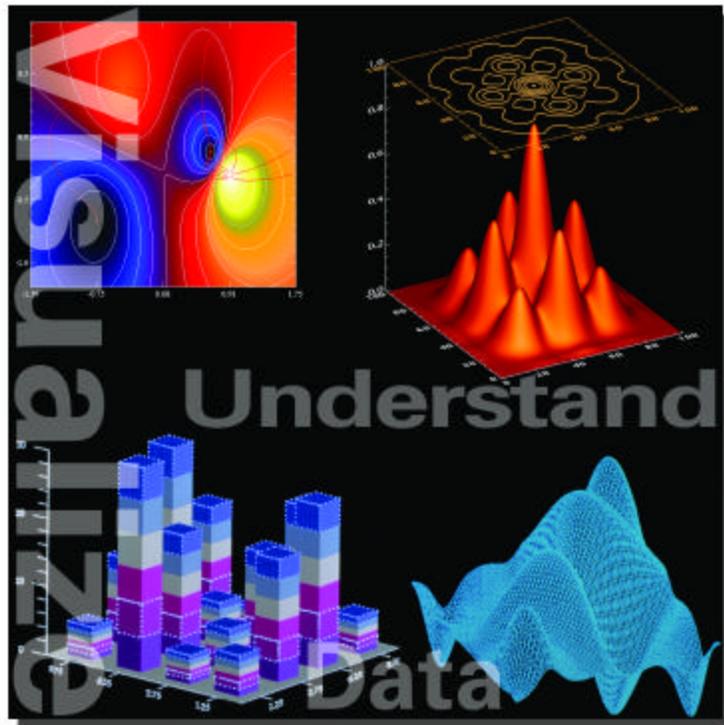




P V - W A V E 7 . 5[®]



P r o g r a m m e r ' s G u i d e

HELPING CUSTOMERS **SOLVE** COMPLEX PROBLEMS

Visual Numerics, Inc.

Visual Numerics, Inc.
2500 Wilcrest Drive
Suite 200
Houston, Texas 77042-2579
United States of America
713-784-3131
800-222-4675
(FAX) 713-781-9260
<http://www.vni.com>
e-mail: info@boulder.vni.com

Visual Numerics, Inc.
7/F, #510, Sect. 5
Chung Hsiao E. Rd.
Taipei, Taiwan 110 ROC
+886-2-727-2255
(FAX) +886-2-727-6798
e-mail: info@vni.com.tw

Visual Numerics S.A. de C.V.
Cerrada de Berna 3, Tercer Piso
Col. Juarez
Mexico, D.F. C.P. 06600
Mexico

Visual Numerics, Inc. (France) S.A.R.L.
Tour Europe
33 place des Corolles
Cedex 07
92049 PARIS LA DEFENSE
FRANCE
+33-1-46-93-94-20
(FAX) +33-1-46-93-94-39
e-mail: info@vni-paris.fr

Visual Numerics International GmbH
Zettachring 10
D-70567 Stuttgart
GERMANY
+49-711-13287-0
(FAX) +49-711-13287-99
e-mail: info@visual-numerics.de

Visual Numerics, Inc., Korea
Rm. 801, Hanshin Bldg.
136-1, Mapo-dong, Mapo-gu
Seoul 121-050
Korea

Visual Numerics International, Ltd.
Suite 1
Centennial Court
East Hampstead Road
Bracknell, Berkshire
RG 12 1 YQ
UNITED KINGDOM
+01-344-458-700
(FAX) +01-344-458-748
e-mail: info@vniuk.co.uk

Visual Numerics Japan, Inc.
Gobancho Hikari Building, 4th Floor
14 Gobancho
Chiyoda-Ku, Tokyo, 102
JAPAN
+81-3-5211-7760
(FAX) +81-3-5211-7769
e-mail: vda-spirt@vnij.co.jp

© 1990-2001 by Visual Numerics, Inc. An unpublished work. All rights reserved. Printed in the USA.

Information contained in this documentation is subject to change without notice.

IMSL, PV-WAVE, Visual Numerics and PV-WAVE Advantage are either trademarks or registered trademarks of Visual Numerics, Inc. in the United States and other countries.

The following are trademarks or registered trademarks of their respective owners: Microsoft, Windows, Windows 95, Windows NT, Fortran PowerStation, Excel, Microsoft Access, FoxPro, Visual C, Visual C++ — Microsoft Corporation; Motif — The Open Systems Foundation, Inc.; PostScript — Adobe Systems, Inc.; UNIX — X/Open Company, Limited; X Window System, X11 — Massachusetts Institute of Technology; RISC System/6000 and IBM — International Business Machines Corporation; Java, Sun — Sun Microsystems, Inc.; HPGL and PCL — Hewlett Packard Corporation; DEC, VAX, VMS, OpenVMS — Compaq Computer Corporation; Tektronix 4510 Rasterizer — Tektronix, Inc.; IRIX, TIFF — Silicon Graphics, Inc.; ORACLE — Oracle Corporation; SPARCstation — SPARC International, licensed exclusively to Sun Microsystems, Inc.; SYBASE — Sybase, Inc.; HyperHelp — Bristol Technology, Inc.; dBase — Borland International, Inc.; MIFF — E.I. du Pont de Nemours and Company; JPEG — Independent JPEG Group; PNG — Aladdin Enterprises; XWD — X Consortium. Other product names and companies mentioned herein may be the trademarks of their respective owners.

IMPORTANT NOTICE: Use of this document is subject to the terms and conditions of a Visual Numerics Software License Agreement, including, without limitation, the Limited Warranty and Limitation of Liability. If you do not accept the terms of the license agreement, you may not use this documentation and should promptly return the product for a full refund. Do not make illegal copies of this documentation. No part of this documentation may be stored in a retrieval system, reproduced or transmitted in any form or by any means without the express written consent of Visual Numerics, unless expressly permitted by applicable law.

Table of Contents

Preface ix

What's in this Manual [ix](#)

Conventions Used in this Manual [xi](#)

Technical Support [xii](#)

Chapter 1: PV-WAVE Programming 1

Where to Find Libraries of PV-WAVE Programs [1](#)

Creating and Running Programs [3](#)

Using PV-WAVE in Runtime Mode [12](#)

Startup Flags [17](#)

Chapter 2: Constants and Variables 19

Constants [19](#)

Variables [24](#)

Chapter 3: Expressions and Operators 29

Operator Precedence [30](#)

Type and Structure of Expressions [31](#)

Structure of Expressions [35](#)

PV-WAVE Operators [37](#)

Chapter 4: Statement Types 47

Components of Statements [47](#)

Assignment Statement [48](#)

Blocks of Statements [53](#)

CASE Statement [55](#)

Common Block Definition Statement	56
FOR Statement	57
Function Declaration Statement	60
Function Definition Statement	60
GOTO Statement	62
IF Statement	62
Procedure Call Statement	64
Procedure Definition Statement	66
REPEAT Statement	67
WHILE Statement	68

Chapter 5: Using Subscripts with Arrays 71

Syntax	71
“Extra” Dimensions	73
Subscript Ranges	74
Structure of Subarrays	76
Arrays as Subscripts to Other Arrays	78
Combining Array Subscripts with Others	79
Storing Elements with Array Subscripts	81
Memory Order	83
Matrices	83

Chapter 6: Working with Structures 89

Introduction to Structures	89
Defining and Deleting Structures	90
Structure References	94
Creating Arrays of Structures	99
Structure Input and Output	100
Advanced Structure Usage	103

Working with Lists and Associative Arrays 104

Chapter 7: Working with Text 113

- Example String Array 113
- Basic String Operations 114
- Concatenating Strings 114
- String Formatting 115
- Converting Strings to Upper or Lower Case 117
- Removing White Space from Strings 118
- Determining the Length of Strings 119
- Manipulating Substrings 120
- Using Non-string and Non-scalar Arguments 122
- Using Regular Expressions 123

Chapter 8: Working with Data Files 131

- Simple Examples of Input and Output 131
- Opening and Closing Files 133
- Logical Unit Numbers (LUNs) 136
- How is the Data File Organized? 139
- Types of Input and Output 144
- Free Format Input and Output 151
- Explicitly Formatted Input and Output 155
- Input and Output of Binary Data 174
- External Data Representation (XDR) Files 188
- Associated Variable Input and Output 194
- Miscellaneous File Management Tasks 199
- UNIX-Specific Information 203
- OpenVMS-Specific Information 204
- Windows-Specific Information 210

Chapter 9: Writing Procedures and Functions 217

- Procedure and Function Parameters 218
- Compiling Procedures and Functions 222
- System Limits and the Compiler 224
- Using the ..LOCALS Compiler Directive 226
- Parameter Passing Mechanism 228
- Procedure or Function Calling Mechanism 229
- Error Handling in Procedures 230
- The Users' Library 231
- OpenVMS Procedure Libraries 233
- Creating OpenVMS Procedure Libraries 234

Chapter 10: Programming with PV-WAVE 237

- Description of Error Handling Routines 237
- Error Signaling 241
- Detection of Math Errors 243
- Hardware-dependent Math Error Handling 247
- Checking for Parameters 248
- Using Program Control Routines 252

Chapter 11: Tips for Efficient Programming 255

- Increasing Program Speed 255
- Avoid IF Statements for Faster Operation 256
- Use Array Operations Whenever Possible 257
- Use System Routines for Common Operations 259
- Use Constants of the Correct Type 259
- Remove Invariant Expressions from Loops 260
- Access Large Arrays by Memory Order 260
- Be Aware of Virtual Memory 261

Running Out of Virtual Memory? 262

Array Operations are Rewarded 266

Chapter 12: Getting Session Information 267

Using the INFO Procedure 267

Chapter 13: Using the PV-WAVE Debugger 275

The Main PV-WAVE Debugger Window 276

Using the Debugger's Online Help System 277

Starting the Debugger 277

Saving Your Work and Stopping the Debugger 278

Loading Files into the Debugger 278

Running an Application 280

Detecting Execution Errors 280

Editing the Source File 280

Setting Breakpoints 281

Controlling Program Execution 283

Examining Variables 283

Obtaining Session Information 286

Customizing the Debugger 286

Chapter 14: Creating an OPI Option 287

Introduction 287

Managing Options 288

The Developer Environment 289

Creating An Option 295

Keyword Processing 302

License Management 303

Adding an Option to the PV-WAVE Search Path 303

Variable Handling Examples	304
Option Programming Interface Language Bindings	307
OPI Function Definitions for PV-WAVE Variables	311
wave_execute	313
wave_compile	314
wave_interp	316
wave_free_WCH	317
wave_assign_num	317
wave_assign_string	317
wave_assign_struct	317
wave_get_WVH	321
wave_get_unWVH	322
wave_free_WVH	323
wvh_name	324
wvh_type	326
wvh_ndims	327
wvh_nelems	328
wvh_dimensions	329
wvh_sizeofdata	330
wave_type_sizeof	330
wvh_is_scalar	332
wvh_is_constant	333
wvh_dataptr	334
wave_wsdh_from_wvh	335
wave_wsdh_from_name	336
wave_free_WSDH	337
wsdh_name	338
wsdh_ntags	339

wsdh_tagname 339
wsdh_sizeofdata 341
wsdh_offset 341
wsdh_element 342
opi_malloc, opi_free, opi_realloc, opi_calloc 344
C Language Error Handling 345
wave_error 346
wave_onerror 348
wave_is_onerror 349
wave_onerror_continue 349
wave_is_onerror_continue 350

Appendix A: FORTRAN and C Format Strings **A-1**

What Are Format Strings? A-1
When to Use Format Strings A-2
What to Do if the Data is Formatted Incorrectly A-2
Example — Using C and FORTRAN Format Strings A-2
Using Format Reversion A-4
Group Repeat Specifications A-6
FORTRAN Formats for Data Import and Export A-7
FORTRAN Format Code Descriptions A-9
C Format Strings for Data Import and Export A-19

Appendix B: Modifying Your Environment **B-1**

Modifying Your PV-WAVE Environment
(UNIX/OpenVMS Only) B-1
Modifying Your PV-WAVE Environment (Windows) B-9

Programmer's Guide Index **21**

Preface

This manual describes the PV-WAVE programming language in detail. PV-WAVE uses an intuitive fourth-generation language (4GL) that analyzes and displays data as you enter commands. With it you can perform complex analysis, visualization, and application development quickly and interactively.

What's in this Manual

This manual covers the following topics:

- **Chapter 1, *PV-WAVE Programming*** — Provides an overview of the basic elements of the command language and a brief discussion of its high-level features.
- **Chapter 2, *Constants and Variables*** — Introduces the different types and structures of variables, constants, and predefined system variables.
- **Chapter 3, *Expressions and Operators*** — Explains expressions, which are one or more variables or constants combined with operators. Expressions are the basic building blocks of PV-WAVE.
- **Chapter 4, *Statement Types*** — Describes the syntax and semantics of PV-WAVE statements, such as FOR and WHILE loops, CASE statements, and assignments.
- **Chapter 5, *Using Subscripts with Arrays*** — Describes how to use the wide variety of subscript types, ranges, and arrays available with PV-WAVE.
- **Chapter 6, *Working with Structures*** — Explains how to define and use structures.

- **Chapter 7, *Working with Text*** — Discusses the system routines used for string processing and gives examples.
- **Chapter 8, *Working with Data Files*** — Describes how to read and write formatted and unformatted data files using the traditional routines such as WRITEU, WRITEF, READU, and READF. In addition, a collection of new routines, the DC_READ_* and DC_WRITE_* functions, provide a greatly simplified alternative to other methods of reading and writing data. These routines are discussed in this chapter as well.
- **Chapter 9, *Writing Procedures and Functions*** — Explains how to write your own PV-WAVE functions and procedures. Topics such as error handling and parameter passing are discussed.
- **Chapter 10, *Programming with PV-WAVE*** — Discusses routines that are useful when programming PV-WAVE applications.
- **Chapter 11, *Tips for Efficient Programming*** — Explains ways to optimize programs written in the PV-WAVE language.
- **Chapter 12, *Getting Session Information*** — Describes how to get information about the current PV-WAVE session.
- **Chapter 13, *Using the PV-WAVE Debugger*** — Explains how to use the PV-WAVE Debugger, a development environment for creating, testing, and maintaining VDA applications written in PV-WAVE.
- **Chapter 14, *Creating an OPI Option*** — Discusses how to use the Option Programming Interface (OPI) to create optional modules that can be loaded explicitly by any PV-WAVE user.
- **Appendix A, *FORTRAN and C Format Strings*** — Discusses the format strings that you can use to transfer data to and from PV-WAVE.
- **Appendix B, *Modifying Your Environment*** — Discusses methods for modifying your PV-WAVE environment for UNIX, OpenVMS, and Windows systems.
- ***Programmer's Guide Index*** — A subject index with hypertext links to information in this manual.

Conventions Used in this Manual

You will find the following conventions used throughout this manual:

- Code examples appear in this typeface. For example:

```
PLOT, temp, s02, Title = 'Air Quality'
```

- Code comments are preceded by a semicolon and are shown in this typeface, immediately below the commands they describe. For example:

```
PLOT, temp, s02, Title = 'Air Quality'  
; This command plots air temperature data vs. sulphur  
; dioxide concentration.
```

- Variables are shown in lowercase italics (*myvar*), function and procedure names are shown in uppercase (XYOUTS), keywords are shown in mixed case italic (*XTitle*), and system variables are shown in regular mixed case type (!Version). For better readability, all GUI development routines are shown in mixed case (WwMainMenu).
- A \$ at the end of a line of PV-WAVE code indicates that the current statement is continued on the following line. By convention, use of the continuation character (\$) in this document reflects its syntactically correct use in PV-WAVE. This means, for instance, that *strings* are never split onto two lines without the addition of the string concatenation operator (+). For example, the following lines would produce an error if entered literally in PV-WAVE.

```
WAVE> PLOT, x, y, Title = 'Average $  
Air Temperatures by Two-Hour Periods'  
; Note that the string is split onto two lines; an error message is  
; displayed if you enter a string this way.
```

The correct way to enter these lines is:

```
WAVE> PLOT, x, y, Title = 'Average ' + $  
'Air Temperatures by Two-Hour Periods'  
; This is the correct way to split a string onto  
; two command lines.
```

- Reserved words, such as FOR, IF, CASE, are always shown in uppercase.

Technical Support

If you have problems installing, unlocking, or running your software, contact Visual Numerics Technical Support by calling:

Office Location	Phone Number
Corporate Headquarters Houston, Texas	713-784-3131
Boulder, Colorado	303-939-8920
France	+33-1-46-93-94-20
Germany	+49-711-13287-0
Japan	+81-3-5211-7760
Korea	+82-2-3273-2633
Mexico	+52-5-514-9730
Taiwan	+886-2-727-2255
United Kingdom	+44-1-344-458-700

Users outside the U.S., France, Germany, Japan, Korea, Mexico, Taiwan, and the U.K. can contact their local agents.

Please be prepared to provide the following information when you call for consultation during Visual Numerics business hours:

- Your license number, a six-digit number that can be found on the packing slip accompanying this order. (If you are evaluating the software, just mention that you are from an evaluation site.)
- The name and version number of the product. For example, PV-WAVE 7.0.
- The type of system on which the software is being run. For example, SPARCstation, IBM RS/6000, HP 9000 Series 700.
- The operating system and version number. For example, HP-UX 10.2 or IRIX 6.5.
- A detailed description of the problem.

FAX and E-mail Inquiries

Contact Visual Numerics Technical Support staff by sending a FAX to:

Office Location	FAX Number
Corporate Headquarters	713-781-9260
Boulder, Colorado	303-245-5301
France	+33-1-46-93-94-39
Germany	+49-711-13287-99
Japan	+81-3-5211-7769
Korea	+82-2-3273-2634
Mexico	+52-5-514-4873
Taiwan	+886-2-727-6798
United Kingdom	+44-1-344-458-748

or by sending E-mail to:

Office Location	E-mail Address
Boulder, Colorado	support@boulder.vni.com
France	support@vni-paris.fr
Germany	support@visual-numerics.de
Japan	vda-sprt@vnij.co.jp
Korea	support@vni.co.kr
Taiwan	support@vni.com.tw
United Kingdom	support@vniuk.co.uk

Electronic Services

Service	Address
General e-mail	info@boulder.vni.com
Support e-mail	support@boulder.vni.com
World Wide Web	http://www.vni.com
Anonymous FTP	ftp.boulder.vni.com
FTP Using URL	ftp://ftp.boulder.vni.com/VNI/
PV-WAVE Mailing List:	Majordomo@boulder.vni.com
To subscribe include:	subscribe pv-wave YourEmailAddress
To post messages	pv-wave@boulder.vni.com

PV-WAVE Programming

PV-WAVE's programming environment is versatile and its syntax is easy to learn. PV-WAVE allows you to concentrate on specialized applications rather than on system design and routine program development, therefore saving valuable time. Instant display of intermediate and final results, either in the form of graphs or images, allows you to deal with the unexpected, to better interpret complex data, and to create and debug programs in an efficient manner.

Furthermore, PV-WAVE provides several ways for you to develop applications with a “user-friendly” graphical user interface (GUI).

TIP If you are just getting started with PV-WAVE, we recommend that you work through the lessons in the *PV-WAVE Tutorial*. The *Tutorial* is designed to teach you the basics of PV-WAVE and prepare you to use PV-WAVE productively.

This chapter discusses methods for creating and running PV-WAVE programs.

Where to Find Libraries of PV-WAVE Programs

Several libraries of procedures and functions written in the PV-WAVE language are available for your use. You can use the routines in these libraries as they are, or you can copy or modify the code for use in your own applications.

The following diagram shows the directory structure for PV-WAVE function and procedure libraries:

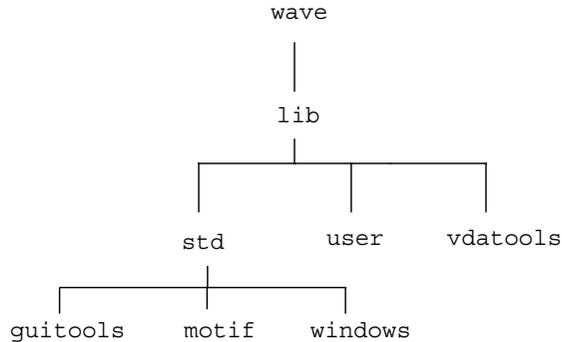


Figure 1-1 The PV-WAVE function and procedure library directory structure.

The rest of this section describes the contents of each of these libraries.

PV-WAVE Function and Procedure Libraries

Library	Description
std	Routines in the Standard library are fully tested and documented by Visual Numerics.
guitools	Contains an assortment of graphical user interface (GUI) routines. These routines perform color table modifications, display surface views, display and manipulate iso-surfaces, and provide access to a variety of other functions. The GUI routines all begin with Wg (e.g., WgSurfaceTool) and are described in the PV-WAVE Reference.
motif windows	Contains the standard WAVE Widgets routines for Motif and Microsoft Windows. For information on WAVE Widgets, see the PV-WAVE Application Developer's Guide.
user	Users' library routines written and submitted by PV-WAVE users. This library contains such entries as routines for compressing images, making pie charts, creating 2D/3D bar graphs, and displaying 3D scatterplots. The routines in the Users' library are not documented in the PV-WAVE documentation set. For information on a routine in the user library, read the header of the .pro file for that routine. For information on adding routines to the Users' Library, see Submitting Programs to the Users' Library on page 232 .
vdatools	Contains routines used to build VDA Tools. These include the VDA Tools Manager (Tm) routines, VDA Utilities (Wo), and a set of prewritten VDA Tools (Wz).

Creating Your Own Library

You can also create your own routines and add them to the library, or create your own library. In fact, creating your own library is recommended so your routines aren't lost when you upgrade to a new version of PV-WAVE.

TIP Be sure to include the new directory in the PV-WAVE search path. You can do this by modifying the !Path system variable.

Creating and Running Programs

You can create program files using a text editor that can save a file in ASCII format, and then execute these programs within PV-WAVE. This method is usually how programs are created because these programs can be saved in files for future use. The types of programs you can create include:

- interactive command line programs
- functions and procedures
- main programs
- command or batch files
- journal files

NOTE For information on creating applications with a Graphical User Interface (GUI), refer to PV-WAVE Application Developer's Guide.

Creating and Running Programs Interactively at the Command Line

Normally, functions and procedures are created in files so that they can be used in future sessions. However, occasionally you may need to create a short program or function that you do not want to save. The .RUN command provides this option.

Here's an example showing how to create a program interactively using the .RUN command:

```
WAVE> .RUN
- FOR I = 0,3 DO BEGIN
- PRINT, 'SQRT of ', I
- PRINT, ' = ', SQRT(I)
```

```

- ENDFOR
- END
% Compiled module: $MAIN$
SQRT OF 0
= 0.00000
SQRT OF 1
= 1.00000
SQRT OF 2
= 1.41421
SQRT OF 3
= 1.73205

```

The example program calculates and prints out the square root for the numbers 0 through 3.

After typing `.RUN` and pressing <Return>, a dash (–) prompt is displayed indicating that you are in program mode. When you have completed the program, you must enter `END` as the last line and press <Return>. The message `%Compiled module: $MAIN$` that displays indicates that this is a main program.

Two other types of programs, procedures and functions, can also be created from the `WAVE>` prompt using the `.RUN` command. Here’s an example of how to create a function that squares a number:

```

WAVE> .RUN
- FUNCTION SQUARE, NUMBER
- RETURN, NUMBER^2
- END
%Compiled module: SQUARE

```

After you type `END` and press <Return>, the message `%Compiled module: SQUARE`, displays. Now, you can use the `SQUARE` function to calculate the square of a number.

A program created interactively like this cannot be saved for use in later sessions unless you have given it a name, such as `SQUARE`. As long as the program has a name, you can enter the following command:

```
SAVE, /Routine
```

and the routine `SQUARE` will be saved along with any other routines that you had already compiled during that session.

Creating and Running a Function or Procedure

Using an ordinary text editor that can save to an ASCII file, you can create files that define procedures and functions.

NOTE For much more information on writing procedures and functions, see [Chapter 9, *Writing Procedures and Functions*](#), [Chapter 10, *Programming with PV-WAVE*](#), and [Chapter 11, *Tips for Efficient Programming*](#). See also [Chapter 13, *Using the PV-WAVE Debugger*](#).

Function Program Example

For example, here's the program listing for a file named `square.pro` that defines a function to square a number:

```
FUNCTION SQUARE, NUMBER
    RETURN, NUMBER^2
END
```

The file automatically compiles and executes when being called at the `WAVE>` prompt:

```
WAVE> x = SQUARE(24) & PRINT, x
% Compiled module: SQUARE.
    576
```

The file automatically compiles and executes only under the following circumstances:

- if the file is in the `!Path` or current directory
- and*
- the filename is the same as the function or procedure name and has a `.pro` extension.

If the file is not the same name as the function, then you must use the `.RUN` command to compile it. See the section in this manual, [WAVE_PATH: *Setting Up a Search Path \(UNIX, OpenVMS\)*](#) on page [B-3](#) for details about search paths.

TIP The PV-WAVE Debugger has a built-in editor that you can use to develop and run programs. See [Chapter 13, *Using the PV-WAVE Debugger*](#).

Creating and Running Main Programs

A main program is a series of statements that are not preceded by a procedure or function heading (PRO or FUNCTION) and is compiled as a unit. Main programs can also be created interactively as indicated in the next section. Since there is no heading, it cannot be called from other routines, and cannot be passed arguments. When PV-WAVE encounters the END statement in a main program as the result of a .RUN executive command, it compiles it into the special program named \$MAIN\$ and immediately executes it as a unit. Afterwards, it can be executed again with the .GO executive command.

Main Program Example

For example, a main program file named `testfile` might contain the following statements:

```
FOR I = 3,5 DO BEGIN
  PRINT, 'Square of ', I, ' = ', I^2
  PRINT, 'Square root of ', I, ' = ', SQRT(I)
ENDFOR
END
```

To compile and run this main program file named `testfile`, enter the following at the WAVE> prompt:

```
WAVE> .RUN testfile
```

The results are:

```
Square of 3 = 9
Square root of 3 = 1.73205
Square of 4 = 16
Square root of 4 = 2.00000
Square of 5 = 25
Square root of 5 = 2.23607
```

Main Program Compared to Function or Procedure

A main program is a series of statements that is not preceded by a procedure or function heading (PRO or FUNCTION) and is compiled as a unit. The Editor window in the Windows version of PV-WAVE is ideal for creating programs like this.

Since a main program has no heading, it cannot be called from other routines, and it cannot be passed arguments. But it can contain commands that communicate with other applications – either off-the-shelf commercially-available software, or

custom applications that you or a coworker have written in C. Furthermore, it can be stored for later use, either by you or by someone else.

For example, suppose that earlier you had saved a PV-WAVE program on a floppy disk in a file named `ditto.pro`. Because the program did not include a `FUNCTION` or `PRO` command, it is considered to be a main program. If the disk drive on your computer is assigned to the `A:` partition, you can run this program by entering the following command at the `WAVE>` prompt:

```
WAVE> .RUN A:\ditto
```

If the file had some other filename extension besides `.pro`, you would have had to supply that, as well (e.g., `ditto.txt` or `ditto.pgm`).

UNIX and OpenVMS USERS For more information about how to enable PV-WAVE to communicate with other applications, refer to the PV-WAVE Application Developer's Guide.

Windows USERS For more information about how to enable PV-WAVE to communicate with other applications, refer to the PV-WAVE Application Developer's Guide.

Main Program Compared to Command File

The differences between a main program and a command file are:

- Main programs must have an `END` statement, they must be executed with the `.RUN` command (or the **File=>Run** option), and they are executed as a unit.
- Command files do not have an `END` statement, are executed by typing `@filename`, and are executed one line at a time.

For more information about command files, and how they can be used during your PV-WAVE session, refer to [Running a Command \(Batch\) File on page 8](#).

Creating and Running a Command (Batch) File

Each line of a command file (also called a batch file) is interpreted exactly as if it had been entered from the `WAVE>` prompt. In the command file mode, PV-WAVE compiles and executes each statement before reading the next statement. This is different than the interpretation of programs, procedures, and functions compiled using `.RNEW` or `.RUN` (explained previously), in which all statements in a program are compiled as a single unit and then executed. Labels, as described in [Statement](#)

[Labels on page 48](#), are not allowed in the command file mode because each statement is compiled independently.

Multi-line statements must be continued on the next line using the \$ continuation character, because in interactive mode PV-WAVE terminates every statement not ending with \$ by an END statement. A common mistake is to include a block of commands in a FOR loop inside a command file:

```
FOR I = 1,10 DO BEGIN
  PRINT, I, ' square root = ', SQRT(I)
  PRINT, I, ' square = ', I^2
ENDFOR
```

In command file mode (this is not the case for functions and procedures), PV-WAVE compiles and executes each line separately, causing syntax errors in the example above because no matching ENDFOR is found on the same line as the BEGIN when the line is compiled. The above example can be made to work by inserting an ampersand between each statement in the block of statements and by terminating each line (except the last) with a \$:

```
FOR I = 1,10 DO BEGIN & $
  PRINT, I, ' square root = ', SQRT(I) & $
  PRINT, I, ' square = ', I^2 & $
ENDFOR
```

NOTE Both the ampersand and the dollar sign are required on every line of a command file except the last line. For example, with just an ampersand at the end of each line, the sample program does not run properly because each line is compiled as a separate entity. Hence, a syntax error results when the ENDFOR statement is compiled because it is seen as a single statement that is not connected to a FOR statement. With the dollar sign at the end of each line, no compilation occurs until the ENDFOR statement. For more information on the dollar sign and ampersand characters, see *Special Characters* in the PV-WAVE Reference.

Running a Command (Batch) File

A command file is simply a file that contains PV-WAVE executive commands and statements. Command files are useful for executing commands and procedures that are commonly used. The commands and statements in the command file are executed as if they were entered from the keyboard at the WAVE> prompt.

There are three ways that you can run a command file:

- You can enter the command file mode (run a command file) by entering the following at the WAVE> prompt:

WAVE> @filename

- From a UNIX or OpenVMS prompt, you can enter the filename in conjunction with the wave command:

wave filename

- If you have created a startup file that has been defined with the environment variable WAVE_STARTUP, then you can enter the wave command at the UNIX or OpenVMS prompt to run the command file. See [WAVE_STARTUP: Using a Startup Command File](#) on page B-5 for details.

NOTE You cannot execute a command file and a PV-WAVE command on the same command line. For instance, if you were to type the following commands at the WAVE> prompt, the command file will execute, but the PRINT command will not.

```
WAVE> @myfile & PRINT, a
```

PV-WAVE reads commands from the specified file until the end is reached. You can nest command files by prefacing the name of the new command file with the @ character. The current directory and then all directories in the !Path system variable are searched for the file. See [WAVE_PATH: Setting Up a Search Path \(UNIX, OpenVMS\)](#) on page B-3.

OpenVMS USERS A semicolon (;) after the @ character can be interpreted as an OpenVMS filename in an OpenVMS environment. Surround the semicolon within blank spaces or tabs to create a comment after the @ sign.

Command file execution may be terminated before the end of the file with control returning to the interactive mode by calling the STOP procedure from within the command file. Calling the EXIT procedure from the command file has the usual effect of terminating PV-WAVE.

Command File Example

An example of a command line that initiates command file execution is:

```
WAVE> @myfile
      ; Use myfile for statement and command input. If not in the current
      ; directory, use the search path !Path.
```

Possible contents of myfile are shown below:

```
.RUN PROGA
      ; Run program A.
```

```
.RUN PROG B
    ; Run program B.

PRINT, avalue, bvalue
    ; Print results.

CLOSE, 3
    ; Close file on logical unit 3.
```

The command file should not contain complete program units such as procedures or functions. However, complete program units can be compiled and run by using the `.RUN` and `.RNEW` commands in the command files, as shown in the previous example.

Creating Journal Files

Journaling provides a record of an interactive PV-WAVE session. All text entered at the `WAVE>` prompt is entered directly into the file, and any text entered from the terminal in response to any other input request (such as with the `READ` procedure) is recorded as a comment. The result is a file that contains a complete description of the PV-WAVE session which can be rerun later.

The `JOURNAL` procedure has the form:

```
JOURNAL [, param]
```

where the string parameter *param* is either a filename (if journaling is not currently in progress), or an expression to be written to the file (if journaling is active).

The first call to `JOURNAL` starts the logging process. If no parameter is supplied, a journal file named `wavesave.pro` is created. If a filename is specified in *param*, the session's commands will be written to a file of that name.

UNIX USERS Under UNIX, creating a new journal file causes any existing file with the same name to be lost. This includes the default file `wavesave.pro`. Use a filename parameter with the `JOURNAL` procedure to avoid destroying existing journal files.

Programmatically Controlling the Journal File

When journaling is not in progress, the value of the system variable `!Journal` is 0. When the journal file is opened, the value of this system variable is set to the logical unit number of the journal file that is opened. This fact can be used by routines to check if journaling is active. You can send any arbitrary data to this file using the normal PV-WAVE output routines. In addition, calling `JOURNAL` with a parameter while journaling is in progress results in the parameter being written to the

journal file as if the PRINTF procedure had been used. In other words, the statement:

```
JOURNAL, param
```

is equivalent to:

```
PRINTF, !Journal, param
```

with one exception—the JOURNAL procedure is not logged to the file (only its output) while a PRINTF statement is logged to the file in addition to its output.

Journaling ends when the JOURNAL procedure is called again without an argument, or when you exit PV-WAVE.

TIP The journal file can be used later as a command input file to repeat the session, and it can be edited with any text editor if changes are necessary.

JOURNAL Procedure Example

As an example, consider the following statements:

```
JOURNAL, 'demo.pro'  
    ; Start journaling to file demo.pro  
PRINT, 'Enter a number: '  
READ, Z  
    ; Read the user response into variable Z.  
JOURNAL, '; This was inserted with JOURNAL.'  
    ; Send a comment to the journal file using the JOURNAL procedure.  
PRINTF, !Journal, '; This was inserted ' + $  
    'with PRINTF.'  
    ; Send another comment using PRINTF.  
JOURNAL  
    ; End journaling.
```

If these statements are executed by a user named *bobf* on a Sun workstation named *peanut*, the resulting journal file `demo.pro` will look something like:

```
; SUN WAVE Journal File for bobf@peanut  
; Working directory: /home/bobf/wavedemo  
; Date: Mon Aug 29 19:38:51 1995  
PRINT, 'Enter a number: '  
; Enter a number:  
READ, Z
```

```
; 100
; This was inserted with JOURNAL.
PRINTF, !Journal, '; This was inserted ' +$
'with PRINTF.'
; This was inserted with PRINTF.
```

NOTE The input data to the READ statement is shown as a comment. In addition, the statement to insert the text using JOURNAL does not appear.

Using PV-WAVE in Runtime Mode

PV-WAVE can interpret and execute two kinds of files: source files and compiled files.

- **Source Files** — Functions and procedures saved as regular ASCII files with a `.pro` filename extension. When a function or procedure of this type is called, it is first compiled, then executed by PV-WAVE.
- **Compiled Files** — Functions and procedures that are first compiled in PV-WAVE, then saved with the `COMPILE` procedure. By default, such files are given a `.cpr` filename extension. Because a file of this type is already compiled, it can be executed more quickly than a `.pro` file.

For detailed information on the `COMPILE` procedure, see its description in the PV-WAVE Reference.

This ability to handle both source and compiled files allows PV-WAVE to be run in two different modes:

- **Interactive mode** — The mode normally used for PV-WAVE application development and direct access to the PV-WAVE command line, and, under Windows, to the Home window and command line.
- **Runtime mode** — Allows direct execution of PV-WAVE applications composed of compiled routines that have been saved with the `COMPILE` procedure. The runtime mode is described in the following sections.

Runtime Mode for UNIX and OpenVMS

NOTE All of the interapplication communication methods described in the PV-WAVE Application Developer's Guide are supported in runtime mode *except* the unidirectional communication routines `wavecmd`, `waveinit`, and `waveterm`.

Starting PV-WAVE in Runtime Mode (UNIX/OpenVMS)

NOTE To execute a runtime mode (compiled) application, you must have a runtime license. Without a runtime license for PV-WAVE, you will be unable to start PV-WAVE in runtime mode as described in this section. For information on obtaining a runtime license for PV-WAVE, please contact Visual Numerics.

In runtime mode, you can run a PV-WAVE application directly from the operating system prompt. When the application is finished running, control returns to the operating system level.

The application must first be compiled and saved with the `COMPILE` procedure. For example, if the procedure called `images` is compiled, the command:

```
COMPILE, 'images'
```

saves a file containing the compiled procedure. By default, this file is named `images.cpr`, and it is saved in the current working directory. For detailed information on the `COMPILE` procedure, see its description in the PV-WAVE Reference.

To execute the compiled, saved application named `images.cpr` from the operating system prompt, enter the following command, where `-r` is a flag that specifies runtime mode:

```
wave -r images
```

When the application is finished running, control is returned to the operating system prompt. Note that the `.cpr` extension is not used when invoking the application.

You can set the default mode to “runtime” with the environment variable `WAVE_FEATURE_TYPE` by typing on a UNIX system:

```
setenv WAVE_FEATURE_TYPE RT
```

On a OpenVMS system, enter:

```
DEFINE WAVE_FEATURE_TYPE RT
```

Now, the `-r` flag is not needed, and you can run the application by entering:

```
wave images
```

The read-only system variable `!Feature_Type` allows you to distinguish between runtime mode and normal, interactive mode. This system variable simply reflects the current setting of the `WAVE_FEATURE_TYPE` environment variable (UNIX) or logical (OpenVMS).

More than one saved compiled file can be executed at a time from the operating system prompt. Just separate the application filenames with spaces, as follows:

```
wave file_1 file_2 file_3 ...
```

The Search Path for Compiled Routine Files (UNIX/OpenVMS)

Whenever a user-written procedure or function is called, PV-WAVE searches *first* for saved, compiled files (.cpr files) with the same name as the called routine. If a saved, compiled file is not found, PV-WAVE searches for a source file (.pro file). PV-WAVE searches the current directory and all directories specified in the !Path directory path.

If you place a .pro file in the current working directory that has the same name as a .cpr file further along the directory path, the .cpr file will always be found and executed first. To explicitly execute the .pro file, compile it with the .RUN command or remove the .cpr file from the !Path directory path.

NOTE The compiled (.cpr) file must have the same name as the called routine. If the calling name of an application program is `images`, then the saved, compiled file must be called `images.cpr`.

Developing Runtime Applications (UNIX/OpenVMS)

Applications developed for operation in runtime mode must adhere to the following guidelines:

- Only PV-WAVE routines that are compiled and saved with the `COMPILE` command can be executed in runtime mode.
- The startup file pointed to by the `WAVE_RT_STARTUP` environment variable (UNIX) or logical (OpenVMS) must be compiled and saved with the `COMPILE` command. The startup file must be in a directory pointed to by the `WAVE_PATH` environment variable (UNIX) or logical (OpenVMS). For more information on this startup file, see [WAVE_RT_STARTUP: Using a Startup Procedure in Runtime Mode](#) on page B-6.
- Executive commands `.RUN`, `.RNEW`, `.GO`, `.STEP`, `.SKIP`, `.CON`, and the `STOP` routine are not recognized in runtime mode.
- Breakpoints specified with the `BREAKPOINT` procedure are not recognized in runtime mode.
- Any errors that occur in runtime mode are reported as usual, and control is returned to the operating system prompt.

Runtime Mode for Windows

Starting PV-WAVE in Runtime Mode (Windows)

In the Windows version of PV-WAVE, you can run a PV-WAVE application directly from the DOS prompt in a Windows command shell window. When the application is finished running, control returns to the operating system level.

The application must first be compiled in PV-WAVE and saved with the COMPILE procedure. For example, if the procedure called `images` is compiled in PV-WAVE, the command:

```
COMPILE, 'images'
```

saves a file containing the compiled procedure. By default, this file is named `images.cpr`, and it is saved in the current working directory. For detailed information on the COMPILE procedure, see its description in the PV-WAVE Reference.

To execute the compiled, saved application named `images.cpr` from the operating system prompt, enter the following command, where `-r` is a flag that specifies runtime mode:

```
wave -r images
```

When the application is finished running, control is returned to the operating system prompt. Note that the `.cpr` extension is not used when invoking the application.

You can set the default mode to “runtime” with the environment variable `WAVE_FEATURE_TYPE` by typing:

```
set WAVE_FEATURE_TYPE=RT
```

Now, the `-r` flag is not needed, and you can run the application by entering:

```
wave images
```

The read-only system variable `!Feature_Type` allows you to distinguish between runtime mode and normal, interactive mode. This system variable simply reflects the current setting of the `WAVE_FEATURE_TYPE` environment variable.

More than one saved compiled file can be executed at a time from the operating system prompt. Just separate the application filenames with spaces, as follows:

```
wave file_1 file_2 file_3 ...
```

The Search Path for Compiled Routine Files (Windows)

Whenever a user-written procedure or function is called, PV-WAVE searches *first* for saved, compiled files (.cpr files) with the same name as the called routine. If a saved, compiled file is not found, PV-WAVE searches for a source file (.pro file). PV-WAVE searches the current directory and all directories specified in the !Path directory path.

The compiled (.cpr) file must have the same name as the called routine. If the calling name of an application program is images, then the saved, compiled file must be called images.cpr.

If you place a .pro file in the current working directory that has the same name as a .cpr file further along the directory path, the .cpr file will always be found and executed first. To explicitly execute the .pro file, compile it with the .RUN command or remove the .cpr file from the !Path directory path.

Developing Runtime Applications (Windows)

Applications developed for operation in PV-WAVE's runtime mode must adhere to the following guidelines:

- Only PV-WAVE routines that are compiled and saved with the COMPILE command can be executed in runtime mode.
- The startup file pointed to by the WAVE_RT_STARTUP environment variable must be compiled and saved with the COMPILE command. The startup file must be in a directory pointed to by the WAVE_PATH environment variable. For more information on this startup file, see [WAVE_STARTUP: Using a Startup Command File](#) on page B-5.
- PV-WAVE executive commands .RUN, .RNEW, .GO, .STEP, .SKIP, .CON, and the STOP routine are not recognized in runtime mode.
- PV-WAVE breakpoints specified with the BREAKPOINT procedure are not recognized in runtime mode.

NOTE Any errors that occur in runtime mode are reported as usual, and control is returned to the operating system prompt.

Runtime Mode for Dynamically Loaded Options

Applications developed with the Option Programming Interface (OPI) can be used in runtime mode.

To load an OPI application in runtime mode, you must include the startup call for the option at the beginning of the runtime procedure. For example, the commands `math_init`, `stat_init`, and `sigpro_init` start PV-WAVE:IMSL Mathematics, PV-WAVE:IMSL Statistics, and the PV-WAVE:Signal Processing Toolkit.

Other Ways to Run the Program

Alternatively, you can achieve the same results by saving the commands in a file, and then compiling that file using the `.RUN` command entered at the `WAVE>` prompt:

```
WAVE> .RUN test_06
```

For details about where to store the file and what to name it, refer to [Creating and Running a Function or Procedure on page 5](#).

Although the file is named `test_06` in this example, it is customary to give the file a name that matches the name of the function or procedure it contains. Otherwise, that function or procedure is not as easy to use from other PV-WAVE programs.

NOTE If the program is a main program (not a named function or procedure), this program can be executed over and over again using the `.GO` executive command. This is true whether you process the file with the `.RUN` command or the **File=>Run** command from the Editor window.

`.RUN` and `.GO` are special commands called executive commands. For more details about using executive commands to control programs, see the PV-WAVE Reference.

Startup Flags

PV-WAVE has flags for Windows and UNIX that can be used with the “wave” command for setting initial values for the number of local variables and for the code size of a WAVE session.

OpenVMS USERS There are no such flags on VMS, but users can get the same effect by setting environment variables.

`wave -lv number` — Sets the initial number of local variables to *number*.

`wave -cs number` — Sets the initial size of the code area to *number* of bytes.

The corresponding environment variables for Windows and UNIX are `WAVE_INIT_LVARS` and `WAVE_INIT_CODESIZE`.

Constants and Variables

Constants and variables are combined with operators and functions to produce results. A constant is a value that does not change during the execution of a program. A variable is a location with a name that contains a scalar or array value. During the execution of a program or an interactive terminal session, numbers, strings, or arrays may be stored into variables and used in future computations.

Constants

The data type of a constant is determined by its syntax, as explained later in this section. In PV-WAVE there are eight basic data types, each with its own form of constant:

- **Byte** — 8-bit unsigned integers.
- **Integer** — 16-bit signed integers.
- **Longword** — 64-bit signed integers on Digital ALPHA UNIX platforms; 32-bit signed integers on all other platforms.
- **Floating-Point** — 32-bit single-precision floating-point.
- **Double-Precision** — 64-bit double-precision floating-point.
- **Complex** — Real-imaginary pair using single-precision floating-point.
- **Double Complex** — Real-imaginary pair using double-precision floating-point.
- **String** — Zero or more eight-bit characters which are interpreted as text.

In addition, structures are defined in terms of the eight basic data types. [Chapter 6, Working with Structures](#), describes the use of structures in detail.

Numeric Constants

This section discusses the different kinds of numeric constants in PV-WAVE and their syntax. The types of numeric constants are:

- Integer constants.
- Floating-point and double-precision constants.
- Complex constants.

Integer Constants

Numeric constants of different types may be represented by a variety of forms. The syntax of integer constants is shown in the following table, where “*n*” represents one or more digits.

Syntax of Integer Constants

Radix	Type	Form	Examples
Decimal	Byte	<i>n</i> B	12B, 34B
	Integer	<i>n</i>	12, 425
	Long	<i>n</i> L	12L, 94L
Hexadecimal	Byte	' <i>n</i> 'XB	'2E'XB
	Integer	' <i>n</i> 'X	'0F'X
	Long	' <i>n</i> 'XL	'FF'XL
Octal	Byte	" <i>n</i> B	"12B
		Integer	" <i>n</i>
	Long	' <i>n</i> 'O	'377'O
		" <i>n</i> L	"12L
		' <i>n</i> 'OL	'777777'OL

Digits in hexadecimal constants may include the letters A through F, for the decimal numbers 10 through 15. Also, octal constants may be written using the same style as hexadecimal constants by substituting an O for the X. The following table illustrates both examples of valid and invalid constants.

Examples of Integer Constants

Correct	Incorrect	Reason
255	256B	Too large, limit is 255
'123'X	'123X	Unbalanced apostrophe
-'123'X	'-123'X	Minus sign inside apostrophe
"123	'03G'x	Invalid character
'27'OL	'27'L	No radix
'650'XL	650XL	No apostrophes
"124	"129	9 is an invalid octal digit

Values of integer constants can range from 0 to 255 for bytes, 0 to $\pm 32,767$ for integers, and 0 to $\pm (2^{31} - 1)$ for longwords. Integers that are initialized with absolute values greater than 32,767 are automatically typed as longword. Any numeric constant may be preceded by a + or a - sign. To ensure cross-platform compatibility, place the + or a - sign outside of the apostrophe.

CAUTION There is no checking for integer overflow conditions when performing integer arithmetic. For example, the statement:

```
print, 32767 + 10
```

will give an incorrect answer and no error message. For more details on overflow conditions and error checking, see [Chapter 10, *Programming with PV-WAVE*](#).

Floating-point and Double-precision Constants

Floating-point and double-precision constants may be expressed in conventional or scientific notation. Any numeric constant that includes the decimal point is a floating-point or double-precision constant.

The syntax of floating-point and double-precision constants is shown in . The notation *sx* represents the sign and magnitude of the exponent, for example: E-2.

Double-precision constants are entered in the same manner, replacing the E with a D. For example, 1.0D0, 1D, 1.D, all represent a double precision one.

Syntax of Floating-point Constants

Form	Example
<i>n .</i>	102.
<i>. n</i>	.102
<i>n .n</i>	10.2
<i>n Esx</i>	10E5
<i>n .Esx</i>	10.E-3
<i>.n Esx</i>	.1E+12
<i>n .n Esx</i>	2.3E12

Complex Constants

Complex constants contain a real and an imaginary part, which can be of single or double-precision floating point numbers. The imaginary part may be omitted, in which case it is assumed to be zero.

The form of a complex constant is:

`COMPLEX(real_part, imaginary_part)`

or:

`COMPLEX(real_part)`

For example, `COMPLEX (1 , 2)`, is a complex constant with a real part of one, and an imaginary part of two. `COMPLEX (1)` is a complex constant with a real part of one and a zero imaginary component.

The ABS function returns the magnitude of a complex expression. To extract the real part of a complex expression, use the FLOAT function; to extract the imaginary part, use the IMAGINARY function. These functions are explained in the PV-WAVE Reference.

String Constants

A string constant consists of zero or more characters enclosed by apostrophes (') or quotation marks ("). The value of the constant is simply the characters appearing between the leading delimiter (' or ") and the next occurrence of the delimiter.

A double apostrophe (' ') or double quotation mark (" ") is considered to be the null string; a string containing no characters.

An apostrophe or quotation mark may be represented within a string that is delimited by the same character, by two apostrophes, or quotation marks.

For example, 'Don ' 't ' produces Don 't; or you can write: "Don 't" to produce the same result.

The following table illustrates valid string constants.

Examples of Correct String Constants

String Value	Correct
Hi there	'Hi there'
Hi there	"Hi there"
Null String	' '
I'm happy	"I'm happy"
I'm happy	'I 'm happy'
counter	'counter'
129	'129'

The following table illustrates invalid string constants.

Examples of Incorrect String Constants

String Value	Incorrect	Reason
Hi there	'Hi there"	Mismatched delimiters
Null String	'	Missing delimiter
I'm happy	'I'm happy'	Apostrophe in string
counter	' 'counter' '	Double apostrophe is null string
129	"129"	Illegal octal constant

NOTE The entry "129" is interpreted as an illegal octal constant. This is because a quotation mark character followed by a digit from 0 to 7 represents an octal numeric constant, not a string, and the character 9 is an illegal octal digit.

Representing Nonprintable Characters with UNIX/OpenVMS

The ASCII characters with values less than 32 or greater than 126 do not have printable representations. Such characters are included in string constants by specifying their octal or hexadecimal values. A character is specified in octal notation as a backslash followed by its three-digit octal value, and in hex as a backslash followed by the x or X character, followed by its two-digit hexadecimal value. In order to construct a character string which actually contains a literal backslash character, it is necessary to enter two consecutive backslash characters. The following table gives examples of using octal or hexadecimal character notation.

Specifying Non-printing Characters

Specified String	Actual Contents	Comment
' \033 [;H\033 [2J '	' <Esc>[;H<Esc>[2J '	Erase — ANSI terminal
' \x1B [;H\x1b [2J '	' <Esc>[;H<Esc>[2J '	Erase — hex notation
' \007 '	Bell	Ring the bell
' \x08 '	Backspace	Move cursor left
' \014 '	Formfeed	Eject current page
' \\hello '	'hello'	Literal backslash

Representing Nonprintable Characters with Windows

The ASCII characters with values less than 32 or greater than 126 do not have printable representations. To include such “nonprintable” characters in a string, you can use the `STRING` function. For example, the bell sound is a nonprintable ASCII “character”. The way to represent this character in a string is:

```
s='This is a bell:' + STRING(7B)
PRINT, s
; The text is printed and the bell rings.
```

The notation “7B” indicates that the parameter is of byte data type. The result is equal to the decimal ASCII code 7, which is the bell character. For more information, see [Using *STRING* with *Byte Arguments* on page 116](#).

Variables

Variables are named repositories where information is stored. A variable may contain a scalar, vector, multidimensional array, or structure of virtually any size.

Arrays may contain elements of any of the eight basic data types plus structures. Variables may be used to store images, spectra, single quantities, names, tables, etc.

The following are the basic data types that variables may have:

- **Byte** — An eight-bit unsigned integer ranging in value from 0 to 255. Pixels in images are commonly represented as byte data.
- **Integer** — A 16-bit signed integer ranging from $-32,768$ to $+32,767$.
- **Longword (Long Integer)** — A 32-bit signed integer ranging in value from approximately minus two billion to plus two billion.

NOTE On Digital ALPHA UNIX platforms only, the longword is 64 bits.

- **Floating Point** — A 32-bit single-precision number in the range of $\pm 10^{38}$, with 7 decimal places of significance.
- **Double Precision** — A 64-bit double-precision number in the range of $\pm 10^{308}$, with 14 decimal places of significance.
- **Complex** — A real-imaginary pair of single-precision floating numbers.
- **Double Complex** — A real-imaginary pair of double-precision floating numbers.
- **String** — A sequence of characters, from 0 to 32,767 characters in length. This data type is used to transfer alphanumeric strings as well as date/time data for calendar-based analysis.
- **Structure** — An aggregation made from the basic data types, other structures, and arrays. Date/time data is handled internally as a structure.

Attributes of Variables

Each variable has a structure and a type, which can change dynamically during the execution of a program or terminal session.

One important advantage that PV-WAVE has over program languages such as C and FORTRAN is that you do not need to declare variables. When a variable is assigned a value, it is automatically declared as a specific data type.

NOTE The dynamic nature of PV-WAVE variables may seem unusual to you if you are used to strongly typed languages such as PASCAL. For example, in PV-WAVE you can write a valid statement that assigns a scalar variable to an array variable, or a string variable to an array variable.

Structure of Variables

A variable may contain either a single value (a scalar), or it may contain a number of values of the same type (an array). Note that one-dimensional arrays are often referred to as *vectors* in the PV-WAVE documentation. Strings are considered a single value and a string array contains a number of fixed-length strings. A single instance of a structure is considered a scalar.

In addition, a variable may associate an array structure with a file; these variables are called associated variables. Referencing an associated variable causes data to be read from or written to the file. Associated variables and the related ASSOC function are described in [Chapter 8, Working with Data Files](#), and in the PV-WAVE Reference.

Type of Variables

A variable may have one and only one of the following types: undefined, byte, integer, longword, floating-point, double-precision floating-point, complex floating-point, string, or structure.

When a variable appears on the left-hand side of an assignment statement its attributes are copied from those of the expression on the right-hand side. For example, the statement:

```
ABC = DEF
```

redefines or initializes the variable ABC with the attributes and value of variable DEF. Attributes previously assigned to the variable are destroyed.

NOTE This is an example of PV-WAVE's loose data typing. This may be confusing to programmers who are used to strongly typed languages where such an assignment statement would produce an error.

Initially, every variable has the single attribute of *undefined*. Attempts to use undefined variables result in an error.

Names of Variables

Variables are named by identifiers that have the following characteristics:

- Each identifier must begin with a letter and may contain from one to 31 characters.
- The second and following characters may be a letter, digit, the underscore character, or the dollar sign.

- A variable name may not contain embedded spaces, because spaces are considered to be delimiters.
- Characters after the first 31 are ignored.
- Names are case insensitive, lowercase letters are converted to uppercase; so the variable name abc is equivalent to the name ABC.
- A variable may not have the same name as a function or reserved word. This will result in a syntax error. The following are reserved words:

Reserved Words

AND	BEGIN	CASE	COMMON
DO	ELSE	END	ENDCASE
ENDELSE	ENDFOR	ENDIF	ENDREP
ENDWHILE	EQ	FOR	FUNCTION
GE	GOTO	GT	IF
LE	LT	MOD	NE
NOT	OF	ON_IOERROR	OR
PRO	REPEAT	THEN	UNTIL
WHILE	XOR		

The following table illustrates examples of valid and invalid variable names.

Examples of Variable Names

Correct	Incorrect	Reason
A	EOF	Conflicts with function name
A6	6A	Doesn't start with a letter
INIT_STATE	_INIT	Doesn't start with a letter
ABC\$DEF	AB@	Illegal character, @
My_variable	ab cd	Embedded space

System Variables

NOTE For detailed information on each system variable, see the PV-WAVE Reference.

System variables are a special class of predefined variables, available to all program units. Their names always begin with the exclamation mark character !. System variables are used to set the options for plotting, to set various internal modes, to return error status, and perform other functions.

System variables have a predefined type and structure which cannot be changed. When an expression is stored into a system variable, it is converted to the type of the variable if necessary and possible.

Certain system variables are read only, and their values may not be changed. You may define new system variables with the DEFSYSV procedure.

Examples of system variable references are:

```
!Prompt = 'Good Morning: '  
    ; Change the standard WAVE> prompt to a new string.  
  
A = !C  
    ; Store value of the cursor system variable !C in A.  
  
PRINT, ACOS(a) * !Radeg  
    ; Use !Radeg, a system variable that contains a radians-to- degrees  
    ; conversion factor, to convert radians to degrees.  
  
!P.Title = 'Cross Section'  
    ; Set default plot title. !P is a structure, in which Title is a field.
```

If an error message appears that refers to the system variables !D, !P, !X, !Y, or !Z, the error message will contain an “expanded” name for the system variable. The “expanded” names of these system variables are:

- Device for !D
- Plot for !P
- Axis for !X, !Y, and !Z

Expressions and Operators

Variables and constants may be combined, using operators and functions, into expressions. Expressions are constructs that specify how results are to be obtained. Expressions may be combined with other expressions, variables, and constants to yield more complex expressions. Unlike FORTRAN or BASIC expressions, PV-WAVE expressions may be scalar or array-valued.

There is a great variety of operators in PV-WAVE. In addition to the standard operators — addition, subtraction, multiplication, division, exponentiation, relations (EQ, NE, GT, etc.), and Boolean arithmetic (AND, OR, NOT and XOR) — operators exist to find minima and maxima, select scalars and subarrays from arrays (subscripting), and to concatenate scalars and arrays to form arrays.

Functions, which are operators in themselves, perform operations that are usually of a more complex nature than those denoted by simple operators. Functions exist for data smoothing, shifting, transforming, evaluation of transcendental functions, etc. For a complete description of the PV-WAVE functions and procedures, see the PV-WAVE Reference.

Expressions may be arguments to functions or procedures. For example:

```
SIN (A * 3.14159)
```

evaluates expression A multiplied by the value of π and then applies the trigonometric sine function. This result may be used as an operand to form a more complex expression or as an argument to yet another function. For example:

```
EXP (SIN (A * 3.14159))
```

evaluates to $e^{\sin \pi a}$.

Operator Precedence

Operators are divided into the levels of algebraic precedence found in common arithmetic. Operators with higher precedence are evaluated before those with lesser precedence, and operators of equal precedence are evaluated from left to right. Operators are grouped into five classes of precedence as shown in the following table:

Operator Precedence

Priority	Operator
First (highest)	^ (exponentiation)
Second	* (multiplication) # (matrix multiplication) / (division) MOD (modulus)
Third	+ (addition) - (subtraction) < (minimum) > (maximum) NOT (Boolean negation)
Fourth	EQ (equality) NE (not equal) LE (less than or equal) LT (less than) GE (greater than or equal) GT (greater than)
Fifth	AND (Boolean AND) OR (Boolean OR) XOR (Boolean exclusive OR)

For example, the expression:

$$4 + 5 * 2$$

yields 14 because the multiplication operator has a higher precedence than the addition operator. Parentheses may be used to override the default evaluation:

$$(4 + 5) * 2$$

yields 18 because the expression inside the parentheses is evaluated first. A useful rule of thumb is when in doubt, parenthesize.

Some examples of expressions are:

A
; The value of variable A.

A + 1
; The value of A plus 1.

A < 2 + 1
; The smaller of A or 2, plus 1.

A < 2 * 3
; The smaller of A and 6; The multiplication operator (*) has a higher
precedence than the minimum operator (<).

2 * SQRT(A)
; Twice the square-root of A.

A + 'Thursday'
; The concatenation of the strings A and 'Thursday'. An error
will result if A is not a string.

Type and Structure of Expressions

Every entity in PV-WAVE has an associated type and structure. The eight atomic data types, in decreasing order of complexity are:

- Complex single-precision floating point
- Complex double-precision floating point
- Double-precision floating point
- Floating point
- Longword integer
- Integer
- Byte
- String

The structure of an expression may be either a scalar or an array. The type and structure of an expression depend upon the type and structure of its operands.

NOTE Unlike many other languages, the type and structure of expressions in PV-WAVE cannot be determined until the expression is evaluated. Because of this, care must be taken when writing programs. For example, a variable may be a scalar byte variable at one point in a program, while at a later point it may be set to a complex array.

PV-WAVE attempts to evaluate expressions containing operands of different types in the most accurate manner possible. The result of an operation becomes the same type as the operand with the greatest precedence or potential precision. For example, when adding a byte variable to a floating point variable, the byte variable is first converted to floating point and then added to the floating point variable, yielding a floating point result. When adding a double-precision variable to a complex variable, the result is complex because the complex type has a higher position in the hierarchy of data types.

When writing expressions with mixed types, caution must be used to obtain the desired result. For example, assume the variable `A` is an integer variable with a value of 5. The following expressions yield the indicated results:

`A / 2`
; Evaluates to 2. Integer division is performed. The remainder is
; discarded.

`A / 2.`
; Evaluates to 2.5. The value of `A` is first converted to floating point.

`A / 2 + 1.`
; Evaluates to 3. Integer division is done first because of operator
; precedence. Result is floating point.

