in this appendix generate privileged instructions and may only be executed from supervisor mode. If you try to run a program using one of these routines while in user mode, you get a privilege-violation error.

Disable Turns off the interrupt enable in the hardware status register. (Extension)

**FORMAT**

**disable** {**disable** is a procedure.}

**ARGUMENT**

**Disable** takes no arguments.

**DESCRIPTION**

**Disable** is a built-in procedure for systems programmers' use. It turns off the interrupt enable in the hardware status register and should be used with its complementary procedure **enable**.

By turning off the interrupt enable, **disable** allows you to prevent an interrupt from coming in while the program is in a critical section. After the critical section finishes, you should use **enable** to turn the interrupt enable back on.

The **disable–enable** pair look like this in code:

```
disable;

    { Critical section.                                  }
    { No interrupts allowed while this section is executing. }

 enable;
```

If you mistakenly use only **disable**, your program will essentially grind to a halt since no interrupt signals will be able to get to it. You should only use the **disable–enable** pair around very small sections of code.

## **Enable** Turns on the interrupt enable in the hardware status register. (Extension)

### FORMAT

**enable** {**enable** is a procedure.}

### ARGUMENT

**Enable** takes no arguments.

### DESCRIPTION

**Enable** is a built-in procedure for systems programmers' use. It turns on the interrupt enable in the hardware status register and usually is used with its complementary procedure **disable.**

By turning on the interrupt enable, **enable** allows your program to receive interrupts. Usually, **disable** will have been used to prevent the reception of interrupts during a critical section of code. After the critical section finishes, **enable** lets the interrupts flow.

The **disable-enable** pair look like this in code:

```
disable;

    { Critical section.                              }
    { No interrupts allowed while this section is executing. }

 enable;
```

Because the interrupt enable is turned on by default, there is no effect if you mistakenly use only **enable** in your program.

**Set_sr** Saves the current value of the hardware status register and then inserts a new one. (Extension)

## FORMAT

*oldsr* := **set_sr**(*newsr*); {**set_sr** is a function.}

## ARGUMENT

| | |
|---|---|
| *oldsr* | The old value of the hardware status register. |
| *newsr* | The new value of the hardware status register. |

## DESCRIPTION

**Set_sr** is a built-in function for systems programmers' use. It reads the hardware status register (SR) and replaces its current value with *newsr*. The original value then is assigned to *oldsr*. This translates to assembly language code something like this:

```
move.w    SR,d0
move.w    newsr,SR
move.w    d0,oldsr
```

The function eliminates six instructions in a time-critical path.

# Appendix F

# 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 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 be different from (though along the lines of) 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

## F.1 Instruction Emulation

The 68040 has an on–chip 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, you should 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.

## F.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 Section 6.4.9.) Applications that may contain emulated instructions include

- FORTRAN applications
- Pascal applications
- C applications that are compiled with the **–D_BUILTINS** option (**/bin/cc**) or the **–def _BUILTINS** option (**/com/cc**) or that include the file **<apollo/builtins.h>**

The emulated instructions correspond to the following arithmetic intrinsic functions listed in Table F–1.

Table F-1. Emulated Intrinsic Functions

| FORTRAN | C | Pascal |
|---|---|---|
| SIN, DSIN | sin() | sin |
| COS, DCOS | cos() | cos |
| TAN, DTAN | tan() | -- |
| ATAN, DATAN | atan() | arctan |
| EXP, DEXP | exp() | exp |
| ALOG, DLOG | log() | ln |
| ALOG10, DLOG10 | log10() | -- |
| AINT, DINT | -- | -- |

If an application does not use any of these intrinsics, then 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. However, use **-cpu any** only if your code must run on all existing Domain 680x0-based workstations because this argument imposes a severe performance penalty, usually a greater penalty than that caused by instruction emulation.

Intrinsic functions other than those listed in Table F-1 (such as **ASIN**, **ACOS**, and hyperbolic functions) 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.

## F.3 How Instruction Emulation Affects Performance

As a rule of thumb, the 68040 will 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.

### F.3.1 Changing from –cpu 3000 (–cpu mathchip) to –cpu mathlib or –cpu mathlib_sr10

If your application makes intensive use of **SIN**, **COS**, **TAN**, **ATAN**, **EXP**, or **LOG**, and is currently compiled with **–cpu 3000**, then 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.

### F.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, then 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.

### F.3.3 Changing from –cpu any to –cpu mathchip (–cpu 3000)

If your application makes intensive use of **SIN**, **COS**, **TAN**, **ATAN**, **EXP**, or **LOG**, and is currently compiled with **–cpu any**, then 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.

## F.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

### F.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. You should 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, then you should recompile if you think your application performs much instruction emulation on the 68040.

## F.4.2 If You Recompile, Which -cpu Argument Should You Use?

If your application needs to run optimally only on 68040-based systems, then 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, then 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, then 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, then 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 F-1 shows how to decide which argument is most suited to your application.