`A / 2. + 1`
; Evaluates to 3.5. Division is done in floating point and then the 1
; is converted to Floating point and added.

NOTE When other types are converted to complex type, the real part of the result is obtained from the original value and the imaginary part is set to zero.

When a string type appears as an operand with a numeric data type, the string is converted to the type of the numeric term. For example:

`'123' + 123.0`

is 246.0,

`'123.333' + 33`

results in a conversion error because 123.333 is not a valid integer constant. In the same manner, `'ABC' + 123` also causes a conversion error.

Type Conversion Functions

PV-WAVE provides a set of functions that convert an operand to a specific type. These functions are useful in many instances, such as forcing the evaluation of an

expression to a certain type, outputting data in a mode compatible with other programs, etc. The conversion functions are shown in the following table.

Type Conversion Functions

Function	Description
STRING	Convert to string
BYTE	Convert to byte
FIX	Convert to integer
LONG	Convert to longword integer
FLOAT	Convert to floating point
DOUBLE	Convert to double-precision Floating point
COMPLEX	Convert to single-precision complex value
DCOMPLEX	Convert to double-precision complex value

For example, the result of the expression `FIX (A)` is of single-precision (16-bit) integer type with the same structure (scalar or array) as the variable. The variable may be of any type. These conversion functions operate on data of any structure: scalars, vectors, or arrays. If `A` lies outside the range of single-precision integers ($-32,768$ to $+32,767$) an error will result.

CAUTION The statement:

```
PRINT, FIX(66000)
```

prints the value 464, which is 66000_{216} , with no indication that an error occurred. The `FINITE` and `CHECK_MATH` functions test floating point results for valid numbers, and check the accumulated math error status respectively. For details on these error-checking functions, see [Chapter 10, *Programming with PV-WAVE*](#).

The statement:

```
A = FLOAT(A)
```

is perfectly legitimate in; its effect is to force the variable `A` to have Floating point type.

Special cases of type conversions occur when converting between strings and byte arrays. The result of the `BYTE` function applied to a string or string array is a byte array containing the ASCII codes of the characters of the string. Converting a byte

array with the `STRING` function yields a string array or scalar with one less dimension than the byte array.

The following table shows examples of conversion on functions.

Examples of Conversion Functions

Example	Result
<code>FLOAT(1)</code>	1.0
<code>FIX(1.3 + 1.7)</code>	3
<code>FIX(1.3) + FIX(1.7)</code>	2
<code>BYTE(1.2)</code>	1
<code>BYTE(-1)</code>	255 (Bytes are modulo 256)
<code>BYTE('01ABC')</code>	[48,49,65,66,67]
<code>STRING([65B, 66B, 67B])</code>	ABC
<code>FLOAT(COMPLEX(1, 2))</code>	1.0
<code>COMPLEX([1, 2], [4, 5])</code>	[COMPLEX(1,4),COMPLEX(2,5)]

Extracting Fields

When called with more than a single parameter, the `BYTE`, `COMPLEX`, `DCOMPLEX`, `FIX`, `LONG`, `FLOAT` and `DOUBLE` functions create an expression of the designated type by extracting fields from the input parameter without performing type conversion. The result is that the original binary information is simply interpreted as being of the new type. This feature is handy for extracting fields of data of one type embedded in arrays or scalars of another type.

The general form of the type conversion functions is:

$$\text{CONV_FUNCTION}(\textit{expression}, \textit{offset} [, \textit{dim}_1, \dots, \textit{dim}_n])$$

Where:

`CONV_FUNCTION` is the name of one of the conversion functions listed previously.

expression — An array or scalar expression of any type from which the field is to be extracted.

offset — Starting byte offset within *expression* of the field to be extracted. Zero is the first byte.

dim_p, \dots, dim_n — The dimensions of the result. If these dimensions are omitted, the result is a scalar. If more than two parameters appear, the third and following parameters are the dimensions of the resulting array.

For example, assume file unit 1 is open for reading on a file containing 112-byte binary records containing the fields shown below:

Example Fields in Open File

Bytes	Type	Name
0 - 7	Double	Time
8	Byte	Type
9 - 10	Integer	Count
11 - 110	Floating	DATA (20-by-5 array)
111	Byte	Quality

The following program segment will read a record into an array and extract the fields.

```
A = BYTARR(112)
    ; Define array variable to contain record, 112 bytes.

READU, 1, A
    ; Read the next record.

TIME = DOUBLE(A, 0)
    ; Extract TIME. Offset = 0, double-precision.

TYPE = BYTE(A, 8)
    ; Extract TYPE. Starting offset is 8.

COUNT = FIX(A, 9)
    ; Count, offset = 9, integer.

DATA = FLOAT(A, 11, 20, 5)
    ; DATA = floating array, dimensions of 20-columns by 5-rows, starting offset is 11 bytes.

QUALITY = BYTE(A, 111)
    ; Last field, single byte.
```

Structure of Expressions

Expressions may contain operands with different structures, just as they may contain operands with different types. Structure conversion is independent of type

conversion. An expression will yield an array result if any of its operands is an array as shown in the following table:

Operands	Result
Scalar : Scalar	Scalar
Array : Array	Array
Scalar : Array	Array
Array : Scalar	Array

Eight functions exist to create arrays of the eight types: BYTARR, INTARR, LONARR, FLTARR, DBLARR, COMPLEXARR, DCOMPLEXARR, and STRARR. The dimensions of the desired array are the parameters to these functions. The result of `FLTARR (5)` is a floating point array with one dimension, a vector, with five elements initialized to zero. `FLTARR (50, 100)` is a two-dimensional array, a matrix, with 50 columns and 100 rows.

The size of an array-valued expression is equal to the smaller of its array operands. For example, adding a 50-point array to a 100-point array gives a 50-point array, the last 50 points of the larger array are ignored. Array operations are performed point-by-point without regard to individual dimensions. An operation involving a scalar and an array always yields an array of identical dimensions. When two arrays of equal size (number of elements) but different structure are operands, the result is of the same structure as the first operand.

For example:

```
FLTARR (4) + FLTARR (1, 4)
```

yields `FLTARR (4)`.

In the above example, a row vector is added to a column vector and a row vector is obtained because the operands are the same size causing the result to take the structure of the first operand.

Here are some examples of expressions involving arrays:

```
ARR + 1
```

```
; Is an array in which each element is equal to the same element in  
; ARR plus 1. The result has the same dimensions as ARR. If ARR is  
; byte or integer the result is of integer type, otherwise the result is  
; the same type as ARR.
```

```
ARR1 + ARR2
```

```
; Is an array obtained by summing two arrays.
```

```
(ARR < 100) * 2
```

; Is an array in which each element is set to twice the smaller of
; either the corresponding element of ARR or 100.

```
EXP (ARR / 10.)
```

; Is an array in which each element is equal to the exponential of the
; same element of ARR divided by 10.

```
ARR * 3. / MAX (ARR)
```

; Is an inefficient way of writing the following line:

```
ARR * (3. / MAX (ARR))
```

; The more efficient way.

In the inefficient example above, each point in ARR is multiplied by 3 and then divided by the largest element of ARR. (The MAX function returns the largest element of its array argument.) This way of writing the statement requires that each element of ARR be operated on twice. If (3. / MAX (ARR)) is evaluated with one division and the result then multiplied by each point in ARR the process requires approximately half the time.

PV-WAVE Operators

The following types of operators are described in this section:

- Assignment, array, numeric, and string operators
- Boolean operators
- Relational operators

Assignment, Array, and Numeric Operators

Summary of Operators

Operator	Meaning
()	Expression grouping
=	Assignment
^	Exponentiation
*	Multiplication
#	Matrix multiplication
/	Division

Summary of Operators (Continued)

Operator	Meaning
+	Addition and string concatenation
-	Subtraction
MOD	Modulo operator
[]	Array concatenation

Parentheses ()

Used in grouping of expressions and to enclose subscript and function parameter lists. Parentheses can be used to override order of operator evaluation as described above. Examples:

```
A(X, Y)
    ; Parentheses enclose subscript lists, if A is defined as a variable.
```

```
SIN(ANG * PI / 180.)
    ; Parentheses enclose function argument lists.
```

```
X = (A + 5) / B
    ; Parentheses specify order of operator evaluation.
```

Assignment Operator =

The value of the expression on the right side of the equal sign is stored in the variable, subscript element, or range on the left side. For more information, see [Assignment Statement on page 48](#).

For example:

```
A = 32
    ; Assigns the value of 32 to variable A.
```

Addition Operator +

Besides arithmetic addition, the addition operator concatenates the strings. For example:

```
B = 3 + 6
    ; Assigns the value of 9 to B.
```

```
B = 'John' + ' ' + 'Doe'
    ; Assigns the string value of "John Doe" to B.
```

Subtraction Operator –

Besides subtraction, the minus sign is used as the unary negation operator. For example:

$C = 9 - 5$
; Assigns the value of 4 to C.

$C = - C$
; Changes the sign of C.

Multiplication Operator *

Multiplies two operands. For example:

$A = 5 * 4$
; Assigns the value of 20 to A.

Division Operator /

Divides two operands. For example:

$A = 20 / 4$
; Assigns the value of 5 to A.

Exponentiation Operator ^

A^B is equal to A to the B power. If B is of integer type, repeated multiplication is applied, otherwise the formula $A^B = e^{B \log A}$ is evaluated. 0^0 is undefined for all types of operands.

Matrix Multiplication Operator #

The rules of linear algebra are followed:

- The two operands must conform in that the second dimension of the first operand must equal the first dimension of the second operand.
- The first dimension of the result is equal to the first dimension of the first operand and the second dimension of the result is equal to the second dimension of the second operand.
- The type of the result is double complex, single complex, double-precision or floating point, in decreasing order of precedence. In mixed-mode operations, the calculations are performed in the mode yielding the greatest precision. If neither operand is of one of these types, the type of the result is floating point.

If a parameter is a one-dimensional vector, it is interpreted as either a row or column vector, whichever conforms to the other operand. If both operands are vectors,

the result of the operation is the outer product of the two vectors. Results in which the second dimension is equal to 1 (row vectors) are converted to vectors.

Use the TOTAL function to obtain the inner product which is the sum of the product of the elements of the vectors. The expression

```
TOTAL (A * A)
```

calculates the inner product of the vector A.

For example, the statement:

```
PRINT, [1, 2, 3, 4] # [1, 2, 3, 4]
```

prints the outer product of two four-element vectors whose elements are the integers 1 to 4, or:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 6 & 8 \\ 3 & 6 & 9 & 12 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

NOTE The notion of columns and rows is reversed from that of linear algebra, although their treatment is consistent. The main reason for this is to allow the X subscript to appear first when subscripting images, as is the convention. Arrays and vectors that are operands for the matrix multiplication operator may be transposed, either by entering them transposed or by using the TRANSPOSE function.

MOD

Modulo operator. $I \text{ MOD } J$ is equal to the remainder when I is divided by J .

When I or J are floating point, double-precision, or complex,

$I \text{ MOD } J = I - J * [I/J]$, where the bracketed value is the largest integer smaller than or equal to the expression in the brackets. For example:

```
A = 9 MOD 5  
; A is set to 4.
```

```
A = (ANGLE + B) MOD (2 * PI)  
; Compute angle modulo  $2\pi$ .
```

Array Concatenation Operators []

Operands enclosed in square brackets and separated by commas are concatenated to form larger arrays. The expression $[A, B]$ is an array formed by concatenating the first dimensions of A and B , which may be scalars or arrays.

Similarly, [A, B, C] concatenates A, B, and C. The second and third dimensions may be concatenated by nesting the bracket levels: [[1, 2], [3, 4]] is a two-by-two array with the first row containing 1 and 2, and the second row containing 3 and 4. Operands must have compatible dimensions: all dimensions must be equal except the dimension that is to be concatenated. For example, [2, INTARR(2, 2)] are incompatible.

For example:

```
C = [-1, 1, -1]
    ; Defines C as three-point vector.

C = [C, 12]
    ; Adds a 12 to the end of C.

C = [12, C]
    ; Inserts a 12 at the beginning.

PLOT, [ARR1, ARR2]
    ; Plots ARR2 appended to the end of ARR1.

KER = [[1, 2, 1], [2, 4, 2], [1, 2, 1]]
    ; Defines a 3-by-3 array.
```

Boolean Operators

Results of relational expressions may be combined into more complex expressions using the Boolean operators AND, OR, NOT, and XOR (exclusive OR). The action of these operators is summarized as follows:

Operator (<i>oper</i>)	T <i>oper</i> T	T <i>oper</i> F	F <i>oper</i> F
AND	T	F	F
OR	T	T	F
XOR	F	T	F

NOT is the Boolean inverse and is a unary operator because it only has one operand. NOT true is false and NOT false is true.

AND

AND is the Boolean operator which results in true whenever both of its operands are true, otherwise the result is false. Any non-zero value is considered true. For integer and byte operands, a bitwise AND operation is performed. For operations on other types, the result is equal to the first operand if the second operand is not equal to zero or the null string. Otherwise, it is zero or the null string.

NOT

NOT is the Boolean complement operator. NOT true is false. NOT complements each bit for integer or byte operands. For floating point operands, the result is 1.0 if the operand is zero, otherwise, the result is zero. NOT is the Boolean inverse and is a unary operator because it only has one operand. NOT true is false and NOT false is true.

OR

OR is the Boolean inclusive operator. For integer or byte operands a bitwise inclusive “or” is performed. For example, 3 OR 5 equals 7. For floating point operands the OR operator returns a 1.0 if neither operand is zero, otherwise zero is the result.

XOR

The Boolean exclusive “or” function. XOR is only valid for integer or byte operands. XOR returns a one bit if the corresponding bits in the operands are different; if they are equal, a zero bit is returned.

Examples

When applied to bytes, integers, and longword operands, the Boolean functions operate on each binary bit.

```
(1 AND 7)
    ; Evaluates to 1.
(3 OR 5)
    ; Evaluates to 7.
(NOT 1)
    ; Evaluates to -2 (twos-complement arithmetic).
(5 XOR 12)
    ; Evaluates to 9.
```

When applied to data types that are not integers, the Boolean operators yield the following results:

```
OP1 AND OP2
    ; Means OP1 if OP2 is true (not zero or not the null string), otherwise
    ; false (zero or the null string).
OP1 OR OP2
    ; Means OP2 if OP2 is true, otherwise OP1.
```

Some examples of relational and Boolean expressions are:

```
(A LE 50) AND (A GE 25)
```

; True if A is between 25 and 50. If A is an array the result is an array
; of ones and zeroes.

(A GT 50) OR (A LT 25)

; True if A is less than 25 or A is greater than 50. This expression is
; the inverse of the first example.

ARR AND 'FF'X

; ANDs the hexadecimal constant FF, (255 in decimal) with the
; array ARR. This masks the lower 8 bits and zeroes the upper bits.

Relational Operators

Operator	Meaning
EQ	Equal to
NE	Not equal to
GE	Greater than or equal to
GT	Greater than
LE	Less than or equal to
LT	Less than
<	Comparison to find minimum
>	Comparison to find maximum

Relational operators apply a relation to two operands and return a logical value of true or false. The resulting logical value may be used as the predicate in IF, WHILE, or REPEAT statements or may be combined using Boolean operators with other logical values to make more complex expressions. For example:

1 EQ 1
is true, and
1 GT 3
is false.

The rules for evaluating relational expressions with operands of mixed modes are the same as those given above for arithmetic expressions. For example, in the relational expression:

(2 EQ 2.0)

the integer 2 is converted to floating point and compared to the Floating point 2.0. The result of this expression is true which is represented by a floating point 1.0.

The value *true* is represented by the following:

- An odd, non-zero value for byte, integer and longword integer data types.
- Any non-zero value for single, double-precision and complex floating.
- Any non-null string.

Conversely, *false* is represented as anything that is not true: zero- or even-valued integers; zero-valued floating point quantities; and the null string.

The relational operators return a value of 1 for true and zero for false. The type of the result is determined by the same rules that govern the types of arithmetic expressions. So,

```
(100. EQ 100.)
```

is 1.0, while

```
(100 EQ 100)
```

is 1, the integer.

Relational operators may be applied to arrays and the result, which is an array of ones and zeroes, may be used as an operand. For example, the expression:

```
ARR * (ARR LE 100)
```

is an array equal to ARR except that all points greater than 100 have been zeroed. The expression (ARR LE 100) is an array that contains a 1 where the corresponding element of ARR is less than or equal to 100, and zero otherwise.

Minimum Operator <

The value of $A < B$ is equal to the smaller of A or B. For example:

```
A = 5 < 3
```

```
; Sets A to 3.
```

```
ARR = ARR < 100
```

```
; Sets all points in array ARR that are larger than 100 to 100.
```

```
X = X0 < X1 < X2
```

```
; Sets X to smallest operand.
```

Maximum Operator >

$A > B$ is equal to the larger of A or B. For example:

```
C = ALOG(D > 1E-6)
```

; Avoids taking logs of 0 or negative numbers.

PLOT, ARR > 0

; Plots only positive points. Negative points are plotted as zero.

EQ

EQ returns true if its operands are equal, otherwise it is false. For floating point operands true is 1.000; for integers and bytes, it is 1. For string operands, a zero-length null string represents false.

GE

GE is the greater than or equal to relational operator. GE returns true if the operand on the left is greater than or equal to the one on the right.

One use of relational operators is to mask arrays:

```
A = ARRAY * (ARRAY GE 100)
```

sets A equal to ARRAY whenever the corresponding element of ARRAY is greater than or equal to 100; if the element is less than 100, the corresponding element of A is set to 0.

Strings are compared using the ASCII collating sequence: “ ” is less than “0”, is less than “9”, is less than “A”, is less than “Z”, is less than “a”, which is less than “Z”.

GT

Greater than relational operator.

LE

Less than or equal to relational operator.

LT

Less than relational operator.

NE

NE is the not equal to relational operator. It is true whenever the operands are not of equal value.

Statement Types

PV-WAVE programs, procedures, and functions are composed of one or more valid statements. Most simple statements may be entered in the interactive mode in response to the WAVE> prompt. The 12 types of statements are:

- Assignment
- Block
- CASE
- Common Block Definition
- FOR
- Function Declaration
- Function Definition
- GOTO
- IF
- Procedure Call
- Procedure Definition
- REPEAT
- WHILE

Components of Statements

Statements may consist of any combination of three parts:

- A label field
- The statement proper
- A comment field

Spaces and tabs may appear anywhere except in the middle of an identifier or numeric constant.

Statement Labels

Labels are the destinations of GOTO statements. The label field, which must appear before the statement or comment, is simply an identifier followed by a colon. A line may consist of only a label field. Label identifiers, as with variable names, may consist of from one to 31 alphanumeric characters. The \$ (dollar sign) and _ (underscore) characters may appear after the first character. Some examples of labels are:

```
Label_1:
LOOP_BACK: A = 12
I$QUIT: RETURN      ;Quit the loop.
           ; Note that comments are allowed after labels.
```

Adding Comments

The comment field, which is ignored, begins with a semicolon and continues to the end of the line. Lines may consist of only a comment field. There are no execution time or space penalties for comments in PV-WAVE.

Assignment Statement

The assignment statement stores a value in a variable. There are four forms of the assignment statement. They are described in detail in this section.

The following table summarizes the four forms of assignment statements.

Syntax	Subscript Structure	Expression Structure	Effect
Form 1: Var = expr	none	Scalar or Array	The expr is stored in Var.
Form 2: Var(subs) = scalar.5	Scalar	Scalar	The scalar expression is stored in a single element of Var.

Syntax	Subscript Structure	Expression Structure	Effect
	Array	Scalar	The scalar expression is stored in the designated elements of Var.
Form 3: Var(range) = expr	Range	Scalar	The scalar is inserted into the subarray.
	Range	Array	Illegal
Form 4: Var(subs) = array	Scalar	Array	The array is inserted in the Var array.
	Array	Array	The elements of the array are stored in the designated elements of Var.

Form 1

The first (and most basic) form of the assignment statement has the form:

```
variable = expression
; Stores the value of the expression in the variable.
```

The old value of the variable, if any, is discarded and the value of the expression is stored in the variable. The expression on the right side may be of any type or structure. Some examples of the basic form of the assignment statement are:

```
MMAX = 100 * X + 2.987
; Stores the value of the expression in MMAX.

NAME = 'MARY'
; Stores the string 'MARY' in the variable NAME.

ARR = FLTARR(100)
; ARR is now a 100-element floating-point array.

ARR = ARR(50:*)
; Discards elements 0 to 49 of ARR. ARR is now a 50-element array.
```

Form 2

The second type of assignment statement has the form:

```
variable(subscripts) = scalar_expression
; Stores the scalar in an element of the array variable.
```

Here, a single element of the specified array is set to the value of the scalar expression. The expression may be of any type and is converted, if necessary, to the type

of the variable. The variable on the left side must be either an array or a file variable.

```
DATA(100) = 1.234999  
; Sets element (100) of DATA to value.
```

```
NAME(INDEX) = 'JOE'  
; Stores a string in the array. NAME must be a string array or an error will result.
```

Using Array Subscripts with the Second Form

If the subscript expression is an array, the scalar value will be stored in the elements of the array whose subscripts are elements of the subscript array. For example, the statement:

```
DATA( [3, 5, 7, 9] ) = 0
```

will zero the four specified elements of DATA: DATA(3), DATA(5), DATA(7), and DATA(9).

The subscript array is converted to longword type if necessary before use. Elements of the subscript array that are negative or greater than the highest subscript of the subscripted array are ignored.

The WHERE function may frequently be used to select elements to be changed. For example, the statement:

```
DATA(WHERE(DATA LT 0)) = -1
```

will set all negative values of DATA to -1 without changing the positive values. The result of the function WHERE(DATA LT 0) is a vector composed of the subscripts of the negative values of DATA. Using this vector as a subscript changes all the negative values to -1 in DATA. Note that if the WHERE function finds no eligible elements, it returns a 1-element vector equal to -1; using this result as a subscript vector changes no elements of the subscripted array; it results in a “subscript out of range” error as negative subscripts are not allowed. For more information on the WHERE function, see the PV-WAVE Reference.

Form 3

The third type of assignment statement is similar to the second, except the subscripts specify a range in which all elements are set to the scalar expression.

variable(subscript_range) = scalar_expression

Stores the scalar in the elements of the array variable designated by the subscript range.

A subscript range specifies a beginning and ending subscript. The beginning and ending subscripts are separated by the colon character. An ending subscript equal to the size of the dimension minus one may be written as `*`.

For example, `ARR (I : J)` denotes those points in the vector `ARR` with subscripts between `I` and `J`. `I` must be less than `J` and greater than or equal to zero. `J` must be less than the size of the array dimension. `ARR (I : *)` denotes the points in `ARR` from `ARR (I)` to the last point.

For more information on subscript ranges, see [Subscript Ranges on page 74](#).

Assuming the variable `B` is a 512-by-512 byte array, some examples are:

```
B (*, I) = 1
    ; Stores ones in the ith row.
B (J, *) = 1
    ; Stores ones in the jth column.
B (200:220, *) = 0
    ; Zeroes all the rows of the columns 200 through 220 of the array B.
B (*) = 100.
    ; Stores the value 100 in all the elements of the array B.
```

Form 4

The fourth type assignment statement is of the form:

variable(subscripts) = array

Inserts the array expression into the array variable starting at the element designated by the subscripts.

Note that this form is syntactically identical to the second type of assignment statement, except the expression on the right is an array instead of a scalar. This form of the assignment statement is used to insert one array into another.

The array expression on the right is inserted into the array appearing on the left side of the equal sign, starting at the point designated by the subscripts.

For example, to insert the contents of an array called `A` into an array called `B`, starting at point `B (13 , 24)`:

```
B (13, 24) = A
    ; If A is a 5-column by 6-row array,
    ; elements B ( 13 : 17 , 24 : 29 ) will be replaced by the contents of the array A.
```

Another example moves a subarray from one position to another:

```
B (100, 200) = B (200:300, 300:400)
```

; A subarray of B, specifically the columns 200 to 300 and
; rows 300 to 400, is moved to columns 100 to 200 and rows
; 200 to 300, respectively.

Using Array Subscripts with the Fourth Form

If the subscript expression applied to the variable is an array and an array appears on the right side of the statement:

$$\text{var}(\text{array}) = \text{array}$$

elements from the right side are stored in the elements designated by the subscript vector. Only those elements of the subscripted variable whose subscripts appear in the subscript vector are changed.

For example, the statement:

$$B([2, 4, 6]) = [4, 16, 36]$$

is equivalent to the following series of assignment statements:

$$B(2) = 4 \ \& \ B(4) = 16 \ \& \ B(6) = 36$$

Subscript elements are interpreted as if the subscripted variable is a vector. For example if A is a 10-by-*n* matrix, the element $A(i,j)$ has the subscript $(i+10j)$. The subscript array is converted to longword type before use if necessary.

As described above for the second type of assignment statement, elements of the subscript array that are negative or larger than the highest subscript are ignored and the corresponding element of the array on the right side of the equal sign is skipped.

As another example, assume that the vector DATA contains data elements and that a data drop-out is denoted by a negative value. In addition, assume that there are never two or more adjacent drop-outs.

The following statements will replace all drop-outs with the average of the two adjacent good points:

```
BAD = WHERE (DATA LT 0)
```

```
    ; Subscript vector of drop-outs.
```

```
DATA(BAD) = (DATA(BAD - 1) + DATA(BAD + 1)) / 2
```

```
    ; Replace drop-outs with average of previous and next point.
```

In this example:

- Elements of the vector BAD are set to the subscripts of the points of DATA that are drop-outs using the WHERE function. The WHERE function returns a vector containing the subscripts of the non-zero elements of its (DATA LT 0). This Boolean expression is a vector that is non-zero where the elements of DATA are negative and is zero if positive.

- The expression `DATA (BAD - 1)` is a vector which contains the subscripts of the points immediately preceding the drop-outs, while similarly, the expression `DATA (BAD + 1)` is a vector containing the subscripts of the points immediately after the drop-outs.
- The average of these two vectors is stored in `DATA (BAD)` — the points that originally contained drop-outs.

Associated Variables in Assignment Statements

A special case occurs when using an associated file variable in an assignment statement. For additional information regarding the `ASSOC` function, see [Chapter 8, Working with Data Files](#). When a file variable is referenced, the last (and possibly only) subscript denotes the record number of the array within the file. This last subscript must be a simple subscript. Other subscripts and subscript ranges, except the last, have the same meaning as when used with normal array variables.

An implicit extraction of an element or subarray in a data record may also be performed:

```
A = ASSOC(1, FLTARR(200))
    ; Variable A associates the file open on unit 1 with records of
    ; 200-element floating point vectors.

X = A(0:99, 2)
    ; X is set to the first 100 points of record number 2, the third record of the file.

A(23, 16) = 12
    ; Sets the 24th point of record 16 to 12.

A(10, 12) = A(10:*, 12) + 1
    ; Points 10 to 199 of record 12 are incremented. Points 0 to 9 of
    ; that record remain unchanged.
```

Blocks of Statements

```
BEGIN
    Statement1
    . . .
    Statementn
END
```

A block of statements is simply a group of statements that are treated as a single statement. Blocks are necessary when more than one statement is the subject of a conditional or repetitive statement, as in the `FOR`, `WHILE`, and `IF` statements.

In general, the format of a FOR statement with a block subject is:

```
FOR variable = expression, expression DO BEGIN
    statement1
    statement2
    ...
    ...
    ...
    statementn
ENDFOR
```

All the statements between the BEGIN and the ENDFOR are the subject of the FOR statement. The group of statements is executed as a single statement and is considered to be a compound statement. Blocks may include other blocks.

Syntactically, a block of statements is composed of one or more statements of any type, started by a BEGIN identifier and ended by an END identifier. PV-WAVE allows the use of blocks wherever a single statement is allowed. Blocks may also be nested within other blocks.

For example, the process of reversing an array in place might be written:

```
FOR I = 0, (N - 1) / 2 DO BEGIN
    T = ARR(I)
    ARR(I) = ARR(N - I - 1)
    ARR(N - I - 1) = T
ENDFOR
```

The three statements between the BEGIN and ENDFOR are the subject of the FOR statement. Each statement is executed one time during each iteration of the loop. If the statements had not been enclosed in a block, only the first statement ($T = \text{ARR}(I)$) would have been executed each iteration, and the remaining two statements would have each been executed only once after the termination of the FOR statement.

To ensure proper nesting of blocks of statements, the END terminating the block may be followed by the block type as shown in the following table. The compiler checks the end of each block, comparing it with the type of the enclosing statement.

NOTE Any block may be terminated by the generic END, although no type checking will be performed.

End Statements

End Statement	Usage
ENDCASE	CASE statement
ENDELSE	IF statement, ELSE clause
ENDFOR	FOR statement
ENDIF	IF statement, THEN clause
ENDREP	REPEAT statement
ENDWHILE	WHILE statement

Listings produced by the PV-WAVE compiler indent each block four spaces to the right of the previous level to improve the legibility of the program structure.

CASE Statement

```
CASE expression OF
    expression: statement
    . . .
    . . .
    expression: statement
ELSE: statement
ENDCASE
```

The CASE statement is used to select one, and only one, statement for execution depending upon the value of the expression following the word CASE. This expression is called the *case selector expression*. Each statement that is part of a CASE statement is preceded by an expression which is compared to the value of the selector expression. If a match is found, the statement is executed and control resumes directly below the CASE statement.

The ELSE clause of the CASE statement is optional. If included, it must be the last clause in the CASE statement. The statement after the ELSE is executed only if none of the preceding statement expressions match. If the ELSE is not included and none of the values match, an error will occur and program execution will stop.

An example of the CASE statement is:

```
CASE NAME OF
    'LINDA': PRINT, 'SISTER'
```

```

        ; Executed if NAME = 'LINDA'
'JOHN' : PRINT, 'BROTHER'
        ; Executed if NAME = 'JOHN'
'HARRY' : PRINT, 'STEP-BROTHER'
ELSE: PRINT, 'NOT A SIBLING'
        ; Executed if no matches.
ENDCASE

```

Another example, below, shows the CASE statement with the number 1 as the selector expression of the CASE. 1 is equivalent to true and is matched against each of the conditionals.

```

CASE 1 OF
(X GT 0) AND (X LE 50): Y = 12 * X + 5
(X GT 50) AND (X LE 100): Y = 13 * X + 4
(X LE 200): BEGIN
Y = 14 * X - 5
Z = X + Y
END
ELSE: PRINT, 'X has illegal value'
ENDCASE

```

In the CASE statement, only one clause is selected, and that clause is the first one whose value is equal to the value of the case selector expression.

Common Block Definition Statement

```
COMMON block_name, var1, var2,..., varn
```

The Common Block Definition statement creates a Common Block with the designated name (up to 31 characters long) and places the variables whose names follow into that block. Variables defined in a Common Block may be referenced by any program unit that declares that Common Block.

A Common Block Definition statement is useful when there are variables which need to be accessed by several procedures. Any program unit referencing a Common Block may access variables in the block as though they were local variables. Variables in a Common statement have a global scope within procedures defining the same Common Block. Unlike local variables, variables in Common Blocks are not destroyed when a procedure is exited.

The number of variables appearing in the Common Block Definition statement determines the size of the Common Block. The first program unit (main program, function, or procedure) defining the Common Block sets the size of the Common

Block, which is fixed. Other program units may reference the Common Block with the same or fewer number of variables.

Common Blocks share the same space for all procedures. Common Block variables are matched variable to variable, unlike FORTRAN, where storage locations are matched. The third variable in a given Common Block will always be the same as the third variable in all declarations of the Common Block regardless of the size, type or structure of the preceding variables.

The two procedures in the following example show how variables defined in Common Blocks are shared:

```
PRO ADD, A
  COMMON SHARE1, X, Y, Z, Q, R
  A = X + Y + Z + Q + R
  PRINT, X, Y, Z, Q, R, A
  RETURN
END

PRO SUB, T
  COMMON SHARE1, A, B, C, D
  T = A - B - C - D
  PRINT, A, B, C, D, T
  RETURN
END
```

The variables X, Y, Z, and Q in the procedure ADD are the same as the variables A, B, C, and D, respectively, in procedure SUB. The variable R in ADD is not used in SUB. If the procedure SUB were to be compiled before the procedure ADD, an error would occur when the COMMON definition in ADD was compiled. This is because SUB has already declared the size of the Common Block, SHARE1, which may not be extended.

Variables in Common Blocks may be of any type and may be used in the same manner as normal variables. Variables appearing as parameters may not be used in Common Blocks. There are no restrictions in regard to the number of Common Blocks used, although each Common Block uses dynamic memory.

FOR Statement

There are two basic forms of the FOR statement:

FOR *var* = *expr*₁, *expr*₂ DO *statement*

Form 1: Increment of 1.

FOR *var* = *expr*₁, *expr*₂, *expr*₃ DO *statement*

Form 2: Variable increment.

The FOR statement is used to execute one or more statements repeatedly while incrementing or decrementing a variable each repetition until a condition is met. It is analogous to the DO statement in FORTRAN. There are two types of FOR statements; one with an implicit increment of 1, and the other with an explicit increment. If the condition is not met the first time the FOR statement is executed, the subject statement is not executed.

NOTE The data type of the statement and the index variable are determined by the type of the initial value expression.

Form 1: Implicit Increment

The FOR statement with an implicit increment of 1 is written as follows:

FOR var = expr₁, expr₂ DO statement

The variable after the FOR is called the index variable and is set to the value of the first expression. The statement is executed, and the index variable is incremented by one, until the index variable is larger than the second expression. This second expression is called the limit expression.

Complex limit and increment expressions are converted to floating-point type. *expr₂* is not evaluated as a Boolean expression with a True/False result, but rather directly compared to the index variable *I* with True returned only if $I \leq expr_2$.

An example of a FOR statement is:

```
FOR I = 1, 4 DO PRINT, I, I^2
```

which produces the output:

```
1 1
2 4
3 9
4 16
```

The index variable *I* is first set to an integer variable with a value of 1. The call to the PRINT procedure is executed, then the index is incremented by 1. This is repeated until the value of *I* is greater than 4, when execution continues at the statement following the FOR statement.

The next example displays the use of a block structure in place of the single statement for the subject of the FOR statement. The example is a common process used for computing a count-density histogram.

```
FOR K = 0, N - 1 DO BEGIN
    C = A(K)
    HIST(C) = HIST(C) + 1
ENDFOR
```

NOTE A HISTOGRAM function is provided in the Standard Library.

Another example is:

```
FOR X = 1.5, 10.5 DO S = S + SQRT(X)
```

In this example, X is set to a floating-point variable and steps through the values (1.5, 2.5, ..., 10.5).

The indexing variables and expressions may be integer, longword integer, floating-point, or double-precision. The type of the index variable is determined by the type of the first expression after the = character.

If you need to use very large integers in a FOR loop condition, be sure to designate them as longword in the FOR loop statement. For example:

```
FOR i=300000L, 700000L DO BEGIN
    . . .
ENDFOR
```

Form 2: Explicit Increment

The format of the second type of FOR statement is:

FOR var = expr₁, expr₂, expr₃ DO statement

The first two expressions describe the range of numbers the variable will assume. The third expression specifies the increment of the index variable. A negative increment allows the index variable to step downward. In this case, the first expression must have a value greater than that of the second expression. If it does not, the statement will not be executed.

The following examples demonstrate the second type of FOR statement:

```
FOR K = 100.0, 1.0, -1 DO ...
    ; Decrement K has the values: 100., 99., ...,2., 1.

FOR LOOP = 0, 1023, 2 DO ...
    ; Increments by 2. LOOP has the values 0, 2, 4,..., 1022.
```

```
FOR MID = BOTTOM, TOP, (TOP - BOTTOM) / 4.0 DO ...  
; Divides range from BOTTOM to TOP by 4.
```

CAUTION If the value of the increment expression is zero an infinite loop will occur. A common mistake resulting in an infinite loop is a statement similar to the following:

```
FOR X = 0, 1, .1 DO ...
```

The variable X is first defined as an integer variable because the initial value expression is an integer zero constant. Then the limit and increment expressions are converted to the type of X, integer, yielding an increment value of zero because .1 converted to integer type is zero. The correct form of the statement is:

```
FOR X = 0., 1, .1 DO ...
```

which defines X as a floating-point variable.

Function Declaration Statement

```
DECLARE FUNC, name1, name2, ..., namen
```

You can declare a function in the program unit in which the function appears. Forward declaration of the function (a declaration that occurs before the function definition) allows the compiler to distinguish between function calls and array element references, which have a similar syntax. As a result, program compilation is more efficient for that particular program unit.

A backward declaration (a declaration that occurs after the function definition) allows the compiler to recognize recursive calls to a function that is defined later in the program unit.

The first call to a function also has the effect of declaring that function and increasing the efficiency of the compiler for consecutive calls to that function.

Function Definition Statement

```
FUNCTION function_name, p1, p2, ..., pn
```

A function may be defined as a program unit containing one or more statements and which returns a value. Once a function has been defined, references to the func-

tion cause the program unit to be executed. All functions return a function value which is given as a parameter in the RETURN statement used to exit the function.

Briefly the format of a function definition is, where *name* can contain up to 31 characters:

```
FUNCTION name, parameter1, ..., parametern
    Statement1
    Statement2
    . . .
    . . .
    RETURN, expression
END
```

For example, to define a function called AVERAGE that returns the average value of an array:

```
FUNCTION AVERAGE, ARR
    RETURN, TOTAL (ARR) /N_ELEMENTS (ARR)
END
```

Once the function AVERAGE has been defined, it is executed by entering the function name followed by its arguments enclosed in parentheses. Assuming the variable X contains an array, the statement:

```
PRINT, AVERAGE (X^2)
```

squares the array X, passes this result to the AVERAGE function, and prints the result.

Functions can take positional and keyword parameters. For more information on parameters and parameter passing, see [Positional Parameters and Keyword Parameters on page 65](#) and [More On Parameters on page 66](#).

For more information on writing functions, see [Chapter 9, Writing Procedures and Functions](#).

Automatic Compilation of Functions and Procedures

PV-WAVE will automatically compile and execute a user-written function or procedure when it is first referenced if *both* of the following conditions are met:

- The source code of the function is in the current working directory or in a directory in the search path defined by the system variable !Path. For more information setting the search path, see [Appendix B, Modifying Your Environment](#). For more information on system variables, see [System Variables on page 28](#).

- The name of the file containing the function is the same as the function name suffixed by `.pro`. The file name should be in lowercase letters.

NOTE User-written functions must be defined before they are referenced, unless they meet the above conditions for automatic compilation. This restriction is necessary in order to distinguish between function calls and subscripted variable references. For more information on compiling functions and procedures, see *Executive Commands* in the PV-WAVE Reference.

GOTO Statement

`GOTO, label`

The GOTO statement is used to transfer program control to the point in the program specified by the label. An example of the GOTO statement is:

```
GOTO, JUMP1
    statements . . .
    .
    .
    .
JUMP1: X = 2000 + Y
```

In the above example, the statement at label JUMP1 is executed after the GOTO statement, skipping intermediate statements. The label may also occur before the reference of the GOTO to that label.

CAUTION Be careful in programming with GOTO statements. It is not difficult to get into a loop that will never terminate if there is not an escape (or test) within the statements spanned by the GOTO (and sometimes even when there is!).

GOTO statements are frequently subjects of IF statements:

```
IF A NEG THEN GOTO, MISTAKE
```

IF Statement

The basic forms of the IF statement are:

`IF expression THEN statement`

IF expression THEN statement ELSE statement

The IF statement is used to execute conditionally a statement or a block of statements.

The expression after the IF is called the condition of the IF statement. This expression (or condition) is evaluated, and if true, the statement following the THEN is executed. If the expression evaluates to a false value the statement following the ELSE clause is executed. Control passes immediately to the next statement if the condition is false and the ELSE clause is not present.

Examples of the IF statement include:

```
IF A NE 2 THEN PRINT, 'A IS NOT TWO'
IF A EQ 1 THEN PRINT, 'A IS ONE' ELSE $
    PRINT, 'A IS NOT ONE'
```

The first example contains no ELSE clause. If the value of A is not equal to 2, A IS NOT TWO is printed. If A is equal to 2, the THEN clause is ignored, nothing is printed, and execution resumes at the next statement. In the second example above, the condition of the IF statement is (A EQ 1). If the value of A is equal to 1, A IS ONE is printed, otherwise NOT ONE is printed.

Definition of True in an IF Statement

The condition of the IF statement may be any scalar expression. The definition of true and false for the different data types is as follows:

- Byte, Integer and Longword — Odd integers are true, even integers are false.
- Floating-point, Double-precision floating-point and Complex — Nonzero values are true, zero values are false. The imaginary part of complex floating numbers is ignored.
- String — Any string with a non-zero length is true, null strings are false.

In the following example, the logical expression is a conjunction of two relational expressions.

```
IF (LON GT -40) AND (LON LE -20) THEN . . .
```

If both conditions — LON being larger than -40 and less than or equal to -20 — are true then the statement following the THEN will be executed.

The THEN and ELSE clauses may also be in the form of a block (or group of statements) with the delimiters BEGIN and END. (See [Blocks of Statements on page 53](#).) To ensure proper nesting of blocks, you may use ENDIF to terminate the block, instead of using the generic END.

Below is an example of the use of blocks within an IF statement.

```

IF (expression) THEN BEGIN
. . .
. . .
. . .
ENDIF ELSE IF (expression) THEN BEGIN
. . .
. . .
. . .
ENDIF ELSE BEGIN
. . .
. . .
. . .
ENDELSE ;End of else clause

```

IF statements may be nested in the following manner:

```

IF P1 THEN S1 ELSE $
IF P2 THEN S2 ELSE $
. . .
. . .
. . .
IF Pn THEN Sn ELSE Sx

```

If condition P1 is true, only statement S1 is executed; if condition P2 is true, only statement S2 is executed, etc. If none of the conditions are true statement Sx will be executed. Conditions are tested in the order they are written. The above construction is similar to the CASE statement except that the conditions are not necessarily related.

Procedure Call Statement

PROCEDURE_NAME, p₁, p₂, ..., p_n

The Procedure Call statement invokes a system, user-written, or externally defined procedure. The parameters which follow the procedure's name are passed to the procedure. Control resumes at the statement following the Procedure Call statement when the called procedure finishes.

Procedures may come from the following sources:

- System procedures built into the PV-WAVE executable file.
- User-written procedures compiled with the .RUN command.

- User-written procedures that are compiled automatically. See [Automatic Compilation of Functions and Procedures on page 61](#).
- Standard Library procedures that are installed with PV-WAVE.

Examples

`ERASE`

This is a procedure call to a subroutine to erase the current window. There are no explicit inputs or outputs. Other procedures have one or more parameters. For example:

`PLOT, Circle`

calls the `PLOT` procedure with the parameter `Circle`.

Positional Parameters and Keyword Parameters

Parameters passed to procedures and functions are identified by their position or by a keyword.

As their name indicates, the position of positional parameters establishes the correspondence of the parameters in the call and those in the definition of the procedure or function.

A keyword parameter is a parameter preceded by a keyword and an equal sign (=) that identifies the parameter.

For example, the `PLOT` procedure can be instructed to not erase the screen and to draw using color index 12 by either of the calls:

`PLOT, X, Y, Noerase = 1, Color = 12`

or:

`PLOT, X, Y, Color = 12, /Noerase`

The two calls produce identical results. Keywords may be abbreviated to the shortest non-ambiguous string. The `/Keyword` construct is equivalent to setting the keyword parameter to the value 1. For example, `/Noerase` is equivalent to `Noerase=1`.

In the above examples, the parameter `X` is the first positional parameter, because it is not preceded by a keyword. `Y` is the second positional parameter.

Calls may mix arguments with and without keywords. The interpretation of keyword arguments is independent of their order. The placement of keyword arguments does not affect the interpretation of positional parameters — keyword parameters may appear before, after, or in the middle of the positional parameters.

Keyword parameters offer the following advantages:

- Procedures and functions may have a large number of arguments, any of which may be optional. Only those arguments that are actually used need be present in the call.
- It is much easier to remember the names of keyword arguments, rather than their order.
- Additional features can be added to existing procedures and functions without changing the meaning or interpretation of other arguments.

More On Parameters

Parameters may be of any type or structure, although some system procedures, as well as user-defined procedures, may require a particular type of parameter for a specific argument.

Parameters may also be expressions which are evaluated, used in the call, and then discarded. For example:

```
PLOT, SIN(Circle)
```

The sine of the array `Circle` is computed and plotted, then the result of the computation is discarded.

Parameters are passed by value or by reference. Parameters that consist of only a variable name are passed by reference. Expressions, constants, and system variables are passed by value. The two passing mechanisms are fundamentally different. The called procedure or function may not return a value in a parameter that is passed by value, as the value of the parameter is evaluated and passed into the called procedure, but is not copied back to the caller. Changes made by the called procedure are passed back to the caller if the parameter is passed by reference. For more details, see [Parameter Passing Mechanism on page 228](#).

Procedure Definition Statement

PRO *name*, *p*₁, *p*₂ ..., *p*_{*n*}

A sequence of one or more statements may be given a name, compiled and saved for future use with the Procedure Definition statement.

Once a procedure has been successfully compiled, it may be executed using a procedure call statement interactively from the `WAVE>` prompt, from a main program, or from another procedure or function.

The general format for the definition of a procedure is, where *name* can be up to 31 characters long:

```
PRO name, param1, ..., paramn
    Statement1,
    Statement2
    . . .
    . . .
    RETURN
END
```

For more information on writing procedures, see [Chapter 9, Writing Procedures and Functions](#).

Calling a user-written procedure that is in a directory in the search path (!Path) causes the procedure to be read from the disk, compiled, saved, and executed, without interrupting program execution.

OpenVMS USERS If you are running under OpenVMS, see [OpenVMS Procedure Libraries on page 233](#) for information on creating libraries of procedures.

REPEAT Statement

REPEAT *subject_statement* UNTIL *condition_expr*

The REPEAT statement repetitively executes its subject statement until a condition is true. The condition is checked after the subject statement is executed. Therefore, the subject statement is always executed at least once.

Below are some examples of the use of the REPEAT statement:

```
A = 1
REPEAT A = A * 2 UNTIL A GT B
```

This code finds the smallest power of 2 that is greater than B. The subject statement may also be in the form of a block, as shown in the following block of code that sorts an array:

```
REPEAT BEGIN
NOSWAP = 1
    ; Init flag to true.
FOR 1 = 0, N - 2 DO IF ARR(I) GT ARR(I + 1)
    THEN BEGIN
        NOSWAP = 0
```

```

        ; Swapped elements, clear flag.
T = ARR (I)
ARR (I) = ARR (I + 1)
ARR (I + 1) = T
ENDFOR
ENDREP UNTIL NOSWAP
        ; Keep going until nothing is moved.

```

The above example sorts the elements of ARR using the inefficient bubble sort method. A more efficient way to sort array elements is to use the SORT function.

NOTE The ending statement for a REPEAT loop is ENDREP, not ENDREPEAT.

WHILE Statement

WHILE expression DO statement

WHILE statements are used to execute a statement repeatedly while a condition remains true. The WHILE statement is similar to the REPEAT statement except that the condition is checked prior to the execution of the statement.

When the WHILE statement is executed, the conditional expression is tested, and if it is true, the statement following the DO is executed. Control then returns to the beginning of the WHILE statement where the condition is again tested. This process is repeated until the condition is no longer true, at which point the control of the program continues at the next statement.

In the WHILE statement, the subject is never executed if the condition is initially false.

Examples of WHILE statements are:

```
WHILE NOT EOF(1) DO READF, 1, A, B, C
```

In this example, data are read until the end-of-file is encountered.

The next example demonstrates one way to find the first point of an array greater than or equal to a selected value assuming the array is sorted in ascending order (the array contains *N* elements):

```

N = N_ELEMENTS (ARR)
    ; Determine number of elements in ARR.
I = 0
    ; Initializes index.

```

```
WHILE (ARR(I) LT X) AND (I LT N)
```

```
DO I = I + 1
```

```
; Increments I until a smaller point is found or the end of  
; the array is reached.
```

Another way to accomplish the same thing is with the statements:

```
P = WHERE (ARR GE X)
```

```
; P is a vector of the array subscripts where ARR(I) GE X.
```

```
I = P (0)
```

```
; Saves first subscript.
```


Using Subscripts with Arrays

Subscripts provide a means of selecting one or more elements of an array variable. The values of one or more selected array elements are extracted when a subscripted variable reference appears in an expression. Values are stored in selected array elements, without disturbing the remaining elements, when a subscript reference appears on the left side of an assignment statement. The section [Assignment Statement on page 48](#) discusses the use of the different types of assignment statements when storing into arrays.

The subscripts of an array element denote the address of the element within the array. In the simple case of a one-dimensional array, an n -element vector, elements are numbered starting at 0 with the first element, 1 for the second element, and running to $n - 1$, the subscript of the last element. Arrays with multiple dimensions are addressed by specifying a subscript expression for each dimension. For example, a two-dimensional $n \times m$ array is addressed with a subscript of the form: (i, j) , where $0 \leq i < n$ and $0 \leq j < m$.

Syntax

The syntax of a subscript reference is:

variable_name (*subscript_list*)

Or:

(*array_expression*) (*subscript_list*)

The subscript list is simply a list of expressions, constants, or subscript ranges which contains the values of the one or more subscripts. Subscript expressions are

separated by commas if there is more than one subscript. In addition, multiple elements are selected with subscript expressions that contain either a contiguous range of subscripts or an array of subscripts.

Subscript Reference Discussion

Subscripts may be used either to retrieve the value of one or more array elements or to designate array elements to receive new values. The expression:

`ARR (12)`

denotes the value of the thirteenth element of `ARR` (because subscripts start at 0), while the statement:

`ARR (12) = 5`

stores the number 5 in the thirteenth element of `ARR` without changing the other elements.

Elements of multidimensional arrays are specified by using one subscript for each dimension. With `PV-WAVE`, like with `FORTTRAN`, the first subscripts vary fastest in memory.

If `A` is a 2-by-3 array, the command: `PRINT, A` prints the array like this:

```
A0,0 A1,0
A0,1 A1,1
A0,2 A1,2
```

On the other hand, the command: `PM, A` prints the array like this:

```
A0,0 A0,1 A0,2
A1,0 A1,1 A1,2
```

But regardless of how the array is printed, the values are stored in memory in the same way: `A0,0`, `A1,0`, `A0,1`, `A1,1`, `A0,2`, `A1,2`. As another example, suppose `B` is a 2 X 2 X 2 array. Then, `B` is stored in memory in the order:

```
B0,0,0, B1,0,0, B0,1,0, B1,1,0, B0,0,1, B1,0,1, B0,1,1, B1,1,1
```

Elements of multidimensional arrays may also be specified using only one subscript, in which case the array is treated as a 1D array with the same number of elements. For instance, in the previous examples, `A (2)` is the same element as `A (0, 1)`, `A (5)` is the same element as `A (1, 2)`, and `B (5)` is the same as `B (1, 0, 1)`.

If an attempt is made to reference a non-existent element of an array using a scalar subscript (a subscript that is negative or larger than the size of the dimension minus 1), an error occurs and program execution stops.

Subscripts may be any type of array or scalar expression. If a subscript expression is not of type integer, a longword integer copy is made and used to evaluate the subscript. For example:

```
A(1.4) = A(1.6) = A(1)
```

Arrays (as well as scalars) can be assigned to an array element referenced by a scalar subscript. For instance: A(S)=ARR, where ARR is an array. In this case, the elements of ARR are stored sequentially into A beginning at the element A(S).

Examples

```
a = INDGEN( 5 ) & a(2) = [ 10, 20 ] & PRINT, a
```

```
0      1      10      20      4
```

```
a = LONARR( 4, 5 ) & a(5) = REPLICATE( 1, 6 ) & PM, a
```

```
0      0      1      0      0
```

```
0      1      1      0      0
```

```
0      1      1      0      0
```

```
0      1      0      0      0
```

```
a = LONARR( 4, 5 ) & a(1,1) = REPLICATE( 1, 2, 3 ) & PM, a
```

```
0      0      0      0      0
```

```
0      1      1      1      0
```

```
0      1      1      1      0
```

```
0      0      0      0      0
```

“Extra” Dimensions

All “degenerate” trailing dimensions of size 1 are eliminated from arrays. Thus, the statements:

```
A = INTARR(10, 5, 5, 1)
INFO, A
```

print the following:

```
A INT = Array(10, 5, 5)
```

This removal of superfluous dimensions is usually convenient, but it can cause problems when attempting to write fully general procedures and functions. There-

fore, you can specify “extra” dimensions for an array as long as the extra dimensions are all zero. For example, consider a vector defined as:

```
ARR = INDGEN(10)
```

The following are all valid references to the 6th element of ARR:

```
ARR(5)
ARR(5, 0)
ARR(5, 0, 0, *, 0)
```

Thus, the automatic removal of degenerate trailing dimensions does not cause problems for routines that attempt to access the resulting array.

Subscripting Scalars

References to scalars may be subscripted. All subscripts must be zero. For example:

```
a = 5
PRINT, a(0)
a(0) = 6
PRINT, a
```

Subscript Ranges

Subscript ranges are used to select a subarray from an array by giving the starting and ending subscripts of the subarray in each dimension.

Subscript ranges may be combined with scalar and array subscripts and with other subscript ranges. Any rectangular portion of an array may be selected with subscript ranges.

There are four types of subscript ranges:

- A range of subscripts, written ($e0 : e1$), denoting all elements whose subscripts range from the expression $e0$ to $e1$. $e0$ must not be greater than $e1$ (but it may equal $e1$).

For example, if the variable VEC is a 50-element vector,

```
VEC(5 : 9)
```

is a 5-element vector composed of

```
[VEC(5), . . . , VEC(9)]
```

- All elements from a given element to the last element of the dimension, written as (*E* : *).

Using the above example,

`VEC(10 : *)`

is a 40-element vector made of

`[VEC(10), ..., VEC(49)]`

- A simple subscript, (*n*). When used with multidimensional arrays, simple subscripts specify only elements with subscripts equal to the given subscript in that dimension.
- All elements of a dimension, written (*). This form is used with multidimensional arrays to select all elements along the dimension.

For example, if `ARR` is a 10-by-12 array,

`ARR(*, 11)`

is a 10-element vector composed of elements

`[ARR(0, 11), ARR(1, 11), ..., ARR(9, 11)]`

Similarly,

`ARR(0, *)`

is the 1-by-12 array,

`[ARR(0, 0), ARR(0, 1), ..., ARR(0, 11)]`

Multidimensional subarrays may be specified using any combination of the above forms. For example,

`ARR(*, 0 : 4)`

is a 10-by-5 array. Or, if `ARR` is a 5 × 10 × 15 × 20 array, then

`ARR(0, 1:2, 3:*, *)` is a 1 × 2 × 12 × 20 array.

Subscript Ranges

Form	Meaning
<i>E</i>	A simple subscript expression
<i>e0</i> : <i>e1</i>	Subscript range from <i>e0</i> to <i>e1</i>
<i>E</i> : *	All points from element <i>E</i> to end
*	All points in the dimension

Structure of Subarrays

The dimensions of an extracted subarray are determined by the size in each dimension of the subscript range. In general, the number of dimensions is equal to the number of subscripts. The size of a dimension is equal to 1 if a simple subscript was used for that dimension; otherwise it is equal to the number of elements selected by the range.

Degenerate dimensions (trailing dimensions whose size is equal to 1) are removed. This was illustrated in the above example by the expression `ARR (*, 11)` which resulted in a vector with a single dimension because the last dimension of the result was 1 and was removed. On the other hand, the expression `ARR (0, *)` became an array with dimensions of (1, 12) because the dimension with a size of 1 does not appear at the end.

Using the examples of `VEC`, a 50-element vector, and `A`, a 10-by-12 array, some typical subscript range expressions are:

```
VEC(5 : 10)
    ; Points 5 to 10 of VEC, a 6-element vector.

VEC(I - 1 : I + 1)
    ; 3-point neighborhood around I: [VEC(I - 1), VEC(I), VEC(I + 1)].

VEC(4 : *)
    ; Points in VEC from VEC(4) to the end, a 50 - 4 = 46-element
    ; vector.

A(3, *)
    ; A 1-by-12 array: [A(3, 0), A(3, 1), ..., A(3, 11)].

A(*, 0)
    ; A 10-element vector.

A(X - 1 : X + 1, Y - 1 : Y + 1)
    ; The 9-point neighborhood surrounding A(X,Y), a 3-by-3 array.

A(3 : 5, *)
    ; A 3-by-12 subarray.
```

One-dimensional range subscripts can be used with multidimensional arrays. For example, if `A` is a `2 X 2 X 2 X 2 X 2` array, then `A (*)` is a 32-element vector containing all the elements of `A`, and `A (5 : *)` is a vector containing the last 27 elements of `A`.

Arrays as well as scalars can be assigned to array elements referenced by range subscripts.

Examples

```
a = FLTARR( 5, 5 ) & a(*) = 1 & a(0:3,1:*) = 2 & PM, a
```

```
1.00000 2.00000 2.00000 2.00000 2.00000
1.00000 2.00000 2.00000 2.00000 2.00000
1.00000 2.00000 2.00000 2.00000 2.00000
1.00000 2.00000 2.00000 2.00000 2.00000
1.00000 1.00000 1.00000 1.00000 1.00000
```

```
a = FLTARR( 2, 4 ) & a(*) = INDGEN( 8 ) & PM, a
```

```
0.00000 2.00000 4.00000 6.00000
1.00000 3.00000 5.00000 7.00000
```

```
a = DBLARR( 3, 3, 3 ) & a(*,1:*,0) = INDGEN( 6 ) & PM, a
```

```
0.0000000 0.0000000 3.0000000
0.0000000 1.0000000 4.0000000
0.0000000 2.0000000 5.0000000

0.0000000 0.0000000 0.0000000
0.0000000 0.0000000 0.0000000
0.0000000 0.0000000 0.0000000

0.0000000 0.0000000 0.0000000
0.0000000 0.0000000 0.0000000
0.0000000 0.0000000 0.0000000
```

```
a(*,*,2) = REPLICATE( 1, 3, 3 ) & PM, a
```

```
0.0000000 0.0000000 3.0000000
0.0000000 1.0000000 4.0000000
0.0000000 2.0000000 5.0000000

0.0000000 0.0000000 0.0000000
0.0000000 0.0000000 0.0000000
0.0000000 0.0000000 0.0000000

1.0000000 1.0000000 1.0000000
1.0000000 1.0000000 1.0000000
1.0000000 1.0000000 1.0000000
```

```
a(0:0,2:2,2:*) = 2 & PM, a
```

```
0.0000000 0.0000000 3.0000000
0.0000000 1.0000000 4.0000000
0.0000000 2.0000000 5.0000000

0.0000000 0.0000000 0.0000000
0.0000000 0.0000000 0.0000000
0.0000000 0.0000000 0.0000000
```

```

1.0000000    1.0000000    2.0000000
1.0000000    1.0000000    1.0000000
1.0000000    1.0000000    1.0000000

```

See the section [Assignment Statement on page 48](#) for more information describing the assigning of values to subarrays.

Arrays as Subscripts to Other Arrays

Arrays may be used to subscript other arrays. Each element in the array used as a subscript selects an element in the subscripted array. When used with subscript ranges, more than one element is selected for each subscript element.

If no subscript ranges are present, the length and structure of the result is the same as that of the subscript expression. The type of the result is the same as that of the subscripted array. If only one subscript is present, all subscripts are interpreted as if the subscripted array has one dimension.

In the simple case of a single subscript, S , which is a vector, the process may be written as:

$$A(S) = \left\{ \begin{array}{l} A_{S_i} \text{ if } 0 \leq S_i < n \\ A_0 \text{ if } S_i < 0 \\ A_{n-1} \text{ if } S_i \geq n \end{array} \right\} \text{ (for } 0 \leq i < m \text{)}$$

assuming that the array A has n elements, and S has m elements. The result $A(S)$ has the same structure and number of elements as does the subscript vector S . Just as with scalar subscripts and range subscripts, array subscripts can represent one-dimensional indices into multidimensional arrays; thus, the dimensionality of A is arbitrary.

If an element of the subscript array is less than or equal to zero, the first element of the subscripted variable is selected. If an element of the subscript array is greater than or equal to the last subscript in the subscripted variable (N , above), the last element is selected.

Example

```
A = [6, 5, 1, 8, 4, 3]
```

```
B = [0, 2, 4, 1, -1, 10]
```

```
C = A(B)
PRINT, C
      6   1   4   5   6   3
```

The first element is 6 because it is in the zero position of A. The second is 1 because the value in B of 2 indicates the third position in A, and so on. The last two elements of C are the endpoints of A, because the last two subscripts of B are out of range.

As another example, assume the variable A is a 10-by-10 array. The expression:

```
A(INDGEN(10) * 11)
```

yields a 10-element vector equal to the diagonal elements of A. The one dimensional subscripts of the diagonal elements, $A_{0,0}$, $A_{1,1}$, ..., $A_{9,9}$ are 0, 11, 22, ..., 99 (the same as elements of the vector `INDGEN(10) * 11`).

The WHERE function, which returns a vector of subscripts, may be used to select elements of an array using expressions similar to:

```
A(WHERE(A GT 0))
```

which results in a vector composed only of the elements of A that are greater than 0.

Combining Array Subscripts with Others

Array subscripts may be combined with:

- Subscript ranges
- Simple scalar subscripts
- Other array subscripts

When it encounters a multidimensional subscript that contains one or more subscript arrays, PV-WAVE builds an array of subscripts by processing each subscript, from left to right. The resulting array of subscripts is then applied to the variable that is to be subscripted.

As with other subscript operations, trailing degenerate dimensions (those with a size of 1) are eliminated.

Combining Array Subscripts with Scalar or Range Subscripts

When combining an n -element subscript array with an m -element subscript range, the resulting subarray is of dimension $n \times m$.

For example, the expression `A([1, 3], 5)` yields the vector $[A_{1,5}, A_{3,5}]$, and the expression `A([1, 3, 5], 7 : 9)` yields a 3-by-3 array composed of the elements:

$$\begin{bmatrix} A_{1,7} & A_{1,8} & A_{1,9} \\ A_{3,7} & A_{3,8} & A_{3,9} \\ A_{5,7} & A_{5,8} & A_{5,9} \end{bmatrix}$$

Each element of the 3-element subscript array (1, 3, 5) is combined with each element of the 3-element range (7, 8, 9).

Examples

The common process of zeroing the edge elements of a two-dimensional n -by- m array is:

```
A(*, [0, M - 1]) = 0
```

```
A([0, N - 1], *) = 0
```

For another example of combining array and range subscripts, consider:

```
A = DBLARR( 5, 10, 5, 10, 5 )
```

```
B = [-1, 0, 5, 3.9]
```

```
INFO, A(B, *, 2:*, 1:3, 0)
```

```
<Expression>    DOUBLE    = Array(4, 10, 3, 3)
```

Combining with Other Subscript Arrays

If all subscripts are arrays, then all these arrays must have the same number of elements; in this case, each element of the first subscript array is combined with the corresponding elements of the other subscript arrays.

For example:

```
a=FINDGEN(6,6) & PM, a([0,2,4], [1,3,5])
```

```
6.00000 (= A0,1)
```

```
20.0000 (= A2,3)
```

```
34.0000 (= A4,5)
```

Or, if A is a 3D array, then:

```
A([0,2], [1,3], [0,2]) = 10
```

assigns the value 10 to the elements A_{010}, A_{232} .

If multiple array subscripts are mixed with scalars or ranges, then the resulting subscript array is the Cartesian product of all of the subscripts. For example, if A is a 3D array, then the expression $A([0, 2, 3], [1, 3], 0)$ yields the 2D array:

$$\begin{bmatrix} A_{0,1,0} & A_{0,3,0} \\ A_{2,1,0} & A_{2,3,0} \\ A_{3,1,0} & A_{3,3,0} \end{bmatrix}$$

Also, note that since extra “0 dimensions” are allowed, a 2D array A can be subscripted with the Cartesian product of two subscript arrays. For example, the expression $A([0, 2, 3], [1, 3], 0)$ yields the 2D array:

$$\begin{bmatrix} A_{0,1} & A_{0,3} \\ A_{2,1} & A_{2,3} \\ A_{3,1} & A_{3,3} \end{bmatrix}$$

Storing Elements with Array Subscripts

One or more values may be stored in selected elements of an array by using an array expression as a subscript for the array variable appearing on the left side of an assignment statement. Values are taken from the expression on the right side of the assignment statement and stored in the elements whose subscripts are given by the array subscript. The right-hand expression may be either a scalar or array.

See [Assignment Statement on page 48](#) for details and examples of storing with vector subscripts.

Examples

$$A([2, 4, 6]) = 0$$

zeroes elements $A(2)$, $A(4)$, and $A(6)$, without changing other elements of A . The following statement:

$$A([2, 4, 6]) = [4, 16, 36]$$

is equivalent to the statements:

$$A(2) = 4$$

$$A(4) = 16$$

$$A(6) = 36$$

One way to create a square n -by- n identity matrix is:

```
A = FLTARR(N, N)
A(INDGEN(N) * (N + 1)) = 1.0
```

The expression `INDGEN(N) * (N + 1)` results in a vector containing the 1D subscripts of the diagonal elements. Yet another way is to use two array subscripts:

```
A = FLTARR(N, N)
A(INDGEN(N), INDGEN(N)) = 1.0
```

The statement:

```
A(WHERE(A LT 0)) = -1
```

sets negative elements of `A` to minus 1.

Consider also the following examples:

```
a = INTARR( 4, 4, 4 ) & a([0,2,3],[1,3],0:1) = 10 & PM, a
```

```
0  10  0  10
0  0  0  0
0  10  0  10
0  10  0  10
```

```
0  10  0  10
0  0  0  0
0  10  0  10
0  10  0  10
```

```
0  0  0  0
0  0  0  0
0  0  0  0
0  0  0  0
```

```
0  0  0  0
0  0  0  0
0  0  0  0
0  0  0  0
```

```
a = INTARR( 4, 4 ) & a([0,2,3],[1,3],0) = INDGEN(6) & PM, a
```

```
0  0  0  3
0  0  0  0
0  1  0  4
0  2  0  5
```

```
a = INTARR( 4, 4 ) & a(INDGEN(4), INDGEN(4)) = 10 & PM, A
```

```
10  0  0  0
0  10  0  0
0  0  10  0
0  0  0  10
```

Memory Order

To facilitate optimum performance, it is useful to know the memory order of the elements in the array. Given a 2-by-3 array created with the statement

```
a = INTARR (2, 3)
```

the elements of A are ordered in memory as:

$$A_{0,0}, A_{1,0}, A_{0,1}, A_{1,1}, A_{0,2}, A_{1,2}$$

Similarly, in arrays of dimension higher than two, the elements are stored such that the first dimension varies fastest, the next dimension varies the next fastest, and so on. For more information, see [Subscript Reference Discussion on page 72](#).

Knowledge of the memory order is also important when attempting to subscript multidimensional arrays with a single subscript, in which case the array is treated as a vector with the same number of elements. In the above example, $A(2)$ is the same element as $A(0,1)$ and $A(5)$ is the same element as $A(1,2)$.

Matrices

If A is an $m \times n$ array, the command

```
PRINT, A
```

yields:

$$\begin{bmatrix} A_{0,0} & \dots\dots & A_{m-1,0} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ A_{0,n-1} & \dots\dots & A_{m-1,n-1} \end{bmatrix}$$

The fact that the array is printed this way may disturb those who are used to the linear algebra convention for listing a matrix. For this reason PV-WAVE is equipped with a set of input/output routines that subscribe to the linear algebra convention. RMF and PMF read and write files according to the linear algebra convention, and RM and PM are the interactive versions of RMF and PMF. For instance, the command:

```
PM, A
```

yields:

$$\begin{bmatrix} A_{0,0} & \dots & A_{0,n-1} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ A_{m-1,0} & \dots & A_{m-1,n-1} \end{bmatrix}$$

NOTE Regardless of how the array is read in or printed out, memory storage order is unaffected. Thus, the distinction between arrays and matrices in PV-WAVE is completely superficial.

Reading and Printing Matrices Interactively

Matrices can be entered interactively using the RM procedure and printed to the screen using PM (see the PV-WAVE Reference). In this example, a matrix is interactively entered and printed along with its inverse.

```
RM, a, 3, 3
    ; Enter 3 by 3 matrix A.

row 0: 3 1 2
row 1: 4 5 1
row 2: 7 3 9
    ; User is prompted to enter the rows of the matrix.
```

```
PM, a
    ; Print the matrix.

    3.00000    1.00000    2.00000
    4.00000    5.00000    1.00000
    7.00000    3.00000    9.00000
```

```
PM, INVERT(a)
    ; Print the inverse of A.

    0.823530   -0.0588235   -0.176471
   -0.568628    0.254902    0.0980392
   -0.450980   -0.0392157    0.215686
```

The matrix multiplication operator is “#”. For instance

```
RM, p, 4, 2
row 0: 2 4
row 1: 1 3
```

```

row 2: 5 6
row 3: 0 7
      ; Enter 4 by 2 matrix P.

RM, q, 2, 3
      ; Enter 2 by 3 matrix Q.

row 0: 1 3 5
row 1: 2 4 6

PM, p # q
      ; Print the matrix product of P and Q.

10.0000      22.0000      34.0000
 7.00000     15.0000     23.0000
17.0000      39.0000     61.0000
14.0000      28.0000     42.0000

```

Matrices also can be entered elementwise, starting with the (0, 0) subscript. As is standard in mathematics, the first subscript refers to the row and the second to the column. For example:

```

w = FLTARR(3, 3)
      ; Allocate w to be a 3 by 3 float array.

w(0, 0) = 1
w(0, 1) = 2
w(0, 2) = 3
w(1, 0) = 4
w(1, 1) = 5
w(1, 2) = 6
w(2, 0) = 7
w(2, 1) = 8
w(2, 2) = 9
      ; Assign values to w.

PM, w
      ; Print W as a matrix.

1.00000      2.00000      3.00000
4.00000      5.00000      6.00000
7.00000      8.00000      9.00000

PRINT, w

1.00000      4.00000      7.00000
2.00000      5.00000      8.00000
3.00000      6.00000      9.00000
      ; Print W as an array. Note that it is the transpose of the previous statement.

```

In a matrix, the elements are stored columnwise; i.e., the elements of the 0-th column are first, followed by the elements of the 1-st row, etc. Continuing the above example, the elements in the 0-th column (1, 4, 7) come first, followed by those in the 1-st column (2, 5, 8), etc.

```
FOR k = 0, 8 DO PRINT, k, w(k)
      0      1.00000
      1      4.00000
      2      7.00000
      3      2.00000
      4      5.00000
      5      8.00000
      6      3.00000
      7      6.00000
      8      9.00000
```

Reading a Matrix From a File

In this example, the RMF procedure is used to read a matrix contained in an external file. The file `cov.dat` contains the following data:

```
1.0   0.523 0.395 0.471 0.346 0.426 0.576 0.434 0.639
0.523 1.0   0.479 0.506 0.418 0.462 0.547 0.283 0.645
0.395 0.479 1.0   0.355 0.27  0.254 0.452 0.219 0.504
0.471 0.506 0.355 1.0   0.691 0.791 0.443 0.285 0.505
0.346 0.418 0.27  0.691 1.0   0.679 0.383 0.149 0.409
0.426 0.462 0.254 0.791 0.679 1.0   0.372 0.314 0.472
0.576 0.547 0.452 0.443 0.383 0.372 1.0   0.385 0.68
0.434 0.283 0.219 0.285 0.149 0.314 0.385 1.0   0.47
0.639 0.645 0.504 0.505 0.409 0.472 0.68  0.47  1.0
```

After reading the matrix, principal components are computed for a nine-variable covariance matrix. (This example uses the PV-WAVE:IMSL Statistics PRINC_COMP function.)

```
OPENR, unit, 'cov.dat', /Get_Lun
RMF, unit, covariances, 9, 9
CLOSE, unit

values = PRINC_COMP(covariances)

PM, values, Title = "Eigenvalues:"

Eigenvalues:
      4.67692
      1.26397
```

```
0.844450
0.555027
0.447076
0.429125
0.310241
0.277006
0.196197
```

Printing a Matrix to a File

This example retrieves a statistical data set using the PV-WAVE:IMSL Statistics function `STATDATA`, then outputs the matrix to the file `stat.dat`.

```
stats = STATDATA(5)
    ; Get the data from STATDATA.

PM, stats
    ; Print the 13 by 5 matrix to standard output.

7.00000 26.0000 6.00000 60.0000 78.5000
1.00000 29.0000 15.0000 52.0000 74.3000
11.0000 56.0000 8.00000 20.0000 104.300
11.0000 31.0000 8.00000 47.0000 87.6000
7.00000 52.0000 6.00000 33.0000 95.9000
11.0000 55.0000 9.00000 22.0000 109.200
3.00000 71.0000 17.0000 6.00000 102.700
1.00000 31.0000 22.0000 44.0000 72.5000
2.00000 54.0000 18.0000 22.0000 93.1000
21.0000 47.0000 4.00000 26.0000 115.900
1.00000 40.0000 23.0000 34.0000 83.8000
11.0000 66.0000 9.00000 12.0000 113.300
10.0000 68.0000 8.00000 12.0000 109.400
    ; Print the 13 by 5 matrix to a file.

OPENW, unit, 'stat.dat', /Get_Lun
PMF, unit, stats
    ; Use PMF to output the matrix.

CLOSE, unit
    ; Close the file.
```

Subarrays

Using subscript ranges, it is possible to extract submatrices. For instance, the 0-th and 2-nd row of matrix w are extracted by using the following statements:

```

PM, w
; Print W as a matrix.
1.00000      2.00000      3.00000
4.00000      5.00000      6.00000
7.00000      8.00000      9.00000

PM, w([0, 2], *)
1.00000      2.00000      3.00000
7.00000      8.00000      9.00000

```

Matrix Expressions

Complicated matrix expressions are possible. Using the matrices defined above, the following statements compute the inverse of a :

```

PM, a
; Print the matrix.
3.00000      1.00000      2.00000
4.00000      5.00000      1.00000
7.00000      3.00000      9.00000

PM, a # INVERT(a)
; AA-1 should be identity. Error due to round off.
1.00000      0.00000      0.00000
1.19209e-07  1.00000      -1.19209e-07
9.53674e-07  -2.98023e-08  1.00000

```

In the following code segment, $(3.5A + W)(Q^TQ)$ is computed:

```

PM, q
1.00000      3.00000      5.00000
2.00000      4.00000      6.00000
; Compute and print (3.5A + W)(QTQ).
PM, (3.5 * a + w) # (TRANSPPOSE(q) # q)
288.000      654.000      1020.00
499.000      1131.00     1763.00
1049.50      2388.50     3727.50

```

Working with Structures

Introduction to Structures

PV-WAVE supports structures and arrays of structures. A structure is a collection of scalars, arrays, or other structures contained in a variable. Structures are useful for representing data in a natural form, for transferring data to and from other programs, and for containing a group of related items of various types.

Before a structure can be used, it must be defined. When you define a structure, you actually create a new data type. The definition includes a structure name and a list of structure fields. Each structure field is given a tag name and tag definition (data type). The tag definition may be an expression or a variable. It defines the data type of the data that can be placed in the field. A structure definition, per se, does not contain any data values; however, a variable of a particular structure type always contains data.

A structure field may be defined as any type of data representable by PV-WAVE. Fields may contain scalars, arrays of the eight basic data types, and even other structures or arrays of structures.

Just as you cannot alter the basic definition of an integer or floating-point data type in PV-WAVE, you cannot alter a structure definition after it has been created. You can, however, delete a structure definition as long as it is not currently being referenced by any variables. See the next section for more information on deleting structure definitions.

When structure definitions are referred to, they must be enclosed in braces. For example:

```
PRINT, {struct_name}
```

The braces distinguish structure definitions from variable names, function names, or other identifiers.

Defining and Deleting Structures

A structure is created by executing a structure definition expression. This is an expression of the following form:

$$\{ \textit{Structure_name}, \textit{Tag_name}_1, : \textit{Tag_def}_1, \dots : \dots, \\ \textit{Tag_name}_n : \textit{Tagdef}_n \}$$

Tag names must be unique within a given structure, although the same tag name may be used in more than one structure. Structure and tag names follow the same rules as all PV-WAVE identifiers: they must begin with a letter, following characters may be letters, digits, or the underscore or dollar sign characters, and case is ignored.

As mentioned above, each tag definition is a constant, variable, or expression whose type and dimension defines the type and dimension of the field. The result of a structure definition expression is a structure definition that is global in scope and can be used to create variables of the particular structure type.

A structure that has already been defined may be referred to by simply enclosing the structure's name in braces:

$$\text{variable} = \{ \textit{Structure_name} \}$$

The variable created as a result of this command is a structure of the designated name with all of its fields filled with zeros or null strings.

The variable created by the above statement and the structure definition $\{ \textit{Structure_name} \}$ are separate entities. The variable is said to be of type $\{ \textit{Structure_name} \}$. The definition $\{ \textit{Structure_name} \}$ is analogous to any data type, such as integer or double. Just as any number of values can be of type integer, any number of variables may reference a given structure definition.

When referring to a structure definition, the tag names need not be present, as in:

$$\text{variable} = \{ \textit{Structure_name}, \textit{expr}_1, \dots, \textit{expr}_n \}$$

All of the expressions are converted to the type and dimension of the original tag definition. If a structure definition of the first form (where the tag names are present) is executed and the structure already exists, each tag name and the struc-

ture of each tag field definition must agree with the original definition or an error will result.

Example of Defining a Structure

Assume a star catalog is to be processed. Each entry for a star contains the following information: Star name, right ascension, declination, and an intensity measured each month over the last 12 months. A structure for this information is defined with the statement:

```
STAR = { CATALOG, NAME: '', RA: 0.0, $
        DEC: 0.0, INTEN: FLTARR(12) }
```

This structure definition is the basis for all examples in this chapter.

The above statement defines a structure type named CATALOG in a variable named STAR, which contains four fields. The tag names are NAME, RA, DEC, and INTEN. The first field, with the tag NAME, contains a scalar string as given by its tag definition; the following two fields each contain floating-point scalars, and the fourth field, INTEN, contains a 12-element floating-point array. Note that the type of the constants, 0.0, is floating point. If the constants had been written as 0 the fields RA and DEC would contain integers.

Defining a Structure within a Structure

The following example shows how to embed or nest a structure within another structure definition.

```
STAR = {CATALOG, NAME:'', RA=0.0}
        ; Create structure, STAR, of type CATALOG.
STAR2 = {CATALOG2, POS:0.0, DEC:0}
        ; Create a second structure, STAR2, of type CATALOG2.
ALL = {TOTAL, TAG1:{CATALOG}, TAG2:STAR2}
        ; Create a third structure ALL which contains the previously defined
        ; structures as fields. Note that the tag definition can be either the
        ; name of a structure definition ({CATALOG}) or a variable of type
        ; structure (STAR2).
```

Deleting a Structure Definition

The DELSTRUCT procedure lets you delete a structure definition, as long as the structure definition is not referenced by any variables. To determine if a structure definition is referenced, use the STRUCTREF procedure. Variables that are local to a procedure or function can be deleted only by exiting the procedure or function.

You can delete variables at the `$MAIN$` level with the `DELVAR` procedure. Because structure definitions can include other structure definitions, the parent structure definition must be deleted before any nested structure definitions can be deleted.

Deleting a structure definition frees all the memory used to store the structure name, the tag names, and the information about the data type of each structure element. If you want to delete a structure to free memory, then you must delete all referenced variables as well. However, if you simply want to reuse the structure name, then you do not have to delete all the referenced variables. Use the *Rename* keyword with the `DELSTRUCT` procedure. This changes the name of the structure to a new unique name and frees the original name for reuse. This new name is chosen by the system. You cannot specify the name directly. All variables that referenced the original structure name will automatically reference the new name.

For more information on `DELSTRUCT` and `STRUCTREF`, see the PV-WAVE Reference.

Creating Unnamed Structures

As noted previously, a typical structure definition consists of a name and a list of fields. You can also create a structure that you do not name.

Unnamed structures are useful if you:

- do not want to use a structure definition globally.
- do not want to invent new names for structure definitions.
- want the structure definition to be deleted automatically when it is no longer referenced.
- want to create a structure-type variable that contains an array field that can vary.

Scope of Named and Unnamed Structures

Named structure definitions are global in scope. A named structure definition is created only once and then can be referenced by any number of variables. It is important to note that a named structure definition is not associated directly with any particular variable.

An unnamed structure, on the other hand, is closely associated with a specific variable. When the variable that is associated with an unnamed structure is deleted, so is the unnamed structure definition.

Syntax of an Unnamed Structure Definition

The syntax of an unnamed structure definition is:

$$x = \{, tag_name_1: tag_def_1, tag_name_n: tag_def_n\}$$

The data type of variable x references the unnamed structure definition. Unlike named structure definitions, when all variables that reference an unnamed structure definition are deleted, the unnamed structure definition is also deleted. If you copy a variable that references an unnamed structure definition (e.g., $y = x$), then both variables reference the same unnamed structure definition. Only when both variables are deleted will the unnamed structure definition be deleted.

Creating Variable-length Array Fields

The unnamed structure definition can be useful if you want to create a structure definition that contains array fields whose lengths can change. For example, suppose you want to create several variables that have the same structure except that one element is an array that you want to have different lengths for different variables. Using named structures, you would have to create a different structure for each case (because named structure definitions cannot be altered). For example:

```
a={structa, xdim:2, ydim:4, arr:intarr(2,4)}
b={structb, xdim:2, ydim:8, arr:intarr(2,8)}
```

However, the unnamed structure allows you to solve this problem. For example, the following function returns a structure-type variable whose tag names are the same, but whose array length is different for each variable:

```
function my_struct, x, y
  RETURN, { , xdim:x,ydim:y, array:intarr(x,y)}
END
```

Now, you can create a and b as follows:

```
a = my_struct(2, 4)
b = my_struct(2, 8)
```

Internal Names of Unnamed Structures

PV-WAVE generates a name internally for an unnamed structure definition. This name always begins with a \$. This ensures that an unnamed structure definition will never conflict with a named structure definition (because identifiers cannot begin with \$).

The INFO command lets you see this internal name:

```
INFO, a, /Struct
*** Structure $2, 3 tags, 20 length:
    XDIM INT      2
    YDIM INT      4
    ARRAY INT     Array(2, 4)
```

CAUTION Do not attempt to use the internal name for an unnamed structure in any other command. For example:

```
c = {$2}
or
PRINT, STRUCTREF({$2})
```

In these cases, the \$ character is interpreted as a line continuation character. The remainder of the line after \$ is ignored, and PV-WAVE waits for you to enter the rest of the command on the next line. No error message is displayed until you enter another line that does not contain a \$.

Structure References

The basic syntax of a reference to a field within a structure is:

Variable_name . Tag_name

Variable_name must be a variable that contains a structure;

Tag_name is the name of the field and must exist for the structure.

If the field referred to by the tag name is itself a structure, the tag name may optionally be followed by one or more additional tag names. For example:

```
VAR . TAG1 . TAG2
```

This nesting of structure references may be continued up to ten levels. Each tag name, except possibly the last, must refer to a field that contains a structure.

Subscripted Structure References

In addition, a subscript specification may be appended to the variable or tag names if the variable is an array of structures, or if the field referred to by the tag contains an array:

Variable_name . Tag_name(Subscripts)

Variable_name(Subscripts) . Tag_name ...

or

$$\text{Variable_name}(\text{Subscripts}) . \text{Tag_name}(\text{Subscripts})$$

Each subscript is applied to the variable or tag name it immediately follows.

The syntax and meaning of the subscript specification is similar to simple array subscripting: it may contain a simple subscript, array of subscripts, or a subscript range. See Chapter 6, *Using Subscripts*, for more information about subscripts.

If a variable or field containing an array is referenced without a subscript specification, all elements of the item are affected. Similarly, when a variable that contains an array of structures is referenced without a subscript but with a tag name, the designated field in all array elements is affected.

The complete syntax of references to structures is:

$$\text{Structure_ref} := \text{Variable_name} [(\text{subscripts})] . \text{Tags}$$
$$\text{Tags} := [\text{Tags} .] \text{Tag}$$
$$\text{Tag} := \text{Tag_name} [(\text{subscripts})]$$

Optional items are enclosed in square brackets, []. For example, all of the following are valid structure references:

A.B

A.B(N, M)

A(12).B

A(3:5).B(*, N)

A(12).B.C(X, *)

The semantics of storing into a structure field using subscript ranges are slightly different than that of simple arrays. This is because the dimension of arrays in fields is fixed. See [Storing into Structure Array Fields on page 97](#).

Examples of Structure References

The name of the star contained in STAR is referenced as STAR.NAME, the entire intensity array is referred to as STAR.INTEN, while the *n*th element of STAR.INTEN is STAR.INTEN(N). The following are valid statements using the CATALOG structure:

```
STAR = {CATALOG, NAME: 'SIRIUS', RA: 30., $
        DEC: 40., INTEN: INDGEN(12)}
; Store a structure of type CATALOG into variable STAR. Define
; the values of all fields.
```

```

STAR.NAME = 'BETELGEUSE'
    ; Set name field. Other fields remain unchanged.
PRINT, STAR.NAME, STAR.RA, STAR.DEC
    ; Print name, right ascension, and declination.
Q = STAR.INTEN(5)
    ; Set Q to the value of the 6th element of STAR.INTEN. Q will be a floating-point scalar.
STAR.RA = 23.21
    ; Set RA field to 23.21.
STAR.INTEN = 0
    ; Zero all 12 elements of intensity field. Because the type and size of
    ; STAR.INTEN are fixed by the structure definition, the semantics of
    ; assignment statements are somewhat different than with normal variables.
B = STAR.INTEN(3:6)
    ; Store 4th through 7th elements of INTEN field in variable B.
STAR.NAME = 12
    ; The integer 12 is converted to string and stored in the name field
    ; because the field is defined as a string.
MOON = STAR
    ; Copy STAR to MOON. The entire structure is copied and MOON
    ; contains a CATALOG structure.

```

Using INFO with Structures

Use `INFO, /Structure` to determine the type, structure, and tag name of each field in a structure. In the example above, a structure was stored into variable `STAR`. The statement:

```
INFO, /Structure, STAR
```

prints the following information:

```

** Structure CATALOG, 4 tags, 60 length:
NAME           STRING   '(null) '
RA             FLOAT    0.0
DEC           FLOAT    0.0
INTEN         FLOAT    Array(12)

```

Calling `INFO` with the *Structure* keyword and no parameters prints a list of all defined structures and tag names. In addition to the *Structure* keyword, the *Userstruct* and *Sysstruct* `INFO` keywords can also be used to obtain information about structures. See [Chapter 12, Getting Session Information](#), for information on these keywords.

Parameter Passing with Structures

As explained in [Parameter Passing Mechanism on page 228](#), PV-WAVE passes simple variables by *reference* and everything else by *value*.

An entire structure is passed by *reference* by simply using the name of the variable containing the structure as a parameter. Changes to the parameter within the procedure are passed back to the caller.

Fields within a structure are passed by value. For example, to print the value of `STAR.NAME`:

```
PRINT, STAR.NAME
```

Any reference to a structure with a subscript or tag name is evaluated into an expression, hence `STAR.NAME` is an expression and is passed by value. This works as expected unless the called procedure returns information in the parameter, as in the call to `READ`:

```
READ, STAR.NAME
```

which *does not* read into `STAR.NAME`, but interprets its parameter as a prompt string. The proper code to read into the field is:

```
B = STAR.NAME  
    ; Copy type and attributes to variable.
```

```
READ, B  
    ; Read into a simple variable.
```

```
STAR.NAME = B  
    ; Store result into field.
```

Storing into Structure Array Fields

As was mentioned above, the semantics of storing into structure array fields is slightly different than storing into simple arrays. The main difference is that with structures a subscript range must be used when storing an array into part of an array field. With normal arrays, when storing an array inside part of another array, use the subscript of the lower-left corner, not a range specification.

Other differences occur because the size and type of a field are fixed by the original structure definition and the normal PV-WAVE semantics of dynamic binding are not applicable.

The rules for storing into array fields are:

Rule 1

$\text{VAR.TAG} = \text{scalar_expr}$

The field TAG is an array. All elements of VAR .TAG are set to *scalar_expr*. For, example:

```
STAR.INTEN = 100
; Sets all 12 elements of STAR.INTEN to 100.
```

Rule 2

$\text{VAR.TAG} = \text{array_expr}$

Each element of *array_expr* is copied to the array VAR.TAG. If *array_expr* contains more elements than does the destination array an error results. If it contains fewer elements than VAR.TAG, the unmatched elements remain unchanged. Example:

```
STAR.INTEN = FINDGEN(12)
; Sets STAR.INTEN to the 12 numbers 0, 1, 2, ..., 11.
STAR.INTEN = [1, 2]
; Sets STAR.INTEN(0) to 1 and STAR.INTEN(1) to 2. The
; other elements remain unchanged.
```

Rule 3

$\text{VAR.TAG}(\text{subscript}) = \text{scalar_expr}$

The value of the scalar expression is simply copied into the designated element of the destination. If *subscript* is an array of subscripts, the scalar expression is copied into the designated elements. Example:

```
STAR.INTEN(5) = 100
; Sets the 6th element of STAR.INTEN to 100.
STAR.INTEN([2, 4, 6]) = 100.
; Sets elements 2, 4, and 6 to 100.
```

Rule 4

$\text{VAR.TAG}(\text{subscript}) = \text{array_expr}$

Unless VAR.TAG is an array of structures, the subscript must be an array. Each element of *array_expr* is copied into the element of VAR.TAG given by the corresponding element *subscript*. Example:

```
STAR.INTEN([2, 4, 6]) = [5, 7, 9]
; Sets elements 2, 4, and 6 to the values 5, 7, and 9.
```

Rule 5

VAR .TAG (*subscript_range*) = *scalar_expr*

The value of the scalar expression is stored into each element specified by the subscript range. Example:

```
STAR.INTEN(8 : *) = 5
; Sets elements 8, 9, 10, and 11, to the value 5.
```

Rule 6

VAR.TAG (*subscript_range*) = *array_expr*

Each element of the array expression is stored into the element designated by the subscript range. The number of elements in the array expression must agree with the size of the subscript range. Example:

```
STAR.INTEN(3 : 6) = findgen(4)
; Sets elements 3, 4, 5, and 6 to the numbers 0, 1, 2, and
; 3, respectively.
```

See [Creating Variable-length Array Fields on page 93](#) for information on placing variable-length arrays in structures.

Creating Arrays of Structures

An array of structures is simply an array in which each element is a structure of the same type. The referencing and subscripting of these arrays (also called *structure arrays*) follow essentially the same rules as simple arrays.

The easiest way to create an array of structures is to use the REPLICATE function. The first parameter to REPLICATE is a reference to the structure of each element. Using the above example of a star catalog and assuming the CATALOG structure has been defined, an array which contains 100 elements of the structure is created with the statement:

```
CAT = REPLICATE({ CATALOG }, 100)
```

Alternatively, since the variable STAR contains an instance of the structure CATALOG:

```
CAT = REPLICATE(STAR, 100)
```

Or, to define the structure and an array of the structure in one step:

```
CAT = REPLICATE({ CATALOG, NAME : '', RA: 0.0, $
DEC : 0.0, INTEN : FLTARR(12) }, 100)
```

The concepts and combinations of subscripts, subscript arrays, subscript ranges, fields, nested structures, etc., are general and lead to many possibilities, only a small number of which can be explained here. In general what seems reasonable usually works.

Examples of Arrays of Structures

Using the above definition in which the variable CAT contains a star catalog of CATALOG structures:

```
CAT.NAME = 'EMPTY'
; Set the NAME field of all 100 elements to EMPTY.

CAT(I) = {CATALOG, 'BETELGEUSE', 12.4, $
54.2, FLTARR(12)}
; Set the ith element of CAT to the contents of the CATALOG structure.

CAT.RA = INDGEN(100)
; Store a 0.0 into CAT(0).RA, 1.0 into CAT(1).RA, ..., 99.0 into CAT(99).RA.

PRINT, CAT.NAME + ', '
; Prints name field of all 100 elements of CAT, separated by commas.

I = WHERE(CAT.NAME EQ 'SIRIUS')
; Find index of star with name of SIRIUS.

Q = CAT.INTEN
; Extract intensity field from each entry. Q will be a 12-by-100 floating point array.

PLOT, CAT(5).INTEN
; Plot intensity of 6th star in array CAT.

CONTOUR, CAT(5 : 50).INTEN(2:8)
; Make a contour plot of the (7, 46) floating-point array taken from
; months (2:8) and stars (5:50).

CAT = CAT(SORT(CAT.NAME))
; Sort the array into ascending order by names. Store the result back into CAT.

MONTHLY = CAT.INTEN # REPLICATE(1,100)
; Determine the monthly total intensity of all stars in array. MONTHLY
; is now a 12-element array.
```

Structure Input and Output

Structures are read and written using the formatted and unformatted I/O procedures READ, READF, PRINT, PRINTF, READU, and WRITEU. Structures and arrays

of structures are transferred in much the same way as simple data types, with each element of the structure transferred in order.

Formatted Input and Output with Structures

Writing a structure with PRINT, or PRINTF and the default format, outputs the contents of each element using the default format for the appropriate data type. The entire structure is enclosed in braces: “{ }”. Each array begins a new line.

For example, printing the variable STAR, as defined in the first example in this chapter, results in the output:

```
{ SIRIUS 30.0000 40.0000
    0.000001.000002.000003.00000
    4.000005.000006.000007.00000
    8.000009.000010.000011.0000
}
```

When reading a structure with READ, or READF and the default format, white space should separate each element. Reading string elements causes the remainder of the input line to be stored in the string element, regardless of spaces, etc.

A format specification may be used with any of these procedures overriding the default formats. The length of string elements is determined by the format specification (i.e., to read the next 10 characters into a string field, use an A10 format). For more information about format specification, see [Explicitly Formatted Input and Output on page 155](#).

Unformatted Input and Output in Structures

Reading and writing unformatted data contained in structures is a straightforward process of transferring each element without interpretation or modification, *except in the case of strings*. Each data type, except strings, has a fixed length expressed in bytes; this length, with the addition of padding, is also the number of bytes read or written for each element.

All instances of structures contain an even number of bytes. As with most C compilers, PV-WAVE begins fields that are not of byte type on an even byte boundary. Thus, a “padding byte” may appear after a byte field to cause the following non-byte type field to begin on an even byte. A padding byte is never added before a byte or byte array field. For example, the structure:

```
{EXAMPLE, T1: 1B, T2: 1}
```

occupies four bytes. A padding byte is added after field T1 to cause the integer field T2 to begin on an even byte boundary.

String Input and Output

Strings are exceptions to the above rules because the length of strings within structures is not fixed. For example, one instance of the { CATALOG } structure may contain a NAME field with a five-character name, while another instance of the same structure may contain a 20-character name.

When reading into a structure field that contains a string, PV-WAVE reads the number of bytes given by the length of the string. If the string field contains a 10-character string, 10 characters are read. If the data read contains a null byte, the length of the string field is truncated, and the null and following characters are discarded.

When writing fields containing strings with the unformatted procedure WRITEU, PV-WAVE writes each character of the string and does *not* append a null byte.

String Length Issues

Reading into or writing out of structures containing strings with READU or WRITEU is tricky when the strings are not the same length. For example, it would be difficult for a C program to read variable-length string data written from a PV-WAVE application because PV-WAVE does not append a null byte to the string when it is written out. And from the other side of the coin, it is not possible to read into a string element using READU unless the number of characters to read is known. One way around this problem is to set the lengths of the string elements to some maximum length using the STRING function with a format specification.

For example, it is easy to set the length of all NAME fields in the CAT array to 20 characters:

```
CAT.NAME = STRING(CAT.NAME, Format='(A20)')
```

This statement will truncate names larger than 20 characters long and will pad with blanks those names shorter than 20 characters. The structure or structure array may then be output in a format suitable to be read by C or FORTRAN programs.

To read into the CAT array from a file in which each NAME field occupies, for example, 26 bytes:

```
CAT = REPLICATE( { CATALOG, STRING(' ', $
  Format='(A26)'), 0., 0., FLTARR(12) }, 100)
; Make a 100-element array of CATALOG structures, storing a
; 26-character string in each NAME field.
```

```
READU, 1, CAT
; Read the structure.
```

As mentioned above, 26 bytes will be read for each NAME field. The presence of a null byte in the file will truncate the field to the correct number of bytes.

Advanced Structure Usage

Facilities exist to process structures in a general way using tag numbers rather than tag names. Tags may be referenced using their index, enclosed in parenthesis, as follows:

Variable_name . (Tag_index)

The tag index ranges from 0 to the number of fields minus 1.

The N_TAGS function returns the number of fields in a structure. The TAG_NAMES function returns a string array containing the names of each tag.

Example of Tag Indices

Using tag indices, and the above-mentioned functions, we specify a procedure which reads into a structure from the keyboard. The procedure prompts you with the type, structure, and tag name of each field within the structure:

```
PRO READ_STRUCTURE, S
; A procedure to read into a structure, S, from the keyboard with
; prompts.

NAMES = TAG_NAMES(S)
; Get the names of the tags.

FOR I = 0, N_TAGS(S)-1 DO BEGIN
; Loop for each field.
A = S.(I)
; Define variable A of same type and structure as the ith field.
INFO, S.(I)
; Use INFO to print the attributes of the field.
READ, 'Enter value for field ', $
NAMES(I), ': ', A
; Prompt user with tag name of this field, and then read into variable A.
S.(I) = A
; Store back into structure from A.
ENDFOR

END
```

Note, in the above procedure the READ procedure reads into the variable A rather than S. (I), because S. (I) is an expression, not a simple variable reference. Expressions are passed by value; variables are passed by reference. The READ procedure prompts you with parameters passed by value and reads into parameters passed by reference.

Working with Lists and Associative Arrays

Lists and associative arrays allow you to create dynamic data structures in PV-WAVE. Lists contain collections of variables and/or expressions. An associative array is like a list, except each element in an associative array is given a unique name. This name is then used to reference its associated array element.

Unlike other kinds of arrays, the elements of a list or associative array do not have to be the same data type. Furthermore, the contents and size of lists and associative arrays can be modified dynamically, while an application is running.

NOTE A list or associative array definition creates a new data type.

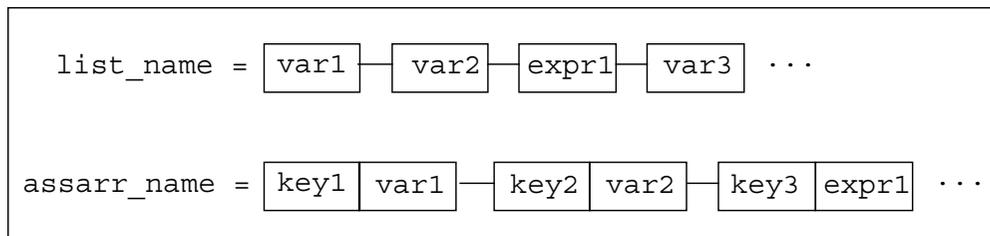


Figure 6-1 A list, shown on top, consists of an array of variables and expressions, which do not have to be of the same data type. An associative array, shown on the bottom, consists of pairs of key names (strings) and values (variables or expressions). A list is referenced using subscript numbers, just like a 1D array. The elements of an associative array are referenced by key name.

Defining a List

Use the LIST function to create a list:

$$result = LIST(expr_1, \dots, expr_n)$$

where $expr_1, \dots, expr_n$ are expressions or variables. These expressions or variables are the elements of the list array.

The elements of a list can be any of the eight basic PV-WAVE data types, other structures or arrays of structures, and other lists or associative arrays. In addition, lists and associative arrays can be used as structure fields.

Example

A list is created using the LIST function. The elements in the list do not have to have the same data type.

```
lst = LIST(1B, 2.2, '3.3', {,a:1, b:lindgen(2)})
```

The INFO command shows the contents of the list.

```
INFO, lst, /Full
LST LIST= List (4)
BYTE= 1
FLOAT = 2.20000
STRING = '3.3'
STRUCT = ** Structure $1, 2 tags, 24 length:
A  INT 1
B  LONG Array (2)
```

The PRINT command also shows the contents of the list.

```
PRINT, lst
{ 1 2.200003.3{ 1 0 1}
```

Defining an Associative Array

Use the ASARR function to create an associative array. You can call ASARR in the following two ways:

$$result = ASARR(key_1, expr_1, \dots, key_n, expr_n)$$

where $key_1, expr_1, \dots, key_n, expr_n$ are pairs of key names (strings) and expressions or variables. A key name is a string that uniquely identifies the expression or variable that immediately follows.

$$result = ASARR(keys_arr, values_list)$$

where $keys$ is an array of key names (strings) and $values$ is a list array containing the expressions and/or variables. The first element in the $keys$ array is paired with (and uniquely identifies) the first element in the $values$ array, and so on.

Example 1

An associative array is created using the first form of the ASARR function, described previously. Key names and values are specified as separate parameters.

```
as = ASARR('byte', 1B, 'float', 2.2, 'string', $
          '3.3', 'struct', {a:1, b:lindgen(2)})
```

Example 2

An associative array, equivalent to the array in Example 1, is created using the second form of the ASARR function, described previously. An array of key names is created first, followed by an array of values. Note that the values do not have to be of the same data type.

```
as=ASARR(['byte', 'float', 'string', 'struct'], $
         LIST(1B, 2.2, '3.3', {a:1, b:lindgen(2)}))
```

The INFO command shows the contents of the associative array.

```
INFO, as, /Full
AS AS. ARR = Associative Array(4)
byte BYTE = 1
struct STRUCT = ** Structure $3, 2 tags, 12 length:
A INT 1
B LONG Array(2)
float FLOAT = 2.20000
string STRING = '3.3'
```

The PRINT command also shows the contents of the array.

```
PRINT, as
{'byte' 1 'struct' { 1 0 1 } 'float' 2.20000 'string'3.3 }
```

Defining a List within a Structure within an Associative Array

The following example shows how to nest a structure and a list within an associative array. This example has applications in GUI tool development, where associative arrays can be used to store information about the attributes of a GUI tool.

```
strDef = {Main_Data_Str, attrs:ASARR(), vars: LIST()}
        ; Define a data structure to hold variables and attributes for a GUI tool.

dataStr = ASARR()
        ; Create an empty associative array that will hold the GUI tool data structure.
```

```
    ; This array will be filled in later, possibly in another procedure.
dataStr('Wg1') = {Main_Data_Str}
    ; Call the tool Wg1.
dataStr('Wg1').attrs('size') = [512, 512]
    ; Set a size attribute for the Wg1 tool.
dataStr('Wg1').vars = LIST('VAR1', 'VAR2')
    ; Set a list of variables for the Wg1 tool.
```

How to Reference a List

To reference elements in a list, follow the same rules as you would to reference elements of any 1D array:

variable_name(subscript_list)

where *variable_name* is a variable that contains a list, and *subscript_list* is a list of expressions, constants, or subscript ranges containing the values of one or more subscripts.

Nested lists are subscripted like multi-dimensional arrays:

variable_name(subscript, subscript, ...)

NOTE If the elements of nested lists are of type structure or associative array, they follow the same rules for referencing structures or associative arrays.

How to Reference an Associative Array

The basic syntax of a reference to a element of an associative array is:

variable_name (key_name)

where *variable_name* must be a variable that contains an associative array, and *key_name* is the name of the key (a string) and must exist for the associative array.

Embedded, nested associative arrays use a subscripting scheme similar to that of multi-dimensional arrays:

variable_name(key_name₁, key_name₂, ...)

NOTE If the elements of nested associative arrays are of type structure or list, they follow the rules for subscripting structures or lists.

Supported Operations for Lists

You can perform the following kinds of operations on lists.

Insert

Insert elements $expr_1, \dots, expr_n$ between the $(i-1)$ -th and i -th element of the list.

```
lst = [lst(0:i-1), expr1, ..., exprn, lst(i:*)]
```

Append

Append elements $expr_1, \dots, expr_n$ after the last element in the list.

```
lst = [lst, expr1, ..., exprn]
```

Prepend

Prepend elements $expr_1, \dots, expr_n$ before the first element in the list.

```
lst = [expr1, ..., exprn, lst]
```

Replace

Replace elements between element *start* and element *end* with the elements $expr_1, \dots, expr_n$.

```
lst = [lst(0:start), expr1, ...exprn, lst(end:*)]
```

Delete

Delete elements between element *start* and element *end*.

```
lst = [lst(0:start), lst(end:*)]
```

Create Sublists

The method for creating sublists is the same as creating subarrays. For example:

```
sublist = lst(from:to)
    ; Create a sublist from a specific range of elements.
toend   = lst(from:*)
    ; Create a sublist from a specified element to the last element.
startto = lst(0:to)
    ; Create a sublist from the first element to a specified element.
sublist = lst([0,2,5])
    ; Create a sublist containing specified elements only.
```

Enumeration

Lists can be referenced in loops just like other kinds of arrays:

```
FOR i = 0, N_ELEMENTS(lst) - 1 do BEGIN
    lst(i) = lst(i) + 1
    ; do something with the variable
ENDFOR
```

Supported Operations For Associative Arrays

You can perform the following kinds of operations on associative arrays:

Create a New Element

If the associative array *asa* exists, the following statement adds a new expression to the array with the given key name:

```
asa('newkey') = expr
```

Concatenate

The following statement concatenates the associative arrays *asa₁*, *asa₂*, ..., *asa_n* in the associative array *asa_{new}*.

```
asa_new = [asa1, asa2, ..., asan]
```

Subset

The following statement creates a new associative array that is a subset of an existing associative array.

NOTE Only the key names are required to specify the subset.

```
asa_sub = asa(['key1', 'key2'])
```

Retrieve Keys

The ASKEYS function returns the key names for an associative array.

```
keys = ASKEYS(asa)
```

The following statements show how you can enumerate associative array elements.

```
keys = ASKEYS(asa)
for i = 0, N_ELEMENTS(asa) - 1 do asa(keys(i)) = expr
```

Test for Keys

The ISASKEY function lets you test for the presence of a key name in an associative array.

```
result = ISASKEY(asa, 'key')
```

Lists, Associative Arrays Input and Output

To read and write lists and associative arrays, use the procedures READ, READF, PRINT, and PRINTF. Lists and associative arrays are transferred in much the same way as simple data types, with each element of the list or associative array transferred in order.

Writing a List or Associative Array with PRINT or PRINTF

You can write a list or an associative array with PRINT or PRINTF. By default, each element of the list or associative array is output in the default format of the element's data type. The entire list or associative array is enclosed in { } (braces). Each value of an associative array element is preceded by the key name enclosed in " " (double quotes).

Example

The contents of an associative array are printed with the PRINT command. This example uses the same code that was used previously to demonstrate nesting.

```
strDef = {Main_Data_Str, attrs:ASARR(), vars: LIST()}  
        ; Define a data structure to hold variables and attributes for a GUI tool.  
  
dataStr = ASARR()  
        ; Create an empty associative array that will hold the GUI tool data structure.  
        ; This array will be filled in later, possibly in another procedure.  
  
dataStr('Wg1') = {Main_Data_Str}  
        ; Call the tool Wg1.  
  
dataStr('Wg1').attrs('size') = [512, 512]  
        ; Set a size attribute for the Wg1 tool.  
  
dataStr('Wg1').vars = LIST('VAR1', 'VAR2')  
        ; Set a list of variables for the Wg1 tool.  
  
PRINT, dataStr  
        ; Print the contents of the associative array.  
  
{'Wg1'{{'size' 512 512}{ VAR1 VAR2}}}
```

Reading a List or Associative Array with READ or READF

You can read data into an associative array or list using the READ or READF function. The method for doing this is similar to reading data into a structure; however, associative array elements must be read in a particular order. You can determine this order with the ASKEYS function, as shown in the following example:

```
as = ASARR('byte', 1B, 'float', 2.2, 'string', $
         'hello', 'struct', {,a:1, b:lindgen(2)})
         ; Create an associative array.

PRINT, ASKEYS(as)
      byte struct float string
         ; Get the correct order of elements in the associative array. Note that the order
         ; is different than the order of the parameters in the ASARR function used to
         ; create the array.

READ, as
: 4 5 6 7 8.8 hello
         ; Read in some data. Input to READ and READF must be separated by at least
         ; one space.

PRINT, as
{"byte" 4 "struct"{ 5 6 7 }
 "float" 8.80000 "string" hello}
```


Working with Text

Working with text in PV-WAVE is equivalent to working with strings. A string is a sequence of 0 to 32,767 characters. Strings have dynamic length (they grow or shrink to fit), and there is no need to declare the maximum length of a string prior to using it. As with any data type, string arrays can be created to hold more than a single string. In this case, the length of each individual string in the array depends only on its own length, and is not affected by the lengths of the other string elements.

Example String Array

In some of the examples in this chapter, it is assumed that a string array named TREES exists. TREES contains the names of seven trees, one name per element, and is created using the statement:

```
TREES = ['Beech', 'Birch', 'Mahogany', $  
        'Maple', 'Oak', 'Pine', 'Walnut']
```

Executing:

```
PRINT, '>' + TREES + '<'
```

results in the output:

```
>Beech< >Birch< >Mahogany< >Maple< >Oak< >Pine< >Walnut<
```

Basic String Operations

PV-WAVE supports several basic string operations.

Concatenating Strings

The addition operator, +, is used to concatenate strings.

Formatting

The STRING function is used to format data into a string.

Converting to Upper or Lower Case

The STRLOWCASE function returns a copy of its string argument converted to lower case. Similarly, the STRUPCASE function converts its argument to upper case.

Removing White Space

The STRCOMPRESS and STRTRIM functions can be used to eliminate unwanted white space (blanks or tabs) from their string arguments.

Determining String Length

The STRLEN function returns the length of its string argument.

Manipulating Substrings

The STRPOS, STRPUT, and STRMID routines locate, insert, and extract substrings from their string arguments.

Concatenating Strings

The addition operator concatenates strings. For example, the command:

```
A = 'This is ' + 'a concatenation example.'  
PRINT, A
```

results in the output:

```
This is a concatenation example.
```

The following statements build a scalar string containing a list of the names found in the TREES string array separated by commas:

```

NAMES = ''
    ; The list of names.
FOR I = 0, 6 DO BEGIN
    IF (I NE 0) THEN NAMES = NAMES + ', '
    ; Add comma before next name.
    NAMES = NAMES + TREES(I)
    ; Add the new name to the end of the list.
ENDFOR
PRINT, NAMES
    ; Show the resulting list.

```

Running the above statements gives the result:

```
Beech, Birch, Mahogany, Maple, Oak, Pine, Walnut
```

String Formatting

The STRING function has the form:

$$result = \text{STRING}(Expression_1, \dots, Expression_n)$$

It converts its parameters to characters, returning the result as a string expression. It is very similar to the PRINT statement, except that its output is placed into a string rather than being output to the screen. As with PRINT, the *Format* keyword can be used to explicitly specify the desired format. See the discussions of free format and explicitly formatted I/O in [Choosing Between Free or Fixed \(Explicitly Formatted\) ASCII I/O on page 148](#) for details on data formatting.

As a simple example, the following statements:

```

A = STRING(Format='("The values are:", $
    /, (I))', INDGEN(5))
    ; Produce a string array.
INFO, A
    ; Show its structure.
FOR I = 0, 5 DO PRINT, A(I)
    ; Print the result.

```

produce the following output:

```

A  STRING = Array(6)
The values are:
  0
  1

```

2
3
4

Using STRING with Byte Arguments

There is a close association between a string and a byte array — a string is simply an array of bytes that is treated as a series of ASCII characters. It is therefore convenient to be able to switch between them easily.

When STRING is called with a single argument of type byte and the *Format* keyword *is not* used, STRING does not work in its normal fashion. Instead of formatting the byte data and placing it into a string, it returns a string containing the byte values from the original argument. Thus, the result has one less dimension than the original argument. A two-dimensional byte array becomes a vector of strings, a byte vector becomes a scalar string. However, a byte scalar also becomes a string scalar. For example, the statement:

```
PRINT, STRING([72B, 101B, 108B, 108B, 111B])
```

produces the output:

```
Hello
```

This occurs because the argument to STRING, as produced by the array concatenation operator [], is a byte vector. Its first element is 72B which is the ASCII code for “H”, the second is 101B which is an ASCII “e”, and so forth.

As discussed in the section [Explicitly Formatted Input and Output on page 155](#), it is easier to read fixed length string data from binary files into byte variables instead of string variables. It is therefore convenient to read the data into a byte array and use this special behavior of STRING to convert the data into string form.

Another use for this feature builds strings that have unprintable characters in them in a way that doesn’t actually require entering the character directly. This results in programs that are easier to read, and which also avoid file transfer difficulties. (Some forms of file transfer have problems transferring unprintable characters).

For example:

```
tab = STRING(9B)
      ; 9 is the decimal ASCII code for the tab character.
bel = STRING(7B)
      ; 7 is the decimal ASCII code for the bell character.
PRINT, 'There is a', tab, 'tab here.', bel
      ; Output a line containing a tab character, and ring the terminal bell.
```

Executing these statements gives the output:

There is a tab here.

and rings the bell.

Applying the `STRING` function to a byte array containing a null (zero) value will result in the resulting string being truncated at that position. Thus, the statement:

```
PRINT, STRING([65B, 66B, 0B, 67B])
```

produces the output:

```
AB
```

because the null byte in the third position of the byte array argument terminates the string and hides the last character.

The `BYTE` function, when called with a single argument of type string, performs the inverse operation to that described here, resulting in a byte array containing the same byte values as its string argument. For additional information about the `BYTE` function, see [Type Conversion Functions on page 32](#).

Converting Strings to Upper or Lower Case

The `STRLOWCASE` and `STRUPCASE` functions convert their arguments to lower or upper case. They have the form:

$$result = \text{STRLOWCASE}(string)$$
$$result = \text{STRUPCASE}(string)$$

where *string* is the string to be converted to lower or upper case.

The following statements generate a table of the contents of `TREES` showing each name in its actual case, lower case, and upper case:

```
FOR I = 0, 6 DO PRINT, TREES(I), STRLOWCASE(TREES(I)), $  
    STRUPCASE(TREES(I)), Format = '(A,T15,A,T30,A)'
```

The resulting output from running this statement is:

Beech	beech	BEECH
Birch	birch	BIRCH
Mahogany	mahogany	MAHOGANY
Maple	maple	MAPLE
Oak	oak	OAK
Pine	pine	PINE
Walnut	walnut	WALNUT

A common use for case folding occurs when writing procedures that require input from the user. By folding the case of the response, it is possible to handle responses written in any case. For example, the following statements can be used to ask “Yes or No” style questions:

```
ANSWER = ''
    ; Create a string variable to hold the response.
READ, 'Answer Yes or No: ', ANSWER
IF (STRUPCASE(ANSWER) EQ 'YES') THEN
    PRINT, 'Yes' else PRINT, 'No'
    ; Compare the response to the expected answer.
```

Removing White Space from Strings

The STRCOMPRESS and STRTRIM functions remove unwanted white space (tabs and spaces) from a string. This can be useful when reading string data from arbitrarily formatted strings.

STRCOMPRESS returns a copy of its string argument with all white space replaced with a single space, or completely removed. It has the form:

$$result = STRCOMPRESS(string)$$

where *string* is the string to be compressed. The default action is to replace each section of white space with a single space. Use of the *Remove_All* keyword causes white space to be completely eliminated. For example:

```
A = ' This is a poorly spaced sentence.'
    ; Create a string with undesirable white space. Such a string might
    ; be the result of reading user input with a READ statement.
PRINT, '>', STRCOMPRESS(A), '<'
    ; Print the result of shrinking all white space to a single blank.
PRINT, '>', STRCOMPRESS(A, /REMOVE_ALL), '<'
    ; Print the result of removing all white space.
```

results in the output:

```
> This is a poorly spaced sentence.<
>Thisisapoorlyspacedsentence.<
```

STRTRIM returns a copy of its string argument with leading and/or trailing white space removed. It has the form:

$$result = STRTRIM(string[, flag])$$

where *string* is the string to be trimmed and *flag* is an integer that indicates the specific trimming to be done. If *flag* is 0, or is not present, trailing white space is removed. If it is 1, leading white space is removed. Both are removed if it is equal to 2.

As an example:

```
A = '   This string has leading and ' + $
    'trailing white space   '
    ; Create a string with unwanted leading and trailing blanks.

PRINT, '>', STRTRIM(A), '<'
    ; Remove trailing white space.

PRINT, '>', STRTRIM(A, 1), '<'
    ; Remove leading white space.

PRINT, '>', STRTRIM(A, 2), '<'
    ; Remove both.
```

Executing these statements produces the output:

```
> This string has leading and trailing white space<
>This string has leading and trailing white space  <
>This string has leading and trailing white space<
```

When processing string data, it is often useful to be able to remove leading and trailing white space and shrink any white space in the middle down to single spaces. STRCOMPRESS and STRTRIM can be combined to handle this:

```
A = ' Yet   another poorly   spaced ' + $
    'sentence.'
    ; Create a string with undesirable white space.

PRINT, '>', STRCOMPRESS(STRTRIM(A, 2)), '<'
    ; Eliminate unwanted white space.
```

Executing these statements gives the result:

```
>Yet another poorly spaced sentence.<
```

Determining the Length of Strings

The STRLEN function obtains the length of a string. It has the form:

$$result = STRLEN(string)$$

where *string* is the string for which the length is required.

For example, the following statement:

```
PRINT, STRLEN('This sentence has 31 ' + $
    'characters')
```

results in the output:

```
31
```

while the following statement prints the lengths of all the names contained in the array TREES:

```
PRINT, STRLEN(TREES)
```

The resulting output from running this statement is:

```
5 5 8 5 3 4 6
```

Manipulating Substrings

The STRPOS, STRPUT, and STRMID routines locate, insert, and extract substrings from their string arguments.

The STRPOS function is used to search for the first occurrence of a substring. It has the form:

$$result = STRPOS(object, search_string[, pos])$$

where *object* is the string to be searched, *search_string* is the substring to search for, and *pos* is the character position (starting with position 0) at which the search is begun. The argument *pos* is optional. If it is omitted, the search is started at the first character (character position 0). The following statements count the number of times that the word dog appears in the string dog cat duck rabbit dog cat dog:

```
ANIMALS = 'dog cat duck rabbit dog cat dog'
    ; The string to search — dog appears 3 times.

I = 0
    ; Start searching in character position 0

CNT = 0
    ; Number of occurrences found

WHILE (I NE -1) DO BEGIN
    I = STRPOS(ANIMALS, 'dog', I)
        ; Search for an occurrence
        IF (I NE -1) THEN BEGIN CNT = CNT + 1 & $
            I = I + 1 & END
        ; If one is found, count it and advance to the next character position.

ENDWHILE

PRINT, 'Found ', cnt, " occurrences of 'dog'"
```

Running the above statements produces the result:

```
Found 3 occurrences of 'dog'
```

The STRPUT procedure inserts the contents of one string into another. It has the form:

```
STRPUT, destination, source [, position]
```

where *destination* is the string to be inserted into, *source* is the string to be inserted, and *position* is the first character position within *destination* at which *source* will be inserted. The argument *position* is an optional argument. If it is omitted, the insertion is started at the first character (character position 0). The following statements use STRPOS and STRPUT to replace every occurrence of the word dog with the word CAT in the string dog cat duck rabbit dog cat dog:

```
ANIMALS = 'dog cat duck rabbit dog cat dog'
```

```
    ; The string to modify — dog appears 3 times.
```

```
WHILE (((I = STRPOS(ANIMALS, 'dog')) NE -1) DO STRPUT, ANIMALS,  
    'CAT', I
```

```
    ; While any occurrence of dog exists, replace it.
```

```
PRINT, ANIMALS
```

Running the above statements produces the result:

```
CAT cat duck rabbit CAT cat CAT
```

The STRMID function extracts substrings from a larger string. It has the form:

```
result = STRMID(expression, position, length)
```

where *expression* is the string from which the substring will be extracted, *position* is the starting position within *expression* of the substring (the first position is position 0), and *length* is the length of the substring to extract. If there are not *length* characters following *position*, then the substring will be truncated. The following statements use STRMID to print a table matching the number of each month with its three-letter abbreviation:

```
MONTHS = 'JANFEBMARAPRPMAYJUNJULAUGSEP' + $  
    'OCTNOVDEC'
```

```
    ; String containing all the month names.
```

```
FOR I = 1, 12 DO PRINT, I, '  
    STRMID(MONTHS, (I - 1) * 3, 3)
```

```
    ; Extract each name in turn. The equation (I-1)* 3 calculates  
    ; the position within MONTH for each abbreviation.
```

The result of executing these statements is:

```
1                JAN
```

2	FEB
3	MAR
4	APR
5	MAY
6	JUN
7	JUL
8	AUG
9	SEP
10	OCT
11	NOV
12	DEC

Using Non-string and Non-scalar Arguments

Most of the string processing routines described in this chapter expect at least one argument, which is the string on which they act.

If the argument is not of string type, it is converted to string type according to the same default formatting rules that are used by the PRINT, or STRING routines. The function then operates on the converted result. Thus, the statement:

```
PRINT, STRLEN(23)
```

returns the result:

```
8
```

because the argument 23 is first converted to the string ' 23 ' which happens to be a string of length eight.

If the argument is an array instead of a scalar, the function returns an array result with the same structure as the argument. Each element of the result corresponds to an element of the argument.

For example, the following statements:

```
A = STRUPCASE(TREES)
    ; Get an uppercase version of TREES.
INFO, A
    ; Show that the result is also an array.
PRINT, TREES
    ; Display the original.
```

```
PRINT, A  
; Display the result.
```

results in the output:

```
A      STRING      = Array(7)  
Beech Birch Mahogany Maple Oak Pine Walnut  
BEECH BIRCH MAHOGANY MAPLE OAK PINE WALNUT
```

For more details on how individual routines handle their arguments, see the individual descriptions in the PV-WAVE Reference.