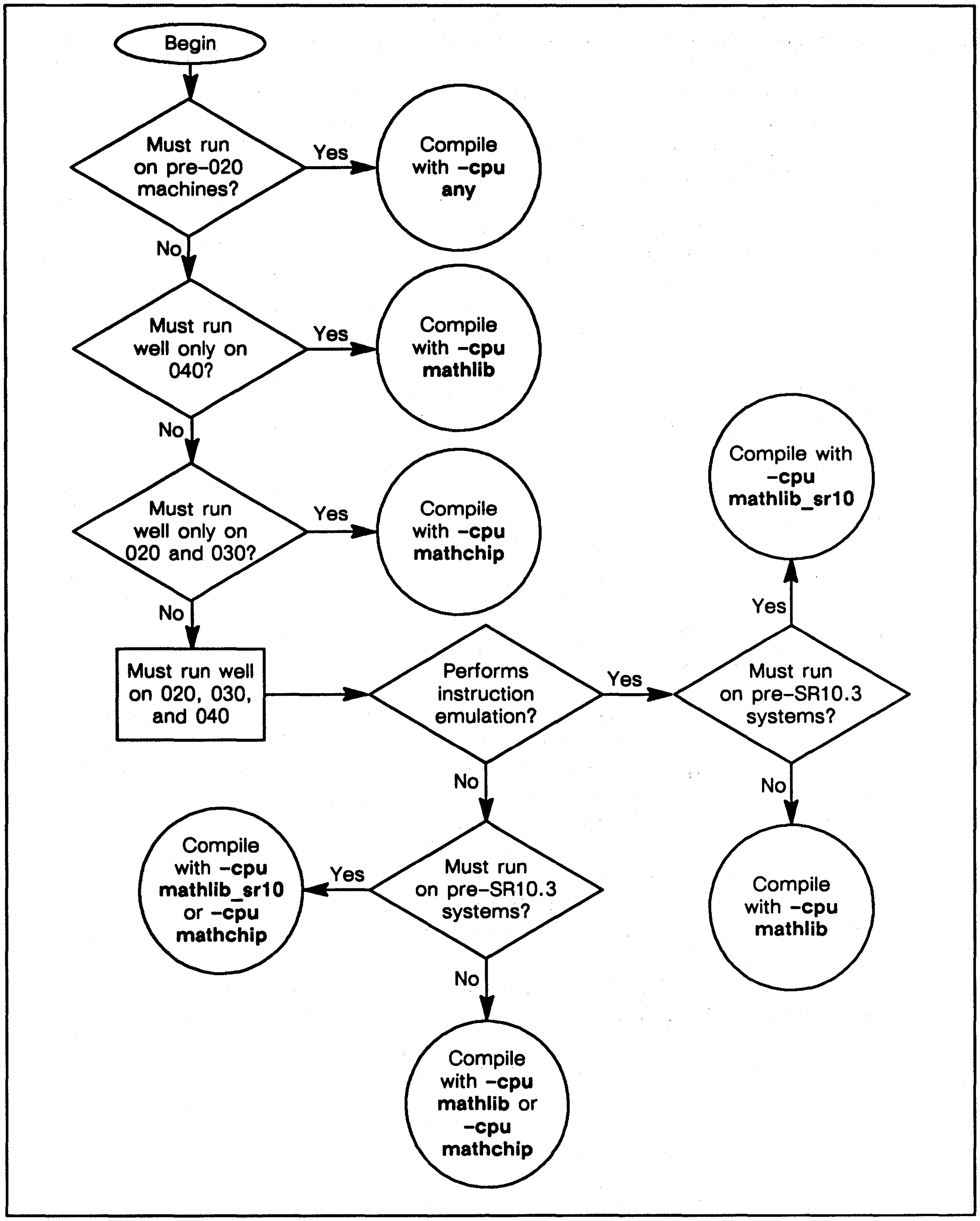
Begin
Must run on pre-020 machines?
Yes
Compile with -cpu any
No
Must run well only on 040?
Yes
Compile with -cpu mathlib
No
Must run well only on 020 and 030?
Yes
Compile with -cpu mathchip
No
Must run well on 020, 030, and 040
Performs instruction emulation?
Yes
Must run on pre-SR10.3 systems?
Yes
Compile with -cpu mathlib_sr10
No
Compile with -cpu mathlib
No
Must run on pre-SR10.3 systems?
Yes
Compile with -cpu mathlib_sr10 or -cpu mathchip
No
Compile with -cpu mathlib or -cpu mathchip

*Figure F-1. Which* **-cpu** *Argument Is Best for Your Application?*

## F.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. Therefore, 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

## 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 Pascal Language Reference*
Order No.: 000792-A01

### User Profile

Your Name ______________________ Title ______________________

Company ______________________________________________

Address ______________________________________________

Telephone number (____) ______________

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 Pascal Language Reference*
Order No.: 000792-A01

## User Profile

Your Name ______________________ Title ______________________

Company ______________________

Address ______________________

Telephone number (____) ______________

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

HEWLETT
PACKARD

Order Number 000792-A01

*000792-A01*