Using Regular Expressions

To use the PV-WAVE string handling functions STRMATCH, STRSPLIT, and STRSUBST, you must understand how *regular expressions* work.

Regular expressions are used in UNIX-based utilities such as `grep`, `egrep`, `awk`, and `ed`. UNIX users are probably familiar with the powerful pattern matching capabilities of regular expressions.

NOTE Regular expressions are not the same as wildcard characters. See the section [Regular Expressions vs. Wildcard Characters on page 127](#) for information on this common source of confusion.

This section provides an elementary introduction to regular expressions. Additional sources of information on regular expressions are listed at the end of this section.

Simple Regular Expressions: A Brief Introduction

This section introduces some simple regular expression examples. More complex examples are presented in [Practical Regular Expression Examples on page 126](#).

Regular expressions can be very complex. Indeed, entire books have been written on the subject of regular expressions. Regular expressions normally consist of characters that you wish to match and special characters that perform specific pattern matching functions. For a list of commonly used special characters see [Basic Special Characters Used In Regular Expressions on page 125](#).

In PV-WAVE, the STRMATCH, STRSPLIT, and STRSUBST commands take regular expression arguments to perform pattern matching operations. The following examples demonstrate the use of regular expressions in the STRMATCH function.

Matching a Single Character

The regular expression special character ' .' (dot) matches any single character except a newline.

For example, the regular expression used in the STRMATCH function:

```
result=STRMATCH(string, '.at')
```

matches any string containing the following sequence of characters:

```
bat  
cat  
mat  
oat
```

Matching Zero or More Characters

The regular expression special character ' * ' (asterisk) matches zero or more of the preceding character.

For example, the regular expression used in the STRMATCH function:

```
result=STRMATCH(string, 'x*y')
```

matches the following strings (zero or more “x” characters, followed by a single “y”):

```
y  
xy  
xxy  
xxxy
```

Matching One or More Characters

The regular expression special character ' + ' (plus) matches one or more of the preceding character.

For example, the regular expression used in the STRMATCH function:

```
result=STRMATCH(string, 'x+y')
```

matches the following strings:

```
xy  
xxy  
xxxy
```

Other Special Characters

Other characters — such as brackets, braces, parentheses, back-slashes and so on — also have meaning in a regular expression, depending on the regular expression syntax used.

See the table in the following section for a list of the most basic regular expression special characters.

Basic Special Characters Used In Regular Expressions

The following table lists the most basic regular expression special characters and explains what they match.

Special Character	Matches
.	any single character except newline
^	the first character of the string (when used as the first character in the regular expression)
\$	the last character of the string (when used as the last character in the regular expression)
*	zero or more of the preceding character. (This character is a modifier, which means that it specifies how many times you expect to see the preceding character. Therefore, this character is only significant if it is preceded by another character.)
+	one or more of the preceding character. (This character is also a modifier, because it must be preceded by another character.)
?	zero or one of the preceding character. (This character is also a modifier, because it must be preceded by another character.)
[...]	a single character that is in the enclosed group of characters; either a list of characters, like [abc], or a range of characters, like [0-9], or both [0-9 ABC w-z]
[^ ...]	any character except those enclosed in the square brackets, like [^0-9]
	acts as an OR operator, separating two regular expressions
()	encloses sub-expressions (used for grouping and for the <i>registers</i> variable in the STRMATCH function)

Escaping Special Characters

To match a special character as you would a normal character, you must “escape it” by preceding it with a backslash (`\`). Note, however, that in PV-WAVE strings, two backslashes translate to a single backslash. For example, to match a period (`.`) in a regular expression in a PV-WAVE function, you must use `' \\ . '`

NOTE To match a single backslash in a PV-WAVE string, you have to use two pairs of backslashes `' \\\\'`. Each pair, in PV-WAVE strings, makes a single backslash, thus you end up with a single escaped backslash. In other words, the first pair of backslashes is the “escape” character, and the second pair is the “escaped” backslash.

TIP If you get confused writing strings with multiple backslashes in PV-WAVE, you can print the string to see what you get. For example:

```
PRINT, '\\\\'
      \\
```

Practical Regular Expression Examples

Assume that `string` is a string array defined in PV-WAVE. The following PV-WAVE commands demonstrate the regular expression pattern matching used in the `STRMATCH` command.

```
result=STRMATCH(string, 'a')
```

Matches any string containing the character `'a'`.

```
result=STRMATCH(string, '^ [CcBb]at')
```

Matches any string beginning (^) with `Cat`, `bat`, and so on: `'Cat Woman'`, `'catatonic'`, `'Batman, the animated series'`; but does not match: `' cat'` (begins with a space), `'cab'`, and so on.

```
result=STRMATCH(string, 'Ll+')
```

Matches any string containing `'L'` followed by one or more occurrences of `'l'`: `'Get a Llama'` matches; `'larry the llama'` does not match (first `l` in `llama` is lower case).

```
result=STRMATCH(string, '^ [^C] . *x$')
```

Matches a string that starts (^) with any character that is not `'C'` (`[^C]`), and is followed by zero or more other characters (`. *`), and ends with `'x'` (`x$`). The patterns `'ux'`, `'under stdfx'`, and `'corx'` match; `'x'` does not match (`x` matches the `[^C]`, but there's nothing to match the `x$`).

```
result=STRMATCH(string, '\\\\.')
```

Matches any string containing a period. '3.14159' matches; 'the quick brown fox' does not match. Remember that it takes two backslashes in a PV-WAVE string to produce the single backslash that "escapes" the dot (.), as explained previously.

```
result=STRMATCH(string, '.')
```

Matches any string containing any character (that is, any non-null string).

```
result=STRMATCH(string, '^$')
```

Matches only empty strings (start and end with nothing in between).

```
result=STRMATCH(string, '^ *$')
```

Matches either blank or null strings (Between the beginning (^) and the end (\$) there are only zero or more spaces (*)).

```
result=STRMATCH(string, '^...$')
```

Matches only three-character strings.

```
result=STRMATCH(string, '^...+$')
```

Matches strings three characters or longer.

```
result=STRMATCH(string, '^[\ \011]*[-+]?[0-9]+[\ \011]*$')
```

This interesting example matches any integer number, possibly surrounded by spaces and/or tabs. This expression means:

From the beginning of the string (^), zero or more spaces or tabs (\011 is the octal ASCII number for a tab character), zero or one sign [-+], one or more digit [0-9], zero or more spaces/tabs, and finally match the end of string.

Regular Expressions vs. Wildcard Characters

Many users understandably confuse wildcard characters and regular expressions, because both are used for pattern matching, and because some of the same characters, like asterisk (*), question mark (?), and square brackets ([]), are used in both, yet have different meanings.

NOTE Wildcard characters are commonly used in file matching contexts on Microsoft Windows systems. On UNIX systems, wildcards are used in the Bourne shell and C shell, as well as in the commands `find` and `cpio`. The most common wildcard is the asterisk (*), which matches any group of characters.

A common misconception is that the asterisk (*) is a wildcard character in regular expressions. In regular expressions, asterisk (*) means "match zero or more of the preceding character."

To make a "wildcard" (that is, an expression that matches anything) with regular expressions, you must use `' .* '` (dot asterisk). This expression means, "match zero or more of any character."

Example of Wildcards vs. Regular Expressions

For example, most computer users have used the asterisk (*) as a wildcard character in system commands such as `ls` and `dir`. For example:

```
dir file.*
```

is a wildcard expression that matches anything that begins with “file.”, such as `file.c`, `file.o`, `file.dat`, `file.pro`, and so on.

However, the regular expression character * means something entirely different from the wildcard character *. In regular expressions, the asterisk means *match zero or more of the preceding character*.

Therefore, the regular expression, 'file.*', would match:

```
file.dat
myfile.c
myfile
myfiles
```

This result is quite different from the wildcard example shown previously.

Regular Expressions are Versatile

You can, of course, construct a regular expression that is equivalent to the wildcard expression shown previously. Here is a regular expression that performs the same pattern matching function as the wildcard expression `file.*`:

```
'^file\\.\\..*'
```

Here, the caret (^) matches the beginning of the string. The “\\.\\.” matches a single dot (.), and the “.*” matches zero or more of any characters.

For More Information

For an excellent explanation of regular expressions, see:

- *UNIX Power Tools*, Jerry Peek, Tim O’Reilly, and Mike Loukides, O’Reilly & Associates/Bantam, 1993.
- *Mastering Regular Expressions: Powerful Techniques for Perl and Other Tools*, Jeffrey Friedl, O’Reilly & Associates, 1997.

Many general books on UNIX programming contain information on regular expressions. In addition, books on the Perl programming language usually explain

regular expressions in detail (Perl uses regular expressions extensively). For example, see:

- *Programming Perl*, Larry Wall, Tom Christiansen, and Randal L. Schwartz, O'Reilly & Associates, Inc., Second Edition, 1996.

UNIX users can find regular expressions explained in the man page for the `ed` command.

Working with Data Files

PV-WAVE provides many alternatives for working with data files. There are few restrictions imposed on data files and there is no unique PV-WAVE format. This chapter describes input and output methods and routines, and gives examples of programs that read and write data using PV-WAVE, C, and FORTRAN commands.

NOTE If you work with Hierarchical Data Format (HDF) files, then refer also to *The PV-WAVE HDF Interface* in the PV-WAVE Reference for details on how to access HDF functions from within PV-WAVE.

Simple Examples of Input and Output

PV-WAVE variables point to portions of memory that are set aside during a session to store data. The first step in analyzing data is usually to transfer it into PV-WAVE variables.

This section provides a “birds-eye view” of how PV-WAVE I/O (Input/Output) works by providing some examples showing how data is transferred in and out of variables.

Example 1 — Input

The following example illustrates how easy it is to read a single column of data points contained in the file `data1.dat` into a variable `flow`. The data points can then be plotted. The file `data1.dat` contains the data points:

```
23.2
34.7
78.1
46.5
44.4
```

Try entering the following commands to read and plot the data points:

```
status = DC_READ_FREE('data1.dat', flow)
; DC_READ_FREE handles the opening and closing of the file. It
; takes the values in the file "data1.dat" and places them into a
; floating-point variable named flow. The variable flow is dimensioned
; to match the number of points read from the file. The returned value
; status can be checked to see if the process completed successfully.
PLOT, flow
; Display the variable flow in a window.
```

With two commands, the data is transferred from the file into the variable `flow` and displayed in a window.

An alternate set of commands that achieves a similar result is shown below.

```
flow = FLTARR(9)
; Define a variable that holds a single column of data containing
; 9 data points. Even though there are only 5 data points in the
; file, the array is made larger so that data points can be added later.
OPENR, 1, 'data1.dat'
; Open the file "data1.dat" for reading.
READF, 1, flow
; Read the data from the file into the variable flow.
CLOSE, 1
; Close the file.
PLOT, flow
; Display the variable flow in a window.
```

Example 2 — Output

Here's a simple example showing how you can transfer data from a variable to a file:

```
yflow = [77, 63, 42, 56]
; Define yflow to be a vector of integers.
status = DC_WRITE_FREE('data2.dat', yflow, $
/Column)
; DC_WRITE_FREE handles the opening and closing of the file. It
; takes the values in "yflow" and stores them in a file named
; "data2.dat". Because the Column keyword was supplied, each
```

```
; value is written on a different line of the file. The returned value  
; status can be checked to see if the process completed successfully.
```

Or you can use the `OPENW` command to create a new file that contains these same values:

```
OPENW, 2, 'data3.dat'  
; Open the file "data3.dat" for writing.  
PRINTF, 2, '77'  
PRINTF, 2, '63'  
PRINTF, 2, '42'  
PRINTF, 2, '56'  
; Write the values to the file, each value on a new line.  
CLOSE, 2  
; Close the file.
```

Now use the following commands to change a data point in the existing file `data3.dat`:

```
OPENU, 1, 'data3.dat'  
; Open the file "data3.dat" for updating.  
PRINTF, 1, '89'  
; Replaces the value 77 with the new value 89.  
CLOSE, 1  
; Close the file.
```

Now the contents of `data3.dat` look like:

```
89  
63  
42  
56
```

Conclusion

These two examples have introduced you to a few of the commands that are available for reading and writing data. The rest of this chapter elaborates on the various commands and concepts that you need to know to confidently transfer data in and out of PV-WAVE.

Opening and Closing Files

PV-WAVE has several commands for opening and closing data files; you select the command that matches the way you intend to use the file.

Opening Files

Before a file can be processed by PV-WAVE, it must be opened and associated with a number called the *logical unit number*, or LUN for short. All I/O is done by specifying the LUN, not the filename.

The LUN is supplied as part of the function call. For example, to open the file named `data.dat` for reading on file unit 1, you would enter the following command:

```
OPENR, 1, 'data.dat'
```

Once the file is opened, you can choose between several I/O routines. Each routine fills a particular need — the one to use depends on the particular situation. Refer to the examples in this chapter to get an idea of how (and when) to open and close data files.

NOTE If you are using one of the I/O routines that start with the letters “DC”, you do not need to explicitly open and close the file, because these steps happen automatically. For more details, refer to [Functions for Simplified Data Connection on page 146](#).

Basic Commands for Opening Files

The three main OPEN commands are listed in the following table:

Procedure	Description
OPENR	Opens an existing file for input only.
OPENW	Opens a new file for input and output. Under UNIX and Windows, if the named file already exists, the previous contents are destroyed. Under OpenVMS, a file with the same name and a higher version number is created.
OPENU	Opens an existing file for input and output.

The general form for using any of the OPEN procedures is:

OPEN x , *unit*, *filename*

where *unit* refers to the logical file unit that will be allocated for opening the file named *filename*, and x is either an R, W, or U, depending on which of the three OPEN commands you choose to use.

NOTE The three commands shown above recognize keywords that modify their normal behavior. Some keywords are generally applicable, while others only have effect under a given operating system. For more information about keywords, refer to the descriptions for the OPENR, OPENW, and OPENU procedures. These descriptions can be found in the PV-WAVE Reference.

When to Open the File for I/O (Input/Output)

Usually you must open the file before any I/O can be performed. But there are two situations where you don't need to open the file before doing any I/O:

- **Reserved LUNs** — There are three file units that are always open — in fact, the user is not allowed to close them. These files are *standard input* (usually the keyboard), *standard output* (usually the workstation's screen), and *standard error output* (usually the workstation's screen). These three files are associated with LUNs 0, -1, and -2 respectively. Because these file units are always open, you do not need to open them prior to using them for I/O. For more information about the three reserved file units, refer to [Reserved Logical Unit Numbers \(-2, -1, 0\) on page 136](#).
- **Simplified I/O Routines** — Any I/O function that begins with the two letters "DC" automatically handles the opening and closing of the file unit. This group of functions has been provided to simplify the process of getting your data in and out of PV-WAVE. For more information about the DC I/O functions, refer to [Functions for Simplified Data Connection on page 146](#).

Closing Files

Always close the file when you are done using it. Closing a file removes the association between the file and its LUN and thus frees the LUN for use with a different file. There is usually an operating-system-imposed limit on the number of files you may have open at once. Although this number is large enough that it rarely causes problems, you may occasionally need to close a file before opening another file. In any event, it is a good idea to only keep needed files open.

Closing a LUN is done with the CLOSE procedure. For example, to close file unit 1, enter this command:

```
CLOSE, 1
```

Also, remember that PV-WAVE closes all open files as it shuts down. Any LUN you allocated is automatically deallocated when you exit PV-WAVE with the EXIT or QUIT command.

NOTE If `FREE_LUN` is called with a file unit number that was previously allocated by `GET_LUN`, it calls `CLOSE` before deallocating the file unit.

Logical Unit Numbers (LUNs)

PV-WAVE logical unit numbers are in the range $\{-2\dots 128\}$; they are divided into three groups:

Reserved Logical Unit Numbers (-2, -1, 0)

0, -1, and -2 are special file units that are always open within PV-WAVE:

- 0 (zero) — The standard input stream, which is usually the keyboard. This implies that the statement:

```
READ, X
```

is equivalent to

```
READF, 0, X
```

The user would then enter the values of X from the keyboard, as shown in the following statements:

```
READ, X
```

```
: 0.2, 0.4, 0.6
```

The line preceded with the colon (:) denotes user input.

- -1 (negative 1) — The standard output stream, which is usually the workstation's screen. This implies that the statement:

```
PRINT, X
```

is equivalent to

```
PRINTF, -1, X
```

The following command can be used to send a message to the screen:

```
PRINT, 'Hello World.'
```

The following line:

```
Hello World.
```

is sent to the workstation's screen.

- `-2` (negative 2) — The standard error stream, which is usually the workstation's screen.

Because the `READ` and `PRINT` procedures automatically use the standard input and output streams (files) by default, basic ASCII I/O is extremely simple.

Operating System Dependencies

The reserved files units have a special meaning which is operating-system dependent, as explained in the following sections:

UNIX

The reserved LUNs are equated to `stdin`, `stdout`, and `stderr` respectively. This means that the normal UNIX file redirection and pipe operations work with PV-WAVE. For example, the shell command:

```
% wave < wave.inp > wave.out &
```

causes PV-WAVE to execute in the background, reading its input from the file `wave.inp` and writing its output to the file `wave.out`.

OpenVMS

The reserved LUNs are equated to `SYS$INPUT`, `SYS$OUTPUT`, and `SYS$ERROR` respectively. This means that the `DCL DEFINE` statement can be used to redefine where PV-WAVE gets commands and writes its output. It also means that PV-WAVE can be used in command and batch files.

Logical Unit Numbers for General Use (1...99)

These are file units for normal interactive use. When using PV-WAVE interactively, you can select any number in this range.

The following statements show how a string, "Hello World.", could be sent to a file named `hello.dat`:

```
OPENW, 1, 'hello.dat'
    ; Open LUN 1 for hello.dat with write access.

PRINTF, 1, 'Hello World.'
    ; Insert the string "Hello World." into the file hello.dat.

CLOSE, 1
    ; You're done with the file, so close it.
```

Logical Unit Numbers Used by GET_LUN/FREE_LUN (100...128)

These are file units that are managed by the GET_LUN and FREE_LUN procedures. GET_LUN and FREE_LUN provide a standard mechanism for routines to obtain a LUN.

GET_LUN allocates a file unit from a pool of free units in the range {100...128}. This unit will not be allocated again until it is released by a call to FREE_LUN. Meanwhile, it is available for the exclusive use of the program that allocated it.

CAUTION When writing procedures and functions, be sure not to explicitly assign file unit numbers in the range {100...128}. If a procedure or function reads or writes to an explicitly assigned file unit, there is a chance it will conflict with other routines that are using the same unit. Always use the GET_LUN and FREE_LUN procedures to manage LUNs.

Sample Usage — GET_LUN and FREE_LUN

A typical procedure that needs a file unit might be structured in the following way:

```
PRO demo
  OPENR, Unit, 'file.dat', /GET_LUN
    ; Get a unique file unit and open the file.
    .
    . (Other commands go here.)
    .
  FREE_LUN, Unit
    ; Return the file unit number. Since the file is still open,
    ; FREE_LUN will automatically call CLOSE.
END
```

NOTE All procedures and functions that open files, including those that you write yourself, should use GET_LUN and FREE_LUN to obtain file units. Never use a file unit in the range {100...128} unless it was previously allocated with GET_LUN.

How is the Data File Organized?

In ASCII files, the file can either be organized by rows or columns; the fact that ASCII files are human-readable helps you interpret their contents. In binary files, however, the organization of the file may be considerably less clear; you need to know something about the application that created the file, and understand the operating system under which the application was running to fully understand the organization of the file.

Column-Oriented ASCII Data Files

A column-oriented data file is one that contains multiple data values arranged in columns; because it is ASCII, the data is human-readable. At the end of each row is a control character, such as Ctrl-J or Ctrl-M, that forces a line feed and carriage return.

In a column-oriented file, the values in each column are related in some way; ultimately, you will probably want to group all the data in each column into a different variable for further analysis. A typical column-oriented data file is shown in [Figure 8-1](#).

NOTE Not all files that contain columns of values contain column-oriented data. For example, if you are reading every value in the file into the same variable, the file is probably a row-oriented file, despite its apparent columnar organization. The organization of row-oriented files is discussed further in [Row-Oriented ASCII Data Files](#) on page 140.

JAN 0	33.4110	0.5382	0.2683
JAN 2	33.7718	0.3849	0.2465
JAN 4	34.2258	0.3116	0.2465
JAN 6	34.6347	1.4532	0.4215
JAN 8	38.8444	2.0452	0.7581
JAN 10	44.7400	0.7629	0.7511
JAN 12	47.4997	0.2935	0.6559
JAN 14	47.5487	0.8376	0.7142
JAN 16	44.5487	0.8376	0.7142
JAN 18	39.4317	1.5540	0.5852
JAN 20	36.9194	0.8124	0.4210
JAN 22	35.4489	0.6462	0.3712
FEB 0	30.4813	0.4902	1.2768
FEB 2	29.8589	1.3381	
FEB 4	29.9985	0.3262	
FEB 6	33.0000		
FEB 8			
FEB 10			

Name: Hour
 Type: Integer
 Dimension 1: *

Name: Month
 Type: String
 Dimension 1: *

Name: Fahrenheit
 Type: Float
 Dimension 1: *

Name: CO
 Type: Float
 Dimension 1: *

Name: SO2
 Type: Float
 Dimension 1: *

Figure 8-1 Typical file organization for a column-oriented ASCII data import file. In this example, the first column of data is associated with a variable named Month, the second column with a variable named Hour, the third column with a variable named Fahrenheit, the fourth column with a variable named CO, and the fifth column with a variable named SO2.

Row-Oriented ASCII Data Files

A row-oriented data file is one that contains multiple data values arranged in a continuous stream; because it is ASCII, the data is human-readable. When reading this kind of file, the size of the variables in the variable list determines how many values get transferred. The data type of the variables also influences how the data gets interpreted, because if the data is not the expected type, PV-WAVE performs type conversion as it reads the data. A typical row-oriented data file is shown in [Figure 8-2](#).

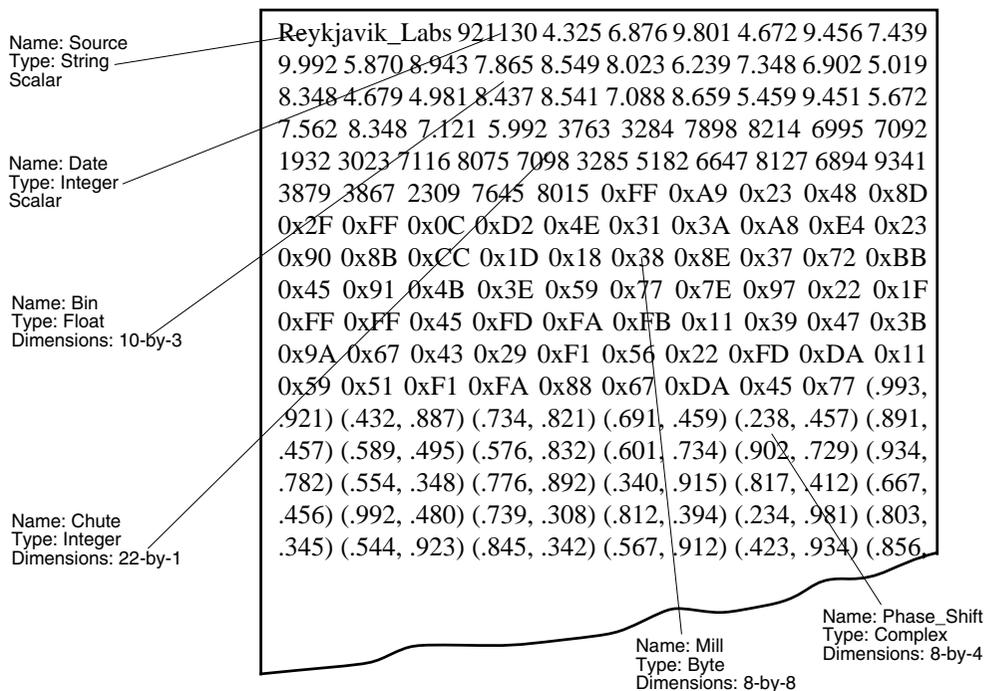


Figure 8-2 Typical file organization for a row-oriented ASCII data import file. Spaces are being used as the delimiter to separate adjacent data values. In this example, the first group of data is associated with a variable named Source, the second group with a variable named Date, the third group with a variable named Bin, the fourth group with a variable named Chute, the fifth group with a variable named Mill, and the sixth group with a variable named Phase_Shift.

How Long is a Record?

It can be important to understand the concept of records, especially if you are performing certain types of I/O. The following sections discuss records, both in the context of formatted and unformatted data.

UNIX and OpenVMS USERS Differences between the UNIX and OpenVMS operating systems are also noted, when they exist.

Record Length in ASCII (Formatted) Files

In an ASCII text file, the end-of-line is signified by the presence of either a Ctrl-J or a Ctrl-M character, and a *record* extends from one end-of-line character to the next. However, there are actually two kinds of records:

- ✓ physical records
- ✓ logical records

For column-oriented files, the amount of data in a *physical record* is often sufficient to provide exactly one value for each variable in the variable list, and then it is a *logical record*, as well. For row-oriented files, the concept of logical records is not relevant, since data is merely read as contiguous values separated by delimiters, and the end-of-line is interpreted as yet another delimiter.

Changing the Logical Record Size

If you are using one of the DC_READ routines for simplified I/O, and you are reading column-oriented data, you can use a command line keyword to explicitly define a different logical record size, if you wish. The “DC” routines are introduced in [Functions for Simplified Data Connection on page 146](#).

NOTE By default, PV-WAVE considers the physical record to be one line in the file, and the concept of a logical record is not needed. So in most cases, you do not need to define a logical record. But if you *are* using logical records, the physical records in the file must all be the same length.

For more details about the keywords that control logical record size, refer to the descriptions for the DC_READ_FIXED and DC_READ_FREE routines; these descriptions are found in the PV-WAVE Reference.

Record Length in Binary (Unformatted) Files

Binary data is a continuous stream of ones and zeros. To fully understand the organization of binary files, you need to know something about the application that created the file, and understand the operating system under which the application was running. You would then choose variables for the variable list that match that organization. The type and size of the variables in the variable list establish a framework by which the ones and zeros in the file are interpreted.

UNIX and OpenVMS USERS For binary files, neither the concept of physical or logical records is relevant, although when using PV-WAVE in an OpenVMS environment, the concept of records (at the operating system level) may still affect

your work. For an example showing why you must consider record length when working in an OpenVMS environment, refer to [Record-Oriented I/O in OpenVMS Binary Files on page 143](#).

For more information about how the operating system affects the transfer of binary data, refer to [Reading UNIX FORTRAN-Generated Binary Data on page 183](#) and [Reading OpenVMS FORTRAN-Generated Binary Data on page 186](#).

Number of Records in a File

OpenVMS USERS In the OpenVMS operating system, the number of records in a file is always known because that information is included in the file header description. For an example of how to view the header description for an OpenVMS file, refer to [Creating Indexed Files on page 207](#).

UNIX USERS In the UNIX operating system, files are not divided into records, unless the application or individual that created it chose to organize it by records when creating the file.

Record-Oriented I/O in OpenVMS Binary Files

All OpenVMS files are divided into records at the operating system level. The basic rule of I/O with record-oriented binary files is that the form of the input and output statements should match. For instance, the statements:

```
WRITEU, unit, A  
WRITEU, unit, B  
WRITEU, unit, C
```

generate three output records, and should be later input with statements of the form:

```
READU, unit, A  
READU, unit, B  
READU, unit, C
```

In contrast, the statement:

```
WRITEU, unit, A, B, C
```

generates a single output record, and should be later input with the single statement:

```
READU, unit, A, B, C
```

NOTE In the examples shown above, it is assumed that the type and size of variables A, B, and C is the same during both the writing and the reading of the data. Otherwise, the data is interpreted differently by the READU commands than it was interpreted previously by the WRITEU commands.

For more information about OpenVMS files, refer to [OpenVMS-Specific Information on page 204](#); that section contains more information on how OpenVMS handles files.

Example — Transferring Record-Oriented Data Under OpenVMS

When writing to OpenVMS files, PV-WAVE always transfers at least a single record of data. If the amount of data required exceeds a single record, more I/O occurs. For example, these commands open a file with 80 character records:

```
OPENW, unit, "filename", 80
```

The statement:

```
WRITEU, unit, FINDGEN(512)
```

causes 2048 bytes to be output (each floating point value takes 4 bytes), and thus causes 26 records to be output ($2048/80 = 25.6$). The last record is not entirely full, and is padded at the end with zeroes.

On later input, the same rule is applied in reverse — 26 records are read, and the unused portion of the last one is discarded.

UNIX USERS This example does not apply to the UNIX operating system, since UNIX files are not record-oriented.

Types of Input and Output

PV-WAVE divides I/O into two categories. These are summarized, along with a brief discussion of advantages and disadvantages, in the following table:

Comparison of Binary and Human-Readable Input/Output

	Advantages	Disadvantages
Binary I/O	<p>Binary I/O is the simplest and most efficient form of I/O.</p> <p>Binary data is more compact than ASCII data</p>	<p>Binary data is not always portable. Binary data files can only be moved easily to and from computers that share the same internal data representation.</p> <p>Binary data is not directly human readable, so you can't type it to a workstation's screen or edit it with a text editor.</p>
ASCII I/O	<p>ASCII data is very portable. It is easy to move ASCII data files to various computers, even computers running different operating systems, as long as they all use the ASCII character set.</p> <p>ASCII data can be edited with a text editor or typed to the workstation's screen because it uses a human readable format.</p>	<p>ASCII I/O is slower than binary I/O because of the need to convert between the internal binary representation and the equivalent ASCII characters.</p> <p>ASCII data requires more space than binary data to store the same information.</p>

Each Type of I/O has Pros and Cons

The type of I/O you use will be determined by considering the advantages and disadvantages of each method. Also, when transferring data to or from other programs or systems, the type of I/O is determined by the application. The following suggestions are intended to give a rough idea of the issues involved, although there are always exceptions:

- Data that needs to be human readable should be written using a human-readable character set. The two main character sets in use are ASCII and EBCDIC; the PV-WAVE documentation assumes that you are using ASCII. The PV-WAVE routines for human-readable I/O are listed in [ASCII I/O — Free Format on page 149](#) and [ASCII I/O — Fixed Format on page 149](#).
- Images and large data sets are usually stored and manipulated using binary I/O in order to minimize processing overhead. The ASSOC function is often the natural way to access such data, and thus is an important function to understand. The ASSOC function is discussed in [Associated Variable Input and Output on page 194](#).

- Images stored in the TIFF format can be easily transferred using the DC_READ_TIFF and DC_WRITE_TIFF functions. Device Independent Bitmap (DIB) images can be transferred with the DC_READ_DIB and DC_WRITE_DIB functions. Other images, either 8-bit or 24-bit, are transferred with the DC_READ_*_BIT and DC_WRITE_*_BIT functions, where the * represents either an 8 or a 24, depending on the type of image data that you have. The various DC routines that can be used to transfer image data are discussed in *Input and Output of Image Data* on page 175.
- Data that needs to be portable should be written using the ASCII character set. Another option is to use XDR (eXternal Data Representation) binary files by specifying the *Xdr* keyword with the OPEN procedures. This is especially important if you intend to exchange data between computers with markedly different internal binary data formats. XDR is discussed in *External Data Representation (XDR) Files* on page 188.
- For ASCII files, freely formatted I/O is easier to use than explicitly formatted I/O, and is almost as easy as binary I/O, so it is often a good choice for small files where there is no strong reason to prefer one method over another. Free format I/O is discussed in *Free Format Input and Output* on page 151.
- The easiest routines to use for the transfer of both images and formatted ASCII data are the DC_READ and DC_WRITE routines. They are easy to use because they automatically handle many aspects of data transfer, such as opening and closing the data file. The “DC” routines are introduced in the next section, *Functions for Simplified Data Connection*.

Functions for Simplified Data Connection

PV-WAVE includes a group of I/O functions that begin with the two letters “DC”; this group of functions has been provided to simplify the process of getting your data in and out of PV-WAVE. This group of I/O functions does not replace the READ, WRITE, and PRINT commands, but does provide an easy-to-understand alternative for most I/O situations.

NOTE The DC_* routines that import and export ASCII data do not support the transfer of data into or from structures. An exception to this is the !DT, or date/time, structure. It is possible to transfer date/time data using DC_* routines.

The functions DC_READ_FREE and DC_READ_FIXED are well-suited for reading column-oriented data; there is no need to use the looping construct necessitated by other PV-WAVE procedures used for reading formatted data. The functions DC_WRITE_FREE and DC_WRITE_FIXED are equally well-suited for writing

column-oriented ASCII data files. To see a figure showing a sample column-oriented file, refer to [Figure 8-1](#) on page 140.

The DC functions are easy to use because they automatically handle many aspects of data transfer, such as opening and closing the data file. Another advantage of the DC I/O commands is that they recognize C-style format strings, even though all other PV-WAVE I/O routines recognize only FORTRAN-style format strings.

NOTE By default, DC_WRITE_FREE generates CSV (Comma Separated Value) ASCII data files, and the corresponding function, DC_READ_FREE, easily reads CSV files.

For specific information about any of the DC routines, refer to examples later in this chapter, or refer to individual function descriptions in the PV-WAVE Reference. For information on the two routines used to perform DC routine error checking, refer to [Other I/O Related Routines](#) on page 150.

Binary I/O Routines

Binary I/O transfers the internal binary representation of the data directly between memory and the file without any data conversion. Use it for transferring images or large data sets that require higher efficiency. The routines for binary I/O are shown in the following table:

Routines for Binary Input/Output

Function	Description
READU	Read binary data from the specified file unit.
WRITEU	Write binary data to the specified file unit.
DC_WRITE_8_BIT DC_READ_8_BIT	Write (or read) binary 8-bit data to (or from) a file without having to explicitly choose a LUN.
DC_WRITE_24_BIT DC_READ_24_BIT	Write (or read) binary 24-bit data to (or from) a file without having to explicitly choose a LUN.
DC_WRITE_TIFF DC_READ_TIFF	Write (or read) TIFF image data. You do not have to explicitly choose a LUN.
ASSOC	Map an array definition to a data file, providing efficient and convenient direct access to binary data.

Routines for Binary Input/Output (Continued)

Function	Description
GET_KBRD	Read single characters from the keyboard.

For more information about the routines shown in the previous table, refer to [Input and Output of Binary Data on page 174](#), [Associated Variable Input and Output on page 194](#), and [Getting Input from the Keyboard on page 203](#).

ASCII I/O Routines

ASCII data is useful for storing data that needs to be human readable or easily portable. ASCII I/O works in the following manner:

- **Input** — ASCII characters are read from the input file and converted to an internal form.
- **Output** — The internal binary representation of the data is converted to ASCII characters that are then written to the output file.

PV-WAVE provides a number of routines for transferring ASCII data; these routines are listed in [ASCII I/O — Free Format on page 149](#) and [ASCII I/O — Fixed Format on page 149](#).

Choosing Between Free or Fixed (Explicitly Formatted) ASCII I/O

ASCII I/O is subdivided further into two categories; the two categories are compared below.

Fixed Format I/O

You provide an explicit format string to control the exact format for the input or output of the data. For a column-oriented data file, with data going into more than one variable, this implies that the values in the input or output file line up in well-defined, fixed-width columns, as shown earlier in [Figure 8-1 on page 140](#).

Because the data values end up being restricted to certain locations on the line, this style of I/O is called *fixed format I/O*. The exact format of the character data is specified to the I/O procedure using a format string (via the *Format* keyword). If no format string is given, default formats for each type of data are applied.

Free Format I/O

PV-WAVE uses default rules to format the data and uses delimiters to differentiate between different data values in the file. During input, the values in the file do not

have to line up with one another because PV-WAVE is not imposing a rigid structure (format) on the file.

You do not have to decide how the data should be formatted because, in the case of input, PV-WAVE automatically looks for delimiters separating data values, and in the case of output, automatically places delimiters between adjacent data values. Because the values are “free” to be anywhere on the line, as long as they are clearly separated by delimiters, this style of I/O is called *free format I/O*.

ASCII I/O — Free Format

Routines for Freely Formatted ASCII I/O

Procedure	Description
PRINT	Write ASCII data to the standard output file (LUN -1).
READ	Read ASCII data from the standard input file (LUN 0).
PRINTF READF	Write (or read) ASCII data to (or from) the specified LUN.
DC_WRITE_FREE DC_READ_FREE	Write (or read) ASCII data to (or from) a file without having to explicitly choose a LUN.

For all the routines listed in the previous table, you do not have to provide a format string to transfer the data. (Because the values in the file are all separated with delimiters, no format string is needed.) The free format I/O routines are discussed in more detail in [Free Format Input and Output on page 151](#).

ASCII I/O — Fixed Format

Routines for Explicitly Formatted ASCII I/O

Procedure	Description
PRINT	Write ASCII data to the standard output file (LUN -1).
READ	Read ASCII data from the standard input file (LUN 0).
PRINTF READF	Write (or read) ASCII data to (or from) the specified LUN.

Routines for Explicitly Formatted ASCII I/O (Continued)

Procedure	Description
DC_WRITE_FIXED DC_READ_FIXED	Write (or read) ASCII data to (or from) a file without having to explicitly choose a LUN.

For all the routines shown in the previous table, you use the *Format* keyword to provide the format string that is used to transfer the data. The first routines listed (PRINT, READ, PRINTF, READF) recognize FORTRAN-like formats; the DC routines accept either C or FORTRAN format strings. The explicit format I/O routines are discussed in more detail in [Explicitly Formatted Input and Output on page 155](#).

NOTE The STRING function can also generate ASCII output that is sent to a string variable instead of a file. For more information about the STRING function, refer to a later section, [Using the STRING Function to Format Data on page 173](#).

Other I/O Related Routines

In addition to performing I/O to an open file, there are several routines that provide other file management capabilities. These additional routines are shown in the following table:

Additional I/O Routines

Procedure	Description
GET_LUN FREE_LUN	Allocate and free LUNs.
FINDFILE	Locate files that match a file specification.
FLUSH	Ensure all buffered data for a LUN has actually been written to the file.
POINT_L UN	Position the file pointer.
EOF	Check for the end-of file condition.
INFO, /Files	Print information about open files.
FSTAT	Get detailed information about any LUN.

Additional I/O Routines (Continued)

Procedure	Description
DC_ERROR_MSG	Returns the text string associated with the negative status code generated by a “DC” data import/export function that does not complete successfully.
DC_OPTIONS	Sets the error message reporting level for all “DC” import/export functions.

For additional information about DC_ERROR_MSG and DC_OPTIONS, refer to their descriptions in the PV-WAVE Reference. For more information about the rest of the routines shown in the previous table, refer to a later section, [Miscellaneous File Management Tasks on page 199](#).

Free Format Input and Output

Free format ASCII I/O is extremely easy to use. The main advantage of free formatted ASCII I/O is that you do not have to provide a format string to format the data, because you assume that adjacent values are separated by delimiters.

The routines for performing freely formatted ASCII I/O are listed in [ASCII I/O — Free Format on page 149](#).

Free Format Input

Input is performed on scalar variables. In other words, array and structure variables are treated as collections of scalar variables. For example:

```
Z_hi = INTARR(5)
READ, Z_hi
```

causes PV-WAVE to read (from the standard input stream) five separate values to fill each element of the variable Z_hi.

Input data must be separated by commas or white space (tabs and blank spaces).

If the current input line is empty and there are variables left to be filled, another line is read. If the current input line is not empty but there are no variables left to be filled, the remainder of the line is ignored.

When reading into a variable with data type String, all characters remaining in the current input line are placed into the string.

When reading into numeric variables, PV-WAVE attempts to convert the input into a value of the expected type. Decimal points are optional and exponential (scientific) notation is allowed. If a floating-point value is provided for an integer variable, the value is truncated.

Importing String Data

When PV-WAVE reads strings using free formats, it reads to the end of the line. For this reason, it is usually convenient to place string variables at the end of the list of variables to be input. For example, if *S* is a string variable and *I* is an integer, do *not* do this:

```
READ, S, I
    ; Read into the string first.

: hello world 34
    ; PV-WAVE prompts for input. The user enters a string value
    ; followed by an integer.

: 34
    ; Because this is a freely formatted read statement, and the READ
    ; procedure does not recognize delimiters inside strings, the entire
    ; previous line was placed into the string variable S, and PV-WAVE
    ; still expects a value to be entered for I. Consequently, PV-WAVE
    ; prompts for another line.

PRINT, S
    ; Show the result of S.
```

results in the output:

```
'Hello world 34'
```

Importing Data into Complex Variables

Complex scalar values are treated as two floating-point values. When reading into a variable of complex type, the real and imaginary parts must be separated by a comma and surrounded by parentheses. If only a single value is provided, it is taken as the real part of the variable, and the imaginary part is set to zero.

Here are some examples of how to enter complex data from the keyboard:

```
Z_lo = COMPLEX(0)
    ; Create a complex variable.

READ, Z_lo
: (3,4)
    ; PV-WAVE prompts for input: Z_lo is set to COMPLEX(3,4).

READ, Z_lo
```

```

: 50
; PV-WAVE prompts for input: Z_lo is set to COMPLEX(50,0).

```

Importing Data into a Structure

The following statements demonstrate how to load data into a complicated structure variable and then print the results:

```

A = {alltypes, a:0b, b:0, c:0L, d:1.0, e:1D,$
     f:complex(0), g:'string', e:fltarr(5)}
; Create a structure named "alltypes" that contains all eight of
; the basic data types, as well as a floating-point array.

READ, A
: 1 2 3 4 5 (6,7) eight
; Read freely formatted ASCII data from the standard input;
; PV-WAVE prompts for input. Enter values for the first six numeric
; fields of A, and the string. Notice that the complex value was
; specified as (6,7). If the parentheses had been omitted, the complex
; field of A would have received the value COMPLEX(6,0), and the 7
; would have been used for the next field. When reading into a string
; variable with the READ procedure, and no format string has been
; provided, PV-WAVE starts from the current point in the input and
; continues to the end of the line. Thus, the values intended for the
; rest of the structure are entered on a separate line, as shown in the next step.

: 9 10 11 12 13
; There are still fields of A that have not received data, so PV-WAVE
; prompts for another line of input.

PRINT, A
; Show the result.

```

Executing these statements results in the following output:

```

{ 1      2      3      4.00000      5.0000000
 (6.00000, 7.00000) eight
 9.00000 10.0000 11.0000 12.0000 13.0000 }

```

When producing the output, PV-WAVE uses default formats for formatting the values, and attempts to place as many items as possible onto each line. Because the variable A is a structure, curly braces, “{” and “}”, are placed around the output. The default formats are shown in [Free Format Output on page 155](#).

Importing Date/Time Data

The following statements show how to read a file that contains some data values and also some chronological information about when those data values were recorded. The name of the file is `events.dat`:

```

01/01/92    05:45:12    10
02/01/92    10:10:10    15.89
05/15/92    02:02:02    14.2

```

This example shows how to use the `DC_READ_FREE` function to read this data. When using `DC_READ_FREE`, the date data and the time data can be placed into the same date/time structure using predefined templates. To see a complete list of the date/time templates, refer to [Date/Time Templates on page 159](#).

To read the date/time from the first two columns into date/time variables and then read the third column of floating point data into another variable, use the following statements:

```

date1 = REPLICATE(!DT, 3)
; The system structure definition of date/time is !DT. Date/time
; variables must be defined as !DT arrays before being used if the
; date/time data is to be read as such.

status = DC_READ_FREE("events.dat", $
    date1, date1, float1, /Column, $
    Dt_Template=[1, -1])
; The variables date1 is used twice, once to read the date data and
; once to read the time data.

```

To see the values of the variables, you can use the `PRINT` command:

```

FOR I = 0,2 DO BEGIN
    PRINT, date1(I), float1(I)
; Print one row at a time.
ENDFOR

```

Executing these statements results in the following output:

```

{ 1992 01 01 05 45 12.00 } 10.0000
{ 1992 02 01 10 10 10.00 } 15.8900
{ 1992 05 15 02 02 02.00 } 14.2000

```

Because `date1` is a structure, curly braces, “{” and “}”, are placed around the output. When displaying the values of `date1` and `float1`, PV-WAVE uses default formats for formatting the values, and attempts to place as many items as possible onto each line.

For more information about the internal organization of the `!DT` system structure, refer to *Working with Date/Time Data* in the PV-WAVE User’s Guide. For more information about using the `DC_READ_FREE` function with date/time data, refer to its description in the *PV-WAVE Reference*.

Free Format Output

The format used to output numeric data is determined by the data type.

Output Formats Used When Writing Data

Data Type	Output Formats Used by PRINT, PRINTF, and DC_WRITE_FREE
Byte	I4
Integer	I8
Long Integer	I12
Float	G13.6
Double	G16.8
Complex	'(, G13.6, ',, G13.6,)'
String	A (character data)

NOTE When writing string data, each string (or element of a string array) is written to the file, flanked with a delimiter on each side. This implies that the strings should not contain delimiter characters if you intend use free format input at a later time to read the file.

The current output line is filled with characters until one of the following happens (in the following order):

- (a) There is no more data to output.
- (b) The output line is full. The line width is controlled by the device characteristics, as determined by the terminal characteristics (tty), or the file's record characteristics (disk file).
- (c) An entire row is output in the case of multidimensional arrays.

When writing the contents of a structure variable to a file, its contents are bracketed with curly braces, “{” and “}”.

Explicitly Formatted Input and Output

Explicit formatting allows a great deal of flexibility in specifying exactly how ASCII data is formatted. Formats are specified using a syntax that is very similar

to that used in FORTRAN or C format statements. Scientists and engineers already familiar with FORTRAN or C will find PV-WAVE formats easy to write.

The routines for performing explicitly (fixed) formatted ASCII I/O are listed in [ASCII I/O — Fixed Format on page 149](#).

All data is handled in terms of basic data types. Thus, an array is considered to be a collection of scalar data elements, and a structure is processed in terms of its basic components. Complex scalar values are treated as two floating-point values.

Using FORTRAN or C Formats for Data Transfer

All formatted ASCII I/O routines recognize FORTRAN-style format strings, and for formatted I/O routines that begin with the prefix “DC”, C-style format strings can be used, as well. The format string specifies the format in which data is to be transferred as well as the data conversion required to achieve that format.

FORTRAN and C data transfer codes are discussed in more detail in [Appendix A, FORTRAN and C Format Strings](#). You can also find examples of using format codes with any of the descriptions of the commands for transferring explicitly formatted data; these descriptions are in the PV-WAVE Reference.

How is the Format String Interpreted?

The variable names provided in a call to an I/O routine comprise the *variable list*. The variable list specifies the data to be moved between memory and the file. The *Format* keyword can be included in the parameter list of an ASCII I/O routine to provide a format string that explicitly specifies the appearance of the transferred data.

The format string is traversed from left to right, processing each record terminator and format code until an error occurs, or until no variables are left in the variable list. In FORTRAN-style formats, the comma field separator serves no purpose except to delimit the format codes.

When reading or writing data from the file, the data is formatted according to the format string. If the data type of the input data does not agree with the data type of the variable that is to receive the result, PV-WAVE performs type conversion if possible, and otherwise, issues a type conversion error and stops.

If the last closing parenthesis of the format string is reached and there are no variables left in the variable list, then format processing terminates. If, however, there are still variables to be processed in the variable list, then part or all of the format specification is reused. This process is called format reversion, and is discussed more in [Format Reversion on page 157](#).

In a FORTRAN-style format string, when a slash (/) or newline (↵) record terminator is encountered, the current record is completed and a new one is started. For output, this means that a new line is started. For input, it means that the rest of the current input record is ignored, and the next input record is read.

When a format code that does not transfer data is encountered, it is processed according to its meaning. When a format code that transfers data is encountered, it is matched up with the next entry in the variable list. All recognized format codes are listed in [Appendix A, FORTRAN and C Format Strings](#).

CAUTION It is an error to specify a variable list with a format string that doesn't contain a format code that transfers data to or from the variable list. Because the command expects to transfer data to the variables in the variable list, an infinite loop would result. For example, consider the following statement:

```
PRINTF, 1, names, years, salary, Format= $
      ('Name', 28X, "Year", 4X, "Total Salary")'
```

This statement results in a message stating that an infinite loop is detected (because no data is being transferred to the named variables), and thus execution is being halted. On the other hand, the following statement is acceptable because there are no variables included as part of the parameter list:

```
PRINTF, 1, Format= $
      ('Name', 28X, "Year", 4X, "Total Salary")'
```

Should I Use a FORTRAN or C Format?

The only functions that recognize the C format strings are those that begin with the prefix “DC”. The DC functions are the ones that have been designed specifically to simplify the process of transferring data.

All other procedures and functions that transfer data recognize only the FORTRAN-style format statements. The FORTRAN format codes that are recognized by PV-WAVE are listed in [Appendix A, FORTRAN and C Format Strings](#).

Format Reversion

Format reversion is a way to transfer a lot of data with a format string that, at first glance, seems to be “too short”. When using format reversion, the current record is terminated, a new one is started, and format control reverts to the first group repeat specification that does not have an explicit repeat factor.

NOTE If you are using a C-style format string, the entire format string is reused.

If the format does not contain a group repeat specification, format control returns to the initial opening parenthesis of the format string. For example, the command:

```
PRINT, Format = $
      ' ("The values are: ", 2("<", I1, ">"))', INDGEN(6)
```

results in the output:

```
The values are: <0><1>
<2><3>
<4><5>
```

The process involved in generating this output is:

- 1) Output the string, “The values are:”.
- 2) Process the group specification and output the first two values. The end of the format specification is encountered, so end the output record. Data remains, so revert to the group specification

```
      2("<", I1, ">")
```

using format reversion.

- 3) Repeat the second step until no data remains, and then for output, end the output record, or for input, stop reading data values.

At this point, format processing is complete. To see other examples of format reversion, refer to [Appendix A, FORTRAN and C Format Strings](#).

Transferring Date/Time Data

PV-WAVE supports the transfer of date/time data in and out of data files. Some examples of date/time data which you may wish to read are:

```
10/20/92 12:00:10.90
21/01/93 11:06:29.0875
10-JAN-1992 12:46
MAR:1993 $25440.0
```

Although there are several ways to read date/time data, you would want to choose the method that makes the most sense for your application and best matches the style of program you are writing:

- **Use classical programming constructs** — With this method, you open the file, loop to read the data, close the file, and run the data through one of the date/time conversion routines. This method is shown below in [Method 1 — Read the File with READF](#) on page 160.

- **Use one of the DC_READ routines** — With this method, you define one or more variables that use the date/time system structure organization, and then use DC_READ_FIXED or DC_READ_FREE to transfer the data into those variables using date/time templates. This method is shown in [Method 2 — Read the File with DC_READ_FIXED on page 162](#).

Method 2 utilizes the DC_READ routines. As discussed in [Functions for Simplified Data Connection on page 146](#), the DC routines have been provided as yet another alternative for the process of transferring data in and out of PV-WAVE.

Date/Time Templates

The templates that can be used with the formatted ASCII I/O routines are shown in the following table.

Templates for Transferring Date/Time Data

Number	Template Description
1	MM*DD*YY[YY]
2	DD*MM*YY[YY]
3	ddd*YY[YY]
4	DD*mmm[mmmmmm]*YY[YY]
5	[YY]YY*MM*DD
-1	HH*MnMn*SS[.SSSS]
-2	HHMnMn

M = Month, D = Day, Y = Year, H = Hour, Mn = Minute, S = Second

The asterisk (*) shown above represents a delimiter that separates the different fields of data. The delimiter can also be a slash (/), a colon (:), a hyphen (-), or a comma (,).

Positive template numbers are for transferring date data, while negative template numbers are for transferring time data. To see examples of the types of data that can be transferred using each of these templates, refer to *Working with Date/Time Data* in the PV-WAVE User's Guide.

Example — Reading Date/Time Data

Assume that you have a file, `chrono.dat`, that contains some data values, including a three-character label showing where the data was recorded, and also some chronological information about when those data values were recorded:

```
LAM 10/02/90 09:32:00 10.00 32767
COS 10/02/90 09:36:00 15.89 99999
SNV 10/02/90 09:37:00 14.22 87654
```

Method 1 — Read the File with READF

To read the label from the first column into a string variable, the date and time from the second and third columns into one date/time variable and read the fourth and fifth columns of data into another two variables, use the following commands:

```
loc = STRARR(3) & calib = LONARR(3)
date1 = STRARR(3) & time1 = STRARR(3)
decibels = FLTARR(3)
    ; Create variables to hold the location, calibration, date, time,
    ; and decibel level.

OPENR, 1, 'chrono.dat'
    ; Open data file for input.

locs = ' ' & date1s = locs & time1s = date1s
    ; Define scalar strings.

calibs = 1L
    ; Define a long integer scalar.

I = 0
    ; Initialize counter.

WHILE (NOT EOF(1)) DO BEGIN
    ; Loop over each record of data.
        READF, 1, locs, date1s, $
            time1s, decibelss, calibs, Format = $
            "(A3, 2(1X, A8), 1X, F5.2, 1X, I5)"
        ; Read scalars; the first three are string variables, the fourth is
        ; a float, and the fifth one is an integer.
            loc(I) = locs & date1(I) = date1s & $
            time1(I) = time1s & calib(I) = calibs $
            & decibels(I) = decibelss
        ; Store in each vector.
            IF I LE 2 THEN I = I+1 ELSE CLOSE, 1 & $
            STOP, "Too many records."
        ; Increment counter and check for too many records.
    ENDWHILE
```

```

CLOSE, 1
    ; Close the file.

my_dt_arr = STR_TO_DT(date1, time1, $
    Date_Fmt=1, Time_Fmt = -1)
    ; Use one of the conversion utilities, STR_TO_DT, to convert
    ; the strings to date/time data. The variable date1 uses
    ; Template 1, while the variable time1 uses Template -1. The
    ; result array, my_dt_arr, holds both the MM/DD/YY and the
    ; HH:MM:SS data.

```

Another alternative is to read the time and date data as integers instead of strings. This is the approach you must take if your time/date data does not have the customary delimiters separating the months, days, and years, or the hours, minutes, and seconds, as shown in the sample file below:

```

LAM 100290 093200 10.00 32767
COS 100290 093600 15.89 99999
SNV 100290 093700 14.22 87654

```

In this situation, instead of defining `date1` and `time1` to be strings, you would define different variables — one for each component of the date/time data:

```

year = INTARR(3) & mon = year & day = year
hour = INTARR(3) & min = hour & sec = hour
    ; Define integer arrays to hold the months, days, years, hours,
    ; minutes, and seconds data.

years = 0 & mons = 0 & days = 0
hours = 0 & mins = 0 & secs = 0
    ; Define integer scalars for use inside the read loop.

loc = STRARR(3) & calib = LONARR(3)
decibels = FLTARR(3)
    ; Create variables to hold the location, calibration, and decibel level.

locs = ' ' & calibs = 1L
    ; Initialize string and long integer scalars.

OPENR, 1, 'chrono.dat'
    ; Open data file for input.

I = 0
    ; Initialize counter.

WHILE NOT EOF(1) DO BEGIN
    ; Beginning of read loop.
        READF, 1, locs, mons, days, years, $
            hours, mins, secs, $
            decibelss, calibs, Format = $
            "(A3, 2(1X, 3(I2)), 1X, F5.2, 1X, I5)"

```

```

; Read scalars; the first one is a string variable, the next six
; are integer variables, the eighth is a float, and the ninth one is an integer.
    year(I) = years & mon(I)= mons
    day(I) = days & hour(I) = hours
    min(I) = mins & sec(I) = secs
; Store in each vector.
    IF I LE 2 THEN I = I+1 ELSE CLOSE, 1 & $
        STOP, "Too many records."
; Increment counter and check for too many records.
ENDWHILE
CLOSE, 1

```

Now that the date/time data has been read into variables, these variables can be used as input to the conversion utility, VAR_TO_DT:

```

my_dt_arr = VAR_TO_DT(year, mon, day, hour, min, sec)
; Use one of the conversion utilities, VAR_TO_DT, to convert the
; variables to date/time format.

```

Regardless of whether you read the data as strings and use the STR_TO_DT function for conversion, or read the data as integer values and use the VAR_TO_DT function for conversion, the value of the my_dt_arr array is the same. You can easily view the contents of my_dt_arr using the PRINT command:

```

PRINT, my_dt_arr
{ 1990 10 2 9 32 0.00000 86946.397 0 }
{ 1990 10 2 9 36 0.00000 86946.400 0 }
{ 1990 10 2 9 37 0.00000 86946.401 0 }

```

Because the variable my_dt_arr is a structure, curly braces, “{” and “}”, are placed around the output. For more information about the internal organization of date/time structures, refer to *Working with Date/Time Data* in the PV-WAVE User’s Guide.

Method 2 — Read the File with DC_READ_FIXED

The following statements present another method for reading date/time data into variables (the same data that was used for Method 1). Because this method utilizes the DC_READ_FIXED function, it is able to use a C-style format string to read the data. The data file is repeated below for your convenience:

```

LAM 10/02/90 09:32:00 10.00 32767
COS 10/02/90 09:36:00 15.89 99999
SNV 10/02/90 09:37:00 14.22 87654

```

This method automatically handles the string to date and string to time conversion, although it does require that the date/time variable, `date1`, be predefined as a date/time system structure:

```
date1 = REPLICATE(!DT, 3)
    ; The system structure definition of date/time is !DT. Date/time
    ; variables must be defined as !DT structure arrays before being
    ; used if the date/time data is to be read as such.

loc = STRARR(3) & calib = LONARR(3)
decibels = FLTARR(3)
    ; Explicitly define the string, integer, and floating-point vectors.

status = DC_READ_FIXED("chrono.dat", $
    loc, date1, date1, decibels, calib, $
    /Column, Format="%s %8s %8s %f %d", $
    Dt_Template=[1, -1])
    ; DC_READ_FIXED handles the opening and closing of the file.
    ; It transfers the values in "chrono.dat" to the variables in the
    ; variable list, working from left to right. The variable date1
    ; appears in the variable list twice, once to read the date data
    ; and once to read the time data.
```

Notice how in this method, the variable `date1` is specified twice. Because `date1` is defined as a date/time structure, it has predefined tags for the various classes of chronological information. By including `date1` in the variable list twice, both the date data and the time data is combined in the same !DT structure, using two different date/time templates (1 for date values and -1 for time values).

For more information about the internal organization of the !DT system structure, refer to *Working with Date/Time Data* in the PV-WAVE User's Guide.

Reading, Sorting, and Printing Tables of Formatted Data

Explicitly formatted I/O has the power and flexibility to handle almost any kind of formatted data. A common use of explicitly formatted I/O is to read and write tables of data.

Example — Reading Data From a Word-Processing Application

Frequently, data files are produced by a word-processing or spreadsheet application program. This example shows how to import this kind of data into variables.

Method 1 — Read the File with READF

Consider a data file containing employee data records. Each employee has a name (String – 16 columns) and the number of years they have been employed (Integer

– 3 columns) on the first line. The next two lines contain their monthly salary for the last twelve months. A sample file named `bullwinkle.wp` with this format might look like:

Bullwinkle			10		
1000.0	9000.97	1100.0			2000.0
5000.0	3000.0	1000.12	3500.0	6000.0	900.0
Boris			11		
400.0	500.0	1300.10	350.0	745.0	3000.0
200.0	100.0	100.0	50.0	60.0	0.25
Natasha			10		
950.0	1050.0	1350.0	410.0	797.0	200.36
2600.0	2000.0	1500.0	2000.0	1000.0	400.0
Rocky			11		
1000.0	9000.0	1100.0	0.0	0.0	2000.37
5000.0	3000.0	1000.01	3500.0	6000.0	900.12

The following statements read data with the above format and produce a summary of its contents:

```

OPENR, 1, 'bullwinkle.wp'
    ; Open data file for input.

name = '' & years = 0 & salary = FLTARR(12)
    ; Create variables to hold the name, number of years, and monthly
    ; salaries. The type of each variable is automatically determined by
    ; the type of initial value it is given.

PRINT, 'Name      Years      Yearly Salary'
    ; Output a heading for the summary.

PRINT, '-----'
    ; Output a ruling line for the heading.

WHILE (NOT EOF(1)) DO BEGIN
    ; Loop over each employee.
        READF, 1, name, years, salary, $
            Format = "(A16, I3, 2(/, 6F10.2))"
        ; Read the data on the next employee.
            PRINT, Format = "(A16, I5, 5X, F10.2)", $
                name, years, TOTAL(salary)
        ; Output the employee information. Use the TOTAL function
        ; to compute the yearly salaries from the monthly salaries.

ENDWHILE
CLOSE, 1

```

The output from executing the statements shown above is:

Name	Years	Yearly Salary
Bullwinkle	10	32501.09
Boris	11	6805.35
Natasha	10	14257.36
Rocky	11	32500.50

DC_READ_FIXED is not used in this method because the file, as it is shown on page 164, is neither a column-organized file or a row-organized file; it falls somewhere in between. In other words, the name and years-of-service data are organized by columns, while the yearly salary data is organized in rows. But the file can be rearranged, as shown below in the next method, and then using DC_READ_FIXED becomes a viable (and time-saving) option.

Method 2 — Read the File with DC_READ_FIXED

Suppose the file was much longer than we are able to show in this example, and you wanted to use PV-WAVE's powerful data connection and table building utilities to read and process the data. If the file was organized a bit differently, DC_READ_FIXED could be used to read the data. Then, the BUILD_TABLE function could be used to quickly organize the data in a table structure. The new file organization is shown below:

Bullwinkle	Boris	Natasha	Rocky		
10	11	10	11		
1000.0	9000.97	1100.0			2000.0
5000.0	3000.0	1000.12	3500.0	6000.0	900.0
400.0	500.0	1300.10	350.0	745.0	3000.0
200.0	100.0	100.0	50.0	60.0	0.25
950.0	1050.0	1350.0	410.0	797.0	200.36
2600.0	2000.0	1500.0	2000.0	1000.0	400.0
1000.0	9000.0	1100.0	0.0	0.0	2000.37
5000.0	3000.0	1000.01	3500.0	6000.0	900.12

The following statements read the data file shown above and display a summary of its contents on the screen:

```

name = STRARR(4) & years = INTARR(4)
salary = FLTARR(12, 4)
    ; Create variables to hold the name, number of years, and monthly
    ; salaries.

status = DC_READ_FIXED('bullwinkle.wp', $
    name, years, salary, Format= "(4A16, " + $
    "/", I3, 3(10X,I3), /, 48(F7.2, 3X))", $
    Ignore=["$BLANK_LINES"])
    ; DC_READ_FIXED handles the opening and closing of the file. It
    ; transfers the values in "bullwinkle.wp" to the variables in the variable
    ; list, working from left to right. The two slashes in the format string
    ; force DC_READ_FIXED to switch to a new record in the input file.
    ; When reading row-oriented data, each variable is "filled up" before
    ; any data is transferred to the next variable in the variable list. The
    ; value of the Ignore keyword insures that all blank lines are skipped
    ; instead of being interpreted as data.

PRINT, 'Name      Years      Yearly Salary'
PRINT, '-----'
    ; Print a heading and ruling line for the heading.

yearly_salary = FLTARR(4)
FOR I = 0,3 DO BEGIN
    ; One row at a time, total the monthly salaries.
    yearly_salary(I) = TOTAL(salary[*],I)
        ; Use array subscripting notation to total all twelve months of
        ; salary for each employee.
ENDFOR

zz = BUILD_TABLE('name, years, yearly_salary')
    ; Create a table structure, with each column of information being an
    ; individual tag of the structure.

FOR I = 0,3 DO BEGIN
    ; Print one row at a time.
    PRINT, Format="(A16, 3X, I5, 5X, F10.2)", $
        zz(I).name, zz(I).years, zz(I).$
        yearly_salary
        ; Print the employee information. Each column of information
        ; is now a tag of the zz table.
ENDFOR

```

NOTE You do not need to understand structures to work with tables. For a comparison of tables and structures, refer to the *Creating and Querying Tables* in the PV-WAVE User's Guide.

Just like in Method 1, the output from executing the statements shown above is:

Name	Years	Yearly Salary
Bullwinkle	10	32501.09
Boris	11	6805.35
Natasha	10	14257.36
Rocky	11	32500.50

Now you could easily enter commands to sort the table, using a variety of criteria. Suppose you want to rearrange the table (in descending order) so that the employee with the highest salary is listed first:

```
by_val = QUERY_TABLE (zz, $
    '* Order By yearly_salary Desc')
FOR I = 0,3 DO BEGIN
    ; Print one row at a time.
    PRINT, Format="(A16, 3X, I5, 5X, F10.2)", $
        by_val(I).name, by_val(I).years, $
        by_val(I).yearly_salary
    ; Print the employee information. Each column of information
    ; is a tag of the by_val table.
ENDFOR
```

The output is now sorted in descending order by yearly salary:

Name	Years	Yearly Salary
Bullwinkle	10	32501.09
Rocky	11	32500.50
Natasha	10	14257.36
Boris	11	6805.35

Now suppose you want to rearrange the table (in ascending alphabetical order) so that the employees are listed alphabetically:

```

by_val = QUERY_TABLE(zz, '* Order By name')
FOR I = 0,3 DO BEGIN
    ; Print one row at a time.
    PRINT, Format="(A16, 3X, I5, 5X, F10.2)", $
        by_val(I).name, by_val(I).years, $
        by_val(I).yearly_salary
    ; Print the employee information.
ENDFOR

```

The output is now sorted in ascending alphabetic order:

Name	Years	Yearly Salary
Boris	11	6805.35
Bullwinkle	10	32501.09
Natasha	10	14257.36
Rocky	11	32500.50

For more information about functions for sorting and organizing table structures, and the keywords that can be used inside the QUERY_TABLE sort string, refer to the PV-WAVE User's Guide.

Reading Records Containing Multiple Array Elements

Frequently, data is written to files with each record containing single elements of more than one array. For example, a file might contain observations of altitude, pressure, temperature, and velocity, with each line (or record) containing a value for each of the four variables. Data files like this are called *record-oriented files*, and PV-WAVE offers several different ways to read them, as shown below.

Example 1 — Column-oriented FORTRAN Write

A FORTRAN program that writes the data and the PV-WAVE program that reads the data are shown below:

FORTRAN Write

This FORTRAN program writes the data by creating an array with as many columns as there are variables and as many rows as there are elements.

```

DIMENSION ALT(100), PRES(100), TEMP(100),
C   VELO(100)

```



```

aptv = BUILD_TABLE('alt, pres, temp, velo')
; Create a table structure, with each column of information being an
; individual tag of the table.

```

For more information about what can be done with data once it is placed into a table structure, refer to an earlier example on page 165, or refer to the PV-WAVE User's Guide.

Notice that the variables were not predefined with the FLTARR function, as they were with Method 1. Because the variables were not predefined, DC_READ_FIXED creates them all as one-dimensional floating-point arrays dimensioned to match the number of records in the file. For example, suppose that each column of data in aptv.dat contained 280 values. All four variables (alt, pres, temp, and velo) would be created and dimensioned as 280 element vectors.

Example 2 — Row-oriented FORTRAN Write

The same data values may be written without the implied DO list, writing all elements for each variable contiguously and simplifying the FORTRAN write program:

FORTRAN Write

```

DIMENSION ALT(100), PRES(100), TEMP(100),
      C   VELO(100)
OPEN (UNIT=1, STATUS='NEW', FILE='aptv.dat')
      .
      .           Other commands go here.
      .
WRITE (1, '(4(1x,G15.5))') ALT, PRES, TEMP,
      C   VELO
END

```

PV-WAVE Read (Method 1)

Read the data as an uninterrupted stream of values. In other words, read the file as though it contains row-oriented data.

```

alt = FLTARR(100)
; Create a floating-point array to hold the data.

pres = alt & temp = alt & velo = alt
; Create more floating-point arrays, all the same size as alt.

OPENR, 1, 'aptv.dat'
; Open file for input.

```

```

READF, 1, alt, pres, temp, velo
    ; Read the data.
CLOSE, 1
    ; Close the file.

```

PV-WAVE Read (Method 2)

DC_READ_FIXED can be used to read row-oriented data; in fact, this happens by default when the *Column* keyword is omitted from the function call. However, when you are reading row-oriented data, the import variables must be pre-dimensioned so that DC_READ_FIXED knows how many values to store in each of the variables included in the variable list:

```

alt = FLTARR(100)
    ; Create a floating-point array to hold the data.

pres = alt & temp = alt & velo = alt
    ; Create more floating-point arrays, all the same size as alt.

status = DC_READ_FIXED('aptv.dat', alt, $
    pres, temp, velo, Format="%f")
    ; DC_READ_FIXED handles the opening and closing of the file. It
    ; reads values from aptv.dat and stores them in the variables alt, pres,
    ; temp, and velo. By default, the data is read as row-oriented data.
    ; The returned value status can be checked to see if the process
    ; completed successfully.

```

The format string shown in this example (Method 2) may be used only if all of the variables in the variable list are typed as floating-point, because the same C format string is used over and over to read all the data values. For more information on format reversion, (the process of re-using format strings when reading or writing data), refer to [Format Reversion on page 157](#).

NOTE If the variable list contained other data types besides floating-point, the format string would have to be more specific, such as the one used in the next example. Another alternative is to use DC_READ_FREE (instead of DC_READ_FIXED) to read the file, and then you aren't required to supply any format string.

Example 3 — Using a FORTRAN Format String to Read Multiple Array Elements

Assume that the data used is the same as that of the previous examples, but a fifth variable, the name of an observer (which is a string), has been added to the variable list. The FORTRAN output routine and PV-WAVE input routine are shown below:

FORTRAN Write

```
DIMENSION ALT(100), PRES(100), TEMP(100),  
  C VELO(100)  
CHARACTER*10 OBS(100)  
OPEN (UNIT = 1, STATUS = 'NEW', FILE =  
  C 'aptvo.dat')  
  .  
  .           Other commands go here.  
  .  
WRITE (1, ' (4(1X,G15.5), 2X, A)') (ALT(I),  
  C PRES(I), TEMP(I), VELO(I), OBS(I), I = 1,100)  
END
```

PV-WAVE Read (Method 1)

This method involves defining the arrays, defining a scalar variable to contain each value in one record, then writing a loop to read each line into the scalars, and finally storing the scalar values into each array:

```
OPENR, 1, 'aptvo.dat'  
  ; Access file. This example reads files containing from 0 to 100 records.  
  
alt = FLTARR(100)  
  ; Create a floating-point array to hold the data.  
  
pres = alt & temp = alt & velo = alt  
  ; Create more floating-point arrays, all the same size as alt.  
  
obs = STRARR(100)  
  ; Define string array.  
  
obss = ' '  
  ; Define scalar string.  
  
I = 0  
  ; Initialize counter.  
  
WHILE NOT EOF(1) DO BEGIN  
  ; Beginning of read loop.  
  READF, 1, alts, press, $  
    temps, velos, obss, $  
    Format="(4(1X, G15.5), 2X, A10)"  
  ; Read scalars; the last one is a string variable, and by default,  
  ; the first four are floating-point variables.  
  alt(I) = alts & pres(I) = press  
  temp(I) = temps & velo(I) = velos  
  obs(I) = obss  
  ; Store in each vector.
```

```

        IF I LE 99 THEN I = I+1 ELSE CLOSE,1 & $
        STOP, "Too many records."
    ; Increment counter and check for too many records.
ENDWHILE
CLOSE, 1
    ; Close the file.

```

If desired, after the file has been read and the number of observations is known, the arrays may be truncated to the correct length using a series of statements similar to:

```
alt = alt(0:I-1)
```

The above represents a worst case example. Reading is greatly simplified by writing data of the same type contiguously and by knowing the size of the file. Another alternative is to use Method 2, shown below.

TIP One frequently used technique is to include the number of observations in the first record so that when reading the data the size is known.

PV-WAVE Read (Method 2)

The DC_READ_FIXED function is ideal for situations such as this one, where the columns are treated as different data types or the number of lines or records in the file is not known.

```

obs = STRARR(100)
    ; Define string array; let other variables use default floating-point data
    ; type.

status = DC_READ_FIXED('aptvo.dat', $
    alt, pres, temp, velo, obs, /Column, $
    Format="(4(1X, G15.5), 2X, A10)", $
    Resize=[1, 2, 3, 4, 5])
    ; DC_READ_FREE handles the opening and closing of the file. It
    ; reads values from aptvo.dat and stores them in the variables alt,
    ; pres, temp, velo, and obs. The data is being read as column
    ; oriented data.
    ; Because the Resize keyword was included with the function call, all
    ; five variables are resizable and are redimensioned to match the
    ; number of values actually transferred from the file. The returned
    ; value status can be checked to see if the process completed successfully.

```

Using the STRING Function to Format Data

The STRING function is very similar to the PRINT and PRINTF procedures. You can even think of it as a version of PRINT that places its ASCII output into a string

variable instead of a file. If the output is a single line, the result is a scalar string. If the output has multiple lines, the result is a string array, with each element of the array containing a single line of the output.

Example 1 — *STRING* Function without Format Keyword

Three variations using the `STRING` function are shown below:

```
abc = STRING([65B,66B,67B])
abc = STRING([byte('A'),byte('B'),byte('C')])
abc = STRING('A'+ 'B'+ 'C')
```

In all three cases, `abc` has the same value, the string scalar 'ABC'.

Example 2 — *STRING* Function with Format Keyword

The following statements:

```
A = STRING(Format='("The values are:", ' + $
    ', (I))', INDGEN(5))
    ; Create a string array named A.

INFO, A
    ; Display information about A.

FOR I = 0, 5 DO PRINT, A(I)
    ; Print the result.
```

produce the following output:

```

                A                STRING      = Array(6)
The values are:
  0
  1
  2
  3
  4
```

For additional details about the `STRING` function, see its description in the PV-WAVE Reference.

Input and Output of Binary Data

Binary I/O involves the transfer of data between a file and memory without conversion to and from a character representation. Binary I/O is used when efficiency is important and portability is not an issue; it is faster and requires less space than human-readable I/O.

NOTE Binary I/O is almost always used for the transfer of image data, such as TIFF images, or 8- and 24-bit images.

PV-WAVE provides many procedures and functions for performing binary I/O; they are listed in [Binary I/O Routines on page 147](#). All of these routines are described in this section except ASSOC and GET_KBRD; these important functions are discussed in [Associated Variable Input and Output on page 194](#) and [Getting Input from the Keyboard on page 203](#).

Input and Output of Image Data

Images are frequently stored using either 8-bit or 24-bit binary data. 8-bit data is capable of displaying 2^8 different colors, while 24-bit data is capable of displaying 2^{24} different colors.

Windows USERS Windows NT does not support the display of 24-bit color.

Images are treated in the same manner as any variable. Images may be either square or rectangular. There is no restriction placed on the size of images; the limiting factors are the maximum amount of virtual memory available to you by the operating system and the processing time required.

8-bit and 24-bit Image Data

Image data is usually stored in either an 8-bit or 24-bit format:

- **8-bit Format** — Images in 256 shades of gray or 256 discrete colors (sometimes known as “pseudo-color”).
- **24-bit Format** — 3-color RGB (8 bits Red/8 bits Green/8 bits Blue) images.

8-bit images must be stored in a 2-dimensional variable, and 24-bit images must be stored in a 3-dimensional variable. For more information about how the RGB information in 24-bit image data is stored, refer to [Image Interleaving on page 178](#).

NOTE Your workstation or device must support 24-bit color mode if you intend to view 24-bit images with PV-WAVE.

Image Data Input

Image data can be imported using either the READU or the ASSOC commands. However, one of the easiest ways to import image data is to use either the

DC_READ_8_BIT or DC_READ_24_BIT functions. For example, if the file `hero.img` contains a 786432 byte 24-bit image-interleaved image, the function call:

```
status = DC_READ_24_BIT('hero.img', hero, Org=1)
```

reads the file `hero.img` and creates a 512-by-512-by-3 image-interleaved byte array named `hero`.

When you do not pre-dimension the variable, PV-WAVE creates either a two- or three-dimensional byte variable, depending on whether you are using DC_READ_8_BIT or DC_READ_24_BIT. It also checks the total number of bytes in the file and automatically dimensions the import variable such that it matches the organization of the file.

To see a complete list of the image sizes that PV-WAVE checks for as it reads image data, refer to the function descriptions for DC_READ_8_BIT and DC_READ_24_BIT; you can find these descriptions in the PV-WAVE Reference.

NOTE If you don't want PV-WAVE guessing the dimensions of the variable, you need to explicitly dimension it.

For 8-bit image data, dimension the variable as w -by- h , where w and h are the width and height of the image in pixels. For 24-bit image data, the image variable should be dimensioned in the following manner:

- **Pixel Interleaved** — Dimension the import variable as 3-by- w -by- h , where w and h are the width and height of the image in pixels.
- **Image Interleaved** — Dimension the import variable as w -by- h -by-3, where w and h are the width and height of the image in pixels.

For a comparison of pixel interleaving and image interleaving, refer to [Image Interleaving on page 178](#).

TIP One popular way of importing binary image data is with the ASSOC command. The advantages of this method are described further in [Advantages of Associated File Variables on page 195](#).

Image Data Output

Image data can be exported using either the WRITEU or the ASSOC commands. However, one of the easiest ways to output image data is to use either the DC_WRITE_8_BIT or DC_WRITE_24_BIT functions. For example, if `fft_flow` is a 600-by-800 byte array containing image data, the function call:

```
status = DC_WRITE_8_BIT('fft_flow1.img', fft_flow)
```

creates the file `fft_flow1.img` and uses it to store the image data contained in the variable `fft_flow`.

The dimensionality of the output image variable should be the same as discussed in the previous section for image data input.

TIP One popular way of exporting binary image data is with the ASSOC command. The advantages of this method are described further in [Advantages of Associated File Variables on page 195](#).

TIFF Image Data

The TIFF (Tag Image File Format) is a standard format for encoding image data. Visual Numerics' TIFF I/O follows the guidelines set forth in a Technical Memorandum, *Tag Image File Format Specification, Revision 5.0 (FINAL)*, published jointly by Aldus™ Corporation and Microsoft® Corporation.

The two functions provided specifically for transferring TIFF images are:

```
DC_READ_TIFF
DC_WRITE_TIFF
```

These functions are easy to use. For example, if the variable `maverick` is a 512-by-512 byte array, the function call:

```
status = DC_WRITE_TIFF('mav.tif', maverick, $
    Class='Bilevel', Compress='Pack')
```

creates the file `mav.tif` and uses it to store the image data contained in the variable `maverick`. The created TIFF file is compressed and conforms to the TIFF Bilevel classification.

For additional details about the `DC_READ_TIFF` and `DC_WRITE_TIFF` functions, see their descriptions in the PV-WAVE Reference.

Compressed TIFF Files

TIFF files can be compressed if you are interested in saving disk space. Compressed TIFF files will take slightly longer to open than uncompressed TIFF files, but are a smart choice if you are willing to trade off a slightly slower access time for reduced file size.

Only TIFF class Bilevel (Class 'B') images can be compressed.

TIFF Conformance Levels

When using `DC_READ_TIFF` and `DC_WRITE_TIFF`, you are able to select the class (level of TIFF conformance) that you wish to follow. The four conformance levels are:

- **Bilevel** — All pixels are either black or white; no shades of gray are supported.
- **Grayscale** — Each pixel is described by eight bits (a byte). With eight bits, 2^8 shades of gray can be represented.
- **Palette Color** — Each pixel is described by eight bits (a byte), so 2^8 discrete colors can be represented. During output, you must supply a colortable that can be stored with the image; you do this using the *Palette* keyword.
- **RGB Full Color** — Each pixel is described by 24 bits (1 byte red, 1 byte green, and 1 byte blue). With 24 bits, 2^{24} full RGB colors can be represented.

If *Palette Color* is selected, you must supply (using the *Palette* keyword) a 3-by-256 array of integers that describes the colortable to be used by the TIFF image.

If *RGB Full Color* is selected, the export variable must be a *w*-by-*h*-by-3 byte image interleaved array. (The letters *w* and *h* denote the width and height of the image, respectively.) Pixel interleaved 24-bit data cannot be exported to a TIFF file. The details of pixel interleaving and image interleaving are described in the next section.

Image Interleaving

Interleaving is the method used to organize the bytes of red, green, and blue image data in a 24-bit image. In other words, each of the basic colors requires 1 byte (8 bits) of storage for each pixel on the screen; the question is whether to store the color data as RGB triplets, or to group all the red bytes together, all the green bytes together, and all the blue bytes together. The two options are shown below:

Pixel Interleaving	Image Interleaving
RGBRGBRGBRGB	RRRRRRRRRRRR
RGBRGBRGBRGB	GGGGGGGGGGGG
RGBRGBRGBRGB	BBBBBBBBBBBB

For more information about how the image variable should be dimensioned to match the various interleaving methods, refer to [Image Data Input on page 175](#).

READU and WRITEU

READU and WRITEU provide basic binary (unformatted) input and output capabilities. WRITEU writes the contents of its variable list directly to the file, and READU reads exactly the number of bytes required by the size of its parameters. Both procedures transfer binary data directly, with no interpretation or formatting.

The general form for using either READU or WRITEU is:

```
READU, unit, var1, ..., varn
WRITEU, unit, var1, ..., varn
```

where *var*_{*i*} represents one or more variables (or expressions in the case of output).

Transferring Data with READU and WRITEU

Example 1 — C Program Writes, PV-WAVE Reads

The following C program produces a file containing employee records. Each record stores the first name of the employee, the number of years they have been employed, and their salary history for the last 12 months.

C Program Write

```
#include <stdio.h>
main()
{
    static struct rec {
        char name [16]; /* Employee's name */
        int years; /* Years with company*/
        float salary[12]; /* Salary for last */
                        /* 12 months */
    } employees[] = {
        {"Bullwinkle", 10,
         {1000.0, 9000.97, 1100.0, 0.0, 0.0, 2000.0, 5000.0, 3000.0,
          1000.12, 3500.0, 6000.0, 900.0} },
        {"Boris", 11,
         {400.0, 500.0, 1300.10, 350.0, 745.0, 3000.0, 200.0, 100.0,
          100.0, 50.0, 60.0, 0.25} },
        {"Natasha", 10,
         {950.0, 1050.0, 1350.0, 410.0, 797.0, 200.36, 2600.0, 2000.0,
          1500.0, 2000.0, 1000.0, 400.0} },
        {"Rocky", 11,
         {1000.0, 9000.0, 1100.0, 0.0, 0.0, 2000.37, 5000.0, 3000.0,
```

```

    1000.01, 3500.0, 6000.0, 900.12} }
};
FILE *outfile;
outfile = fopen("bullwinkle.dat", "w");
(void) fwrite(employees, sizeof(employees), 1, outfile);
(void) fclose(outfile);
}

```

Running this program creates the file `bullwinkle.dat` containing the employee records.

PV-WAVE Read

The following PV-WAVE statements can be used to read the data in `bullwinkle.dat`:

```

str16 = STRING(REPLICATE(32b,16))
    ; Create a string with 16 characters so that the proper number
    ; of characters will be input from the file. REPLICATE is used to
    ; create a byte array of 16 elements, each containing the ASCII
    ; code for a space (32). STRING turns this byte array into a string containing 16 blanks.
A = REPLICATE({employees, name:str16, $
    years:0L, salary:fltarr(12)}, 4)
    ; Create a structure of four employee records to receive the
    ; input data.
OPENR, 1, 'bullwinkle.dat'
    ; Open the file for input.
READU, 1, A
    ; Read the data.
CLOSE, 1
    ; Close the file.

```

For other examples of how to read `bullwinkle.dat` with PV-WAVE, refer to [Reading, Sorting, and Printing Tables of Formatted Data on page 163](#).

Example 2 — PV-WAVE Writes, C Program Reads

PV-WAVE Write

The following PV-WAVE program creates a binary data file containing a 5-by-5 array of floating-point values:

```

OPENW, 1, 'float.dat'
    ; Open a file for output.

```

```
WRITEU, 1, FINDGEN(5, 5)
    ; Write a 5-by-5 array with each element set equal to its one-dimensional index.
CLOSE, 1
    ; Close the file.
```

C Program Read

The file `float.dat` can be read and printed by the following C program:

```
#include <stdio.h>
main()
{
    float data[5][5];
    FILE *infile;
    int i, j;
    infile = fopen("float.dat", "r");
    (void) fread(data, sizeof(data), 1, infile);
    (void) fclose(infile);
    for (i = 0; i < 5; i++)
    {
        for (j = 0; j < 5; j++)        printf("%8.1f",
            data[i][j]);
        printf("\n");
    }
}
```

Running this program results in the following output:

```
0.0    1.0    2.0    3.0    4.0
5.0    6.0    7.0    8.0    9.0
10.0   11.0   12.0   13.0   14.0
15.0   16.0   17.0   18.0   19.0
20.0   21.0   22.0   23.0   24.0
```

Binary Transfer of String Variables

The only basic data type that does not have a fixed size is the string data type. A string variable has a dynamic length that is dependent only on the length of the string currently assigned to it. Thus, although it is always possible to know the length of the other types, string variables are a special case. PV-WAVE uses the following rules to determine the number of characters to transfer:

- **Input** — Input enough bytes to fill the currently defined length of the string variable.

- **Output** — Output the number of bytes contained in the string. This number is the same number that would be returned by the STRLEN function. In other words, the output string contains only the characters in the string and does not include a terminating null byte.

These rules imply that when reading into a string variable from a file, you must usually know the length of the original string so as to be able to initialize the destination string to the correct length. The following example demonstrates the problem and shows how to use the STRLEN function to programmatically initialize the string length.

Examples of Binary String Data Transfer

For example, the following statements:

```
OPENW, 1, 'temp.txt'
    ; Open a file.
WRITEU, 1, 'Hello World'
    ; Write an 11-character string.
POINT_LUN, 1, 0
    ; Rewind the file.
A = '          '
    ; Prepare a 9-character string.
READU, 1, A
    ; Read the string in again.
PRINT, A
    ; Show what was input.
CLOSE, 1
```

produces the following output because the receiving variable A was not long enough:

```
Hello Wor
```

The only solution to this problem is to know the length of the string being input. One way to do this is to store the length of the string(s) in the file at the time the file is created. The following statements demonstrate a technique for doing this:

```
hello = 'Hello World'
    ; Define a string variable that contains the desired string.
len = 0
len = STRLEN(hello)
    ; Initialize an integer variable, and then use it to store the length of
    ; the string variable.
```

```

OPENW, 1, 'temp.txt'
    ; Open a file.
WRITEU, 1, len
    ; Write the string length to the file.
WRITEU, 1, hello
    ; Now write the string to the file.

```

Now that the string length (an integer), followed by the string, have been stored in the file, prepare to read the string back into PV-WAVE:

```

len_input = 0
READU, 1, len_input
    ; Initialize an integer variable, and then use it to read the string length.
A = STRING(REPLICATE(32b, len_input))
    ; Create a string of the desired length, initialized with blanks. The
    ; result of the call to REPLICATE is a byte array with the necessary
    ; number of elements, each element initialized to 32, which is the
    ; ASCII code for a blank. When this byte array is passed to STRING,
    ; it is converted to a scalar string containing this number of blanks.
READU, 1, A
    ; Read the string.
PRINT, A
    ; Show what was input.
CLOSE, 1

```

produces the following output:

```
Hello World
```

This example takes advantage of the special way in which the BYTE and STRING functions convert between byte arrays and strings. See the descriptions of the BYTE and STRING functions for additional details. These descriptions are alphabetically arranged in the PV-WAVE Reference.

Reading UNIX FORTRAN-Generated Binary Data

Although the UNIX operating system considers all files to be an uninterpreted stream of bytes, FORTRAN considers all I/O to be done in terms of logical records. In order to reconcile the FORTRAN need for logical records with the UNIX operating system, UNIX FORTRAN programs add a longword count before and after each logical record of data. These longwords contain an integer count giving the number of bytes in that record.

The use of the *F77_Unformatted* keyword with the OPENR statement informs PV-WAVE that the file contains binary data produced by a UNIX FORTRAN program. When a file is opened with this keyword, PV-WAVE interprets the longword counts properly, and is able to read and write files that are compatible with FORTRAN.

Example — UNIX FORTRAN Program Writes, PV-WAVE Reads

The following UNIX FORTRAN program produces a file containing a 5-by-5 array of floating-point values, with each element set to its one-dimensional subscript. It is thus a FORTRAN implementation of the FINDGEN function for the special case of a 5-by-5 array.

FORTRAN Write

```
INTEGER I, J
REAL DATA(5, 5)
OPEN(1, STATUS = "new", FILE = "mydata",
     FORM = "unformatted")
DO 100 J = 1, 5
  DO 100 I = 1, 5
    DATA(I,J) = ((J-1) * 5) + (I-1)
100 CONTINUE
WRITE(1) DATA
END
```

Running this program creates a file `mydata` that contains the array of numbers.

PV-WAVE Read (Method 1)

The following PV-WAVE statements can be used to read this file and print its contents:

```
OPENR, 1, 'mydata', /F77_Unformatted
; Open the file. The F77_Unformatted keyword lets PV-WAVE know
; that the file contains binary data produced by a UNIX FORTRAN
; program.

A = FLTARR(5, 5, /Nozero)
; Create an array to hold the data. The command executes faster
; because the Nozero keyword disables the automatic zeroing of
; each value that normally occurs.

READU, 1, A
; Read the data in a single input operation.

PRINT, A
; Print the result.
```

```
CLOSE, 1
; Close the file.
```

Executing these PV-WAVE statements results in the following output:

```
0.0000    1.0000    2.0000    3.0000    4.0000
5.0000    6.0000    7.0000    8.0000    9.0000
10.0000   11.0000   12.0000   13.0000   14.0000
15.0000   16.0000   17.0000   18.0000   19.0000
20.0000   21.0000   22.0000   23.0000   24.0000
```

PV-WAVE Read (Method 2)

Because binary data produced by UNIX FORTRAN programs are interspersed with these “extra” longword record markers, it is important that the PV-WAVE program read the data in the same way that the FORTRAN program wrote it. For example, consider the following attempt to read the above data file one row at a time:

```
OPENR, 1, 'mydata', /F77_Unformatted
; Open the file. The F77_Unformatted keyword lets PV-WAVE know
; that the file contains binary data produced by a UNIX FORTRAN
; program.

A = FLTARR(5, /Nozero)
; Create an array to hold one row of the array.

FOR I = 0, 4 DO BEGIN
; One row at a time.
  READU, 1, A
; Read a row of data.
  PRINT, A
; Print the row.
ENDFOR

CLOSE, 1
; Close the file.
```

Executing these PV-WAVE statements produces the output:

```
0.00000 1.00000 2.00000 3.00000 4.00000
%End of file encountered. Unit: 1.
      File: mydata
%Execution halted at $MAIN$ (READU).
```

This program read the single logical record written by the FORTRAN program as if it were written in five separate records. Consequently, it reached the end of the file after reading the first five values of the first record.

For information about using similar commands to read a segmented record file created on an OpenVMS system, refer to the example in the next section.

Reading OpenVMS FORTRAN-Generated Binary Data

By default, OpenVMS FORTRAN programs create data files using *segmented records*, a scheme used by FORTRAN to write data records with lengths that exceed the actual record lengths allowed by OpenVMS.

In segmented record files, a single segmented record is written as one or more actual OpenVMS records. Each of the actual records has a two-byte control field prepended that allows FORTRAN to reconstruct the original record.

Example — OpenVMS FORTRAN Program Writes, PV-WAVE Reads

OpenVMS FORTRAN Write

The following OpenVMS FORTRAN program produces a file containing a 5-by-5 array of floating-point values, with each element set to its one-dimensional subscript. It is thus a FORTRAN implementation of the PV-WAVE FINDGEN function for the special case of a 5-by-5 array:

```
INTEGER I, J
REAL DATA(5, 5)
OPEN(1, STATUS = "new", FILE = "mydata",
     FORM = "unformatted")
DO 100 J = 1, 5
  DO 100 I = 1, 5
    DATA(I,J) = ((J-1) * 5) + (I-1)
100 CONTINUE
WRITE(1) DATA
END
```

Running this program creates a file `mydata` that contains the array of numbers.

PV-WAVE Read (Method 1)

PV-WAVE is able to read and write segmented record files if the OPEN statement used to access the file includes the *Segmented* keyword. The following PV-WAVE statements can be used to read this file and print its contents to the screen:

```

OPENR, 1, 'data.dat', /Segmented
    ; Open the file. The Segmented keyword lets PV-WAVE know that
    ; the file contains OpenVMS FORTRAN segmented records.

A = FLTARR(5, 5, /Nozero)
    ; Create an array to hold the data. The command executes faster
    ; because the Nozero keyword disables the automatic zeroing of
    ; each value that normally occurs.

READU, 1, A
    ; Read the data in a single input operation.

PRINT, A
    ; Print the result.

CLOSE, 1
    ; Close the file.

```

Executing these PV-WAVE statements results in the following output:

```

 0.0000  1.0000  2.0000  3.0000  4.0000
 5.0000  6.0000  7.0000  8.0000  9.0000
10.0000 11.0000 12.0000 13.0000 14.0000
15.0000 16.0000 17.0000 18.0000 19.0000
20.0000 21.0000 22.0000 23.0000 24.0000

```

PV-WAVE Read (Method 2)

As with all record-oriented I/O, it is important that the PV-WAVE program read the data in the same way that the OpenVMS FORTRAN program wrote it. For example, consider the following attempt to read the above data file one row at a time:

```

OPENR, 1, 'mydata', /Segmented
    ; Open the file. The Segmented keyword lets PV-WAVE know that
    ; the file contains OpenVMS FORTRAN segmented records.

A = FLTARR(5, /Nozero)
    ; Create an array to hold one row of the array.

FOR I = 0, 4 DO BEGIN
    ; One row at a time.
    READU, 1, A
        ; Read a row of data.
    PRINT, A
        ; Print the row.
ENDFOR

CLOSE, 1
    ; Close the file.

```

Executing these PV-WAVE statements produces the output:

```
0.00000 1.00000 2.00000 3.00000 4.00000
%End of file encountered. Unit: 1.
      File: mydata
%Execution halted at $MAIN$ (READU).
```

This program read the single logical record written by the FORTRAN program as if it were written in five separate records. Consequently, it reached the end of the file after reading the first five values of the first record.

Reading and Writing Long Integers Under Digital UNIX

Internal C long integers are 8 bytes on Digital UNIX, versus 4 bytes on the other supported UNIX platforms. To accommodate this difference, the long variable type in PV-WAVE on Digital UNIX has increased precision, allowing you to calculate expressions up to 9223372036854775807 without overflow.

Please note that binary dumps of long values or structures with long values will not be retrievable directly from other supported UNIX machines and vice versa.

For example, if you use WRITEU to write out a structure, a long, or a series of longs, to a file from Digital UNIX, and then try to use READU to read those values with PV-WAVE on a Sun platform, you will not retrieve those values, since the Sun will read in 4 bytes instead of the full 8 bytes.

If you need to be able to read that type of file on Digital UNIX and other platforms, you can find an example of how to manipulate the bytes in a structure in the following file, near line 80.

```
(UNIX)      $WAVE_DIR/lib/std/polycontour.pro
```

You may experience this problem especially for files containing headers, like Sun raster files, where the header describes the content of the file.

External Data Representation (XDR) Files

Normally, binary data is not portable between different machine architectures because of differences in the way different machines represent binary data. It is, however, possible to produce binary files that are portable, by specifying the *Xdr* keyword with the OPEN procedures. XDR represents a compromise between the extremes of ASCII and binary I/O.

XDR (*eXternal Data Representation*, developed by Sun Microsystems, Inc.) is a scheme under which all binary data is written using a standard “canonical” repre-

sensation. PV-WAVE understands this standard representation and has the ability to convert between it and the internal representation of the machine upon which it runs.

XDR converts between the internal and standard external binary representations for data, instead of simply using the machine's internal representation. Thus, it is much more portable than pure binary data, although it is still limited to those machines that support XDR.

NOTE XDR is not as efficient as pure binary I/O because it does involve the overhead of converting between the external and internal binary representations. Nevertheless, it is still much more efficient than ASCII I/O because conversion to and from ASCII characters is much more involved than converting between binary representations.

Opening XDR Files

Since XDR adds extra “bookkeeping” information to data stored in the file, and because the binary representation used may not agree with that of the machine being used, it does not make sense to access an XDR file without using the *Xdr* keyword.

To use the XDR format, you must specify the *Xdr* keyword when opening the file. For example:

```
OPENW, /Xdr, 1, 'data.dat'
```

NOTE OPENW and OPENU normally open files for both input and output. However, XDR files can only be open in one direction at a time. Thus, using these procedures with the *Xdr* keyword results in a file open for output only, and the only I/O data transfer routines that can be used is WRITEU. OPENR works in the usual way.

Transferring Data To and From XDR Files

The primary differences in the way PV-WAVE I/O procedures work with XDR files, as opposed to other data files, are listed below:

- The only I/O data transfer routines that can be used with a file opened for XDR are READU and WRITEU.
- The length of strings is saved and restored along with the string. This means that you do not have to initialize a string of the correct length before reading a

string from the XDR file. (This is necessary with normal binary I/O, and is described in *Binary Transfer of String Variables* on page 181.)

- For the sake of efficiency, byte data is transferred as a single unit. Therefore, byte variables must be initialized to a length that matches the data to be input. Otherwise, an error message is displayed. See the following example for more details.

Example — Reading Byte Data from an XDR File

For example, given the statements:

```
OPENW, /Xdr, 1, 'data.dat'  
    ; Open a file for XDR output.
```

```
WRITEU, 1, BINDGEN(10)  
    ; Write a 10-element byte array.
```

```
CLOSE, 1  
    ; Close the file ...
```

```
OPENR, /Xdr, 1, 'data.dat'  
    ; ... and re-open it for input.
```

the following statements:

```
b = 0B  
    ; Define b as a byte scalar.
```

```
READU, 1, b  
    ; Try to read the first byte only.
```

```
CLOSE, 1  
    ; Close the file.
```

will result in the error:

```
%End of file encountered. Unit: 1.  
    File: data.dat  
%Execution halted at $MAIN$ (READU).
```

Instead, it is necessary to read the entire byte array back in one operation using statements such as:

```
b = BYTARR(10)  
    ; Define b as a byte array.
```

```
READU, 1, b  
    ; Read the whole array back at once.
```

```
CLOSE, 1  
    ; Close the file.
```

NOTE This restriction (in other words, the necessity of transferring byte data as a single unit) does not apply to the other data types.

Example — Reading C-generated XDR Data with PV-WAVE

C Program Write

The following C program produces a file containing different types of data using XDR. The usual error checking is omitted for the sake of brevity.

```
#include <stdio.h>
#include <rpc/rpc.h>

[xdr_wave_complex() and xdr_wave_string() included here]
/* For more information about xdr_wave_complex( ) and
   xdr_wave_string( ), refer to a later section that follows this
   example.*/

main()
{
    static struct {          /* output data */
unsigned char c;
    short s;
    long l;
    float f;
    double d;
    struct complex { float r, i } cmp;
    char *str;
} data = {1, 2, 3, 4, 5.0, { 6.0, 7.0}, "Hello" };
u_int c_len = sizeof (unsigned char);
/* Length of a character */
char *c_data = (char *) &data.c;
/* Address of byte field */
FILE *outfile;
/* stdio stream pointer */
XDR xdrs;
/* XDR handle */
/* Open stdio stream and XDR handle */
outfile = fopen("data.dat", "w");
xdrstdio_create(&xdrs, outfile, XDR_ENCODE);
```

```

/* Output the data */
(void) xdr_bytes(&xdrs, &c_data, &c_len, c_len);
(void) xdr_short(&xdrs, (char *) &data.s);
(void) xdr_long(&xdrs, (char *) &data.l);
(void) xdr_float(&xdrs, (char *) &data.f);
(void) xdr_double(&xdrs, (char *) &data.d);
(void) xdr_wave_complex(&xdrs, (char *) &data.cmp);
(void) xdr_wave_string(&xdrs, &data.str);
/* Close XDR handle and stdio stream */
xdr_destroy(&xdrs);
(void) fclose(outfile);
}

```

Running this program creates the file `data.dat` containing the XDR data.

PV-WAVE Read

The following PV-WAVE statements can be used to read this file and print its contents to the screen:

```

data = {s, c:0B, s:0, l:0L, f:0.0, d:0.0D, $
        cmp:COMPLEX(0), str:' '}
        ; Create structure containing correct types.

OPENR, /Xdr, 1, 'data.dat'
        ; Open the file for input.

READU, 1, data
        ; Read the data.

CLOSE, 1
        ; Close the file.

PRINT, data
        ; Show the results.

```

Executing these PV-WAVE statements produces the output:

```

{ 1      2      3      4.00000      5.0000000
  (6.00000, 7.00000) Hello}

```

For further details about XDR, consult the XDR documentation for your machine. If you are a Sun workstation user, consult the *Network Programming* manual.

XDR Conventions for Programmers

PV-WAVE uses certain conventions for reading and writing XDR files. If you use XDR only to exchange data in and out of PV-WAVE, you don't need to be concerned about these conventions because PV-WAVE takes care of it for you.

However, if you want to create PV-WAVE compatible XDR files from other languages, you need to know the actual XDR routines used by PV-WAVE for various data types. These routine names are summarized in the following table:

XDR Routines Used by PV-WAVE

Data Type	XDR Routine
BYTE	xdr_bytes()
INT	xdr_short()
LONG	xdr_long()
FLOAT	xdr_float()
DOUBLE	xdr_double()
COMPLEX	xdr_wave_complex() *
STRING	xdr_wave_string() *

The asterisk (*) indicates compound routines.

XDR Routines for Transferring Complex and String Data

The routines used for types complex and string are not primitive XDR routines. Their definitions are shown in the following C code:

```
bool_t xdr_wave_complex(xdrs, p)
    XDR *xdrs;
    struct complex { float r, i } *p;
{
    return(xdr_float(xdrs, (char *) &p->r)&&
           xdr_float(xdrs, (char *) &p->i));
}

bool_t xdr_wave_string(xdrs, p)
    XDR *xdrs;
    char **p;
{
    int input = (xdrs->x_op == XDR_DECODE);
    short length;
    /* If writing, obtain the length */
```

```

if (!input) length = strlen(*p);
/* Transfer the string length */
if (!xdr_short(xdrs, (char *) &length)) return(FALSE);
/* If reading, obtain room for the string */
if (input)
    {
    *p = malloc((unsigned) (length + 1));
    *p[length] = '\0';/* Null termination */
    }
/* If nonzero, return string length */
return (length ? xdr_string(xdrs, p, length) : TRUE);
}

```

Associated Variable Input and Output

Binary data stored in files often consists of a repetitive series of arrays or structures. A common example is a series of images or a series of arrays. PV-WAVE associated file variables offer a convenient and efficient way to access data that comprises a sequence of identical arrays or structures.

An associated variable is a variable that maps the definition of an array or structure variable onto the contents of a file. The file is treated as an array of these repeating units of data. The first array or structure in the file has an index of 0, the second has index 1, and so on. The general form for using ASSOC is:

ASSOC(*unit*, *array_definition* [, *offset*])

For examples showing how to use the *offset* parameter, refer to a later section, [Using the Offset Parameter on page 198](#).

Associated variables do not use memory like a normal variable. Instead, when an associated variable is subscripted with the index of the desired array or structure within the file, PV-WAVE performs the I/O operation required to access that entire block of data.

OpenVMS USERS OpenVMS fixed-length record files must be accessed by ASSOC either on record boundaries or an integer multiple of the number of data elements on a record boundary.

Advantages of Associated File Variables

Associated file variables offer the following advantages over READU and WRITEU for binary I/O. For these reasons, associated variables are the most efficient form of I/O.

- I/O occurs whenever an associated file variable is subscripted. Thus, it is possible to perform I/O within an expression, without a separate I/O statement.
- The size of the data set is limited primarily by the maximum size of the file containing the data, instead of the maximum memory available. Data sets too large for memory can be easily accommodated.
- You do not have to declare the maximum number of arrays or structures contained in the file.
- Associated variables simplify access to the data. Direct access to any element in the file is rapid and simple — there is no need to calculate offsets into the file and/or position the file pointer prior to performing the I/O operation.

Working with Associated File Variables

Assume that a file named `today.dat` exists, and that this file contains a series of 10-by-20 arrays of floating-point data. The following two statements open the file and create an associated file variable mapped to the file:

```
OPENU, 1, 'today.dat'  
    ; Open the file.  
  
A = ASSOC(1, FLTARR(10, 20, /Nozero))  
    ; Define an associated file variable. Using the Nozero keyword with  
    ; FLTARR increases efficiency since ASSOC ignores the value of the  
    ; resultant array, anyway.
```

NOTE The order of these two statements is not important — it would be equally valid to call ASSOC first and then open the file. This is because the association is between the variable and the logical file unit, not the file itself.

You may opt to close the file, open a new file using the same LUN, and then use the associated variable without first executing a new ASSOC. Naturally, an error occurs if the file is not open when the file variable is subscripted in an expression, or if the file is open for the wrong type of access (for example, trying to assign to an associated file variable with a file opened with OPENR for read-only access).

As a result of executing the two statements above, the variable A is now an associated file variable. Executing the statement:

INFO, A

produces the following response:

```
A   FLOAT = File<today.dat> Array(10, 20)
```

The associated variable `A` maps the definition of a 10-by-20 floating-point array onto the contents of the file `today.dat`. Thus, the response from the INFO procedure shows it to be a two-dimensional floating-point array.

NOTE Only the form of the array is used by ASSOC. The value of the expression is ignored.

The ASSOC command doesn't require that you use a particular combination of dimensions to index into a file, although you may have reasons to prefer one combination of dimensions over another. For example, assume a number of 128-by-128 byte images are contained in a file. The command:

```
row = ASSOC(1, BYTARR(128))
```

maps the file into rows of 128 bytes each. Thus, `row(3)` is the fourth row of the first image, and `row(128)` is the first row of the second image. On the other hand, the command:

```
image = ASSOC(1, BYTARR(128,128))
```

maps the file into entire images. Now, `image(4)` is all 16384 values of the fifth image.

How Data is Transferred into Associated Variables

Once a variable has been associated with a file, data is read from the file whenever the associated variable appears in an expression with a subscript. The position of the array or structure read from the file is given by the value of the subscript. The following statements give some examples of using file variables:

```
Z = A(0)
```

```
    ; Copy the contents of the first array into the normal variable Z. Z is  
    ; now a 10-by-20 floating-point array.
```

```
FOR I = 1,9 DO Z = Z + A(I)
```

```
    ; Compute the sum of the first 10 arrays (Z was initialized in the previous statement  
    ; to the value of the first array. This statement adds the following nine to it.).
```

```
PLOT, A(3)
```

```
    ; Read the fourth array and plot it.
```

```
PLOT, A(5) - A(4)
```

```
    ; Subtract array 4 from array 5, and plot the result. The result of the
```

; subtraction is not saved after the plot is displayed.

An associated file variable only performs I/O to the file when it is subscripted. Thus, the following two statements do not cause I/O to happen:

B = A

; This assignment does not transfer data from the file to variable B
; because A is not subscripted. Instead, B becomes an associated file
; variable with the same dimensions, and to the same LUN, as A.

B = 23

; This assignment does not result in the value 23 being transferred to
; the file because variable B (which became an associated file var
; able in the previous statement) is not subscripted. Instead, B
; becomes a scalar integer variable containing the value 23. It is no
; longer an associated file variable.

Subscripting Associated File Variables During Input

When the associated file variable is defined to be an array, it is possible to subscript into the array being accessed during input operations. For example, for the variable A defined above:

Z = A(0,0,1)

; Assigns the value of the first floating-point element of the second
; array within the file to the variable Z. The rightmost subscript is
; taken as the index into the file causing PV-WAVE to read the entire
; second array into memory. This resulting array expression is then
; further subscripted by the remaining subscripts.

NOTE Although this ability can be convenient, it can also be very slow because every access to an individual array element causes the entire array to be read from disk. Unless only one element of the array is desired, it is much faster to assign the contents of the array to another variable by subscripting the file variable with a single subscript, and then access the individual array elements from the variable.

Efficiency in Accessing Arrays

To increase the efficiency of reading arrays, make their length an integer multiple of the physical block size of the disk holding the file. Common values are 512, 1024, and 2048 bytes. For example, on a disk with 512-byte blocks, one benchmark program required approximately one-eighth of the time required to read a 512-by-512 byte image that started and ended on a block boundary, as compared to a similar program that read an image that was not stored on even block boundaries.

Using the Offset Parameter

The *offset* parameter to ASSOC specifies the position in the file at which the first array starts. It is useful when a file contains a header followed by data records.

Specifying Offsets Under UNIX and Windows

The offset is given in bytes. For example, if a file uses the first 1024 bytes of the file to contain header information, followed by 512-by-512 byte images, the statement:

```
image = ASSOC(1, BYTARR(512, 512), 1024)
```

skips the header by providing a 1024 byte offset before any image data is read.

Specifying Offsets Under OpenVMS

Under OpenVMS, stream files and RMS block mode files have their offset given in bytes, and record-oriented files have it specified in records. Thus, the example above would have worked for OpenVMS if the file was a stream or block mode file. Assume however, that the file has 512-byte fixed-length records. In this case, skipping the first 1024 bytes is equivalent to skipping the first 2 records:

```
image = ASSOC(1, BYTARR(512, 512), 2)
```

For more information about OpenVMS files, refer to [OpenVMS-Specific Information on page 204](#); that section contains an overview of how OpenVMS handles files.

Writing Associated Variable Data

When a subscripted associated variable appears on the left side of an assignment statement, the expression on the right side is written into the file at the given array position. For example:

```
A(5) = FLTARR(10,20)
      ; Zeroes sixth record. By default, every value in a newly created
      ; floating-point array is set equal to zero, unless the Nozero keyword is supplied.
```

```
A(5) = ARR
      ; Writes ARR into the sixth record after any necessary type
      ; conversions.
```

```
A(J) = (A(J) + A(J+1)) / 2
      ; Averages records J and J+1 and writes the result into record J.
```

NOTE When writing data, only a single subscript (specifying the index of the affected array or structure in the file) is allowed. Thus, it is not possible to index

individual elements of associated arrays during output, although it is allowed during input. To update individual elements of an array within a file, assign the contents of that array to a normal array variable, modify the copy, and write the array back by assigning it to the subscripted associated variable.

Binary Data from UNIX FORTRAN Programs

Binary data files generated by FORTRAN programs under UNIX contain an extra longword before and after each logical record in the file. ASSOC does not interpret these extra bytes, but considers them to be part of the data. Therefore, do not use ASSOC to read such files; use READU and WRITEU instead. You can find an example of using PV-WAVE to read data generated by FORTRAN programs under UNIX in [Reading UNIX FORTRAN-Generated Binary Data on page 183](#).

Miscellaneous File Management Tasks

This section describes a variety of utility commands that have been provided to simplify your interaction with data files. It also describes the FSTAT command, which is a valuable source of information about open files.

Locating Files

The FINDFILE function returns an array of strings containing the names of all files that match its parameter list. The parameter list may contain any wildcard characters understood by your system. For example, to determine the number of procedure files that exist in the current directory:

```
PRINT, '# PV-WAVE.pro files:', $  
      N_ELEMENTS(FINDFILE('* .pro'))
```

Flushing File Units

To increase efficiency, PV-WAVE buffers its I/O in memory. This means that when data is output, there is a brief interval of time during which data is in memory, but has not actually been placed into the file. Normally, this behavior is transparent to the PV-WAVE user (except for the improved performance).

The FLUSH routine exists for those rare occasions where a program needs to be certain that the data has actually been written to the file immediately. For example, to flush file unit 1:

```
FLUSH, 1
```

Positioning File Pointers

Each open file unit has a file pointer associated with it. This file pointer indicates the position in the file at which the next I/O operation will take place.

The POINT_LUN procedure allows the file pointer to be positioned arbitrarily. The file position is specified as the number of bytes from the start of the file. The first position in the file is position 0 (zero).

The following statement rewinds file unit 1 to its beginning:

```
POINT_LUN, 1, 0
```

while the following sequence of statements will position it at the end of the file:

```
tmp = FSTAT(1)
POINT_LUN, 1, tmp.size
```

UNIX USERS Moving the file pointer to a position beyond the current end-of-file causes a UNIX file to grow by that amount. (This is the standard UNIX practice.)

Testing for End-of-File

The EOF function is used to test a file unit to see if it is currently positioned at the end of the file. EOF returns true (1) if the end-of-file condition is true, and false (0) otherwise. For example, to read the contents of a file and print it on the screen:

```
OPENR, 1, 'demo.doc'
    ; Open file demo.doc for reading.
line = ''
    ; Create a variable of type string.
WHILE (not EOF(1)) DO BEGIN READF, 1, line & $
    PRINT, line & END
    ; Read and print each line, until the end of the file is encountered.
CLOSE, 1
    ; Done with the file.
```

Getting Information About Files

Using the INFO Procedure

Information about currently open file units is available by using the Files keyword with the INFO procedure. If no parameters are provided, information about all cur-

rently open user file units (units 1-128) is given. For example, to get information about the three special units (-2, -1, and 0), the command:

```
INFO, /Files, -2, -1, 0
```

causes the following to be displayed on the screen if you are running PV-WAVE in a UNIX environment:

Unit	Attributes	Name
-2	Write, Truncate, Tty, Reserved	<stderr>
-1	Write, Truncate, Tty, Reserved	<stdout>
0	Read, Tty, Reserved	<stdin>

or causes the following to be displayed on the screen if you are running under Windows:

Unit	Attributes	Name
-2	Write, Truncate, Tty, Reserved	<stderr>
-1	Write, Truncate, Tty, Reserved	<stdout>
0	Read, Tty, Reserved	<stdin>

For more information about the INFO command, refer to [Chapter 12, Getting Session Information](#).

Use the Information from FSTAT

FSTAT is a structure that contains details about all currently allocated LUNs. You can use the FSTAT function to get more detailed information, including information that can be used from within a PV-WAVE program. It returns an expression of type structure with a name of FSTAT containing information about the file. For example, to get detailed information about the standard input, the command:

```
INFO, /Structures, FSTAT(0)
```

causes the following to be displayed on the screen:

```
** Structure FSTAT, 10 tags, 32 length:
```

UNIT	LONG	0
NAME	STRING	'<stdin>'
OPEN	BYTE	1
ISATTY	BYTE	1
READ	BYTE	1
WRITE	BYTE	0
TRANSFER_COUNT	LONG	0
CUR_PTR	LONG	35862
SIZE	LONG	0

Since PV-WAVE allows keywords to be abbreviated to the shortest non-ambiguous number of characters,

```
INFO, /St, FSTAT(0)
```

will also work (and save some typing). The fields of the FSTAT structure are defined as part of its description in the PV-WAVE Reference.

Sample Usage — FSTAT Function

The following function can be used to read single-precision floating point data from a file into a vector when the number of elements in the file is not known. This function uses FSTAT to get the size of the file in bytes and then divides by 4 (the size of a single-precision floating-point value) to determine the number of values:

```
FUNCTION read_data, file
    ; Read_data reads all the floating-point values from file and returns
    ; the result as a floating-point vector.

OPENR, /Get_Lun, unit, file
    ; Get a unique file unit and open the data file.

status = FSTAT(unit)
    ; Retrieve the file status.

data = FLTARR(status.size / 4.0)
    ; Make an array to hold the input data. The size tag of status gives
    ; the number of bytes in the file and single-precision floating-point
    ; values are four bytes each.

READU, unit, data
    ; Read the data.

FREE_LUN, unit
    ; Deallocate the file unit and close the file.

RETURN, data
    ; Return the data.

END
```

Assuming that a file named `herc.dat` exists and contains 10 floating-point values, the following statements:

```
a = read_data('herc.dat')
    ; Read floating-point values from herc.dat.

INFO, a
    ; Show the result.
```

will produce the following output:

```
A          FLOAT          = Array(10)
```

Getting Input from the Keyboard

The `GET_KBRD` function returns the next character available from the standard input (file unit 0) as a single character string. It takes a single parameter named *Wait*. If *Wait* is zero, `GET_KBRD` returns the null string if there are no characters in the terminal typeahead buffer. If *Wait* is nonzero, the function waits for a character to be typed before returning.

Sample Usage — GET_KBRD Function

A procedure that updates the screen and exits when <Return> is typed might appear as:

```
PRO UPDATE, ...
    ; Procedure definition.

WHILE 1 DO BEGIN
    ; Loop forever, updating the screen as needed.
    CASE GET_KBRD(0) OF
        ; Read character, no wait.
        'A' : ..... ; Process letter A.
        'B' : ..... ; Process letter B.
        ... : ..... ; Process other alternatives.
        STRING("15B) : RETURN
        ; Exit if <Return> is detected (ASCII code = 15 octal).
    ELSE:
        ; Ignore all other characters.
    ENDCASE
ENDWHILE
END
```

UNIX-Specific Information

UNIX offers only a single type of file. All files are considered to be an uninterrupted stream of bytes, and there is no such thing as record structure at the operating system level. (By convention, records of text are simply terminated by the linefeed character, which is referred to as “newline”.) It is possible to move the current file pointer to any arbitrary position in the file and to begin reading or writ-

ing data at that point. This simplicity and generality forms a system in which any type of file can be manipulated easily, using a small set of file operations.

Reading FORTRAN-Generated Binary Data

Although the UNIX operating system views all files as an uninterrupted stream of bytes, FORTRAN considers all I/O to be done in terms of logical records. In order to reconcile FORTRAN's need for logical records with UNIX files, UNIX FORTRAN programs add a longword count before and after each logical record of data. These longwords contain an integer count giving the number of bytes in that record.

The use of the *F77_Unformatted* keyword with the OPENR statement informs PV-WAVE that the file contains binary data produced by a UNIX FORTRAN program. When a file is opened with this keyword, PV-WAVE interprets the longword counts properly, and is able to read and write files that are compatible with FORTRAN. To see an example showing the use of the *F77_Unformatted* keyword with the OPENR statement, refer to [Reading UNIX FORTRAN-Generated Binary Data on page 183](#).

OpenVMS-Specific Information

OpenVMS I/O is a relatively complex topic, involving a large number of formats and options. OpenVMS files are record oriented, and it is necessary to take this into account when writing applications, especially those that will run under other operating systems. This section discusses the various characteristics that an OpenVMS user must consider when transferring data in and out of PV-WAVE.

Organization of the File

An OpenVMS file can be organized in the following ways:

- ✓ sequential
- ✓ relative
- ✓ indexed

The organization controls the way in which data is placed in the file, and determines the options for random access. PV-WAVE is able to read data from all three types, and is able to create sequential or indexed files.

In addition, it is possible to bypass the organization and access a file in *block mode*; this is equivalent to interpreting the file as if it were simply a stream of uninterrupted bytes. This is very similar to stream files, although considerably more efficient (because most OpenVMS file processing is bypassed).

CAUTION With some file organizations, OpenVMS intermingles housekeeping information with data. When accessing such a file in block mode, it is easy to corrupt this information and render the file unusable in its usual mode. However, block mode will always work, and thus, avoiding such file corruption becomes your responsibility.

Access Mode

The access mode controls how the data in a file is accessed. OpenVMS supports the following types of access:

- ✓ sequential access
- ✓ random access by key value (indexed files)
- ✓ relative record number (relative files)
- ✓ relative file address (all file organizations)

Random access for sequential files is allowed by file address using the POINT_LUN procedure. PV-WAVE does not support access by relative record number — files are accessed sequentially or via key value.

Record Format

All OpenVMS files are record oriented; for an overview of how PV-WAVE handles record-oriented data files, refer to an earlier section, [Record-Oriented I/O in OpenVMS Binary Files on page 143](#).

OpenVMS supports the following types of record formats:

- ✓ fixed-length records
- ✓ variable-length records
- ✓ variable-length with fixed-length control field (VFC)
- ✓ stream format

Of these, the fixed-length and variable-length record formats are the most useful and are fully supported by PV-WAVE.

It is possible to read the data portion of a VFC file, but not the control field. All access to stream mode files under PV-WAVE is done via the Standard C Library.

TIP OpenVMS stream files are record oriented (and therefore, fail to provide much of the flexibility of UNIX stream files) although the OpenVMS standard C library (upon which PV-WAVE is implemented) does a good job of concealing this limitation. Our experience indicates that I/O using OpenVMS stream mode files is dramatically slower than the other options, and should be avoided when possible. For binary data, using block mode can provide the flexibility you need while maintaining an efficient rate of data transfer.

Record Attributes

When a record is output to the screen or printer, OpenVMS uses its carriage control attributes to determine how to output each line:

- **Explicit carriage control** — Specifies that OpenVMS should do nothing, and you (the user) will provide the appropriate carriage control (if any) in the data.
- **Carriage Return carriage control** — Specifies that each line should be preceded by a line feed and followed by a <Return>.
- **FORTTRAN carriage control** — Indicates that the first byte of each record contains a FORTRAN carriage control character. The possible values of this byte are listed in the following table.

OpenVMS FORTRAN Carriage Control

Byte Value	ASCII Character	Meaning
0	(null)	No carriage control — output data directly.
32	(space)	Single-space. A linefeed precedes the output data, and a <Return> follows.
48	0	Double-space. Two linefeeds precede the output data, and a <Return> follows.
49	1	Page eject. A formfeed precedes the data, and a <Return> follows.
40	+	Overprint. A <Return> follows the data, causing the next output line to overwrite the current one.

OpenVMS FORTRAN Carriage Control (Continued)

Byte Value	ASCII Character	Meaning
36	\$	Prompt. A linefeed precedes the data, but no <Return> follows.
Other		Same as ASCII space character. Single-space carriage control.

NOTE The default for PV-WAVE is Carriage Return carriage control.

File Attributes

There are many file attributes that can be adjusted to suit various requirements. These attributes allow specifying such things as the default name, the initial size of new files, the amount by which files are extended, whether the file is printed or sent to a batch queue when closed, and file sharing between processes.

For more information about OpenVMS file attributes, refer to *Record Management Services (RMS), File System, Volume 6B*.

Creating Indexed Files

Although PV-WAVE can read and write indexed files, it cannot create them. So you must use the OpenVMS CREATE/FDL command to create the file. FDL stands for File Definition Language, and is the standard method for specifying OpenVMS file attributes. The options for creating indexed files are too numerous to cover in this document, but the *OpenVMS File Definition Language Facility Manual* describes FDL in detail.

TIP It is often useful to start with the FDL description for an existing file and then modify it to suit your new application. The command:

```
$ ANALYZE/RMS_FILE/FDL file.dat
```

will produce a file named `file.fdl` containing the FDL description for `file.dat`.

The following is a FDL description for an indexed file named `wages.dat` with two keys. The first key is a 32 character string containing an employee name. The second key is a 4 byte integer containing the current salary for that employee:

```
FILE
```

```

NAME wages.dat
ORGANIZATION indexed

RECORD
  SIZE 36

KEY 0
  NAME "Name"
  SEGO_LENGTH 32
  SEGO_POSITION 0
  TYPE string

KEY 1
  CHANGES yes
  NAME "Salary"
  SEGO_LENGTH 4
  SEGO_POSITION 32
  TYPE bin4

```

Assume that this description resides in a file named `wages.fdl`. The following statement can be used to create `wages.dat`:

```
SPAWN, 'CREATE/FDL=wages.fdl'
```

Once the file exists, it can be opened within PV-WAVE using the KEYED keyword with the OPENR or OPENU procedures.

Accessing Magnetic Tape

Under OpenVMS, PV-WAVE offers procedures to directly access magnetic tapes. Data is transferred between the tape and PV-WAVE arrays without using RMS. Optionally, tapes from IBM mainframe compatible systems may be read or written with odd/even byte reversal.

The routines used for directly accessing magnetic tape are shown in the following table:

Routines for Directly Accessing Magnetic Tape

Procedure	Description
REWIND	Rewind a tape unit.
SKIPF	Skip records or files.
TAPRD	Read from tape.
TAPWRT	Write to tape.
WEOF	Write an end-of-file mark on tape.

To use the magnetic tape procedures you must define a logical name “MT n ” to be equivalent to the actual name of the tape drive you wish to use. This definition must be done *before* you start PV-WAVE. You must also have the tape mounted as a *foreign* volume.

Example 1 — Mounting a Tape Drive

For example, if you wish to access the tape drive MUA0 : as tape unit number 5, issue the following OpenVMS commands before running PV-WAVE:

```
$ MOUNT/FOREIGN MUA0 :  
$ DEFINE MT5 MUA0 :
```

Or, you can combine the two commands:

```
$ MOUNT MUA0 :/FOR MT5
```

This command serves to both mount the tape and to associate the logical name MT5 with it, thus making it unit 5 from within PV-WAVE. The MOUNT command must be issued to OpenVMS before entering PV-WAVE. Then, within PV-WAVE, refer to the tape as unit number 5. The unit number n , should be in the range {0..9}.

NOTE These unit numbers are not the same as the LUNs (logical unit numbers) used by the other I/O routines. The unit numbers used by the magnetic tape routines are completely unrelated, and come from the last letter of the MT n logical name used to refer to it.

Example 2 — Skipping Forward on the Tape

The following statements skip forward 30 records on the tape mounted on the drive with the logical name MT2, and print a message if an end-of-file is encountered.

```
SKIPF, 2, 30, 1  
    ; Skip forward over 30 records on unit 2.  
  
IF !ERR NE 30 THEN PRINT, 'End of file found.'  
    ; Print a message if the requested number of records were not  
    ; skipped.
```

Example 3 — Skipping Backward on the Tape

The following statements skip two records backwards on the tape mounted on the drive with the logical name MT0, and then position the tape immediately after the second record mark encountered in reverse:

```
SKIPF, 0, -2  
    ; Go backwards two records.
```

```
IF !ERR EQ -2 THEN SKIPF, 0, 1
    ; Reposition tape if two records were actually skipped.
```

Example 4 — Reading Blocks of Image Data

The following code segment reads a 512-by-512 byte image from the tape which is assigned the logical name MT5. It is assumed that the data is stored in 2048 byte tape blocks:

```
A = BYTARR(512, 512)

    ; Define image array.
B = BYTARR(512, 4)
    ; Define an array to hold one tape block worth of data.
FOR I=0, 511, 4 DO BEGIN
    ; Loop to read data.
    TAPRD, B, 5
    ; Read next record.
    A(0,I) = B
    ; Insert four rows starting at ith row.
ENDFOR
```

Windows-Specific Information

Exchanging Image Data Using the Clipboard

Under Windows, the Clipboard provides a convenient mechanism for transferring image data to and from PV-WAVE. You can copy data to and from the Clipboard using either 1) command line functions, or 2) functions on the graphics window Control menu.

Any graphics application that accepts Device Independent Bitmap (DIB) or enhanced-format metafile (EMF) graphics can be exchanged with PV-WAVE through the Clipboard. For example, you can copy graphics from a graphics window into a Microsoft Paintbrush window using the copy and paste functions provided by these two applications.

Depending on the options you have selected, the graphics that are copied to and from the Clipboard are either in bitmap (DIB) or metafile (EMF) format. By default, bitmap graphics are displayed in graphics windows; however, providing the */Meta* keyword with either the DEVICE or WINDOW procedure creates a

metafile. The following table highlights some of the differences between DIB and EMF graphics:

Format	Advantages	Disadvantages
Bitmap (DIB)	Fast display of complex graphics images.	Graphics cannot be resized interactively. Greater storage requirements.
Metafile (EMF)	Allows graphics to be resized interactively. Smaller storage requirements	Slower display time for complex graphics.

Command Line Clipboard Functions

Either of these commands can be entered at the WAVE> prompt in the Console window:

- **WCOPY** — Copies the contents of a graphics window to the Clipboard. The following example copies the contents of window number 1 to the Clipboard:

```
status=WCOPY (1)
```

- **WPASTE** — Pastes the contents of the Clipboard into a specified graphics window. The following example pastes the contents of the Clipboard into the graphics window with index number 1.

```
status=WPASTE (1)
```

For detailed information on these functions, see their descriptions in the PV-WAVE Reference.

Clipboard Functions on the Graphics Window Control Menu

Either of these options can be selected from the Control menu of any graphics window:

- **Copy to Clipboard** — Copies the contents of a graphics window to the Clipboard.
- **Paste from Clipboard** — Pastes the contents of the Clipboard into a specified graphics window.

Input and Output of DIB and Metafile Images

Several commands are provided for the transfer of Device Independent Bitmap (DIB) and enhanced-format metafile (EMF) images. These commands fall into two

general categories: 1) commands that transfer data between variables and files, and 2) commands that transfer data between graphics windows and files.

Commands that Transfer Data Between Variables and Files

The following functions allow you to import and export DIB data between variables and files. For more information on these functions, see their descriptions in the PV-WAVE Reference.

- `DC_READ_DIB` — Reads data from a DIB format file into a variable.
- `DC_WRITE_DIB` — Writes image data from a variable to a DIB format file.

Commands that Transfer Data Between Files and Windows

The following functions allow you to import and export DIB and EMF data between data files and graphics windows. For more information on these functions, see their descriptions in the PV-WAVE Reference.

- `WREAD_DIB` — Loads a DIB from a file into a graphics window.
- `WWRITE_DIB` — Saves the contents of a graphics window to a file using the DIB format.
- `WREAD_META` — Loads an EMF image from a file into a graphics window. (EMF graphics cannot be pasted into 16-bit applications.)
- `WWRITE_META` — Saves the contents of a graphics window to a file using the EMF format. (EMF graphics cannot be pasted into 16-bit applications.)

The graphics window Control menu contains two functions that let you interactively import and export DIB and EMF data:

- **Export Graphics** — Writes the contents of the graphics window to a file. Your choice of output format depends on how the window was created. If the window was created with a metafile option (either by specifying the *Meta* or *Redraw* keyword), then the exported file can either be a DIB or an EMF format file. The filename extension, `.bmp` for DIB format and `.emf` for EMF format, determines the type of output file. If the window was created without a metafile option, then the only choice is to export DIB format data.
- **Import Graphics** — Reads the contents of a file into a PV-WAVE graphics window. The file's contents must be in either DIB or EMF format.

By default, the Import Graphics dialog box looks for files with a `.bmp` or `.emf` extension. If the graphics window was created with the */Meta* or */Redraw* option enabled, then `.emf` files are expected. Otherwise, `.bmp` files are expected. For information on the */Meta* and */Redraw* options, see the dis-

cussions of the DEVICE and WINDOW procedures in the PV-WAVE Reference.

Transferring Data from PV-WAVE to Microsoft® Excel

This section describes a method for exporting data from PV-WAVE into a Microsoft Excel spreadsheet. You can apply this method with any application that allows you to import comma-separated value (CSV) data.

First, have PV-WAVE write the data to a file; the output must be in CSV format. You can do this with the DC_WRITE_FREE function.

```
data = INDGEN(10, 10)
      ; Create some example data, such as a 10-by-10 array of integers.
status = DC_WRITE_FREE('data.csv', data, /Column)
```

This DC_WRITE_FREE command writes a file called `data.csv`, which contains CSV data that can be read directly into a Microsoft Excel spreadsheet.

Now you can import the file into Excel as a CSV format file.

TIP It is also possible to run Excel directly from PV-WAVE using the SPAWN procedure. SPAWN lets you execute external programs from within PV-WAVE. For detailed information on SPAWN, see the PV-WAVE Reference. The following SPAWN command tells Excel to start and load the CSV data file that was created using PV-WAVE:

```
WAVE> SPAWN, 'C:\excel.exe C:\data.csv', /Nowait
```

The input file to Excel must be in CSV format and must have an `.csv` extension.

For more information on DC_WRITE_FREE, see its description in the PV-WAVE Reference. For information on how to transfer data from Microsoft Excel to PV-WAVE, see [Transferring Data from Microsoft® Excel to PV-WAVE on page 213](#).

Transferring Data from Microsoft® Excel to PV-WAVE

This section describes a method for reading comma-separated value (CSV) data from a Microsoft Excel spreadsheet into PV-WAVE. Once the data is imported into PV-WAVE, you can use graphics functions such as PLOT and OPLOT to visualize it. This method of transferring data from another application to PV-WAVE will work with any application as long as the application can export CSV data.

First, save the data from your Excel spreadsheet using the **Save File as Type: CSV** option on the Save As dialog box. This option saves the data in a text file containing

values separated by commas. Blank cells are written as “ , , ” (a pair of commas) which PV-WAVE interprets as zeros.

For example, if your spreadsheet contains the following array of cells:

Q1	Q2	Q3	Q4
50		25	200
40	100	50	
200	50	50	100

when it is saved as CSV data, the resulting file looks like this:

```
Q1, Q2, Q3, Q4
50,, 25, 200
40, 100, 50,,
200, 50, 50, 100
```

NOTE The blank cells in the spreadsheet are saved as “ , , ” (a pair of commas). These blank values are interpreted as zeros when they are imported into PV-WAVE.

Now, you can read this CSV data file into PV-WAVE. First, create arrays to hold the text of the column headings, then create an array to hold the data:

```
headings = STRARR(4)
sales = INTARR(4, 3)
```

Next, use the DC_READ_FREE function to read the data into the arrays headings and sales. For detailed information on the DC_READ_FREE function, see its description in the PV-WAVE Reference.

```
status = DC_READ_FREE('region1.csv', headings, sales)
; The first parameter is the name of the data file to read. The next two
; parameters are the names of the variables into which the data is read.
```

Enter the following commands to check that the data was read correctly:

```
PRINT, headings
    Q1 Q2 Q3 Q4

PRINT, sales
    50, 0, 25, 200
    40, 100, 50, 0
    200, 50, 50, 100
```

Notice that the empty cells have been replaced by zeros.

For information on how to transfer data from PV-WAVE to Microsoft Excel, see *Transferring Data from PV-WAVE to Microsoft® Excel* on page 213.

Writing Procedures and Functions

A procedure or function is a self-contained module that performs a well-defined task. Procedures and functions break large tasks into manageable smaller tasks. Writing modular programs simplifies debugging and maintenance and minimizes the amount of new code required for each application.

New procedures and functions may be written in PV-WAVE and called in the same manner as the system-defined procedures or functions (i.e., from the keyboard or from other programs). When a procedure or function is finished, it executes a RETURN statement which returns control to its caller.

The following directory contains procedures and functions that can be accessed by all PV-WAVE users:

(UNIX) <wavedir>/lib/user

(OpenVMS) <wavedir>:[WAVE.LIB.USER]

(Windows) <wavedir>\lib\user

Where <wavedir> is the main PV-WAVE directory.

UNIX and OpenVMS USERS This subdirectory is automatically placed in the environment variable or logical WAVE_PATH by the PV-WAVE initialization routine, and is also automatically placed in the system variable !Path. The user subdirectory is placed at the end of the search path and is thus searched last. For more information on the search path, see [Modifying Your Environment](#) on page B-1.

PV-WAVE automatically compiles and executes a user-written function or procedure when it is first referenced if:

- The source code of the routine is in the current working directory or in a directory in the search path defined by the system variable !Path.
- and
- The name of the file containing the routine is the same as the routine name suffixed by .pro.

NOTE User-written functions must be compiled (e.g., with .RUN) before they are referenced, unless they meet the above conditions for automatic compilation. This restriction is necessary in order to distinguish between function calls and subscripted variable references.

A procedure is called by a procedure call statement, while a function is called via a function reference. A function always returns an explicit result. For example, if ABC is a procedure and XYZ is a function:

```

ABC, A, 12
    ; Calls procedure ABC with two parameters.

A = XYZ (C/D)
    ; Calls function XYZ with one parameter. The result of XYZ is stored
    ; in variable A.

```

Procedure and Function Parameters

The variables and expressions passed to the function or procedure from its caller are parameters. *Actual parameters* are those appearing in the procedure call statement or the function reference. In the above examples, the actual parameters in the procedure call are the variable A and the constant 12, while the actual parameter in the function call is the value of the expression (C/D).

The procedure and function definition statements notify the compiler that a user-written program module follows. The syntax of these definition statements is:

```
PRO Procedure_name, p1, p2, ..., pn
```

```
FUNCTION Function_name, p1, p2, ..., pn
```

Formal parameters are the variables declared in the procedure or function definition. The same procedure or function may be called using different actual parameters from a number of places in other program units.

Correspondence Between Formal and Actual Parameters

Correspondence between the caller's actual parameters and the called procedure's formal parameters is established by *position* or by *keyword*.

A *keyword parameter*, which may be either actual or formal, is an expression or variable name preceded by a keyword and an equal sign that identifies which parameter is being passed. When calling a procedure with a keyword parameter, you can abbreviate the keyword to its shortest unambiguous abbreviation. Keyword parameters may also be specified by the caller with the syntax */Keyword*, which is equivalent to setting the keyword parameter to 1 (e.g., *Keyword = 1*).

A *positional parameter* is a parameter without a keyword. Just as its name implies, the position of positional parameters establishes the correspondence. The *n*th formal positional parameter is matched with the *n*th actual positional parameter.

NOTE Do not use reserved words for keywords. If you use a reserved word as a keyword, a syntax error will result. For a list of the reserved words in PV-WAVE, see [Names of Variables on page 26](#).

Example of Using Positional and Keyword Parameters

A procedure is defined with a keyword parameter named `Test`:

```
PRO XYZ, A, B, Test = T
```

The caller can supply a value for the format parameter `T` with the calls:

```
XYZ, Test = A
```

```
    ; Supplies only the value of T. A and B are undefined inside the  
    ; procedure.
```

```
XYZ, Te = A, Q, R
```

```
    ; The value of A is copied to formal parameter T (note the  
    ; abbreviation for Test), Q to A, and R to B.
```

```
XYZ, Q
```

```
    ; Variable Q is copied to formal parameter A. B and T are undefined  
    ; inside the procedure.
```

Copying Actual Parameters into Formal Parameters

When a procedure or function is called, the actual parameters are copied into the formal parameters of the procedure or function and the module is executed. On exit, via a `RETURN` statement, the formal parameters are copied back to the actual parameters if they were not expressions or constants. Parameters may be inputs to

the program unit; or they may be outputs in which the values are set or changed by the program unit; or they may be both inputs and outputs.

When a RETURN statement is encountered in the execution of a procedure or function, control is passed back to the caller immediately after the point of the call. In functions, the parameter of the RETURN statement is the result of the function.

OpenVMS USERS Under OpenVMS, PV-WAVE procedures and functions must be defined with at least one formal parameter. Function calls must also have at least one actual parameter while procedure call statements may have zero or more actual parameters.

Number of Parameters Required in Call

A procedure or a function may be called with less arguments than were defined in the procedure or function. For example, if a procedure is defined with ten parameters, the user (or another procedure) may call the procedure with zero to ten parameters.

Parameters that are not used in the actual argument list are set to be *undefined* upon procedure or function entry. If values are stored by the called procedure into parameters not present in the calling statement, these values are discarded when the program unit exits. The number of actual parameters in the calling list may be found by using the system function N_PARAMS. Use the N_ELEMENTS function to determine if a variable is defined or not. The functions KEYWORD_SET and PARAM_PRESENT can be used to determine if parameters are used or not in a function or procedure call.

Example of a Function

An example of a function to compute the digital gradient of an image is shown below. The digital gradient approximates the two-dimensional gradient of an image and emphasizes the edges. This simple function consists of three lines, corresponding to the three required components of procedures and functions: 1) the procedure or function declaration, 2) the body of the procedure or function, and 3) the terminating END statement.

```
FUNCTION GRAD, IMAGE
    ; Defines a function called GRAD.

RETURN, ABS (IMAGE - SHIFT (IMAGE, 1, 0)) + $
    ABS (IMAGE - SHIFT (IMAGE, 0, 1))
    ; Evaluates and returns the result. Result = abs (dz/dx) + abs (dz/dy) which is
    ; the sum of the absolute values of the derivative in the x and y directions.
```

```
END
; End of function.
```

The function has one parameter called IMAGE. There are no local variables. (Local variables are variables within a module that are not parameters and are not contained in Common Blocks.)

The result of the function is the value of the expression appearing after the RETURN statement. Once compiled, the function is called by referring to it in an expression. Some examples might be:

```
A = GRAD(B)
; Store gradient of B in A.

TVSCL, GRAD(ABC + DEF)
; Display gradient of image sum.
```

Example Using Keyword Parameters

A short example of a function that exchanges two columns of a 4-by-4 homogeneous coordinate transformation matrix is shown. The function has one positional parameter, the coordinate transformation matrix, T. The caller can specify one of the keywords *XYexch*, *XZexch*, or *YZexch*, to interchange the XY, XZ, or YZ axes of the matrix. The result of the function is the new coordinate transformation matrix:

```
FUNCTION SWAP, T, XYEXCH = XY, $
  XZEXCH = XZ, YZEXCH = YZ
; Function to swap columns of T. If XYEXCH is specified swap
; columns 0 and 1, XZEXCH swaps 0 and 2, and YZEXCH
; swaps 1 and 2.
IF KEYWORD_SET(XY) THEN S = [0, 1]
; Swap columns 0 and 1?
ELSE IF KEYWORD_SET(XZ) THEN S = [0, 2]
; XZ set?
ELSE IF KEYWORD_SET(YZ) THEN S = [1, 2]
; YZ set?
ELSE RETURN, T
; Nothing is set, just return.
R = T
; Copy matrix for result.
R(S(1), 0) = T(S(0), *)
R(S(0), 0) = T(S(1), *)
; Exchange two columns using matrix insertion operators and
; subscript ranges.
```

```

RETURN, R
; Return result.

END

```

Typical calls to SWAP are:

```

Q = SWAP(!P.T, /XYexch)
Q = SWAP(Q, /XYexch)
Q = SWAP(INVERT(Z), YZexch = 1)
Q = SWAP(Z, XYexch = I EQ 0, YZexch = I EQ 2)

```

The last example sets one of the three keywords, according to the value of the variable I.

This function example uses the system function KEYWORD_SET to determine if a keyword parameter has been passed and it is non-zero. This is similar to using the condition

```
IF N_ELEMENTS(P) NE 0 THEN IF P THEN ... ..
```

to test if keywords that have a true/false value are both present and true.

TIP Use the PARAM_PRESENT function in conjunction with KEYWORD_SET to test if a keyword was actually used in a function or procedure call. For information on PARAM_PRESENT, see the PV-WAVE Reference.

Compiling Procedures and Functions

There are three ways procedures and functions can be compiled:

- Using .RUN with a filename
- Compiling automatically
- Compiling with interactive mode

Using .RUN with a Filename

Procedures and functions can be compiled using the executive command .RUN. The format of this command is:

```
.RUN file1, file2, ...
```

From one to ten files, each containing one or more program units, may be compiled with the .RUN command. Consult *Executive Commands* in the PV-WAVE Reference for more information on the .RUN command.

Compiling Automatically

Generally, however, you create a procedure or function file with a filename that matches the actual procedure or function name. Then you do not need to use `.RUN` to compile the procedure or function file if this file is contained in the current working directory, PV-WAVE library directory, or in the `!Path` directory. The procedure or function automatically compiles when first called. For example, a function named `CUBE` which calculates the cube of a number has the file name `cube.pro`. The file looks like this:

```
FUNCTION CUBE, NUMBER
RETURN, NUMBER ^ 3
END
```

When the function is initially used within a PV-WAVE session the file is automatically compiled. A compiled module message displays on the screen. For example, using the function for the first time at the `WAVE>` prompt results in:

```
WAVE> z = cube(4) & print, z
% Compiled module: CUBE
64
```

If you change the source file of a routine that is currently compiled in memory, then you have to explicitly recompile it with `.RUN` or `.RNEW`.

Compiling with Interactive Mode

You can enter the procedure or function text directly at the keyboard (interactively) by simply entering `.RUN` in response to the `WAVE>` prompt. Rather than executing statements immediately after they are entered, PV-WAVE compiles the program unit as a whole. See [Creating and Running a Function or Procedure on page 5](#).

Procedure and function definition statements may not be entered in the single statement mode but must be prefaced by either `.RUN` or `.RNEW` when being created interactively.

The first non-empty line the compiler reads determines the type of the program unit: procedure, function, or main program. If the first non-empty line is not a procedure or function definition, the program unit is assumed to be a main program.

The name of the procedure or function is given by the identifier following the keyword *Pro* or *Function*. If a program unit with the same name is already compiled, it is replaced by the newer program unit.

Note Regarding Functions

User-defined functions must be compiled using the .RUN command *before* the first reference to the function is made. There are two exceptions:

- As discussed previously in [Compiling Automatically on page 223](#), if the file-name is the same as the function name and is located in the current working directory or in the !Path directory, the file automatically compiles.
- The file is located in the PV-WAVE library directory.

Otherwise the function must be compiled using .RUN because the compiler is unable to distinguish between a reference to a subscripted variable and a call to a presently undefined user function with the same name. For example, in the statement:

```
A = XYZ(5)
```

it is impossible to tell if XYZ is an array or a function by context alone.

Always compile the lowest-level functions (those that call no other functions) first, then higher-level functions, and finally procedures. PV-WAVE searches the current directory and then those in the directory path (!Path) for function definitions when encountering references that may either be a function call or a sub-scripted variable, thereby avoiding this restriction in the case of library functions.

System Limits and the Compiler

When a program is compiled (using .RUN, for example) output is directed to two areas: the code area and the data area. The code area holds internal instruction codes and the data area holds symbols for variables, common blocks, and keywords. If these areas become full, the compile is halted, and you will see one of the following messages:

```
Program code area full.
```

```
Program data area full.
```

Methods of handling these errors are discussed in the following sections.

Program Code Area Full

This message indicates that the code area (a block of memory that is allocated for use by the compiler to store instruction codes) has been exceeded. As a result, the compile cannot be completed. The method used to correct this condition depends on the type of program you are compiling:

If You are Compiling a Procedure or Function

There are two solutions:

- Break the procedure or function into smaller procedures or functions, or
- Use the `.SIZE` executive command to increase the original size of the code area. The `.SIZE` command is described in *Executive Commands* in the PV-WAVE Reference.

Compiling Main Programs

If you use `.RUN` or `.RNEW` to compile a file that contains statements that are not inside a function or procedure, and you receive the `Program code area full` message, you have these options:

- Use the `.SIZE` executive command to increase the original size of the code area. The `.SIZE` command is described in *Executive Commands* in the PV-WAVE Reference.
- Place the statements that will be executed at the `$MAIN$` level — those that are not contained in a procedure or function — into a file that does not contain any procedure or function definitions, then execute the program as a command file using the `@` command. For example:

@filename

The `@` command compiles and executes the commands in the file one at a time, which does not require much code area space.

Program Data Area Full

This message indicates that the data area (a block of memory that is allocated for use by the compiler to store symbolic names of variables and common blocks) has been exceeded. As a result, the compile cannot be completed. The method used to correct this condition depends on the type of program you are compiling:

If You are Compiling a Procedure or Function

There are three solutions:

- Break the procedure or function onto smaller procedures or functions.
- Use the `.SIZE` executive command to increase the original size of the data area. The `.SIZE` command is described in *Executive Commands* in the PV-WAVE Reference.

- Use the `..LOCALS` executive command to increase the original size of the data area. The `..LOCALS` command is described in *Executive Commands* in the PV-WAVE Reference.

If You are Using the EXECUTE Function in a Program

The EXECUTE function uses a string containing a PV-WAVE command as its argument. The command passed to EXECUTE is not compiled until EXECUTE itself is executed. At that time, you may see the `Program data area full` message if the data area is already full and EXECUTE tries to create a new variable or common block.

If this occurs, you have the following options:

- If the program is a main program, then use `..SIZE` or `..LOCALS` to increase the size of the data area.
- If the program is a function or procedure, then it is necessary to use the `..LOCALS` compiler directive in the function or procedure. The `..LOCALS` compiler directive creates additional data area space at runtime.

NOTE In general, if you use EXECUTE to create variables or common blocks in a function or procedure, then it is likely that you will need to use `..LOCALS`. This is because the data area is compressed immediately after compilation to accommodate only the symbols that are known at compile time. Thus, if EXECUTE is used to create variables or common blocks, there may not be space for any new symbols to be created. The `..LOCALS` command is discussed in the next section.

Using the ..LOCALS Compiler Directive

The syntax of the `..LOCALS` compiler directive is:

```
..LOCALS local_vars common_symbols
```

This command is useful when you want to place the EXECUTE function inside a procedure or function. EXECUTE takes a string parameter containing a PV-WAVE command. This command argument is compiled and executed at runtime, allowing the possibility for command options to be specified by the user. Because the data area is compressed after compilation, there may not be enough room for additional local variables and common block symbols created by EXECUTE. The `..LOCALS` command provides a method of allocating extra space for these additional items.

The `..LOCALS` compiler directive is similar to the `.LOCALS` executive command, except:

- `..LOCALS` is only used inside procedures and functions.
- Its arguments specify the number of *additional* local variables and common block symbols that will be needed at “interpreter” time (when the already-compiled instructions are interpreted).
- It is used in conjunction with the `EXECUTE` function, which can create new local variables and common block symbols at runtime.

Example 1

In this example, `..LOCALS` is not needed. This simple procedure does not use the `EXECUTE` function to create new variables or common blocks.

```
PRO mypro1, a
  COMMON c, c1, c2
  a=10
END
```

Example 2

In this example, a new common block (`d`) and a new variable (`x`) are created with two calls to the `EXECUTE` function. The `..LOCALS` directive creates additional space for one variable (`x`) and two common block symbols (`d1` and `d2`).

```
PRO mypro2, a
  ..LOCALS 1 2
  COMMON c, c1, c2
  j=EXECUTE('COMMON d, d1, d2')
  a=10
  b=20
  j=EXECUTE('x=30')
END
```

Example 3

The following procedure can create up to 20 new local variables, as specified by the user at runtime. This example is more realistic than the previous one, because here you do not know how many new variables will be needed until runtime. In this case, however, if `i` is greater than 20, the data area may fill up.

```
PRO mypro3, i
```

```

    ..LOCALS 20
    for j=1, i DO BEGIN
    k=EXECUTE('var'+STRTRIM(STRING(j),2)+'=0')
    ENDFOR
END

```

This procedure creates *i* local variables:

```

VAR1=0
VAR2=0
VAR3=0
VARi=0

```

Parameter Passing Mechanism

Parameters are passed to system and user-written procedures and functions by *value* or by *reference*. It is important that you recognize the distinction between these two methods.

- Expressions, constants, system variables, and subscripted variable references are passed by *value*.
- Variables are passed by *reference*.

Parameters passed by value may only be inputs to program units; results may not be passed back to the caller via these parameters. Parameters passed by reference may convey information in either or both directions. For example consider this trivial procedure:

```

PRO ADD, A, B
A = A + B
RETURN
END

```

This procedure adds its second parameter to the first, returning the result in the first. The call:

```
ADD, A, 4
```

adds 4 to A and store the result in variable A. The first parameter is passed by reference and the second parameter, a constant is passed by value. The call:

```
ADD, 4, A
```

does nothing because a value may not be stored in the constant “4” which was passed by value. No error message is issued.

Similarly, if ARR is an array, the call:

```
ADD, ARR(5), 4
```

does not achieve the desired effect (adding 4 to element ARR (5)) because sub-scripted variables are passed by value. A possible alternative is:

```
TEMP = ARR(5)
```

```
ADD, TEMP, 4
```

```
ARR(5) = TEMP
```

Procedure or Function Calling Mechanism

When a user-written procedure or function is called, the following actions take place:

- All of the actual arguments in the user procedure call list are evaluated and saved in a temporary location.
- The actual parameters that were saved are substituted for the formal parameters given in the definition of the called procedure. All other variables local to the called procedure are set to undefined.
- The procedure is executed until a RETURN or RETALL statement is encountered. The result of a user-written function is passed back to the caller by specifying it as the parameter of a RETURN statement. RETURN statements in procedures may not have parameters.
- All local variables in the procedure (i.e., those variables that are neither parameters nor common variables) are deleted.
- The new values of the parameters that were passed by reference are copied back into the corresponding variables. Actual parameters that were passed by value are deleted.
- Control resumes in the calling procedure after the procedure call statement or function reference.

Recursion

Recursion is supported with both procedures and functions.

Example Using Variables in Common Blocks

Here is an example of a procedure that reads and plots the next vector from a file. This example illustrates how to use common variables to store values between calls, as local parameters are destroyed on exit. It assumes that the file containing

the data is open on logical unit 1 and that the file contains a number of 512-element floating-point vectors.

```
PRO NXT, Recno
    ; Read and plot the next record from file 1. If Recno is specified, set
    ; the current record to its value and plot it.

COMMON Nxt_Com, Lastrec
    ; Save previous record number.
    IF N_PARAMS(0) GE 1 THEN Lastrec = Recno
        ; Set record number if parameter is present.
    IF N_ELEMENTS(Lastrec) LE 0 THEN $
        Lastrec = 0
        ; Define Lastrec if this is first call.
    AA = ASSOC(1, FLTARR(512))
        ; Define file structure.
    PLOT, AA(Lastrec)
        ; Read record and plot it.
    Lastrec = Lastrec + 1
        ; Increment record for next time.
    RETURN
        ; All finished.

END
```

Once you have opened the file, typing `NXT` will read and plot the next record. Typing `NXT, n` will read and plot record number *n*.

Error Handling in Procedures

Whenever an error occurs during the execution of a user-written procedure, a description of the error is printed and execution of the procedure halts. You can change the environment that is restored after an error occurs with the `ON_ERROR` procedure. The four possible actions are:

- 0 — Stop at the statement in the procedure that caused the error, the default action.
- 1 — Return all the way back to the main program level.
- 2 — Return to the caller of the program unit which established the `ON_ERROR` condition.
- 3 — Return to the program unit which established the `ON_ERROR` condition.

If ON_ERROR is not called by a parent of the procedure in which an error occurs, the procedure is *not exited*, and the current variables are those of the halted procedure, not of the caller. To return to the calling unit, or to the single statement mode if the procedure was called from the single statement mode, you should enter a RETURN or RETALL statement from the terminal.

Calling ON_ERROR from the main level or from a procedure sets the default error action for all modules called from that level. For example, if you always wish to return to the main level after an error, simply issue the statement:

```
ON_ERROR, 1
```

from the main level, or from your startup procedure.

Many library procedures issue an ON_ERROR, 2 call to return to their caller if an error occurs.

Error Signaling

Use the MESSAGE procedure in user-written procedures and functions to issue errors. For detailed information on this procedure, see [Error Signaling on page 241](#).

“Disappearing Variables”

PV-WAVE novices are frequently dismayed to find that all their variables have seemingly disappeared after an error occurs inside a procedure or function. The misunderstood subtlety is that after the error occurs PV-WAVE’s context is inside the called procedure, not in the main level. Typing RETURN or RETALL will make the lost variables reappear.

RETALL is best suited for use when an error is detected in a procedure and you want to return immediately to the main program level despite nested procedure calls. RETALL issues RETURNS until the main program level is reached.

The Users’ Library

In order to support and encourage development and sharing of PV-WAVE programs in scientific and technical disciplines, Visual Numerics has established the Users’ Library. The purpose of the library is to help users solve common problems and avoid duplicating the efforts of others. Users are encouraged to submit PV-WAVE procedures and functions they believe are particularly valuable or are of general interest to Visual Numerics for incorporation into the library. Coordinate submis-

sions through the Visual Numerics' Customer Support Group or through your local Technical Support Engineer. The library is updated periodically and distributed free of charge to all PV-WAVE sites.

The Users' Library is located in:

(UNIX) <wavedir>/lib/user

(OpenVMS) <wavedir>:[WAVE.LIB.USER]

(Windows) <wavedir>\lib\user

Where <wavedir> is the main PV-WAVE directory.

Procedures and functions in this subdirectory are automatically compiled when they are referenced from within PV-WAVE.

OpenVMS USERS If you are running PV-WAVE under OpenVMS, you must load any new or modified Users' Library routines into the PV-WAVE text library. Text libraries are explained previously in this chapter.

NOTE PV-WAVE searches for Users' Library procedures and functions along the path specified by the !Path system variable. In most cases this means the first directory searched is the current directory. If a procedure or function with the same name as a Users' Library routine is found in the current directory (or in any directory searched before the Users' Library directory), it is compiled and used in place of the Users' Library routine. This is different from the way system routines are called and used.

Submitting Programs to the Users' Library

The major requirement for a procedure or function to be submitted is that a standardized template be included in the program source. The purpose of the template is to describe the program in enough detail that others may use the program with little difficulty. An empty template is stored in the file `template`, located in the `lib` subdirectory of the main PV-WAVE directory.

Try to write routines in as general a manner as possible. For example, dedicated logical units should never be used. Instead, the `GET_LUN` and `FREE_LUN` procedures should be used to allocate and deallocate logical units. Routines should be able to handle as many different types and structures of data as possible — this includes performing parameter checking to ensure the parameters are the correct type. It is a good idea to use the `ON_ERROR` procedure to return to the caller in the event of an error. The code itself should also be liberally commented.

Procedures and functions conforming to the above requirements will be included in periodic PV-WAVE releases.

Support for Users' Library Routines

Visual Numerics provides minimal testing and no documentation of the procedures and functions submitted to the Users' Library. The library is provided as a service, and users are advised to use caution when incorporating these routines into their own programs. The Users' Library routines do not enjoy the same level of support or confidence Visual Numerics reserves for system procedures and functions in the Standard Library (`std`).

OpenVMS Procedure Libraries

The information in this section applies to PV-WAVE running under OpenVMS only.

When a procedure or function call to an unknown module is encountered, PV-WAVE searches known text libraries for the module. If a module with the same name as the procedure or function is found in a library, the module is extracted from the library and compiled. Program execution resumes with execution of the newly compiled procedure. If the module is not found, an error results and execution stops.

Libraries are searched for functions if the first reference to the function is made with a parameter list and the name has not been defined as a variable. If a variable has the same name as a function defined in one of the libraries, and the first reference is made with a subscript list (indistinguishable from a parameter list), then the name will be set to function type and the variable will be inaccessible in all program units.

The logical names `WAVE$LIBRARY` and `WAVE$LIBRARY_n` are used by PV-WAVE to find the actual text libraries. Up to ten libraries may be active at one time. The library search method is similar to that used by the VAX Linker program when searching for user libraries.

Libraries are searched in the following order:

□ Process logical name table:

```
WAVE$LIBRARY  
WAVE$LIBRARY_1  
WAVE$LIBRARY_2  
...
```

```
WAVE$LIBRARY_9
```

❑ Group logical name table:

```
WAVE$LIBRARY  
WAVE$LIBRARY_1  
...  
WAVE$LIBRARY_9
```

❑ System logical name table:

```
WAVE$LIBRARY  
WAVE$LIBRARY_1  
...  
WAVE$LIBRARY_9
```

An attempt is made to translate each logical name into an actual device and filename. If the attempt fails, indicating that the logical name has not been assigned, searching is terminated in the current logical name table and is started at the next level. Libraries are searched in the above order when locating procedures and functions. For example, if a procedure is defined in both system and process-level libraries, it will be taken from the library defined in the process logical name table because it is searched first.

Assign the logical name `WAVE$LIBRARY` to the actual filename of your PV-WAVE library. For example, if the primary library is `$DISK0 : [SMITH] WAVE . TLB` and the secondary library is `$DISK1 : [JONES] WAVE . TLB`, the following logical assignments should be made before running PV-WAVE:

```
DEFINE WAVE$LIBRARY $DISK0 : [SMITH] WAVE . TLB  
DEFINE WAVE$LIBRARY_1 $DISK1 : [JONES] WAVE . TLB
```

The library in `[SMITH]` will be searched first, then the library in `[JONES]`, followed by any libraries in the group or system logical name tables.

The above assignments may be made in the login command file, system start-up command file, or manually by directly entering DCL commands.

Creating OpenVMS Procedure Libraries

The information in this section applies to PV-WAVE running under OpenVMS only.

PV-WAVE procedure libraries are simply standard OpenVMS text libraries. A text library is a file containing a number of text modules and an index. Text libraries are built and maintained with the VAX Librarian Utility.

Modules, each containing a single procedure or function, may be inserted, replaced, deleted, and extracted using the Librarian Utility. Each module must be named with the name of the program unit it contains and may contain only one program unit. If necessary, use the `/Module` switch to explicitly specify the name.

To create a procedure library, use a text editor to create one or more files each containing a PV-WAVE procedure or function. Once the PV-WAVE code has been debugged, use the Librarian to create a text library from your procedure files. For example, the OpenVMS command:

```
LIBRARY /CREATE /INSERT /TEXT WAVE abc.pro, def.pro
```

invokes the library utility to create a new text library which will be named `WAVE.TLB`, and to insert the files `abc.pro` and `def.pro`. The two modules in the library are named `ABC` and `DEF`, as the module name defaults to the file name.

To extract the module `ABC` from the library file `abc.pro` use:

```
LIBRARY /EXTRACT = ABC /TEXT /OUT = abc.pro WAVE
```

The file may be edited and then replaced in the library with the command:

```
LIBRARY /REPLACE /TEXT WAVE abc.pro
```

Use the `/Module` qualifier to specify the module name if the procedure or function does not have the same name as its filename.

For example, to insert a function called `POLY_FIT`, contained in a file called `polyfit` into the library, use the following library command:

```
LIBRARY /TEXT WAVE polyfit.pro /Module = POLY_FIT
```

Consult the *VAX-11 Utilities Reference Manual* for more information concerning the Librarian.

Programming with PV-WAVE

The routines discussed in this chapter are characterized by the fact that they are useful primarily (though not exclusively) in PV-WAVE procedures and functions. They are rarely used during interactive use. They provide information about variables and expressions, give the programmer control over how errors are handled, and perform other useful operations.

Routines may be loosely categorized into the following groups:

- *Error handling routines* such as ON_ERROR, ON_IOERROR, FINITE, and CHECK_MATH.
- *Informational routines* which return information about variables, expressions, parameters, etc. These routines are N_ELEMENTS, SIZE, N_PARAMS, PARAM_PRESENT, and KEYWORD_SET. In addition, TAG_NAMES and N_TAGS supply information about structure variables.
- *Program control routines* such as EXIT, EXECUTE, STOP, and WAIT.

Description of Error Handling Routines

PV-WAVE divides execution errors into three categories: input/output, math, and all others. ON_ERROR gives control over how regular errors are handled. The ON_IOERROR procedure allows you to change the default way in which I/O errors are handled. FINITE, and CHECK_MATH give control over math errors.

Default Error Handling Mechanism

In the default case, whenever an error is detected by PV-WAVE during the execution of a program, program execution stops and an error message is printed. The variable context is that of the program unit, (procedure, function, or main program), in which the error occurred.

As explained in [Error Handling in Procedures on page 230](#), novices are frequently dismayed to find that all their variables have seemingly disappeared after an error occurs inside a procedure or function. The misunderstood subtlety is that after the error occurs, the context is that of the called procedure or function, not the main level. All variables in procedures and functions, with the exception of parameters and common variables, are *local* in scope.

Sometimes it is possible to recover from an error by manually entering statements to correct the problem. Possibilities include setting the values of variables, closing files, etc., and then entering the .CON executive command, which will resume execution at the beginning of the statement that caused the error.

As an example, if an error stop occurs because an undefined variable is referenced, simply define the variable from the keyboard and then continue execution with .CON.

Controlling Errors

The two procedures ON_ERROR and ON_ERROR_GOTO determine the action to be taken when an error is detected inside a procedure or function. This section provides a brief introduction to these procedures. For more details on these procedures, see their descriptions in the PV-WAVE Reference.

The ON_ERROR procedure specifies an action to take after an error occurs inside a procedure or function. The following table lists the basic options for error recovery. These options specify an action and cause program execution to stop.

Options for Error Recovery

Value	Action
0	Stop immediately in the context of the procedure or function that caused the error. This is the default action.
1	Return to the main program level and stop.
2	Return to the caller of the program unit that called ON_ERROR and stop.
3	Return to the program unit that called ON_ERROR and stop.

Additional options, specified in conjunction with the *Continue* keyword, specify an action to take and allow the program to continue executing after the action. These options are listed in the following table:

Options for Error Recovery and Program Continuation

Value	Action
0	Continue in the procedure that caused the error.
1	Return to \$MAIN\$ and continue.
2	Return to the caller that established the ON_ERROR condition and continue.
3	Return to the program unit that established the ON_ERROR condition and continue.

One useful option is to use ON_ERROR to cause control to be returned to the caller of a procedure in the event of an error. The statement:

```
ON_ERROR, 2
```

placed at the beginning of a procedure will have this effect. It is a good idea to include this statement in library procedures, and any routines that will be used by others, but only after debugging is completed. Debugging a routine after using the ON_ERROR procedure is made more difficult because the routine is exited as soon as an error occurs. Therefore, it should be added once the code is completely tested.

The ON_ERROR_GOTO procedure transfers program control to a specified statement label after an error occurs. A label, as explained in [Statement Labels on page 48](#), is an identifier followed by a colon. A label may exist on a line by itself.

For example:

```
PRO Proc1
    . . .
    . . .
    ON_ERROR_GOTO, Proc1_Failed
        ; If error occurs here, go to the statement label Proc1_Failed.
    Proc1_Failed:
    PRINT, !Err, !Err_String
END
```

Error Handling in WAVE Widgets Applications

WAVE Widgets procedures normally try to continue executing after an error has occurred. WAVE Widgets procedures report the error and traceback information, and return a value indicating a failure has occurred. Usually, the value 0L is returned as the widget ID. It is the programmer's responsibility to check returned values and take appropriate action.

WAVE Widgets callbacks, event and input handlers, timers, and work procedures by default return from the given procedure first and then continue execution after an error. When an error is detected, execution is continued at the statement specified by the input parameter of the last `ON_ERROR` call or by the `ON_ERROR_GOTO` statement in the given callback, handler, timer, or work procedure routine.

It is the programmer's responsibility to set `ON_ERROR` or `ON_ERROR_GOTO` appropriately. To override the default error handling behavior of callbacks, handlers, timers, or work procedures, set:

```
ON_ERROR, Continue=0
```

NOTE If a WAVE Widgets application stops during execution, you can use the `RETALL` procedure to stop all the currently running WAVE Widgets applications and return to the main program level.

Controlling Input and Output Errors

The default action for handling input/output errors is to treat them exactly like regular errors and follow the error handling strategy set by `ON_ERROR`. You can alter the default handling of I/O errors using the `ON_IOERROR` procedure to specify the label of a statement to which execution should jump if an I/O error occurs. When PV-WAVE detects an I/O error and an error handling statement has been established, control passes directly to the given statement without stopping program execution. In this case, no error messages are printed.

When writing procedures and functions that are to be used by others, it is good practice to anticipate and gracefully handle errors caused by the user. For example, the following procedure segment, which opens a file specified by the user, handles the case of a non-existent file or read error:

```
FUNCTION READ_DATA, file_name
    ; Define a function to read and return a 100-element floating-point array.
    ON_IOERROR bad
    ; Declare error label.
```

```

OPENR, UNIT, file_name, /Get_Lun
    ; Use the Get_Lun keyword to allocate a logical file unit.
A = FLTARR(100)
    ; Define data array.
READU, UNIT, A
    ; Read it.
GOTO, DONE
    ; Clean up and return.
bad: PRINT, !Err_string
    ; Exception label. Print the error message.
DONE:
Free_Lun, UNIT
    ; Close and free the I/O unit.
RETURN, A
    ; Return the result. This will be undefined if an error occurred.
END

```

The important things to note are that the *Free_Lun* procedure is always called, even in the event of an error, and that this procedure always returns to its caller. It returns an undefined value if an error occurred, causing its caller to encounter the error.

Error Signaling

Use the MESSAGE procedure in user-written procedures and functions to issue errors. It has the form:

```
MESSAGE, text
```

where *text* is a scalar string describing the error.

MESSAGE issues error and informational messages using the same mechanism employed by built-in routines. By default, the message is issued as an error, the message is output, and PV-WAVE takes the action specified by the ON_ERROR procedure. As a side effect of issuing the error, the system variables !Err and !Error are set and the text of the error message is placed in the system variable !Err_String.

As an example, assume the statement:

```
MESSAGE, 'Unexpected value encountered.'
```

is executed in a procedure named CALC. The result would be the following:

```
% CALC: Unexpected value encountered.
```

MESSAGE accepts several keywords which modify its behavior. See the description of MESSAGE in the PV-WAVE Reference for additional details.

Another use for MESSAGE involves resignaling trapped errors. For example the following code uses ON_IOERROR to read from a file until an error (presumably end of file) occurs. It then closes the file and reissues the error:

```
OPENR, UNIT, 'data.dat', /Get_Lun
    ; Open the data file.

ON_IOERROR, eod
    ; Arrange for jump to label eod when an I/O error occurs.

TOP: READF, UNIT, LINE
    ; Read every line of the file.

GOTO, TOP
    ; Go read the next line.

eod: ON_IOERROR, NULL
    ; An error has occurred. Cancel the I/O error trap.

Free_Lun, UNIT
    ; Close the file.

MESSAGE, !Err_string, /Noname, /Ioerror
    ; Reissue the error. !Err_string contains the appropriate text. The
    ; keyword causes it to be issued as an I/O error. Use of Noname
    ; prevents MESSAGE from tacking the name of the current routine
    ; to the beginning of the message string, since !Err_String already
    ; contains it.
```

Obtaining Traceback Information

It is sometimes useful for a procedure or function to obtain information about its caller(s). The INFO procedure returns, in a string array, the contents of the procedure stack, when the *Calls* keyword parameter is specified. The first element of the resulting array contains the module name, source file name and line number of the current level. The second element contains the same information for the caller of the current level, and so on back to the level of the main program.

For example, the following code fragment prints the name of its caller, followed by the source file name and line number of the call:

```
INFO, CALLS = a
PRINT, 'Called from: ', a(1)
    ; Print 2nd element.
```

Resulting in a message of the following form:

Called from: DIST <wave/lib/dist.pro (27)>

For this example, the commands were entered on a UNIX system; on a Windows or OpenVMS system the pathname will appear differently.

Programs can readily parse the traceback information to extract the source file name and line number.

Detection of Math Errors

On Windows Systems

PV-WAVE detects the following six mathematical errors conditions:

- Integer divide by zero.
- Integer overflow.
- Floating-point divide by zero.
- Floating-point underflow.
- Floating-point overflow.
- Floating-point operand error. (An illegal operand was encountered, such as a negative operand to the SQRT or ALOG functions; or an attempt to convert to integer a number whose absolute value is greater than $2^{31}-1$.)

When an error is detected, PV-WAVE prints an error message indicating the source of the statement that caused the error and continues program execution. Up to eight messages are printed before program execution stops.

On UNIX and OpenVMS Systems

The detection of math errors, such as division by 0, overflow, and attempting to take the logarithm of a negative number, is hardware dependent. Some machines, such as the VAX/OpenVMS, trap on all math errors, while others never trap.

On machines that handle floating-point exceptions and integer math errors properly, PV-WAVE prints an error message indicating the source statement that caused the error and continues program execution. Up to eight error messages are printed.

Checking the Accumulated Math Error Status

PV-WAVE maintains an accumulated math error status. This status, which is implemented as a longword, contains a bit for each type of math error that is detected by the hardware. PV-WAVE checks and clears this indicator each time the interactive prompt is issued, and if it is non-zero, prints an error message. A typical message is:

```
% Program caused arithmetic error: Floating divide by 0
```

This means that a floating division by 0 occurred since the last interactive prompt.

The CHECK_MATH function, described below, allows you to check and clear this accumulated math error status when desired. It is used to control how PV-WAVE treats floating-point exceptions on machines that don't properly support them. It can also disable printing of math error messages.

Special Values for Undefined Results

Under Windows, and on any machine which implements the IEEE standard for binary floating-point arithmetic, such as the Sun, there are two special values for undefined results, *NaN* (Not A Number), and *Infinity*. Infinity results when a result is larger than the largest representation. *NaN* is the result of an undefined computation such as zero divided by zero, taking the square-root of a negative number, or the logarithm of a non-positive number. These special operands propagate throughout the evaluation process. The result of any term involving these operands is one of these two special values.

Check the Validity of Operands

Use the FINITE function to explicitly check the validity of floating-point or double-precision operands. This works under Windows and on machines which use the IEEE floating-point standard. For example, to check the result of the EXP function for validity:

```
a = EXP(expression)
    ; Perform exponentiation.

IF NOT FINITE(a) THEN PRINT, $
    'overflow occurred'
    ; Print error message, or if a is an array do the following:

IF FINITE(a) NE N_ELEMENTS(a) THEN error
```

Check for Overflow in Integer Conversions

When converting from floating to byte, short integer or longword types, if overflow is important, you *must* explicitly check to be sure the operands are in range. Conversions to the above types from floating-point, double-precision, complex, and string types *do not* check for overflow; they simply convert the operand to longword integer and extract the low 8, 16, or 32 bits.

UNIX USERS When run on a Sun workstation, the program:

```
a = 2.0 ^ 31 + 2
PRINT, LONG(a), LONG(-a), FIX(a), FIX(-a), $
      BYTE(a), BYTE(-a)
```

which creates a floating-point number two larger than the largest positive longword integer, will print the following incorrect results:

```
2147483647 -2147483648 -1 0 255 0
% Program caused arithmetic error: Floating illegal operand
```

CAUTION No error message will appear if you attempt to convert a floating number whose absolute value is between 2^{15} and $2^{31}-1$ to short integer even though the result is incorrect. Similarly, converting a number in the range of 256 to $2^{31}-1$ from floating, complex or double to byte type produces an incorrect result but no error message. Furthermore, integer overflow is usually not detected. If integer overflow is a problem, your programs must guard explicitly against it.

Trap Math Errors with the CHECK_MATH Function

As mentioned previously, the CHECK_MATH function lets you test the accumulated math error status. It is also used to enable or disable traps. Each call to this function returns and clears the value of this status.

NOTE CHECK_MATH does not properly maintain an accumulated error status on machines that do not implement the IEEE standard for floating-point math.

It is good programming practice to bracket segments of code which might produce an arithmetic error with calls to CHECK_MATH to properly handle ill-conditioned results.

Its call is:

```
result = CHECK_MATH([print_flag, message_inhibit])
```

If an error condition has been detected, and the first optional parameter is present and non-zero, an error message is printed and program execution continues. Otherwise, the routine runs silently.

If the second optional parameter, *message_inhibit*, is present and non-zero, error messages are disabled for subsequent math errors. The accumulated math error status is maintained, even when error messages are disabled. When the program completes and exits back to the PV-WAVE prompt, accumulated math error messages which have been suppressed are printed. To suppress this final message, call CHECK_MATH to clear the accumulated error status before returning to the interactive mode.

The error status is encoded as an integer, where each binary bit represents an error, as shown in the following table:

NOTE Not all machines detect all errors.

Error Status Code Values

Value	Condition
0	No errors detected since the last interactive prompt or call to CHECK_MATH.
1	Integer divide by zero.
2	Integer overflow.
16	Floating-point divide by zero.
32	Floating-point underflow.
64	Floating-point overflow.
128	Floating-point operand error. An illegal operand was encountered, such as a negative operand to the SQRT or ALOG functions; or an attempt to convert to integer a number whose absolute value is greater than $2^{31}-1$.

Enable and Disable Math Traps

To enable trapping:

```
junk = CHECK_MATH(TRAP = 1)
```

To disable trapping:

```
junk = CHECK_MATH(TRAP = 0)
```

Examples Using the CHECK_MATH Function

For example, assume that there is a critical section of code that is likely to produce an error. The following code shows how to check for errors, and if one is detected to repeat the code with different parameters:

```
junk = CHECK_MATH(1,1)
    ; Clear error status from previous operations and print error
    ; messages if an error exists. Also, disable automatic printing of
    ; subsequent math errors.

again ...
    ; Critical section goes here.

IF CHECK_MATH(0,0) NE 0 THEN BEGIN
    ; Did an arithmetic error occur? Also, re-enable the printing of subsequent math errors.

PRINT, 'Math error occurred in critical ' + 'section'

READ, 'Enter new values: ', ...
    ; Input new parameters from user.

GOTO, again
    ; And retry.

ENDIF
```

Hardware-dependent Math Error Handling

Error Handling on a Sun-4 (SPARC) Running SunOS Version 4

Improper floating-point operations are trapped if traps are enabled. By default, traps *are* enabled. The result of an improper operation contains *garbage*, not an IEEE special value as would be expected if traps are enabled. When traps are disabled, via CHECK_MATH, the correct special values result, but no error message results until the next interactive prompt is issued.

Only integer divide by 0 is detected. Integer overflow is not detected.

Digital Workstation Error Handling

Integer divide by 0 is always trapped. Integer overflows produce no indication of error, and 0 results.

Floating-point traps may be enabled (the default) or disabled. The result of an improper floating-point operation that occurs when traps are enabled is *garbage*. If traps are not enabled, the correct IEEE special value results.

VAX/OpenVMS Error Handling

The VAX does not implement the IEEE floating point standard. The special values *NaN* and *Infinity* cannot occur. The FINITE function always returns a value of 1.

Most floating point functions check for and report overflow or illegal operands. Traps may be disabled, in which case math error messages are not immediately printed. Integer overflow is not detected, while integer division by 0 is.

Error Handling for Silicon Graphics Workstations Running IRIX 5.3

Some PV-WAVE routines may cause “Floating Illegal Operand” arithmetic errors. These errors are caused by a sensitivity to *NaN* comparisons. These messages do not halt PV-WAVE execution; the return values are correct. To disable this error trapping and reduce the number of “Floating Illegal Operand” messages to a final summary message, use the following PV-WAVE command:

```
WAVE> junk = CHECK_MATH (Trap=0)
```

Checking for Parameters

The informational routines, N_ELEMENTS, SIZE, N_PARAMS, PARAM_PRESENT, and KEYWORD_SET, are useful in procedures and functions to check if arguments are supplied. Procedures should be written to check that all required arguments are supplied, and to supply reasonable default values for missing optional parameters.

Checking for Parameters

PARAM_PRESENT tests if a parameter was actually present in the call to a procedure or function. It returns a nonzero value (TRUE) if the specified parameter was present in the call to the current procedure or function. If the specified parameter is not present, this function returns zero, or FALSE.

PARAM_PRESENT is a useful compliment to the functions KEYWORD_SET and N_ELEMENTS, described later in this section. PARAM_PRESENT can be

used to distinguish between the different cases in which those two routines return FALSE. Examples of this use are shown in the following sections.

Checking for Keywords

The `KEYWORD_SET` function returns a 1 (TRUE), if its parameter is defined and non-zero. The function returns 0 (FALSE) in the following cases:

- 1) When the keyword was set to zero.
- 2) When the keyword was not used in the call.

For example, assume that a procedure is written which performs a computation and returns the result. If the keyword *Plot* is present and non-zero the procedure also plots its result:

```
PRO XYZ, result, Plot=Plot
    ; Procedure definition. Compute result.
IF KEYWORD_SET(Plot) THEN PLOT, result
    ; Plot result if keyword parameter is set.
END
```

A call to this procedure that produces a plot is:

```
xyz, r, /Plot
```

The `PARAM_PRESENT` function lets you distinguish between the two FALSE cases of `KEYWORD_SET`. It returns TRUE when the keyword is set to 0 and FALSE when the keyword was not used.

Checking for Number of Positional Parameters

The `N_PARAMS` function returns the number of positional parameters (not keyword parameters) present in a procedure or function call. A frequent use is to call `N_PARAMS` to determine if all arguments are present, and if not to supply default values for missing parameters. For example:

```
PRO XPRINT, xx, yy
    ; Print values of xx and yy. If xx is omitted, print values of yy versus
    ; element number.
CASE N_PARAMS() OF
    ; Check number of arguments.
1: BEGIN
    ; Single argument case.
    y = xx
    ; First argument is y values.
```

```

        x = INDGEN(N_ELEMENTS(y))
        ; Create vector of subscript indices.
    END
    2: BEGIN & y = yy & x = xx & END
        ; Two argument case. Copy parameters to local arguments.
    ELSE: BEGIN
        ; Wrong number of arguments.

        PRINT, 'XPRINT - Wrong number of ' + $
            'arguments'
        ; Print message.

        RETURN
        ; Give up and return.
    END
ENDCASE
...
; Remainder of procedure.
END

```

Checking for Number of Elements

The `N_ELEMENTS` function returns the number of elements contained in any expression or variable. Scalars always have one element, even if they are scalar structures. The number of elements in arrays or vectors is equal to the product of the dimensions. The `N_ELEMENTS` function returns zero if its parameter is an undefined variable. The result is always a longword scalar.

For example, the following expression is equal to the mean of a numeric vector or array:

```
result = TOTAL(arr) / N_ELEMENTS(arr)
```

The `N_ELEMENTS` function provides a convenient method of determining if a variable is defined, as illustrated in the following statement. The following statement sets the variable `abc` to zero if it is undefined, otherwise the variable is not changed.

```
IF N_ELEMENTS(abc) EQ 0 THEN abc = 0
```

`N_ELEMENTS` is frequently used to check for omitted positional and keyword arguments. `N_PARAMS` can't be used to check for the number of keyword parameters because it returns only the number of positional parameters. An example of using `N_ELEMENTS` to check for a keyword parameter is:

```
PRO zoom, image, Factor=Factor
; Display an image with a given zoom factor. If Factor is omitted use 4.
```

```
IF N_ELEMENTS(Factor) EQ 0 THEN Factor = 4
    ; Supply default for missing keyword.
```

If `N_ELEMENTS` is used to check for the number of keyword parameters, it returns 0 in the following cases:

- 1) When the keyword or parameter was present but is an undefined variable.
- 2) When the keyword or parameter was not present in the call.

The `PARAM_PRESENT` function lets you distinguish between the two cases when `N_ELEMENTS` returns zero (0). `PARAM_PRESENT` returns `TRUE` when the keyword was present by is an undefined variable. It returns `FALSE` when the keyword was not present in the call.

Checking for Size and Type of Parameters

The `SIZE` function returns a vector that contains information indicating the size and type of the parameter. The returned vector is always of longword type. The first element is equal to the number of dimensions of the parameter, and is 0 if the parameter is a scalar. The following elements contain the size of each dimension. After the dimension sizes, the last two elements indicate the type of the parameter and the total number of elements respectively. The type is encoded as shown in the following table:

Type Code	Data Type
1	Byte
2	Integer
3	Longword integer
4	Floating-point
5	Double-precision floating
6	Complex floating
7	String
8	Structure

Example of Checking for Size and Type of Parameters

Assume A is an integer array with dimensions of (3, 4, 5). After executing, the statement:

```
B = SIZE(A)
```

assigns to the variable B a six-element vector containing:

b_0	3	Three dimensions
b_1	3	First dimension
b_2	4	Second dimension
b_3	5	Third dimension
b_4	2	Integer type
b_5	60	Number of elements = $3*4*5$

A code segment that checks that a variable, A, is two-dimensional, and extracts the dimensions is:

```
s = SIZE(A)
    ; Get size vector.
IF s(0) NE 2 THEN BEGIN
    ; Two-dimensional?
    PRINT, 'Variable A is not two-dimensional'
    ; Print error message.
    RETURN
    ; And exit.
ENDIF
nx = s(1) & ny = s(2)
    ; Get number of columns and rows.
```

Using Program Control Routines

The program control procedures are largely self-explanatory, with the exception of the EXECUTE function. The EXIT procedure exits the PV-WAVE session. STOP terminates execution of a program or batch file, and prints the values of its optional parameters. WAIT, as its name implies, pauses execution for a given amount of time, specified in seconds.

Executing One or More Statements

The EXECUTE function compiles and executes one or more PV-WAVE statements contained in its string parameter during run-time.

The result of the EXECUTE function is true (1), if the string was successfully compiled and executed. If an error occurred during either phase the result is false (0). If an error occurs, an error message is printed.

Use the & character to separate multiple statements in the string. GOTO statements and labels are not allowed.

Example of Executing Multiple Statements in a Single Command

This example, taken from the Standard Library routine SVDFIT, calls a function whose name is passed to SVDFIT as a string in a keyword parameter. If the keyword parameter is omitted, the function POLY is called:

```
FUNCTION SVDFIT, ..., Funct = Funct
    ; Function declaration.
...
IF N_ELEMENTS(Funct) EQ 0 THEN Funct = 'POLY'
    ; Use default name, POLY, for function if not specified.
z = EXECUTE('a = ' + Funct + '(x, m)')
    ; Make a string of the form "a = funct(x,m)", and execute it.
...
```


Tips for Efficient Programming

Techniques for writing efficient programs in PV-WAVE are identical to those in other computer languages, with the addition of the following simple guidelines:

- Use array operations rather than loops wherever possible. Try to avoid loops with high repetition counts.
- Use PV-WAVE system functions and procedures wherever possible.
- Access array data in machine-address order.

Attention must also be given to algorithm complexity and efficiency, as this is usually the greatest determinant of resources used.

NOTE In this chapter, we give the result of timing various examples. Such timings are influenced by many factors and may not be the same for your machine. However, all timings were made on the same machine under the same conditions. Therefore, the exact times may differ, but the timings given for the various examples can be used to evaluate the relative efficiency of the examples given.

Increasing Program Speed

The order in which an expression is evaluated can have a large effect on program speed. Consider the following statement, where A is an array:

```
B = A * 16. / MAX(A)
      ; Scale A from 0 to 16.
```

This statement first multiplies every element in A by 16, and then divides each element by the value of the maximum element. The number of operations required is twice the number of elements in A. This statement took 24 seconds to execute on a 512-by-512 single-precision floating-point array. A much faster way of computing the same result is:

```
B = A * (16. / MAX(A))  
    ; Scale A from 0 to 16 using only one array operation.
```

OR:

```
B = 16. / MAX(A) * A  
    ; Operators of equal priority are evaluated from left to right. Only one  
    ; array operation is required.
```

The faster method only performs one operation for each element in A, plus one scalar division. It took 14 seconds to execute on the same floating-point array as above.

Avoid IF Statements for Faster Operation

It pays to attempt to code as much as possible of each program in array expressions, avoiding scalars, loops, and IF statements. Some examples of slow and fast ways to achieve the same results are:

Example

Add all positive elements of B to A:

```
FOR I = 0, (N - 1) DO IF B(I) GT 0 THEN  
    A(I) = A(I) + B(I)  
    ; Slow way uses a loop.
```

```
A = A + (B GT 0) * B  
    ; Fast way: Mask out negative elements using array operations.
```

```
A = A + (B > 0)  
    ; Faster way: Add B > 0.
```

Often an IF statement appears in the middle of a loop, with each element of an array in the conditional. By using logical array expressions, the loop may sometimes be eliminated.

Example

Set each element of C to the square-root of A if A (I) is positive, otherwise, set C (I) to minus the square-root of A (I) :

```

FOR I = 0, (N - 1) DO IF A(I) LE 0 THEN
  C(I) = -SQRT(-A(I)) ELSE
  C(I) = SQRT(A(I))
      ; Slow way uses an IF statement.
C = ((A GT 0) * 2 - 1) * SQRT(ABS(A))
      ; Fast way.

```

For a 10,000-element floating-point vector, the statement using the IF took 8.5 seconds to execute, while the version using the array operation took only 0.7 seconds.

The expression (A GT 0) has the value of 1 if A(I) is positive and is 0 if A(I) is not. (A GT 0) * 2 - 1 is equal to +1 if A(I) is positive or minus 1 if A(I) is negative, accomplishing the desired result without resorting to loops or IF statements.

Another method is to use the WHERE function to determine the subscripts of the negative elements of A and negate the corresponding elements of the result:

```

NEGS = WHERE (A LT 0)
      ; Get subscripts of negative elements.
C = SQRT(ABS(A))
      ; Take root of absolute value.
C(NEGS) = -C(NEGS)
      ; Fix up negative elements.

```

This version took 0.8 seconds to process the same 10,000-element floating-point array.

Use Array Operations Whenever Possible

Whenever possible, vector and array data should always be processed with array operations instead of scalar operations in a loop. For example, consider the problem of inverting a 512-by-512 image. This problem arises because about half of the available image display devices consider the origin to be the lower-left corner of the screen, while the other half use the upper-left corner.

NOTE This example is for demonstration only. The system variable !Order should be used to control the origin of image devices. The *Order* keyword to the TV procedure serves the same purpose.

A programmer without experience in using PV-WAVE might be tempted to write the following nested loop structure to solve this problem:

```

FOR I = 0, 511 DO FOR J = 0, 255 DO BEGIN
    TEMP = IMAGE (I, J)
        ; Temporarily save pixel image.
    IMAGE(I, J) = IMAGE(I, 511 - J)
        ; Exchange pixel in same column from corresponding row at
        ; bottom.
    IMAGE(I, 511 - J) = TEMP
ENDFOR

```

Executing this code required 143 seconds.

A more efficient approach to this problem capitalizes on PV-WAVE's ability to process arrays as a single entity.

```

FOR J = 0, 255 DO BEGIN
    SW = 511 - J
        ; Index of corresponding row at bottom.
    TEMP = IMAGE(*, J)
        ; Temporarily save current row.
    IMAGE(0, J) = IMAGE(*, SW)
        ; Exchange row with corresponding row at bottom.
    IMAGE(0, SW) = TEMP
ENDFOR

```

Executing this revised code required 11 seconds, which is 13 times faster.

At the cost of using twice as much memory, things can be simplified even further:

```

IMAGE2 = BYTARR(512, 512)
    ; Get a second array to hold inverted copy.
FOR J = 0, 511 DO IMAGE2(0, J) = $
    IMAGE(*, 511 - J)
        ; Copy the rows from the bottom up.

```

This version also ran in 11 seconds.

Finally, using the built-in ROTATE function:

```

IMAGE = ROTATE(IMAGE, 7)
    ; Inverting the image is equivalent to transposing it and rotating it
    ; 270° clockwise.

```

This simple statement took 0.6 seconds to execute.

Use System Routines for Common Operations

PV-WAVE supplies a number of built-in functions and procedures to perform common operations. These system-supplied routines have been carefully optimized and are almost always much faster than writing an equivalent operation with loops and subscripting.

Example

A common operation is to find the sum of the elements in an array or subarray. The TOTAL function directly and efficiently evaluates this sum at least ten times faster than directly coding the sum:

```
SUM = 0. & FOR I = J, K DO SUM = SUM + ARRAY(I)
```

; Slow way: Initialize SUM and sum each element.

```
SUM = TOTAL(ARRAY(J : K))
```

; Efficient, simple way.

Using a 10,000-element floating-point vector and summing all of its elements took 4 seconds with the first statement and .09 seconds with the second.

Similar savings result when finding the minimum and maximum elements in an array (MIN and MAX functions), sorting (SORT function), finding zero or non-zero elements (WHERE function), etc.

Use Constants of the Correct Type

As explained in [Chapter 2, *Constants and Variables*](#), the syntax of a constant determines its type. Efficiency is adversely affected when the type of a constant must be converted during expression evaluation. Consider the following expression:

```
A + 5
```

If the variable A is of floating-point type, the constant 5 must be converted from short integer type to floating-point type each time the expression is evaluated.

The type of a constant also has an important effect in array expressions. Care must be taken to write constants of the correct type. In particular, when you are performing arithmetic on byte arrays and want to obtain byte results, be sure to use byte constants (e.g., nB). For example, if A is a byte array, the result of the expression A + 5B is a byte array, while A + 5 yields a 16-bit integer array.

Remove Invariant Expressions from Loops

Expressions whose values do not change in a loop should be moved outside the loop. In the following loop:

```
FOR I = 0, N - 1 DO ARR(I, 2 * J - 1) = ...
```

the expression $(2 * J - 1)$ is invariant and should be evaluated only once before the loop is entered:

```
TEMP = 2 * J - 1
FOR I = 0, N - 1 DO ARR(I, TEMP) = ...
```

Access Large Arrays by Memory Order

When an array's size is larger than or near the working set size, it should always, if possible, be accessed in memory-address order.

To illustrate some side-effects of the virtual memory environment, consider the process of transposing a large array. Assume the array is a 512-by-512 byte image and there is a 100-kilobyte working set. The array requires 512 x 512 or approximately 250 kilobytes. Clearly, less than half of the image may be in memory at any one instant.

In the transpose operation, each row must be interchanged with the corresponding column. The first row, containing the first 512 bytes of the image, will be read into memory, if necessary, and written to the first column.

Because arrays are stored in row order (the first subscript varies the fastest), one column of the image spans a range of addresses almost equal to the size of the entire image. In order to write the first column, 250,000 bytes of data must be read into physical memory, updated, and written back to the disk. This process must be repeated for each column, requiring the entire array be read and written almost 512 times!

The time required to transpose the array using the above naive method will be on the order of minutes. The TRANSPOSE function transposes large arrays by dividing them into subarrays smaller than the working set size and will transpose a 512-by-512 image in less than 10 seconds.

Example

Consider the operation of the statement:

```
FOR X = 0, 511 DO FOR Y = 0, 511
DO ARR(X, Y) = ...
```

This statement will require an extremely large amount of time to execute because the entire array must be transferred between memory and the disk 512 times. The proper form of the statement is to process the points in address order:

```
FOR Y = 0, 511 DO FOR X = 0, 511
    DO ARR(X, Y) = ...
```

The time savings are at least a factor of 50 for this example.

Be Aware of Virtual Memory

The PV-WAVE programmer and user must be aware of the characteristics of virtual memory computer systems to avoid penalty. Virtual memory allows the computer to execute programs that require more memory than is actually present in the machine by keeping those portions of programs and data that are not being used on the disk. Although this process is transparent to the user, it can greatly affect the efficiency of the program.

Arrays are stored in dynamically allocated memory. Although the program can address large amounts of data, only a small portion of that data actually resides in physical memory at any given moment—the remainder is stored on disk. The portion of data and program code in real physical memory is commonly called the working set.

When an attempt is made to access a datum in virtual memory that does not currently reside in physical memory, the operating system suspends PV-WAVE, arranges for the page of memory containing the datum to be moved into physical memory, and then allows PV-WAVE to continue. This process involves deciding where in memory the datum should go, writing the current contents of the selected memory page out to the disk, and reading the page containing the datum into the selected memory page. A page fault is said to occur each time this process takes place. Because the time required to read from or write to the disk is very large in relation to the physical memory access time, page faults become an important consideration.

When using PV-WAVE with large arrays it is important to have a generous amount of physical memory and a large swapping area. If you suspect that these parameters are causing problems, consult your system manager.

Running Out of Virtual Memory?

Whenever you define a variable or perform an operation, PV-WAVE asks the operating system for some virtual memory in which to store the data or operation. (Internally, PV-WAVE calls the C function `malloc` to allocate the additional memory.) With each additional definition or operation, the amount of memory allocated to the PV-WAVE process grows. If you typically process large arrays of data and use the vendor-supplied default system parameters, sooner or later the following error will occur:

```
% Unable to allocate memory
```

This error message means that PV-WAVE was unable to obtain from the operating system enough virtual memory to hold all of your data. In general, this situation arises because of the way in which all C applications interact with the operating system. That is, allocated memory that is freed (via a call to the C routine `free`) results in a fragmented or discontinuous pool of memory within the application.

You have two basic options to resolve this error:

- First, you can try deleting all unneeded variables, functions, procedures, and structures. This option may be effective in many cases; however, it will not be effective in all cases. Because of the memory fragmentation described previously, it is not always possible to free sufficient space for a large variable or routine by deleting other smaller variables or routines. PV-WAVE requires a chunk of *contiguous* memory large enough to hold any given array or routine.
- If the first option does not work, you will have to exit PV-WAVE. Before exiting, use the `SAVE` procedure to save only the variables and routines that you need. When you restore the session with the `RESTORE` procedure, the saved variables and routines will be stored in memory in a less fragmented manner, which may create sufficient space for you to continue your work. If PV-WAVE still cannot allocate enough memory for your data, you can try exiting without first saving the session.

To delete structures, procedures, and functions, use the `DELSTRUCT`, `DELPROC`, and `DELFUNC` procedures. Use the `DELVAR` procedure to delete variables. For information on these procedures, see the PV-WAVE Reference. Another method of freeing memory is to assign the value of a large array variable to a scalar value.

TIP The `INFO, /Memory` procedure will tell you how much virtual memory you have allocated. For example, a 512-by-512 complex floating array requires 8×512^2 or about 2 megabytes of virtual memory because each complex element requires 8 bytes.

NOTE Again, with the deletion and reassignment of large variables (as well as structure definitions, compiled procedures, and functions), the memory available to PV-WAVE processes will become fragmented. Eventually, you will not be able to obtain sufficient memory for a given large variable. At this point, you can try deleting unneeded variables, procedures, functions, and structures. If that does not solve the problem, you must exit from PV-WAVE to clear out the memory.

Controlling Virtual Memory System Parameters under UNIX

The size of the swapping area(s) determines how much virtual memory your process is allowed. To increase the amount of available virtual memory, you must either increase the size of the swap device (sometimes called the swap partition), or use the `swapon (8)` command to add additional swap areas. Increasing the size of a swap partition is a time consuming task which should be planned carefully. It requires saving the contents of the disk, reformatting the disk with the new file partition sizes, and restoring the original contents. Consult the documentation that came with your system for details. Some systems (SunOS) allow you to swap to a normal file by using the `mkfile (8)` command in conjunction with `swapon`. This is a considerably easier solution.

Controlling Virtual Memory System Parameters under OpenVMS

VMS, as it comes from Digital, is not tuned for image processing. To get the best performance from PV-WAVE, you should increase the OpenVMS `SYSGEN` parameters, file sizes, and `AUTHORIZE` quotas which restrict the virtual memory system. This discussion is on the most elementary level and the appropriate OpenVMS manuals should be consulted for more detail.

The first step is to determine how much virtual memory you require.

For example, if you do complex FFTs on 512-by-512 images, each complex image requires 2 megabytes. Suppose that during a typical session you need to have four images stored in variables, and require enough memory for two images to hold temporary results, resulting in a total of six images or 12 megabytes. Rounding up to 16 megabytes gives a reasonable goal. The following parameters and quotas should be changed to increase the amount of virtual memory available:

SYSGEN Parameters

- **WSMAX** — Sets the maximum number of pages of any working set on a system-wide basis. The working set is that portion of virtual memory used by a process that is actually in physical memory. Although this is an over-simplification, small working set sizes cause page faulting. Page faults waste time and potentially require disk accesses. Increasing the working set to a size of three times the size of the largest array to be processed, or at least 2000 blocks, can cause dramatic speed improvements. Many MicroVAX systems have from 8 to 16 megabytes of physical memory. Subtracting approximately 2 megabytes for OpenVMS leaves the rest available to be divided up among the user processes. On many MicroVAX systems, there are only one or two users, so large working sets of from 3 to 7 megabytes (6000 to 14000 pages) may be used.
- **VIRTUALPAGECNT** — This parameter sets the maximum number of virtual pages (512 bytes/page) that can be used by any one process.

To change the values of *SYSGEN* parameters, Digital recommends that you run the *AUTOGEN* command procedure after adding lines to set the new values of changed parameters to the end of the file *SYS\$SYSTEM:MODPARAMS.DAT*.

System Files

The sizes of the system page and swap files (*SYS\$SYSTEM: PAGEFILE.SYS* and *SWAPFILE.SYS*) must be large enough to contain the virtual memory used by all active processes. In any event, you cannot have more virtual memory than will fit in the page file.

You can increase the size of these files or create secondary system files on a disk other than the system disk.

If you get the error message:

```
Page file fragmented - continuing
```

on the system console your page file is too small.

To increase the size of these files, use the command procedure *SYS\$UPDATE:SWAPFILES*. Use the *SYSGEN INSTALL* command to activate system files created on disks other than the system disk. *AUTOGEN* may also be used to change the file sizes.

Quotas

The following quotas, all of which may be changed on a per user or system basis using the *AUTHORIZE* utility, affect virtual page limits and working set sizes:

- PGFLQUO — The page file quota for each user expressed in blocks. If you increase the size of the page file, be sure to increase the page file quotas for the users requiring more virtual memory. Be sure that the page file size is at least as large as the sum of the quotas of each active user.
- WSQUO — The working set quota for each user. This quota may be used to allow some users a larger working set than others. WSQUO must not be larger than WSMAX.

Minimize the Virtual Memory Used

If virtual memory is a problem, try to tailor your programming to minimize the number of images held in variables.

Keep in mind that PV-WAVE creates temporary arrays to evaluate expressions involving arrays. For example, when evaluating the statement:

```
A = (B + C) * (E + F)
```

PV-WAVE first evaluates the expression $B + C$, and creates a temporary array if either B or C are arrays. In the same manner, another temporary array is created if either E or F are arrays. Finally, the result is computed, the previous contents of A are deleted and the temporary area holding the result is saved as variable A . Note that during the evaluation of this statement enough virtual memory to hold two array's worth of data is required in addition to normal variable storage.

It is a good idea to delete the allocation of a variable that contains an image and that appears on the left side of an assignment statement. For example, in the program:

```
FOR I = ... DO BEGIN
    ; Loop to process an image.
    ...
    ; Processing steps.
    A = 0
    ; Delete old allocation for A.
    A = Image Expression
    ; Compute image expression and store.
    ...
ENDFOR
```

the purpose of the statement $A = 0$ is to free the old memory allocation for the variable A before computing the image expression in the next statement. Because the old value of A is going to be wiped out in the next statement, it makes sense to free A 's memory allocation before executing the next statement. For more information on the effects of freeing memory by deleting or reassigning large array variables, see [Running Out of Virtual Memory? on page 262](#).

Array Operations are Rewarded

PV-WAVE programs are compiled into a low-level abstract machine code, which is interpretively executed. The dynamic nature of variables in PV-WAVE and the relative complexity of the operators precludes the use of directly executable code. Statements are only compiled once, regardless of the frequency of their execution.

The PV-WAVE interpreter emulates a simple stack machine with approximately 50 operation codes. When performing an operation, the interpreter must determine the type and structure of each operand, and branch to the appropriate routine. The time required to properly dispatch each operation may be longer than the time required for the operation itself.

Array-array and array-scalar operations are implemented by generating and executing optimized machine code in a temporary buffer. The characteristics of the time required for array operations is similar to that of vector computers and array processors. There is an initial set-up time, followed by rapid evaluation of the operation for each element. The time required per element is shorter in longer arrays because the cost of this initial set-up period is spread over more elements.

The speed of PV-WAVE is comparable to that of optimized FORTRAN insofar as array operations are considered. When data are treated as scalars, efficiency degrades by a factor of 30 or more.

As an example, the processes of evaluating the square-root of a 10,000-element floating-point vector and adding 2 to each element of a 512-by-512 byte image was timed using scalar operations, array operations, and FORTRAN. The times for these operations are shown in the following table:

Processing Times

Method Used	Square-Root Time	Addition Time
PV-WAVE with scalars and FOR statement	3.10	138
PV-WAVE with array operation	0.16	0.49
Sun FORTRAN (optimized)	0.11	0.80

As can be seen above, there is a large penalty for using scalar operations when array operations are appropriate. There is little difference in the times required by PV-WAVE array operations and FORTRAN.

Getting Session Information

Using the INFO Procedure

The INFO procedure provides you with information about many different aspects of the current PV-WAVE session. Entering

```
INFO
```

with no parameters prints an overview of the current state, including the definitions of all current variables. Calling INFO with one or more parameters displays the definitions of the parameters. INFO also displays other information about the current session if you call it with a keyword parameter indicating the topic. Only one topic keyword can be specified at a time. The available topics are described in the following sections.

Calling INFO with No Parameters

When INFO is called without any positional or keyword parameters, it provides an overview of the current state. The information provided is:

- A traceback showing the current procedure and function nesting.
- Amount of code area, number of local variables, and number of parameters. (When PV-WAVE reads a procedure or function for the first time, it compiles it into executable code. Every routine has a code area where the executable code is placed and a data area where information about all locally available variables (including common block variables) resides. The amount of room

used in each of these areas is reported to the current routine, along with the number of local variables and parameters.)

- A one-line description of every current variable.
- A description of all currently accessible common blocks.
- The names of all saved procedures and functions.

As an example of a typical PV-WAVE session, the command

```
INFO
```

might result in output similar to:

```
% At $MAIN$(0).
Code area used: 0.00% (0 / 30000), Data area used: 4.88% (100 / 8000)
# local variables (including 0 parameters: 4/250)
# common symbols: 3/8
B  BYTE = Array(256)
G  BYTE = Array(256)
I  BYTE = Array(512, 512)
R  BYTE = Array(256)
X(CBLK) INT = 0
Y(CBLK) INT = 11
Z(CBLK) INT = 12
Common Blocks:
  CBLK(3)
Saved Procedures:
  COLOR_EDIT COLOR_EDIT_BACK INTERP_COLORS READ_SRF SHOW3
Saved Functions:
  AVG BILINEAR CORRELATE CURVEFIT
```

This session summary provides the following information:

- The current routine is \$MAIN\$, meaning that you are currently at the main program level and that no called routine is executing.
- The second and third lines indicate that the code area is empty (zero bytes used out of 30,000 available) and approximately 95% of the data area is also free (100 bytes used out of 8,000 available). The code area is empty because you are currently at the \$MAIN\$ level and no \$MAIN\$ program has been entered.
- The fourth line shows how many local variables the current data area can accommodate. It also indicates the total number of local variables including the number of parameters. In this example, 4/250 means that there is space for a total of 250 local variables and only four are currently being used.

- The sixth line shows how many common block symbols are being used and how many there are space for.
- The next seven lines give one-line descriptions of all locally available variables. The first four variables (B, G, I, and R) are local variables, while the other three (X, Y, and Z) are contained in the common block CBLK. Note that the one-line descriptions of scalar variables gives their values, while the descriptions of arrays shows their dimensions. Use the PRINT procedure to look at the contents of arrays.
- Following the descriptions of variables is the list of available common blocks. In this session, the only common block is named CBLK, and it contains three variables.
- The final information printed is the names of all saved procedures and functions.

Calling INFO with Positional Parameters

If you call INFO with parameters (but without any keyword parameters), it simply provides a one-line description of each parameter. Hence, for the PV-WAVE session described earlier, the command:

```
INFO, 12.0 * 23, R, I, Z, !D
```

gives the output:

```
<Expression> FLOAT = 276.000
R BYTE = Array(256)
I BYTE = Array(512, 512)
Z(CBLK) INT = 12
<Expression> STRUCT = -> !Device
```

As noted earlier, the one-line description of scalars prints their values, while for arrays, you see their dimensions. For structure variables, the name of the structure definition associated with the variable is printed, as shown in the last line of this example. Use INFO with the *Structures*, *Sysstruct*, or *Userstruct* keywords to see the form of a structure variable. These keywords are described later in this chapter.

Calling INFO with Keyword Parameters

INFO, /Device

The command:

```
INFO, /Device
```

gives information about the current graphics device. This information depends on the abilities of the current device, but the name of the device is always given. Other parameters to INFO are ignored when *Device* is selected. As an example of the type of information supplied, the commands:

```
SET_PLOT, 'PS'  
; Select PostScript output.
```

```
INFO, /Device  
; Get device information.
```

yield:

```
Current graphics device: PS  
File: <none>  
Mode: Portrait, Non-Encapsulated  
Offset (X, Y): (1.905,12.7) cm.  
Size (X, Y): (17.78,12.7) cm.  
Scale Factor: 1  
Font Size: 12  
Font: Helvetica  
# bits per image pixel: 4
```

INFO, /Files

The *Files* keyword provides information about file units. If no parameters are supplied, information on all open file units is displayed. If parameters are provided, they are assumed to be integer file unit numbers, and information on the specified file units is given. For example, the command:

```
INFO, -2, -1, 0, /Files
```

gives information about the default file units. For example, under UNIX, the output might look like this:

Unit	Attributes	Name
-2	Write, Truncate, Tty, Reserved	<stderr>
-1	Write, Truncate, Tty, Reserved	<stdout>
0	Read, Tty, Reserved	<stdin>

The attributes column tells about the characteristics of the file. For instance, the file connected to logical file unit -2 is called `stderr`, and is the standard error file. It is opened for write access (`Write`), is a new file (`Truncate`), is a terminal (`Tty`), and cannot be closed via the `CLOSE` command (`Reserved`).

INFO, /Keys

The *Keys* keyword provides current function key definitions, as set with the `DEFINE_KEY` procedure. For information on `DEFINE_KEY`, see the PV-WAVE Reference.

If no parameters are supplied, information on all function keys is displayed. If parameters are provided, they must be scalar strings containing the names of function keys, and information on the specified keys is given.

For example, you can define the <F12> key to execute the command `INFO, /Keys` with the statement:

```
DEFINE_KEY, /Terminate, 'F12', 'INFO, /Keys'
```

the `INFO, /Keys` command produces output that includes the line:

```
F12 <\033[P> = INFO, /Keys <Terminate>
```

showing the new key definition.

Parameters to `INFO` are ignored when *Keys* is selected.

INFO, /Memory

PV-WAVE uses dynamic (heap) memory to store items such as programs and variables. The *Memory* keyword reports the amount of dynamic memory currently in use by the PV-WAVE session, and the number of times dynamic memory has been allocated and deallocated. A typical response to the

```
INFO, /Memory
```

command might look like:

```
heap memory in use: 14572, calls to MALLOC: 64, FREE: 3
```

Other parameters to `INFO` are ignored when *Memory* is selected.

INFO, /Recall_Commands

PV-WAVE saves the last 20 lines of input in a buffer. These lines can be recalled for command line editing. The *Recall_Commands* keyword causes the `INFO` procedure to display the contents of this buffer. Other parameters to `INFO` are ignored when *Recall_Commands* is selected.

UNIX and OpenVMS USERS For more information on reviewing and re-entering previously entered commands, see *Getting Started: UNIX and OpenVMS* in the PV-WAVE User's Guide.

Windows USERS For more information on reviewing and re-entering previously entered commands, see *Getting Started: Windows* in the PV-WAVE User's Guide.

INFO, /Routines

The *Routines* keyword causes INFO to print a list of all compiled procedures and functions with their parameter names. Keyword parameters accepted by each module are enclosed in quotation marks. Other parameters to INFO are ignored when *Routines* is selected. For the typical session described earlier, the result of the command

```
INFO, /Routines
```

would appear as:

```
Saved Procedures:
```

```
  COLOR_EDIT          "HLS" "HSV"
  COLOR_EDIT_BACK
  INTERP_COLORS       pts npts colors
  READ_SRF            file image r g b
  SHOW3               image "INTERP"
```

```
Saved Functions:
```

```
  AVG                 array dimension
  BILINEAR            p ix jy
  CORRELATE           x y
  CURVEFIT            x y w a sigmaa
```

INFO, /Structures

The *Structures* keyword provides information about structure variables. If no parameters are provided, all currently defined structures are shown. If parameters are provided, the structure of those variables are displayed. For example, the command

```
INFO, /Structures, !D
```

shows the contents and structure of the system variable !D:

```
** Structure !Device, 14 tags, 60 length:
```

```
NAME          STRING      'X'
  X_SIZE      LONG        640
  Y_SIZE      LONG        512
  X_VSIZE     LONG        640
  Y_VSIZE     LONG        512
  X_CH_SIZE   LONG         6
  Y_CH_SIZE   LONG         9
```

X_PX_CM	FLOAT	40.0000
Y_PX_CM	FLOAT	40.0000
N_COLORS	LONG	256
TABLE_SIZE	LONG	256
FILL_DIST	LONG	1
WINDOW	LONG	-1
UNIT	LONG	0
FLAGS	LONG	444
ORIGIN	LONG	Array(2)
ZOOM	LONG	Array(2)

TIP It is often more convenient to use `INFO, /Structures` instead of `PRINT` to look at the contents of a structure variable because it shows the names of the fields as well as the data. For instance, the command:

```
PRINT, !D
```

gives the output:

```
{X 640 512 640 512 6 9 40.0000 40.0000 256 256 1
   -1 0 444 0 0 1 1}
```

which is less readable.

See also the *Sysstruct* and *Userstruct* keywords described later.

INFO, /System_Variables

The *System_Variables* keyword causes `INFO` to show all system variables and their values. Other parameters to `INFO` are ignored when the *System_Variables* keyword is selected. The command:

```
INFO, /System_Variables
```

displays the current values of system variables.

INFO, /Sysstruct

The *Sysstruct* keyword displays only the system structures (structures that begin with “!”). The output of this command is a subset of the `INFO, /Structures` command described previously.

INFO, /Traceback

The *Traceback* keyword displays the current nesting of procedures and functions. Other parameters to `INFO` are ignored when *Traceback* is selected.

INFO, /Userstruct

The *Userstruct* keyword displays only the regular user-defined structures (structures that do not begin with “!”). The output of this command is a subset of the `INFO, /Structures` command described previously.

Using the PV-WAVE Debugger

The PV-WAVE Debugger is a development environment for creating, testing, and maintaining VDA applications written in PV-WAVE. With easy-to-use mouse and menu driven functions, the Debugger helps you to become a more productive PV-WAVE application developer. With the Debugger you can:

- Edit source files using a built-in editor or an editor of your choice, such as *emacs* or *vi*.
- Copy, cut, paste, select, and search for text.
- Run an application, step through it line by line, skip lines, or stop execution.
- Set breakpoints and examine variable contents during program execution.
- List information about system variables, structure definitions, open files, and compiled routines.
- Print source code files.

This section is an introduction to the Debugger and provides enough information to get you started loading and debugging your application programs. Additional information on the functions discussed here, as well as other functions not discussed in this section, is available through the Debugger's context sensitive online help system.

The Main PV-WAVE Debugger Window

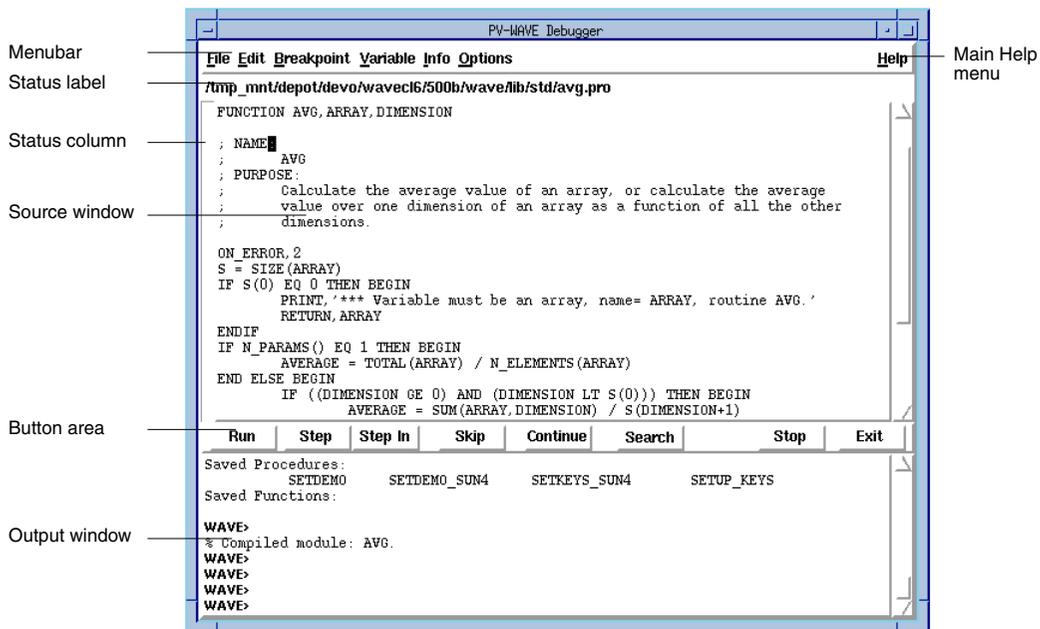


Figure 13-1 The main window appears when you start the Debugger.

- **Menu bar** — Contains menu functions that you can select during a debugging session.
- **Status label** — Displays the currently loaded source file and, during program execution, displays the execution line number and status.
- **Status column** — Shows which lines have breakpoints set.
- **Source window** — A full-featured editing window used to display and edit source files.
- **Button area** — Lets you control program execution.
- **Output window** — The PV-WAVE command line: echoes PV-WAVE commands being executed; displays error, informational, and user input messages; lets you enter PV-WAVE commands.

Using the Debugger's Online Help System

Online help displays information about the Debugger's menu items, dialog boxes, and buttons. Start with the main **Help** menu; it provides information on the menus in the Menu bar, the program execution buttons, and other topics. Each dialog box contains a **Help** button that you can click on to display context sensitive information.

Starting the Debugger

To start the Debugger on UNIX, enter the following command at the operating system prompt:

```
(UNIX)    wavedbg
```

To start the Debugger on Windows, go to:

Start>Programs>PV-WAVE 7.x>PV-WAVE Debugger

where x is the current revision level of PV-WAVE.

After a few moments, the Debugger main window appears. The Source window is empty until you load an application or source file. In the Output window, you will see the normal PV-WAVE startup messages.

Changing the Working Directory

By default, the Debugger recognizes the working directory as the directory from which it was started. To change the working directory, use the **File=>Set Directory** function. This function brings up a File Selection dialog box in which you can specify the new directory path.

Loading an Application at Startup

You can also open an application and one or more source files directly when you start the Debugger by specifying the file name(s) as a parameter to the `wavedbg` command. For example:

```
wavedbg appl source1 source2 source3
```

where *appl* is the name of the main application file and *source1*, *source2*, and *source3* are source files for functions called by the application. The difference between application and source files is discussed further in [Loading Files into the Debugger on page 278](#).

When the Debugger window opens, the specified source file appears in the Source window and is automatically compiled. (Whenever a source file is loaded into the Debugger, it is automatically compiled.)

Executing a Command File at Startup

To execute a PV-WAVE command file (batch file) when you start the Debugger, specify the command file as a parameter to `wavedbg`. You must precede the command filename with the `@` symbol. For example:

```
wavedbg @comfile
```

The command file is automatically executed when the Debugger starts.

Saving Your Work and Stopping the Debugger

Before stopping the Debugger, save your work by selecting the **File=>Save** or **File=>Save As** function. To stop the Debugger, select **File=>Quit**.

The **File=>Save As** function lets you specify a name for the source file before saving it. Use this for new, previously unnamed source files or for a file whose name you want to change.

The **File=>Save** function simply saves the currently loaded source file using its existing filename.

Loading Files into the Debugger

Your PV-WAVE application may consist entirely of one source file, or it may consist of multiple files. In the case of a multiple file application, one file is usually the main application file — the one you call initially to start the application and that calls functions in other files.

Loading a Single-File Application

If your application consists entirely of one file, then you can load it into the Debugger using **File=>Application**. Enter the name of the file in the File Selection dialog box, and it is loaded into the Source window and compiled. At this point, you can run (by clicking the **Run** button) and debug the program.

TIP Most of the menu functions in the Debugger have keyboard accelerators (short cuts) associated with them. Whenever an accelerator exists for a menu item, it is listed on the menu to the right of the menu item name. For example, the accelerator for the **File=>Application** function (the **Application** function on the **File** menu) is **C-x C-a**. This means that you can select this function by holding down the `<Control>` key and pressing the `<x>` and then the `<a>` key.

Loading a Multi-File Application

If you have an application that is broken into several source files (see [Figure 13-2](#)), load the *main application file* using the **File=>Application** function. The file is loaded into the Debugger Source window and compiled. At this point, you can run (by clicking the **Run** button) and debug the program.

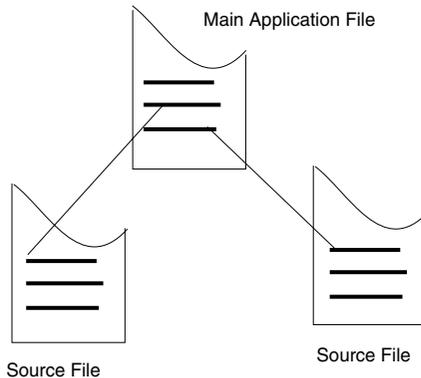


Figure 13-2 An application that consists of a main application file and multiple separate source files can be loaded into the Debugger.

While debugging the application, you may need to open one or more of the application's separate source files. To load one of these files, use the **File=>Source** function. The source file that you specify is loaded into the Source window where you can edit it.

TIP Another way to open source files is to use the **File=>Find Source** function. Type the name of a function in the Find Source dialog box, and the Debugger searches for the function's source file in the current directory and in the directories specified in the !Path system variable. If the function is located, it is displayed in the Source window. Another way to use this function is to hold down the <Shift> and <Control> keys and double-click on the name of a function in the Source window.

If you click the **Run** button, the Debugger will always try to execute the main application file — the file you loaded with **File=>Application**, regardless of which file is currently loaded in the Source window. You do not have to explicitly reload the main application file to execute it.

Running an Application

When you load a program into the debugger, it is compiled automatically, but not executed. To execute the program (either the main program of a multifile application or a single-file program), click the **Run** button.

To specify parameters and keywords for the application, enter them with the **File=>Parameters** function before clicking **Run**.

NOTE The Debugger does not have a function that is equivalent to the .RUN command. A program is automatically compiled whenever it is loaded into the debugger. To force a program to be recompiled, select the **File=>Reload** function.

Detecting Execution Errors

Execution error messages appear in the Output window. The line number of the error is reported in the error message.

Use the function **Edit=>Go To Line** to go directly to the line number of the error in the Source window.

If the error is easily corrected, you can do so directly in the Source window. Simply make the change, recompile the main program by selecting **File=>Reload** or **File=>Save**, and then click **Run**. (See the next section *Editing the Source File* for more information.)

If the error is not obvious, while execution is stopped you can set breakpoints and examine variables to help track down the source of the error (see [Setting Breakpoints on page 281](#) and [Examining Variables on page 283](#) for more information).

Editing the Source File

You can edit source code directly in the Source window, or you can edit files using the text editor of your choice, such as *emacs* or *vi*.

Editing in the Source Window

You can choose either basic or *emacs* key bindings for the Source window. Key bindings, or “mappings”, define the functions performed by keys on your keyboard.

The basic key bindings are limited, but easy to learn. They are useful if you intend to make simple changes in the Source window. The *emacs* bindings give you the editing power of the standard *emacs* editor, which is found on most UNIX systems.

To choose the type of key bindings used in the Source window, select **Options=>Key Bindings** and choose either **Basic** or **Emacs**.

For a complete description of the basic and *emacs* key bindings, refer to online help by selecting **Help=>Key Bindings**.

NOTE After any text in the Source window, the word “edited” appears in the Status label above the Source window. This is done simply to remind you that the file has been modified.

Using a Separate Text Editor

If you do not want to use the Source window to edit source files, you can bring up a separate text editor, such as *vi* or *emacs*.

First, choose the text editor you want to use by selecting **Options=>Editor** and type the name of the text editor in the dialog box. Then, to run the editor, select **File=>Edit**. The text editor you selected appears in a separate window, and the currently loaded application or source file is automatically loaded into the editor.

TIP You can specify the default editor by setting the EDITOR environment variable. For example:

```
setenv EDITOR emacs
```

After you exit the separate editor, you are returned to the Debugger. To load the edited source file into the Debugger, select **File=>Reload**.

Setting Breakpoints

A breakpoint stops program execution at a preselected line number, allowing you to check on the status of variables or other program elements. With the PV-WAVE

Debugger, it is easy to set breakpoints in your source code. Just click on the line where you want to insert a breakpoint and select **Breakpoint=>Set Break** from the main menu. Then click **OK** in the Set Breakpoint dialog box. A small icon ([Figure 13-3](#)) appears in the Status column just to the left of the breakpoint line:

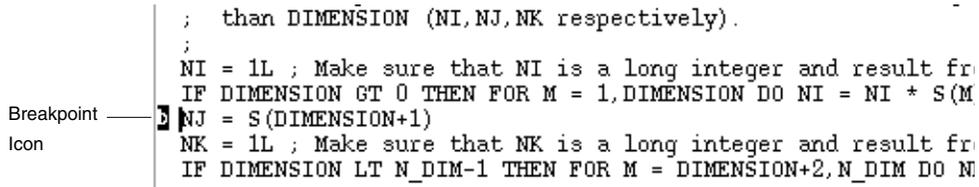


Figure 13-3 The breakpoint icon indicates the lines for which breakpoints are set.

The next time you run the program (e.g., by clicking the **Run** button), it will halt at the breakpoint line.

TIP Keyboard accelerators make setting and unsetting breakpoints easy. To set a breakpoint, press the <Shift> key and double-click on a line of code. To clear a breakpoint, press the <Control> key and double click on a line containing a breakpoint.

Other functions on the **Breakpoint** menu include:

Show Break — Lists each line of source code containing a breakpoint in the Output window.

Clear Break — Clears the breakpoint from a specified line.

Clear All — Clears all breakpoints in the Source window.

TIP While execution is halted, use the functions on the **Variable** and **Info** menus to examine the contents of variables, structures, and other program elements. (See [Examining Variables on page 283](#).) To continue execution, use one of the Button area buttons. For example, use the **Step** button to execute one line at a time or the **Continue** button to execute the rest of the program. These functions are discussed further in the next section.

Controlling Program Execution

Use the functions in the Button area to control program execution. Most of these functions have underlying PV-WAVE commands that are straightforward to understand. The following list gives a brief description of each button.

Run — Executes an application that has been loaded into the Debugger. First, the Debugger tries to find a procedure or function to execute that has the same name as the currently loaded application name. If none is found, the Debugger checks to see if a main program is currently loaded in memory. If a main program exists in memory, it is executed. If not, you are given a chance to enter the name of an application to execute.

Step — Executes the currently loaded program one line at a time. When **Step** encounters a line with a procedure or function call, it executes the procedure or function, but does not enter it. That is, the source code for the called function is not displayed in the Source window. To enter procedures and functions, use **Step In**.

Step In — Executes the currently loaded program one line at a time. When **Step In** encounters a line with a procedure or function call, it displays the procedure or function's source code in the Source window. At the end of the procedure or function, the Debugger redisplay the calling routine.

Skip — Skips the next line in the program, then single-steps after that. This command is useful for skipping over program statements that caused an error.

Continue — Continues the execution of a program that has stopped because of an error, a **Stop** command, or other interruption, such as a breakpoint.

Search — Lets you search for a text string in the Source window or the Output window.

Stop — Stops the execution of an application and returns from nested procedures and functions until the main program level is reached.

Exit — Stops the execution of the current application and ends the PV-WAVE session. To restart the PV-WAVE session, select **File=>Restart**. This command does not exit the Debugger, it only exits the PV-WAVE session running in the Debugger. To exit the Debugger, select **File=>Quit**.

Examining Variables

When you are debugging a program, it is often necessary to examine the contents of variables at various points while the program is running. The Debugger provides

several methods you can use to examine variables. The methods include showing a single variable, monitoring a variable, and “listing” information about variables.

Showing a Single Variable

Whenever program execution is stopped (e.g., by a breakpoint, a **Step** command, or an error), you can examine the contents of variables.

To examine the contents of a variable when program execution is stopped, double click on the variable’s name (in the Source window) and select **Variable=>Show**. Alternatively, you can select **Variable=>Show** first, and then enter the name of a variable in the dialog box.

All the elements of structures and up to 5000 array elements are displayed. You can control the array elements displayed by entering an array sub-expression in the **Variable=> Show** dialog box.

Monitoring a Variable

Another way to examine variables is with the **Variable=>Monitor** function. This method uses a window called the Monitor window in which the values of selected variables are displayed. Whenever program execution stops (e.g., with a breakpoint, or after a **Step** or **Skip**), these values are updated.

To monitor a variable, double click on the variable’s name and select **Variable=>Monitor**. Alternatively, you can select **Variable=>Monitor** first, and then enter the name of a variable in the dialog box.

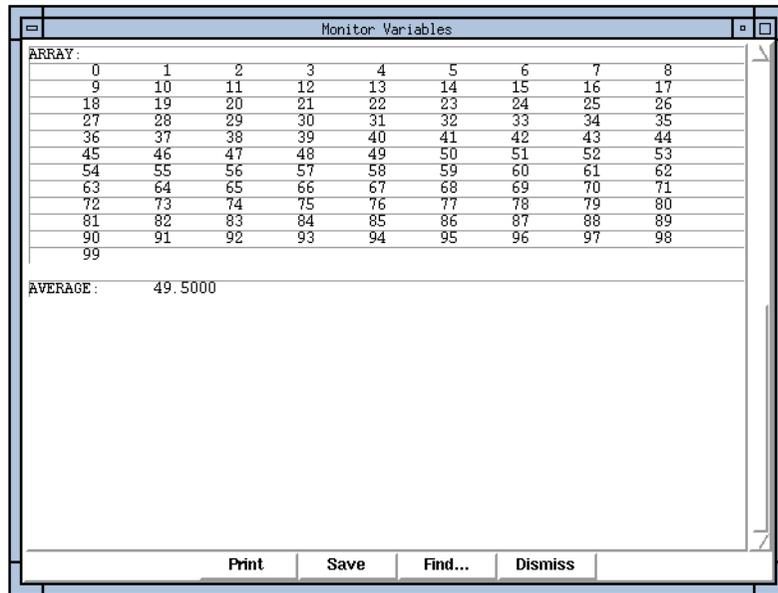


Figure 13-4 The Monitor window displays user variables.

The way in which variables in the Monitor window are displayed (i.e., using highlighting and relief), depends on the status of the variable, as follows:

- Variables that have changed since the last program interruption and are defined within the current procedure are highlighted and shown with sunken relief (i.e., they appear pushed in).
- Variables that are defined within the current procedure that have not changed are shown with sunken relief.
- Variables that are undefined in the current procedure are not highlighted.

All the elements of structures and up to 5000 array elements are displayed in the Monitor window. You can control the array elements displayed by entering sub-array expressions in the **Variable=>Monitor** dialog box.

Listing Variables and Structures

Several additional functions on the **Variable** menu let you list all user-defined and system variables, and structures. These functions work exactly like the PV-WAVE

INFO command when called with its particular keywords. For example, the **Variables=>List Structures** function works just like the PV-WAVE command:

```
INFO, /Structures
```

The output from these commands appears in the Output window. The INFO command is discussed in the PV-WAVE Reference.

Obtaining Session Information

The functions on the **Info** menu let you display information on the current Debugger session.

The functions on this menu work exactly like the PV-WAVE INFO command when called with its particular keywords. For example, the **Info=>Files** function works just like the PV-WAVE command:

```
INFO, /Files
```

The requested information appears in the Output window. The INFO command is discussed in the PV-WAVE Reference.

Customizing the Debugger

Use the functions on the **Options** menu to customize appearance of the Debugger window, how Debugger output is displayed, the editor you prefer to use, and key bindings. You can control the amount of text that the “undo buffer” holds, which affects how many commands can be “undone” and “redone” with the **Edit=>Undo** and **Edit=>Redo** commands.

For detailed information on each of the functions on the **Option** menu, refer to their description in the Debugger’s online help system.

Creating an OPI Option

Introduction

This chapter is for developers who want to create optional modules that can be loaded explicitly by any PV-WAVE user. These optional modules can be written in C or FORTRAN, and can contain new system functions or other primitives.

NOTE FORTRAN connectivity is not available for Windows.

The primary goals of the Option Programming Interface (OPI) are:

Release Independence	Options can be released independently of PV-WAVE.
Extensibility	New Options do not require changes to be made to the PV-WAVE kernel.
Centralized Licensing	Calls to the license manager are transparent to the Option developer and are centralized.
Option Manageability	The user can easily configure an Option, load it, unload it, and manage it.
Performance	Performance of routines developed with OPI compares well with that of regular PV-WAVE system routines.
Hardware Independence	Options run on all supported platforms.

Managing Options

The following PV-WAVE routines are used to explicitly manage Options developed with OPL.

- `LOAD_OPTION` — Explicitly loads a module created as an Option to PV-WAVE.
- `UNLOAD_OPTION` — Explicitly unloads an Option module.
- `SHOW_OPTIONS` — Lists the loaded Options and their associated functions and procedures.

Loading and Unloading an Option

Assume that you have created a simple Option module that contains the functions

- `PLUS_TWO`
- `PLUS_THREE`

and the procedures

- `ADD_TWO`
- `ADD_THREE`
- `ADD_FOUR`

The Option is called `SAMPLE`, and this is the version 1.0 of `SAMPLE`.

In PV-WAVE, the functions and procedures of `SAMPLE` are not yet available, because the Option has not yet been loaded. As expected, the `SHOW_OPTIONS` procedure returns nothing:

```
WAVE> SHOW_OPTIONS, /Function, /Procedure
WAVE>
```

However, once you load the `SAMPLE` module, the functions are available. Here, the `LOAD_OPTION` procedure is used to explicitly load the Option `SAMPLE`, and now the `SHOW_OPTIONS` procedure returns a list of the functions and procedures of `SAMPLE`:

```
WAVE> LOAD_OPTION, 'SAMPLE'
WAVE> SHOW_OPTIONS, /Function, /Procedure
% Option: SAMPLE 1.000000
% Functions:
% PLUS_THREE
```

```
% PLUS_TWO
% Procedures:
% ADD_FOUR
% ADD_THREE
% ADD_TWO
```

The functions and procedures of SAMPLE are ready for use at the command line:

```
WAVE> p = 10
WAVE> ADD_THREE, p
WAVE> PRINT, p, PLUS_THREE(p)
13 16
```

Now, the UNLOAD_OPTION procedure is used to unload SAMPLE. When this is done, the functions and procedures of SAMPLE are no longer available at the command line:

```
WAVE> UNLOAD_OPTION, 'SAMPLE'
WAVE> PRINT, PLUS_THREE(p)
% Variable is undefined: PLUS_THREE.
% Execution halted at $MAIN$ .
```

As you can see, the user has control over when the Option is loaded and unloaded. For detailed information on the routines LOAD_OPTION, UNLOAD_OPTION, and SHOW_OPTIONS, see the *PV-WAVE Reference*.

The Developer Environment

The Directory Structure

The developer must be able to develop Options for UNIX, OpenVMS, Windows NT, and other platforms that support explicit loading.

To facilitate this goal, we recommend that for each Option the developer set up a directory structure containing some common files and some operating system specific files.

NOTE The Option directory structure must be located in the VNI_DIR directory. This is the main directory where all Visual Numerics products are installed.

The directory structure of an Option called SAMPLE is shown in [Figure 14-1](#).

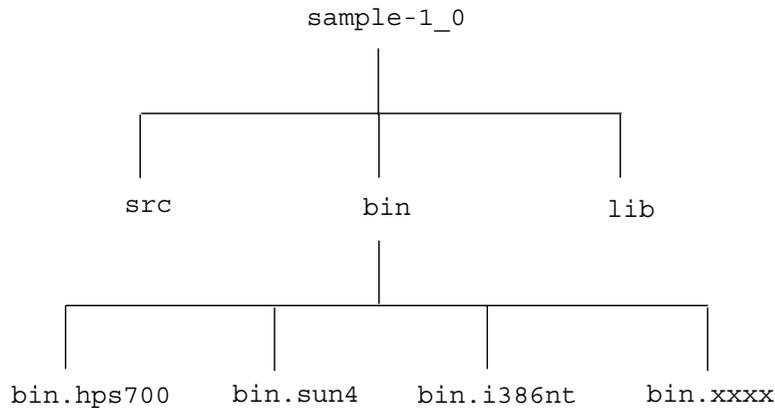


Figure 14-1 The Option directory structure for an Option called SAMPLE.

Makefiles

Makefiles can be set to be platform dependent, so that building the shared libraries is done according to the operating system specific flags and include files. We advise you to place them under the Option main directory, here `sample-1_0`.

The bin Directory

The name `SAMPLE` is used by the `LOAD_OPTION` command to choose the appropriate shared library. The shared libraries are located in the `bin` directory:

```

sample-1_0/bin/bin.hps700
sample-1_0/bin/bin.sun4
sample-1_0\bin\bin.i386nt
  
```

The src Directory

The directory `sample-1_0/src` contains the source code for the Option.

The lib Directory

If an Option requires PV-WAVE code (for example, for keyword processing), it should be located in the `lib` subdirectory.

Main Directory Requirements

NOTE You must place the Option directory in the main Visual Numerics directory. This is the main directory where your Visual Numerics products are installed.

The main Option directory must be named according to the following naming convention. The directory name is used to locate the Option.

<OptionName>-<Version>_<Release>

<i><Option_Name></i>	The name of the Option.
<i><Version></i>	The major version number of the Option.
<i><Release></i>	The minor version number of the Option.

For the example Option described in this chapter, the main directory name is `sample-1_0`.

Required Files

Under the main Option directory, you must have the files:

`bin/bin.<platform_name>`

where the platform names are the same as the platform names used for the main PV-WAVE executable.

For example, `bin/bin.hps700` or `bin/bin.i386nt`.

NOTE For OpenVMS platforms, the underscore (`_`) must be used instead of the dot (`.`) in the platform subdirectory (for example [`.bin.bin_axpvms`], or [`.bin.bin_vaxvms`]).

Assume that you are building an Option that will run under HP-UX and SunOS. The required directory structure for the Option is shown in [Figure 14-2](#).

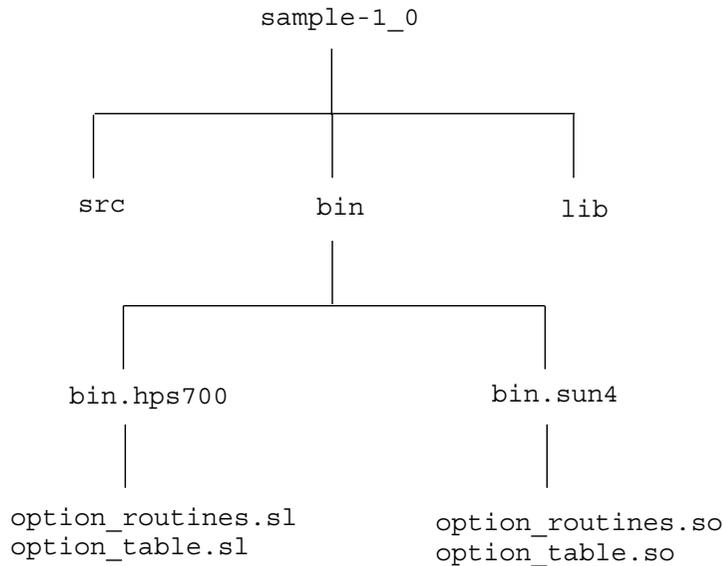


Figure 14-2 The directory structure needed to run the Option, including the shared library files.

The shared library files:

```
option_routines.xx
option_table.xx
```

are explicitly loaded when the `LOAD_OPTION` command is invoked.

- `option_table.xx` — Contains a PV-WAVE kernel table that describes the content of the Option: the number of functions, the number of procedures, the feature and version for the license manager, the names of the functions, and the names of the procedures.

In this particular example (the Option called `SAMPLE`), the file `option_table.xx` returns the fact that this Option has three functions and two procedures, named respectively:

```
PLUS_TWO (function)
PLUS_THREE (function)
PLUS_FOUR (function)
ADD_TWO (procedure)
ADD_THREE (procedure)
```

- `option_routines.xx` — Contains the actual code for these functions and procedures. This code contains OPI wrapper calls that implement the functions and procedures of the module.

The filename extension for the shared library files is operating system dependent. The following table lists the filename extensions for the operating systems that Visual Numerics supports:

Operating System	Shared Library Filename Extension
HP-UX	.sl
SunOS	.so
Solaris	.so
SGI	.so
Digital UNIX	.so
AIX	.so
NT	.dll
OpenVMS	.exe

NOTE AIX requires that the file `option_table.o` be placed in the `bin/bin.rs6000/object` directory.

Option Example

This section presents an example of an Option and describes the flow of control and data through the different components associated with the Option.

This example does not present all of the OPI features available to an Option developer. The example is based on the IMSL C/Stat/Library `normality_test` function as it might be implemented with OPI.

The user enters the following at the PV-WAVE command line:

```
x = SIN(DINDGEN(1000))
norm = IMSLS_NORMALITY_TEST(x, /Double)
```

`IMSLS_NORMALITY_TEST` is a PV-WAVE Option function (contained in a `.pro` file). The purpose of this function is to prepare parameters for the PV-WAVE

Option system functions for this IMSL C/Stat/Library function. The Option system functions are named:

`OPT_FLOAT_IMSLS_NORMALITY_TEST`, and

`OPT_DOUBLE_IMSLS_NORMALITY_TEST`

`IMSLS_NORMALITY_TEST` checks the keyword parameters, makes sure the positional parameters are the correct type and, in this case, calls:

```
value = OPT_DOUBLE_IMSLS_NORMALITY_TEST(x, $
      n_observations)
```

Up until this point the data and control flows have been identical to past versions of PV-WAVE. At this point the PV-WAVE kernel will convert the list of PV-WAVE variables passed to `OPT_DOUBLE_IMSLS_NORMALITY_TEST` to a list of PV-WAVE variable handles (WVH).

Inside of `OPT_DOUBLE_IMSLS_NORMALITY_TEST`, PV-WAVE variable handles (WVH) and PV-WAVE structure definition handles (WSDH) are used by the OPI C or FORTRAN functions to access PV-WAVE variables and PV-WAVE structure definitions.

In the example, the function definition for `OPT_DOUBLE_IMSLS_NORMALITY_TEST` is as follows:

```
WVH opt_double_imsls_normality_test(int argc, WVH *argv);
```

When `OPT_DOUBLE_IMSLS_NORMALITY_TEST` is called in the example the parameters will have the following values:

```
argc =2;
WVH *argv = {
    <WVH for the WAVE variable "x">,
    <WVH for the WAVE variable "n_observations">
};
```

The following assignment statements will get a C pointer to the data associated with the first two PV-WAVE handles:

```
x = (double *) wvh_dataptr(argv[0]);
n_observations =
    (int *) wvh_dataptr( argv[1]);
```

These pointers are then passed to the `imsls_d_normality_test` function in the C/Stat library. The return value from this function is a scalar double. For the purpose of the example the variable name of the return value will be *result*.

```
result = imsls_d_normality_test(n_observations, x);
```

To return *result* to the PV-WAVE kernel the Option must create an unnamed PV-WAVE variable, assign *result* to this variable, then return the unnamed variable's WVH. The following functions accomplish this task:

```
status = wave_get_unWVH(return_value);  
  
status = wave_assign_num(return_value,  
    TYP_DOUBLE, 0, NULL, (char *)  
    result, FALSE);  
  
return(return_value);
```

At this point control returns to the IMSLS_NORMALITY_TEST PV-WAVE Option function. The unnamed variable, with the WVH name *return_value*, has been assigned to the PV-WAVE variable *value*.

IMSLS_NORMALITY_TEST now returns *value* to the original assignment statement, as entered at the PV-WAVE command line:

```
norm = IMSLS_NORMALITY_TEST(x, /Double)
```

Creating An Option

The Option developer must create two shared objects. The first is the Option Table, which contains information regarding the number of functions, the number of procedures, the names of the functions and the names of the procedures and the feature name and version. The second shared object contains the actual code for the routines described in the Option Table.

Option development consists of the following steps:

- Create a new Option directory structure. This can be done using a template that Visual Numerics has provided.
- Modify the template files for the new Option
- Develop the Option code
- Define the Option table
- Build the new Option
- Test the new Option

Step 1: Create a New Option Directory Structure

To begin, copy the `option-templates` directory tree in the main Visual Numerics directory (the directory to which the `VNI_DIR` environment variable/

logical points). Give the new directory a name that follows the naming convention outlined in the section *Main Directory Requirements* on page 291.

For example, to create the new Option directory tree for the SAMPLE Option, enter the commands shown at the system prompt:

UNIX

```
% cd $VNI_DIR
% cp -r option-templates sample-1_0
```

OpenVMS

```
$ SET DEF VNI_DIR
$ CREATE/DIRECTORY [.SAMPLE-1_0]
$ COPY [.OPTION-TEMPLATES...]*.* [SAMPLE-1_0...]
```

Windows

```
> cd %VNI_DIR%
> xcopy option-templates sample-1_0
```

The new Option directory structure contains a number of files needed to build the Option. Some of these files must be modified for the new Option. The procedure for modifying the template files is described in the next step. The following tables list the files that were copied from the template directory tree:

UNIX Platform Files

Files	Used for:
Makefile	Controls the building of an Option.
buildmachine	Defines a build machine for a given UNIX platform.
init.mkinc	Common macros, variables for Makefile.
src/Makefile	Controls the building of the source files.
src/option_info.h	The Option definition template.
src/option_table.c	The Option Table definition.
src/option_routines.c	The Option user routines template.
src/option_rs6000.exp	The exported symbols for the Option Table (AIX only).

Files	Used for:
src/option_rs6000.imp	The imported symbols for the Option (AIX only).
src/depend.mkinc	Dependencies target for make.
src/flags.mkinc	Defines cc, ld flags based on platform.
src/axposf.mkcfg	Digital Alpha specific make flags.
src/hps700.mkcfg	HP-UX specific make flags.
src/rs6000.mkcfg	RS6000/AIX specific make flags.
src/sgi.mkcfg	SGI/IRIX specific make flags.
src/solaris.mkcfg	SPARC/Solaris specific make flags.
src/sun4.mkcfg	SPARC/SunOS specific make flags.
bin/Makefile	Controls the building of the Option shareable libraries.
lib/Makefile	Controls the compilation of the PV-WAVE procedures.

OpenVMS Platform Files

Files	Used for:
build.com	Controls the building of an Option.
[.src]build.com	Controls the building of the source files.
[.src]option_info.h	The Option definition template.
[.src]option_table.c	The Option Table definition.
[.src]option_routines.c	The Option user routines template.
[.src]table_transfer.mar	Transfer vectors for the Option Table (VAX only).
[.src]option_transfer.mar	Transfer vectors for the Option (VAX only).
[.bin]link.com	Controls the building of the Option shareable libraries.
[.bin]option_table.opt	Linker option file for the Option Table (VAX only).

Files	Used for:
[.bin]option_table_alpha.opt	Linker option file for the Option Table (Digital Alpha only).
[.bin]options.opt	Linker option file for the Option (VAX only).
[.bin]options_alpha.opt	Linker option file for the Option (Digital Alpha only).

Windows Platform Files

Files	Used for:
makefile.nt	Controls the building of an Option.
src\makefile.nt	Controls the building of the source files.
src\option_info.h	The Option definition template.
src\option_table.c	The Option Table definition.
src\option_routines.c	The Option user routines template.
src\option_table.def	The module definition file for the Option Table.
src\option_routines.def	The module definition file for the Option.

Step 2: Modify the Template Files

Next you need to modify some of the files that were copied from the template directory. In general, you will change generic names given in the template files to the name of your Option. The following tables list the files that you need to modify and tell you exactly what modifications to make to each file.

UNIX Platform Files to Modify

Files	Modification
Makefile	Change the <i>ROOTNAME</i> variable to the name of your Option directory.
buildmachine	Define a build machine for the supported platform(s).

Files	Modification
init.mkinc	Change the <i>ROOTNAME</i> variable to the name of your Option directory.
src/Makefile	Change the <i>ROOTNAME</i> variable to the name of your Option directory.
src/option_info.h	See the section Step 4: Define the New Option Table on page 301 .
src/option_routines.c	See the section Step 3: Develop the Option Code on page 300 .
src/option_rs6000.imp	Add (or delete) the IMPORTed PV-WAVE routines used in your Option (AIX only).
bin/Makefile	Change the <i>ROOTNAME</i> variable to the name of your Option directory.
lib/Makefile	Change the <i>ROOTNAME</i> variable to the name of your Option directory.

OpenVMS Platform Files to Modify

Files	Modification
[.src]option_info.h	See the section Step 4: Define the New Option Table on page 301 .
[.src]option_routines.c	See the section Step 3: Develop the Option Code on page 300 .
[.src]option_transfer.mar	Define the transfer vectors for the Option routines (VAX only).
[.bin]link.com	Change the <i>Option_Name</i> variable to define the name of the Option shareable libraries.
[.bin]options.opt	Override the default PSECT attribute setting for global variables (if needed).
[.bin]options_alpha.opt	Define the symbol vectors for the Option routines (Digital Alpha only).

Windows Platform Files to Modify

Files	Modification
<code>makefile.nt</code>	Change the <i>ROOTNAME</i> variable to the name of your Option directory.
<code>src\makefile.nt</code>	Change the <i>ROOTNAME</i> variable to the name of your Option directory.
<code>src\option_info.h</code>	See the section Step 4: Define the New Option Table on page 301.
<code>src\option_routines.c</code>	See the section Step 3: Develop the Option Code on page 300.
<code>src\option_routines.def</code>	Define the EXPORTed Option routines, and the IMPORTed PV-WAVE routines used in your Option.

Step 3: Develop the Option Code

The template file `option_routines.c` is available for developing the Option procedures in C. If you have a limited number of Option routines written in C, it is recommended that you place them in the `option_routines.c` file.

If you want to split your Option routines into several separate files, you have to modify the appropriate build files:

(UNIX) `src/Makefile`
(OpenVMS) `[.SRC]BUILD.COM`
(Windows) `src\makefile.nt`

NOTE Windows platforms require the `LibMain` function to be defined in one of the files containing the Option routine code.

NOTE AIX/RS6000 platforms require a list of the Option procedures and/or functions to be defined in the `set_sys_table` function defined in the `option_routines.c` file (for details, refer directly to the file `src/option_routines.c`).

If PV-WAVE procedure files are part of your Option, they should be placed in the `lib` subdirectory. That is because PV-WAVE automatically appends all option directories located in the directory pointed to by the `VNI_DIR` environment variable/logical and that contain the subdirectory `lib` to the `!Path` system variable. The

template Makefile is available in the lib subdirectory to create PV-WAVE compiled files (.cpr) from PV-WAVE procedure files.

Step 4: Define the New Option Table

The Option Table needs to contain information regarding the number of functions and procedures in the Option, the names of the functions and procedures in the Option, the feature name, and version of the Option.

This information must be entered into a template file, option_info.h, located in src subdirectory. Just open this file and fill in the required information as indicated in the comments. Here is a sample of the file with some additional notes:

```
/* Option Feature Identifier: Place the string between the double quotes.
```

NOTE Leave this string blank for unlicensed options.

```
*/
static char feature[] = "";
/* Option Version Identifier: Replace the 0.0 with the option's version
*/
static double version = 0.0;
/* Option Functions: Enter the number of option functions and the option function names below. Enter one name per string in the "function_names" array.
```

NOTE Function names must be in upper case and listed in alphabetical order.

```
*/
static int nm_functions = 0;
static char * function_names[] = {
};
/* Option Procedures: Enter the number of option procedures and the option procedure names below. Enter one name per string in the "procedure_names" array.
```

NOTE Procedure names must be in upper case and listed in alphabetical order.

```
*/
static int nm_procedures = 0;
static char * procedure_names[] = {
};
```

Step 5: Build the New Option

The procedure for building an Option, as explained in this section, is platform dependent.

To build the Option shareable libraries for UNIX platforms use:

```
% cd $VNI_DIR/<Option_Dir_Name>
% gmake all TARGARCH = '<platform name>'
```

The log file `BuildLog.<platform>` contains the results of the build.

NOTE The UNIX Makefiles are written for `gmake`, the FSF GNU version of `make`.

To build the Option shareable libraries for OpenVMS platforms use:

```
$ SET DEFAULT VNI_DIR: [<my option top dir>]
$ @BUILD
```

To build the Option shareable libraries for Windows platforms use:

```
Z:\VNI> cd <Option_Dir_Name>
$ nmake all -f makefile.nt
```

NOTE The environment variables/logicals `VNI_DIR` and `WAVE_DIR` must be set before building the Option. Refer to the *PV-WAVE User's Guide* if you any questions about these variables.

Step 6: Test the New Option

Place the tests written for the new Option in a `test` subdirectory.

Keyword Processing

A new Option may require the creation of some PV-WAVE procedure files to handle keyword processing.

The C or FORTRAN compiled code for Options will not support keyword parameters; however, handling keywords in an Option can be managed with PV-WAVE procedure files. The PV-WAVE language provides functions to check keyword parameters and easily convert parameters from one data type to another.

Place any `.pro` files associated with an Option in the `lib` subdirectory.

License Management

NOTE License Management support is available only for internal Visual Numerics and for explicitly licensed third party development.

The information that allows for the unlocking of an Option should be protected from the user. The user should not be able to use an Option license to unlock another Option.

A license seat must be checked out in order for an Option function to be accessible to the user; however, to compile, or load the functions in PV-WAVE, a license seat is not necessary. In that sense, checking out seats is independent from the loading for an Option.

The name of the Option as specified in the LOAD_OPTION PV-WAVE command is used as the license feature name of the Option, and must match the feature name of the Option in the license file.

Adding an Option to the PV-WAVE Search Path

The `lib` subdirectory of any properly named and located Option directory is automatically appended to the `!Path` and `!Option_Path` system variables. This allows direct access to the PV-WAVE procedures in the Option using PV-WAVE's loading mechanism.

The PV-WAVE system variable `!Option_Path` must be set to point to specific Option(s). The `!Option_Path` system variable is used to locate the Option(s) shareable executable when an Option is loaded using the PV-WAVE LOAD_OPTION procedure.

NOTE Multiple versions of an Option are not supported. If multiple directories with the same Option name are found in the directory pointed to by the `VNI_DIR` environment variable/logical, the Option with the highest version number is appended to the `!Path` and `!Option_Path` system variables.

Variable Handling Examples

This section lists example files that show how to use variable handling functions. These files are located in:

(UNIX) `$VNI_DIR/wave/demo/interapp/opi`

(OpenVMS) `VNI_DIR: [WAVE.DEMO.INTERAPP.OPI]`

(Windows) `%VNI_DIR%\wave\demo\interapp\win32\opi`

For UNIX Only

For UNIX Only

File	Description
<code>opiforunix4.f</code>	Example FORTRAN module that illustrates how to use the FORTRAN OPI to PV-WAVE variables on UNIX platforms with 4-byte long integers only.
<code>opiforunix8.f</code>	Example FORTRAN module that illustrates how to use the FORTRAN OPI to PV-WAVE variables on UNIX platforms with 8-byte long integers only.
<code>opiforunix.build</code>	C-shell script to compile and link <code>opiforunix4.f</code> into a sharable object module that can be called from PV-WAVE via the LINKNLOAD command.
<code>opiforunix.pro</code>	PV-WAVE procedure that illustrates how to use the PV-WAVE LINKNLOAD command to call the FORTRAN functions in <code>opiforunix.f</code> .
<code>opiforunix.export</code>	List of symbols exported by <code>opiforunix4.f</code> , required by <code>ld</code> on Silicon Graphics platforms only.
<code>opicunix.build</code>	C-shell script to compile and link <code>opicunix.c</code> into a sharable object module that can be called from PV-WAVE via the LINKNLOAD command.
<code>opicunix.c</code>	Example C module that illustrates how to use the C OPI to access PV-WAVE variables on UNIX platforms.
<code>opicunix.export</code>	List of symbols exported by <code>opicunix.c</code> , required by <code>ld</code> on Silicon Graphics platforms only.
<code>opicunix.pro</code>	PV-WAVE procedure that illustrates how to use the PV-WAVE LINKNLOAD command to call the C functions in <code>opicunix.f</code> .

For UNIX Only (Continued)

File	Description
<code>cwavec_unix.build</code>	C-shell script to compile and link an example of using the OPI functions from an application that uses PV-WAVE's <code>cwavec</code> functionality.
<code>cwavec_unix.c</code>	Example C module that uses PV-WAVE's <code>cwavec</code> functionality. This C module also calls the functions in <code>opicunix.c</code> and shows that the same OPI functions that are used via LINKNLOAD in the <code>opi*</code> examples can also be used from a <code>cwavec</code> application.

For VAX OpenVMS Only

For VAX OpenVMS Only

File	Description
<code>opiforvaxvms.f</code>	Example FORTRAN module that illustrates how to use the FORTRAN OPI to PV-WAVE variables on VAX OpenVMS platforms only.
<code>opiforvaxvms.com</code>	DCL script to compile and link <code>opiforvaxvms.f</code> into a sharable object module that can be called from PV-WAVE via the LINKNLOAD command.
<code>opiforvaxvms.pro</code>	PV-WAVE procedure that illustrates how to use the PV-WAVE LINKNLOAD command to call the FORTRAN functions in <code>opiforvaxvms.f</code>
<code>opicvaxvms.c</code>	Example C module that illustrates how to use the C OPI to access PV-WAVE variables on VAX OpenVMS platforms only.
<code>opicvaxvms.com</code>	DCL script to compile and link <code>opicvaxvms.c</code> into a sharable object module that can be called from PV-WAVE via the LINKNLOAD command.
<code>opicvaxvms.pro</code>	PV-WAVE procedure that illustrates how to use the PV-WAVE LINKNLOAD command to call the C functions in <code>opicvaxvms.c</code> .
<code>setup.com</code>	DCL script to assign logicals used by <code>opiforvaxvms.pro</code> and <code>opicvaxvms.pro</code> .
<code>cwavec_vaxvms.com</code>	DCL script to compile and link an example of using the OPI functions from an application that uses PV-WAVE's <code>cwavec</code> functionality.

For Digital Alpha OpenVMS Only

For Digital Alpha OpenVMS Only

File	Description
<code>opiforaxpvms.f</code>	Example FORTRAN module that illustrates how to use the FORTRAN OPI to PV-WAVE variables on Digital Alpha OpenVMS platforms only.
<code>opiforaxpvms.com</code>	DCL script to compile and link <code>opiforaxpvms.f</code> into a sharable object module that can be called from PV-WAVE via the LINKNLOAD command.
<code>opiforaxpvms.pro</code>	PV-WAVE procedure that illustrates how to use the PV-WAVE LINKNLOAD command to call the FORTRAN functions in <code>opiforaxpvms.f</code> .
<code>opicaxpvms.com</code>	DCL script to compile and link <code>opicunix.c</code> into a sharable object module that can be called from PV-WAVE via the LINKNLOAD command.
<code>opicaxpvms.pro</code>	PV-WAVE procedure that illustrates how to use the PV-WAVE LINKNLOAD command to call the C functions in <code>opicunix.c</code> .
<code>setup.com</code>	DCL script to assign logicals used by <code>opiforaxpvms.pro</code> and <code>opicaxpvms.pro</code> .
<code>cwavec_axpvms.com</code>	DCL script to compile and link an example of using the OPI functions from an application that uses PV-WAVE's <code>cwavec</code> functionality.

For Windows Only

For Windows Only

File	Description
<code>opicwin32.def</code>	Module definition file that declares symbols that are exported by the DLL.
<code>opicwin32.c</code>	Example C module that illustrates how to use the C OPI to access PV-WAVE variables on Win32 platforms.
<code>opicwin32.pro</code>	PV-WAVE procedure that illustrates how to use the LINKNLOAD command to call the C functions in <code>opicwin32.dll</code> .

For Windows Only (Continued)

File	Description
<code>cwavec_unix.c</code>	Example C module that uses PV-WAVE's <code>cwavec</code> functionality. This C module also calls the functions in <code>opicwin32.c</code> and shows that the same OPI functions that are used via LINKNLOAD in the <code>opi*</code> examples can also be used from a <code>cwavec</code> application.

NOTE FORTRAN connectivity is not available for Windows.

Files That Are Not Platform-specific

Files That Are Not Platform-specific

File	Description
<code>opi_c_devel.h</code>	Include file for C applications using OPI for C.
<code>opi_f_devel.h</code>	(UNIX and OpenVMS only) Equivalent of <code>opi_c_devel.h</code> but for FORTRAN.

Option Programming Interface Language Bindings

This section presents an overview of the architecture of an OPI style Option and describes the C and FORTRAN application programming interfaces for OPI style Options. For both C and FORTRAN, this section discusses the background, data types, and OPI functions used to manipulate PV-WAVE variables and structures from an Option.

NOTE FORTRAN connectivity is not available for Windows.

OPI Variable Handling

OPI consists of C and FORTRAN callable functions that can be used to:

- get information about existing PV-WAVE variables
- create new PV-WAVE variables
- modify existing PV-WAVE variables

- allow existing PV-WAVE variables to be used as parameters to other PV-WAVE functionality

These functions can be called from user-written programs, and may be used in conjunction with `cwavec`, `cwavefor`, or the PV-WAVE LINKNLOAD command. These functions make it easier for user-written code to access PV-WAVE variables and use other PV-WAVE functionality.

NOTE OPI variable handling functions are designed to be used with OPI options; however, these functions can be used in any code that calls PV-WAVE (e.g., via `cwavec` or `cwavefor`) and in code called from PV-WAVE (e.g., via LINKNLOAD).

Use of Opaque Handles

OPI makes use of opaque handles for elements in the PV-WAVE internal code that remain hidden from the Option developer. This makes it possible for Visual Numerics to change the underlying implementation in the future without breaking this interface definition. The following abbreviations are used for these handles:

- ***WVH*** — PV-WAVE variable handle. This handle gives Option developers access to everything they need to know about a PV-WAVE variable.
- ***WSDH*** — PV-WAVE structure definition handle. This handle gives the Option developer access to all elements of a PV-WAVE structure definition.

NOTE *WSDH* is not a PV-WAVE variable of structure type. Structure definitions exist independently of PV-WAVE variables.

CAUTION The routines described later in this chapter have been implemented as functions that return a value indicating the success or failure of the function call. The possible return values and their meanings are described for each function. It is extremely important that any code using these functions looks at the returned values and takes the appropriate action. Ignoring the returned value and continuing to execute after an error condition has occurred can result in memory trashing and PV-WAVE crashes. Ignoring return values that indicate other PV-WAVE states can result in incorrect PV-WAVE behavior.

FORTRAN Variable Handling

NOTE FORTRAN connectivity is not available for Windows.

The FORTRAN-callable OPI functions are actually written in C and are part of the PV-WAVE code base. Basically, these functions take care of all parameter passing differences between C and FORTRAN and then call one of the C-callable OPI functions.

There is not an exact one-to-one correspondence between C-callable and FORTRAN-callable functions due to basic language differences. However, there is no functionality missing from the FORTRAN-callable interface.

All FORTRAN-callable functions have names beginning with `LF_`. Their C-callable equivalent has the same name but without the `LF_` prefix.

Passing and returning string values is the biggest difference between FORTRAN and C. While the C-callable functions can return string values, the FORTRAN-callable functions must pass a string argument which will be filled in with the string value that would be returned by the equivalent C-callable function.

In all the following function descriptions where one or more of the arguments is a string, the string argument is shown to be declared as a `CHARACTER*31` FORTRAN type. The maximum length of a PV-WAVE variable or structure definition name is 31 characters. However, for each string argument passed, FORTRAN also passes the size of the string argument. When filling in a string argument for return to the calling function, these PV-WAVE functions will not “overflow” the string.

For example, if you pass a `CHARACTER*10` as the string argument to be filled by the `LF_WVH_NAME` function and the PV-WAVE variable name is longer than 10 characters, you will get only the first 10 characters of the PV-WAVE variable name. Also, if you pass a `CHARACTER*10` variable as the string argument to the `LF_WSDH_OFFSET` function, the `LF_WSDH_OFFSET` function still works as expected.

Digital Alpha Digital UNIX FORTRAN Specifics

All the FORTRAN-callable functions are names begin with `LF_` because they return long integer values. On all OpenVMS and UNIX platforms *except Digital Alpha Digital UNIX*, a long integer in C is equivalent to an `INTEGER*4` in FORTRAN. For Digital Alpha Digital UNIX, a long integer in C is equivalent to an `INTEGER*8` in FORTRAN.

NOTE All INTEGER*4 function and parameter declarations must be changed to INTEGER*8 for use on Digital Alpha Digital UNIX platforms.

OpenVMS FORTRAN Specifics

On OpenVMS, differences with string arguments are further confused by the use of string descriptors in OpenVMS. Therefore, each FORTRAN-callable function that passes one or more string arguments has yet another version of the function on OpenVMS. Functions whose names begin with LFD_ are used designed for use under OpenVMS and are the equivalent of LF_ functions, except that string descriptors are passed instead of string pointers.

Include Files

The file `opi_devel.h` is a C include file containing `#define` macros, `typedef`'s and `extern` declarations needed by the OPI functions. Just `#include` this in your program.

The file `opi_f_devel.h` lists FORTRAN type declarations and PARAMETER statements needed by the FORTRAN OPI functions. Copy the needed statements from this file into your program.

These files are located in:

(UNIX) `$VNI_DIR/wave/src/priv`
(OpenVMS) `VNI_DIR[WAVE.SRC.PRIV]`
(Windows) `%vni_dir%\wave\src\priv`

Examples

For examples showing the use of these functions, see the files in:

(UNIX) `$VNI_DIR/wave/demo/interapp/opi`
(OpenVMS) `VNI_DIR:[WAVE.DEMO.INTERAPP.OPI]`
(Windows) `%vni_dir%\wave\demo\interapp\win32\opi`

OPI Function Definitions for PV-WAVE Variables

Summary

`wave_execute` ([page 313](#))

Executes a PV=WAVE command.

`wave_compile` ([page 314](#))

Compiles a PV=WAVE command.

`wave_interp` ([page 316](#))

Executes a compiled PV=WAVE command.

`wave_free_WCH` ([page 317](#))

Frees the compiled PV=WAVE command.

`wave_assign_num`, `wave_assign_string`, `wave_assign_num` ([page 317](#))

Assigns data to an existing PV=WAVE variable.

`wave_get_WVH` ([page 321](#))

Gets a PV=WAVE variable handle for an existing named PV=WAVE variable.

`wave_get_unWVH` ([page 322](#))

Creates an unnamed PV=WAVE variable and returns its PV=WAVE variable handle.

`wave_free_WVH` ([page 323](#))

Frees memory associated with a PV=WAVE variable handle.

`wvh_name` ([page 324](#))

Returns the variable name of a PV=WAVE variable handle.

`wvh_type` ([page 326](#))

Returns the variable type of a PV=WAVE variable handle.

`wvh_ndims` ([page 327](#))

Returns the number of dimensions in a PV=WAVE variable.

`wvh_nelems` ([page 328](#))

Returns the number of elements in a PV=WAVE variable.

`wvh_dimensions` ([page 329](#))

Returns the number of dimensions and the size of each dimension.

`wvh_sizeofdata` ([page 330](#))

Returns the size in bytes of the data area of a PV=WAVE variable.

`wave_type_sizeof` ([page 330](#))

Returns the size in bytes associated with a PV=WAVE variable type.

[wvh_is_scalar \(page 332\)](#)
Tests if a PV=WAVE variable is a scalar not an array.

[wvh_is_constant \(page 333\)](#)
Tests if a PV=WAVE variable is a constant.

[wvh_dataptr \(page 334\)](#)
Returns a pointer to the data area of a PV=WAVE variable.

[wave_wsdh_from_wvh \(page 335\)](#)
Returns a PV=WAVE structure definition handle for a PV=WAVE structure variable.

[wave_wsdh_from_name \(page 336\)](#)
Returns a PV=WAVE structure definition handle given the name of a PV=WAVE structure variable.

[wave_free_WSDH \(page 337\)](#)
Frees space associated with a PV=WAVE structure definition handle when it is no longer needed.

[wsdh_name \(page 338\)](#)
Gets the structure name.

[wsdh_ntags \(page 339\)](#)
Gets the number of tags in a structure.

[wsdh_tagname \(page 339\)](#)
Gets the name of a structure tag.

[wsdh_sizeofdata \(page 341\)](#)
Gets the size of data area associated with a structure.

[wsdh_offset \(page 341\)](#)
Gets the byte offset of the data area for a named tag in a structure.

[wsdh_element \(page 342\)](#)
Creates a PV=WAVE variable handle for a tag in a structure.

[wave_error \(page 346\)](#)
Reports an error condition for the Option to PV=WAVE.

[wave_onerror \(page 348\)](#)
Sets the value of PV=WAVE error action.

[wave_is_onerror \(page 349\)](#)
Returns the current value of PV=WAVE error action.

[wave_onerror_continue \(page 349\)](#)
Sets the value of PV=WAVE error continue flag of the ON_ERROR condition.

`wave_is_onerror_continue` ([page 350](#))

Returns the current value of PV-WAVE error continue flag of the ON_ERROR condition.

`opi_malloc`, `opi_free`, `opi_realloc`, `opi_calloc` ([page 344](#))

Provide memory allocation for OPIs.

wave_execute

Executes a PV-WAVE command.

C Usage

long `wave_execute(any_wave_cmd)`

char *`any_wave_cmd`;

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

INTEGER*4 `LF_WAVE_EXECUTE(any_wave_cmd)`

CHARACTER*(*) `any_wave_cmd`

OpenVMS FORTRAN Usage

INTEGER*4 `LFD_WAVE_EXECUTE(any_wave_cmd)`

Input Parameters

any_wave_cmd — A string containing a PV-WAVE command to be executed.

Returned Status Codes

OPI_SUCCESS — Successful.

OPI_FAILURE — Errors occurred but execution can continue.

OPI_DO_NOT_PROCEED — Errors occurred: cease execution. The calling C-code should do its cleaning up (free `malloc`'s space, free handles, etc.) and return to its caller immediately.

In either of the error cases, PV-WAVE has already done its normal error processing which includes printing a message and setting the appropriate system variables.

Discussion

The `wave_execute` function takes the string argument it is given and treats it as a PV-WAVE statement. The statement will be compiled and interpreted as if it were the argument to a PV-WAVE EXECUTE function at the current interpreter position.

The purpose of this function is to give your C function the ability to use all PV-WAVE functionality from C code without having to write a separate interface to every piece of PV-WAVE functionality. For example, suppose your C function needs the transpose of a PV-WAVE variable that was passed as an argument to the function. If you have access to the PV-WAVE TRANSPOSE function, you do not have to write your own transpose routine inside the C code of your function.

For example, suppose your code has some related PV-WAVE system variables and you want to change the value of a system variable without requiring that the system variable be an argument to your function. You can use `wave_execute` to assign a new value to a PV-WAVE system variable.

NOTE `wave_execute` happens in the context of the currently active PV-WAVE procedure or function. It knows about only those variables that are in the scope of the currently active PV-WAVE procedure or function. Between the time `wave_execute` was called and when it returns, many PV-WAVE context changes could have occurred. But when `wave_execute` returns, you are again in the context of the PV-WAVE procedure or function that was active when `wave_execute` was called.

UNIX USERS If you use `wave_execute` in a `cwavec` or `cwavefor` application, you must call either `cwavec` or `cwavefor` at least one time before using `wave_execute`.

wave_compile

Compiles a PV-WAVE command.

C Usage

```
long wave_compile(any_wave_cmd, WCHptr)
```

```
char *any_wave_cmd;
```

```
WCH *WCHptr;
```

FORTRAN Usage

Not available.

Input Parameters

any_wave_cmd — A string containing a PV-WAVE command to be executed.

Output Parameters

WCHptr — A PV-WAVE handle to the compiled code ready to be executed.

Returned Status Codes

OPI_SUCCESS — Successful.

OPI_FAILURE — Errors occurred during compilation.

OPI_DO_NOT_PROCEED — Errors occurred and execution should not continue. The calling C-code should do its cleaning up (free malloc's space, free handles, etc.) and return to its caller immediately.

In either of the error cases, PV-WAVE has already done its normal error processing which includes printing a message and setting the appropriate system variables.

Discussion

`wave_execute` compiles the PV-WAVE statement each time it is called. This is quite ineffective if `wave_execute` is called several times (e.g., in the loop) with the same PV-WAVE statement. It is more efficient to compile the PV-WAVE statement once using `wave_compile`, which returns the handle *WCHptr*. Then execute the compiled PV-WAVE code pointed to by *WCHptr* several times using `wave_interp`.

The `wave_compile` function takes the string argument it is given and treats it as a PV-WAVE statement. The statement will be compiled and *WCHptr* is returned. If

the compilation of the PV-WAVE statement fails, `OPI_FAILURE` is returned, and `WCHptr` is undefined.

wave_interp

Executes a compiled PV-WAVE command.

C Usage

```
long wave_interp(wch)
```

```
WCH wch;
```

FORTTRAN Usage

Not available.

Input Parameters

wch — A PV-WAVE handle to the compiled code that is ready to be executed.

Returned Status Codes

`OPI_SUCCESS` — Successful.

`OPI_FAILURE` — Errors occurred but execution can continue.

`OPI_DO_NOT_PROCEED` — Errors occurred and execution should not continue. The calling C-code should do its cleaning up (free `malloc`'s space, free handles, etc.) and return to its caller immediately.

In either of the error cases, PV-WAVE has already done its normal error processing, which includes printing a message and setting the appropriate system variables.

Discussion

The `wave_interp` function executes the PV-WAVE code previously compiled by `wave_compile`, and pointed to by the *wch* handle. If the *wch* handle is not a valid handle, `OPI_FAILURE` is returned. If the execution of the compiled code fails, `OPI_DO_NOT_PROCEED` is returned.

wave_free_WCH

Frees the compiled PV-WAVE command.

C Usage

```
void wave_free_WCH(WCHptr)
```

```
WCH *WCHptr;
```

FORTRAN Usage

Not available.

Input Parameters

WCHptr — A PV-WAVE handle to the compiled code that is ready to be executed.

Discussion

The *wave_free_WCH* function frees the PV-WAVE resource allocated for the *WCH* handle that was allocated in *wave_compile*.

wave_assign_num

wave_assign_string

wave_assign_struct

Assigns data to an existing PV-WAVE variable.

C Usage

```
long wave_assign_num(wave_variable, type, ndims, dims, data, make_copy)
```

```
long wave_assign_string(wave_variable, ndims, dims, data, make_copy)
```

```
long wave_assign_struct(wave_variable, ndims, dims, wsdh, data, make_copy)
```

```
WVH wave_variable;
```

```
long type;  
long ndims;  
long dims[OPI_MAX_ARRAY_DIMS];  
long make_copy;  
WSDH wsdh;  
char *data;
```

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

INTEGER*4 LF_WAVE_ASSIGN_NUM(*l_wvh*, *l_type*, *l_ndims*, *l_dims*, *value*)

INTEGER*4 LF_WAVE_ASSIGN_STRING(*l_wvh*, *l_ndims*, *l_dims*, *c_value*)

INTEGER*4 LF_WAVE_ASSIGN_STRUCT(*l_wvh*, *l_ndims*, *l_dims*, *l_wsdh*,
s_value)

INTEGER*4 *l_wvh*, *l_type*, *l_ndims*, *l_wsdh*

INTEGER*4 *l_dims*(L_OPI_MAX_ARRAY_DIMS)

INTEGER*1 *value*

CHARACTER*(*) *c_value*

OpenVMS FORTRAN Usage

INTEGER*4 LFD_WAVE_ASSIGN_STRING(*l_wvh*, *l_ndims*, *l_dims*, *c_value*)

Input Parameters

wave_variable — A PV-WAVE variable handle pointing to the PV-WAVE variable to assign.

type — The numerical type of the assignment for `wave_assign_num`. Must be one of the `OPI_TYP_*` macros (PARAMETERS) defined in `opi_devel.h` or `opi_f_devel.h`.

ndims — The number of dimensions you want the PV-WAVE variable to have, and also indicates the number of elements in `dims []` (the number of elements used in

dims). For example, if `ndims` is 2, then the first 2 elements of the `dims` array must contain the dimensions you want the PV-WAVE variable to have.

dims — An array containing the dimensions of the PV-WAVE variable.

data — A pointer to the data you want to assign to the PV-WAVE variable.

It is declared as a `(char *)` but the argument you pass will more commonly be a pointer to whatever type of data you are passing. For the `wave_assign_string` function, it should be a pointer to a string if you are assigning a scalar string, and it should be a pointer to an array of string pointers if you are assigning an array of strings.

make_copy — If `TRUE`, then `wave_assign_*` will make a copy of the data area for use in the PV-WAVE kernel. If `make_copy` is `FALSE`, the PV-WAVE kernel will use the memory pointed to by `data`. This means that the memory pointed to by `data` must be free-able by the PV-WAVE kernel and must no longer be used outside the kernel.

NOTE Due to limitations of the OpenVMS/VAX operating system, PV-WAVE can not free a pointer that was `malloc'd` outside the PV-WAVE address space. Therefore, the `wave_assign_*` functions ignore the `make_copy` argument and will always make a copy of the users data area.

The FORTRAN `LF_WAVE_ASSIGN_*` functions will always make a copy of the value argument for use in the PV-WAVE kernel. Due to the dynamic nature of PV-WAVE variables, PV-WAVE must be able to free the memory associated with a PV-WAVE variable whenever it needs to. Since FORTRAN does not support dynamic allocation of memory, the values of FORTRAN variables must be copied into dynamically-allocated space in PV-WAVE if they are to be used as PV-WAVE variables.

wsdh — This is a PV-WAVE structure definition handle. The structure must already exist in the current PV-WAVE session. Use the `wave_wsdh_from_name` or `wave_wsdh_from_wvh` function to get the PV-WAVE structure definition handle of an existing PV-WAVE structure variable.

Returned Status

`OPI_SUCCESS` — Successful assignment occurred.

`OPI_FAILURE` — `wave_assign_*` considers the arguments to be invalid or inconsistent.

OPI_DO_NOT_PROCEED — Catastrophic errors occurred and execution should not continue. The calling C-code should do its cleaning up (free `malloc`'s space, free handles, etc.) and return to its caller immediately.

In either of the error cases, PV-WAVE will have already done its normal error processing which includes printing a message and setting the appropriate system variables.

Discussion

The `wave_assign_*` functions are wrappers to the function `wave_assign`. These wrappers give the Options developer access to the PV-WAVE assignment statement from the C language. These functions can be used to change the type, dimensions and/or contents of an existing PV-WAVE variable. The PV-WAVE variable to be modified must already exist within the scope of the current PV-WAVE procedure or function. It can be a named variable in the currently active PV-WAVE procedure or function or it can be an unnamed variable. Use the `wave_get_wvh` function to get a PV-WAVE variable handle for a named PV-WAVE variable. Use the `wave_get_unwvh` function to get a PV-WAVE variable handle for an unnamed PV-WAVE variable.

Use `wave_assign_num` to assign numeric values to a PV-WAVE variable. Use `wave_assign_string` to assign string values to a PV-WAVE string variable. Use `wave_assign_struct` to assign a structure value to a PV-WAVE variable.

The `wave_assign_struct` function has an addition argument, `stdef`. This is the handle of a PV-WAVE structure definition. The structure definition must already exist in the current PV-WAVE session. Use the `wave_wsdh_from_name` or `wave_wsdh_from_wvh` function to get the handle of an existing PV-WAVE structure definition.

In all the `wave_assign_*` functions, the data argument must be a pointer to the data you want to assign to the PV-WAVE variable. It is declared as a `(char *)` but the argument you pass will more commonly be a pointer to whatever type of data you are passing. For the `wave_assign_string` function, it should be a pointer to a string if you are assigning a scalar string and it should be a pointer to an array of string pointers if you are assigning an array of strings.

If the `make_copy` argument is `TRUE`, then `wave_assign_*` will make a copy of the data area for use in the PV-WAVE kernel. If `make_copy` is `FALSE`, the PV-WAVE kernel will use the memory pointed to by data. This means that the memory pointed to by data must be free-able by the PV-WAVE kernel and must no longer be used outside the kernel.

Array Indexing in C

PV-WAVE and C language array indexing is opposite. That is `WAVE_ARRAY(i, j, k)` is the same as `carray[k][j][i]`. So, for example, if the PV-WAVE array is:

```
WAVEARRAY(2,3,4,5,6,7,8,9)
```

then the corresponding C array is:

```
carray[9][8][7][6][5][4][3][2]
```

The `dims` array argument to `wave_assign_*` determines the array dimensions in PV-WAVE. If you were using `wave_assign_num` to assign `carray[]` to the PV-WAVE variable named `WAVEARRAY`, then the `dims` argument to `wave_assign_num` should have the values:

```
long dims[] = {2,3,4,5,6,7,8,9};
```

even though PV-WAVE and C language array indexing is indexed as above.

If the `wave_assign_*` functions are used to populate an undefined variable (for example a new unnamed variable created using `wave_get_unWVH`), set `ndims=0, dims=NULL` to create a scalar variable (`ndims=1, dims[0]=1` creates a one dimensional, one element array).

wave_get_WVH

Gets a PV-WAVE variable handle for an existing named PV-WAVE variable.

C Usage

```
long wave_get_WVH(wave_var_name, WVHptr)
```

```
char *wave_var_name;
```

```
WVH *WVHptr;
```

FORTTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

```
INTEGER*4 LF_WAVE_GET_WVH(wave_var_name, l_wvh)
```

```
CHARACTER*31 wave_var_name
```

INTEGER*4 *l_wvh*

OpenVMS FORTRAN Usage

INTEGER*4 LFD_WAVE_GET_WVH(*wave_var_name*, *l_wvh*)

Input Parameters

wave_var_name — A string containing the name of an existing PV-WAVE variable.

Output Parameters

WVHptr — A PV-WAVE variable handle pointing to the variable named by *wave_var_name*.

Returned Status

OPI_SUCCESS — PV-WAVE variable successfully found, and *WVHptr* is returned.

OPI_FAILURE — *wave_var_name* was not found, and *WVHptr* is not defined.

OPI_DO_NOT_PROCEED — Catastrophic errors occurred and execution should not continue. The calling C-code should do its cleaning up (free `malloc`'s space, free handles, etc.) and return to its caller immediately.

In either of the error cases, PV-WAVE will have already done its normal error processing, which includes printing a message and setting the appropriate system variables.

Discussion

The `wave_get_wvh` function looks for *wave_var_name* in the symbol table of the currently active PV-WAVE procedure or function. If found, a PV-WAVE variable handle pointing to this variable is returned in *WVHptr*.

The variable named in *wave_var_name* must already exist in PV-WAVE.

wave_get_unWVH

Creates an unnamed PV-WAVE variable and returns its PV-WAVE variable handle.

C Usage

```
long wave_get_unWVH(WVHptr)
```

```
WVH *WVHptr;
```

FORTRAN Usage

Not available.

Output Parameters

WVHptr — A PV-WAVE variable handle pointing to the unnamed variable created by `wave_get_unWVH`.

Returned Status

`OPI_SUCCESS` — The PV-WAVE variable was successfully created, and *WVHptr* is returned.

`OPI_DO_NOT_PROCEED` — Catastrophic errors occurred and execution should not continue. The calling C-code should do its cleaning up (free `malloc`'s space, free handles, etc.) and return to its caller immediately.

Discussion

The `wave_get_unWVH` function creates a new unnamed PV-WAVE variable for limited use in the system procedure or function. This unnamed PV-WAVE variable is like any other PV-WAVE variable in the currently active procedure or function except that it does not require space in the current symbol table. Since it is unnamed, it can not be accessed by the command line user.

The PV-WAVE variable handle for the unnamed variable can be used as any other PV-WAVE variable handle in any of the routines described in this document. The type, dimensions, etc., of the unnamed variable are undefined until they are defined via the `wave_assign_*` functions.

wave_free_WVH

Frees memory associated with a PV-WAVE variable handle.

C Usage

```
void wave_free_WVH(WVHptr)
```

```
WVH *WVHptr;
```

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

```
INTEGER*4 LF_WAVE_FREE_WVH(l_wvh)
```

```
INTEGER*4 l_wvh
```

Input Parameters

WVHptr — A PV-WAVE variable handle of a named or unnamed PV-WAVE variable to free.

Discussion

This function frees the space associated with both named and unnamed PV-WAVE variable handles. Before an option procedure or function returns, it *must call this routine* to free all PV-WAVE variable handles obtained via the `wave_get_WVH` and `wave_get_unWVH` functions. The option developer should *not* free a WVH which will be used as a return value.

When a named variable's WVH is passed to this function, only the space associated with the PV-WAVE variable handle is freed; the PV-WAVE variable itself is not altered in any way. When an unnamed variable's PV-WAVE variable handle is passed to this function, space associated with both the PV-WAVE variable handle and the unnamed variable are freed; the unnamed variable no longer exists after this function returns.

wvh_name

Returns the variable name of a PV-WAVE variable handle.

C Usage

```
char *wvh_name(wvh)
```

WVH *wvh*;

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

INTEGER*4 LF_WVH_NAME(*l_wvh*, *wave_variable*)

INTEGER*4 *l_wvh*

CHARACTER*31 *wave_variable*

OpenVMS FORTRAN Usage

INTEGER*4 LFD_WVH_NAME(*l_wvh*, *wave_variable*)

Input Parameters

wvh — The PV-WAVE variable handle of the PV-WAVE variable.

Returned Value — C

Returns a pointer to the PV-WAVE variable name. NULL is returned on failure.

NOTE For maximum speed and space efficiency, this function returns a pointer to a string stored internally in PV-WAVE. *DO NOT* modify the string in any way. Any modification of the string pointed to by this returned value may cause PV-WAVE to crash.

NOTE Unnamed variables do have a name string, but it is an internal identifier that can not be recognized at the PV-WAVE command line.

Returned Value — FORTRAN

The variable name is returned in the *wave_variable* string.

The function returns -1 on failure, if *l_wvh* is not a valid PV-WAVE variable handle.

wvh_type

Returns the variable type of a PV-WAVE variable handle.

C Usage

```
long wvh_type(wvh)
```

WVH *wvh*;

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

```
INTEGER*4 LF_WVH_TYPE(l_wvh)
```

```
INTEGER*4 l_wvh
```

Input Parameters

wvh — The PV-WAVE variable handle of the variable.

Returned Values for C

Returns the PV-WAVE variable type. On failure, -1 is returned.

The valid variable type codes (in C) and their corresponding PV-WAVE types are:

OPI_TYP_UNDEFINED — Undefined.

OPI_TYP_BYTE — Byte.

OPI_TYP_SHORT — Integer (FIX).

OPI_TYP_LONG — Long integer.

OPI_TYP_FLOAT — Floating point.

OPI_TYP_DOUBLE — Double precision.

OPI_TYP_COMPLEX — Complex.

OPI_TYP_DCOMPLEX — Double-precision complex.

OPI_TYP_STRING — String.

OPI_TYP_STRUCT — Structure.

Returned Values for FORTRAN

In FORTRAN, the valid type codes are:

```
L_OPI_TYP_UNDEFINED
L_OPI_TYP_BYTE
L_OPI_TYP_SHORT
L_OPI_TYP_LONG
L_OPI_TYP_FLOAT
L_OPI_TYP_DOUBLE
L_OPI_TYP_COMPLEX
L_OPI_TYP_DCOMPLEX
L_OPI_TYP_STRING
L_OPI_TYP_STRUCT
```

All possible type codes are listed as `#define`'s in `opi_devel.h` (PARAMETERS in `opi_f_devel.h`).

wvh_ndims

Returns the number of dimensions in a PV-WAVE variable.

C Usage

```
long wvh_ndims(wvh)
```

WVH *wvh*;

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

```
INTEGER*4 LF_WVH_NDIMS(l_wvh)
```

INTEGER*4 *l_wvh*

Input Parameters

wvh — The PV-WAVE variable handle of the variable.

Returned Values

Returns the number of dimensions in the variable. Returns 1 for scalar variables.

On failure, -1 is returned.

wvh_nelems

Returns the number of elements in a PV-WAVE variable.

C Usage

```
long wvh_nelems(wvh)
```

WVH *wvh*;

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

```
INTEGER*4 LF_WVH_NELEMS(l_wvh)
```

```
INTEGER*4 l_wvh
```

Input Parameters

wvh — The PV-WAVE variable handle of the variable.

Returned Values

Returns the total numbers of elements in the variable. Returns 1 for scalar variables

On failure, -1 is returned.

wvh_dimensions

Returns the number of dimensions and the size of each dimension.

C Usage

```
long *wvh_dimensions(wvh, dims)  
WVH wvh;  
long dims[OPI_MAX_ARRAY_DIMS];
```

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

```
INTEGER*4 LF_WVH_DIMENSIONS(l_wvh, l_dims)  
INTEGER*4 l_wvh  
INTEGER*4 l_dims(L_OPI_MAX_ARRAY_DIMS);
```

Input Parameters

wvh— The PV-WAVE variable handle of the variable.

Output Parameters

dims — An array of the size of each dimension.

Returned Values

Returns the number of dimensions.

If *wvh* is not a valid PV-WAVE variable handle, the returned value is -1.

If 0 is returned, the variable is a scalar for all types except structures.

See Also

`wvh_is_scalar`

wvh_sizeofdata

Returns the size in bytes of the data area of a PV-WAVE variable.

C Usage

```
long wvh_sizeofdata(wvh)
```

WVH *wvh*;

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

```
INTEGER*4 LF_SIZEOFDATA(l_wvh)
```

```
INTEGER*4 l_wvh
```

Input Parameters

wvh — The PV-WAVE variable handle of the variable.

Returned Values

Returns the size in bytes of the data area of the variable. Note that for scalar strings, `wvh_sizeofdata` returns the size of a `(char *)` and for string arrays, `wvh_sizeofdata` returns `wvh_nelems*sizeof(char *)`. To get the length of the actual string, you must get the data pointer and use the PV-WAVE `STRLEN` function or some other method of finding the length of the string pointed to by the data pointer.

On failure, `-1` is returned.

wave_type_sizeof

Returns the size in bytes associated with a PV-WAVE variable type.

C Usage

```
long wave_type_sizeof(opi_type)
```

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

INTEGER*4 LF_WAVE_TYPE_SIZEOF(*l_opi_type*)

INTEGER*4 *l_opi_type*

Input Parameters

opi_type — A valid variable type code, excluding OPI_TYP_STRUCT. This function does not recognize OPI_TYP_STRUCT since the structure definition name is needed to know its size (see [wsdh_sizeofdata](#) on page 341).

The valid variable type codes (in C) and their corresponding PV-WAVE types are:

OPI_TYP_UNDEFINED — Undefined.

OPI_TYP_BYTE — Byte.

OPI_TYP_SHORT — Integer (FIX).

OPI_TYP_LONG — Long integer.

OPI_TYP_FLOAT — Floating point.

OPI_TYP_DOUBLE — Double precision.

OPI_TYP_COMPLEX — Complex.

OPI_TYP_DCOMPLEX — Double-precision complex.

OPI_TYP_STRING — String.

In FORTRAN, the valid type codes are:

L_OPI_TYP_UNDEFINED

L_OPI_TYP_BYTE

L_OPI_TYP_SHORT

L_OPI_TYP_LONG

L_OPI_TYP_FLOAT

L_OPI_TYP_DOUBLE

```
L_OPI_TYP_COMPLEX
L_OPI_TYP_DCOMPLEX
L_OPI_TYP_STRING
```

All these type codes are listed as `#define`'s in `opi_devel.h` or `(PARAMETERS in opi_f_devel.h)`.

Returned Values

Returns the size in bytes needed to store a scalar PV-WAVE variable of the type given.

Returns `-1` if `opi_type` is not one of the basic types, i.e., `OPI_TYP_BYTE` through `OPI_TYP_STRING`.

This function does not recognize `OPI_TYP_STRUCT` since the structure definition name is needed to know its size (see [*wsdh_sizeofdata* on page 341](#)).

wvh_is_scalar

Tests if a PV-WAVE variable is a scalar (not an array).

C Usage

```
long wvh_is_scalar(wvh)
```

WVH *wvh*;

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

```
INTEGER*4 LF_WVH_IS_SCALAR(l_wvh)
```

```
INTEGER*4 l_wvh
```

Input Parameters

wvh — The PV-WAVE variable handle of the variable.

Returned Values

Returns 1 if the variable is a scalar. Returns 0 if the variable is not a scalar and returns -1 if *wvh* is not a valid PV-WAVE variable handle.

For scalar structures, this function returns 1 (TRUE) while *wvh_dimensions* does not return 0. This is because internally PV-WAVE stores structure-type variables in an array even if the variable is a scalar (1 element). So, if you want to know if a structure-type variable is a scalar (1 element) or an array of type structure, use this function rather than *wvh_dimensions*. If you use *wvh_dimensions* and it returns 1, you need to look at the `dims []` array to determine if it is a scalar variable or an array of type structure.

wvh_is_constant

Tests if a PV-WAVE variable is a constant.

C Usage

```
long wvh_is_constant(wvh)
```

WVH *wvh*;

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

```
INTEGER*4 LF_WVH_IS_CONSTANT(l_wvh)
```

```
INTEGER*4 l_wvh
```

Input Parameters

wvh — The PV-WAVE variable handle of the variable.

Returned Values

Returns TRUE if the variable is a constant. A constant PV-WAVE variable can not have its type or value changed. `wave_assign_*` will fail if you try to assign a value to a PV-WAVE constant.

On failure, -1 is returned.

wvh_dataptr

Returns a pointer to the data area of a PV-WAVE variable.

C Usage

```
char *wvh_dataptr(wvh)
```

```
WVH wvh;
```

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

```
INTEGER*4 LF_WVH_DATAPTR(l_wvh)
```

```
INTEGER*4 l_wvh
```

Input Parameters

wvh — The PV-WAVE variable handle of the variable

Returned value

Returns a pointer to the data area of the variable. You need to cast this to the proper data type, according to `wvh_type`.

NULL is returned on failure. Will return NULL if WVH is a PV-WAVE variable handle for a PV-WAVE structure.

In FORTRAN, LF_WVH_DATAPTR returns -1 on failure.

Discussion

In FORTRAN, you should pass the data pointer using `%VAL ()` to a function or subroutine, which can then declare it to be the proper data type.

Examples

For examples of how to use the data pointer in FORTRAN, see the examples in the directory:

```
(UNIX)      $VNI_DIR/wave/demo/interapp/opi
```

(OpenVMS) VNI_DIR: [WAVE.DEMO.INTERAPP.OPI]

NOTE FORTRAN connectivity is not available for Windows.

Specifically, look at the `WLNL*_*` functions (and how they are called) in one of the `opifor*.f` example programs.

wave_wsdh_from_wvh

Returns a PV-WAVE structure definition handle for a PV-WAVE structure variable.

C Usage

long *wave_wsdh_from_wvh(wvh, WSDHptr)*

WVH *wvh*;

WSDH **WSDHptr*;

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

INTEGER*4 *LF_WAVE_WSDH_FROM_WVH(l_wvh, l_wsdh)*

INTEGER*4 *l_wvh, l_wsdh*

Input Parameters

wvh — A PV-WAVE variable handle of a named or unnamed PV-WAVE structure variable.

Output Parameters

WSDHptr — The PV-WAVE structure definition for *wvh*.

Returned Values

If the PV-WAVE variable exists and is of type structure, `OPI_SUCCESS` will be returned and a PV-WAVE structure definition handle will be returned in the `WSDHptr` argument. `OPI_FAILURE` will be returned if the PV-WAVE variable

handle is invalid or the variable is not of type structure. `OPI_DO_NOT_PROCEED` will be returned for catastrophic failures such as PV-WAVE being unable to allocate memory or any other resource.

wave_wsdh_from_name

Returns a PV-WAVE structure definition handle given the name of a PV-WAVE structure variable.

C Usage

```
long wave_wsdh_from_name(struct_def_name, WSDHptr)
```

```
char *struct_def_name;
```

```
WSDH *WSDHptr;
```

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

```
INTEGER*4 LF_WAVE_WSDH_FROM_NAME(struct_def_name, l_wsdh)
```

```
CHARACTER*31 structure_def_name
```

```
INTEGER*4 l_wsdh
```

OpenVMS FORTRAN Usage

```
INTEGER*4 LFD_WAVE_WSDH_FROM_NAME(struct_def_name, l_wsdh)
```

Input Parameters

struct_def_name — A string name of a PV-WAVE structure variable.

Output Parameters

WSDHptr — The PV-WAVE structure definition handle for *struct_def_name*.

Returned value

If there is a PV-WAVE structure definition with the given name, `OPI_SUCCESS` will be returned and a PV-WAVE structure definition handle will be returned in the `WSDHptr` argument. Otherwise, `OPI_FAILURE` will be returned.

`OPI_DO_NOT_PROCEED` will be returned for catastrophic failures such as PV-WAVE being unable to allocate memory or any other resource.

Discussion

Since PV-WAVE structure definitions exist independently of PV-WAVE variables, it is not necessary to have a PV-WAVE variable handle in order to access a structure definition.

To create a new PV-WAVE structure definition, use the `wave_execute` function such as:

```
wave_execute('a = {new_struct, tag1:0, ... }')
```

wave_free_WSDH

Frees space associated with a PV-WAVE structure definition handle when it is no longer needed.

C Usage

```
void wave_free_WSDH(WSDHptr)
```

```
WSDH *WSDHptr;
```

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

```
SUBROUTINE LF_WAVE_FREE_WSDH(l_wsdh)
```

```
INTEGER*4 l_wsdh
```

Input Parameters

WSDHptr — A PV-WAVE structure definition handle.

Discussion

This function must be called to free the space associated with the PV-WAVE structure definition handle when it is no longer needed. This does not destroy the structure definition in PV-WAVE, it only frees the space associated with the handle.

wsdh_name

Gets the structure name.

C Usage

```
char *wsdh_name(wsdh)
```

```
WSDH wsdh;
```

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

```
INTEGER*4 LF_WSDH_NAME(l_wsdh, string)
```

```
INTEGER*4 l_wsdh
```

```
CHARACTER*31 string
```

OpenVMS FORTRAN Usage

```
INTEGER*4 LFD_WSDH_NAME(l_wsdh, string)
```

Input Parameters

wsdh — A PV-WAVE structure definition handle.

Returned Value — C

Returns the name of the structure definition associated with the PV-WAVE structure definition handle.

Returns NULL on failure, if the handle argument is not a valid PV-WAVE structure definition handle.

Returned Value — FORTRAN

Fills the string argument with the name of the structure definition associated with the PV-WAVE structure definition handle.

The function will return -1 on failure, if the handle argument is not a valid PV-WAVE structure definition handle.

wsdh_ntags

Gets the number of tags in a structure.

C Usage

```
long wsdh_ntags(wsdh)
```

```
WSDH wsdh;
```

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

```
INTEGER*4 LF_WSDH_NTAGS(l_wsdh)
```

```
INTEGER*4 l_wsdh
```

Input Parameters

wsdh — A PV-WAVE structure definition handle.

Returned Value

Returns the number of tags in the structure definition.

Returns -1 on failure, if the handle argument is not a valid PV-WAVE structure definition handle.

wsdh_tagname

Gets the name of a structure tag.

C Usage

```
char *wsth_tagname(wsth, N)
```

```
WSDH wsth;
```

```
long N;
```

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

```
INTEGER*4 LF_WSDH_TAGNAME(l_wsth, N, string)
```

```
INTEGER*4 l_wsth
```

```
INTEGER*4 N
```

```
CHARACTER*31 string
```

OpenVMS FORTRAN Usage

```
INTEGER*4 LFD_WSDH_TAGNAME(l_wsth, N, string)
```

Input Parameters

wsth — A PV-WAVE structure definition handle.

N — The number of the desired tag.

Returned Value — C

Returns the name of the *N*th tag in the structure definition. Tag numbers start at 0.

Returns NULL on failure, if the handle argument is not a valid PV-WAVE structure definition handle.

Returned Value - FORTRAN

Fills the string argument with the name of the *N*th tag in the structure definition. Tag numbers start at 0.

The function will return -1 on failure, if the handle argument is not a valid PV-WAVE structure definition handle.

wsdh_sizeofdata

Gets the size of data area associated with a structure.

C Usage

```
long wsdh_sizeofdata(wsdh)
```

```
WSDH wsdh;
```

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

```
INTEGER*4 LF_WSDH_SIZEOFDATA(l_wsdh)
```

```
INTEGER*4 l_wsdh
```

Input Parameters

wsdh — A PV-WAVE structure definition handle.

Returned Value

Returns the size in bytes of the data area needed to store a scalar PV-WAVE variable of this structure type.

Returns -1 on failure, if the handle argument is not a valid PV-WAVE structure definition handle.

wsdh_offset

Gets the byte offset of the data area for a named tag in a structure.

C Usage

```
long wsdh_offset(wsdh, tagname)
```

```
WSDH wsdh;
```

```
char *tagname;
```

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

INTEGER*4 LF_WSDH_OFFSET(*l_wsdh*, *tagname*)

INTEGER*4 *l_wsdh*

CHARACTER*31 *tagname*

OpenVMS FORTRAN Usage

INTEGER*4 LFD_WSDH_OFFSET(*l_wsdh*, *tagname*)

Input Parameters

wsdh — A PV-WAVE structure definition handle.

tagname — The name of the desired tag in the structure.

Returned Value

Returns the offset in bytes of the tag's data area.

Returns -1 on failure, if the handle argument is not a valid PV-WAVE structure definition handle, or if the tag doesn't exist in the structure.

Discussion

If the given *tagname* exists in the structure definition, `wsdh_offset` returns an offset which can be used to find the data associated with this *tagname*. The unit of offset is number of bytes. Given that `(char *) P` holds a pointer to the data area of a PV-WAVE variable of this structure type, `(P+offset)` points to the data corresponding to this *tagname*. Note that `(P+offset)` must be cast to the type of the associated *tagname*. The pointer to the data area of a PV-WAVE variable can be retrieved from the `wvh_dataptr` function and the type of the associated *tagname* can be retrieved from the `wsdh_element` function.

wsdh_element

Creates a PV-WAVE variable handle for a tag in a structure.

C Usage

long *wsdh_element*(*wsdh*, *tagname*, *WVHptr*)

WSDH *wsdh*;

char **tagname*;

WVH **WVHptr*;

FORTRAN Usage

NOTE FORTRAN connectivity is not available for Windows.

INTEGER*4 *LF_WSDH_ELEMENT*(*l_wsdh*, *tagname*, *l_wvh*)

INTEGER*4 *l_wsdh*, *l_wvh*

CHARACTER*31 *tagname*

OpenVMS FORTRAN Usage

INTEGER*4 *LFD_WSDH_ELEMENT*(*l_wsdh*, *tagname*, *l_wvh*)

Input Parameters

wsdh — A PV-WAVE structure definition handle.

tagname — The name of the desired tag in the structure.

Output Parameters

WVHptr — A PV-WAVE variable handle that describes this one element of the structure.

Returned Status

If the function returns *OPI_SUCCESS*, then the *WVH* will be returned in the *WVHptr* argument. If the function can not allocate enough memory to create the *WVH*, it will return *OPI_DO_NOT_PROCEED*. If the function fails because it does not find a tag whose name matches *tagname*, then it returns *OPI_FAILURE* and *WVHptr* is undefined.

Discussion

If `tagname` is a valid tag name in the structure definition referred to by `wsdh`, then `wsdh_element` will create a PV-WAVE variable handle that describes this one element of the structure definition. Since one element of a structure definition has all the characteristics of a PV-WAVE variable (except it has no data area), a WVH is a convenient way to describe it. All the `WVH_*` functions that return information about a PV-WAVE variable can also be used to return information about a structure element. `wvh_datapttr` will return `NULL` if it is passed a WVH for a structure element since structure elements do not have data areas. Since creating a WVH requires allocating some memory, `wsdh_element` may fail if the allocation request fails.

opi_malloc, opi_free, opi_realloc, opi_calloc

Allocates memory for an OPI.

C Usage

```
void *opi_malloc (unsigned int len);
```

```
void opi_free (void * buff);
```

```
void *opi_realloc (void * buff, unsigned int len );
```

```
void *opi_calloc (unsigned int l1, unsigned int l2 );
```

Input Parameters

Refer to the corresponding system functions: `malloc`, `free`, `realloc`, and `calloc`.

Returned value

Refer to the corresponding system functions: `malloc`, `free`, `realloc`, and `calloc`.

Discussion

On some platforms (OpenVMS VAX, Windows 95, Windows NT) dynamic memory allocation in an OPI that is built as a shared library uses a different address space than the memory allocation in the main executable, in this case the

PV-WAVE kernel. The consequence is that memory allocated from the OPI's own `malloc` may only be freed by the same OPI's `free` function. The same applies for the functions `realloc` and `calloc`.

NOTE The heap administration of the kernel is corrupted when the rule is violated; namely, by allocating memory in an OPI, passing a pointer to the kernel, and allowing the kernel to free this memory.

The functions `opi_malloc`, `opi_free`, `opi_realloc` and `opi_calloc` give the OPI access to the kernel's memory allocation routines. In order to avoid changing existing C-code, an include file is provided (`opi_malloc.h`). This include file has preprocessor directives which change functions calls to `malloc`, `free`, `realloc` and `calloc` into the corresponding OPI versions. You must reference the include file in an include directive (`#include 'opi_malloc.hi'`) after all system include files but before the first memory allocation function call.

C Language Error Handling

This section describes the methods an Option uses for handling C Language error conditions.

Whenever an Option's C procedures or functions fail, the Option developer needs to be able to call the PV-WAVE error handling mechanism. The variables that would have been returned by the Option will be undefined variables when an error occurs on the Option side.

When an OPI Call Fails

Most OPI calls return a status. It is up to the Option developer to handle the status codes appropriately. A status can have the following values:

- `OPI_SUCCESS` — The call is successful.
- `OPI_FAILURE` — The call failed, but execution can continue.
- `OPI_DO_NOT_PROCEED` — Catastrophic failure has occurred. For example, PV-WAVE is unable to allocate memory or any other PV-WAVE resource.

Recovering from Errors Inside the Option Code

OPI provides access to the standard PV-WAVE error handling functionality (see [wave_error on page 346](#)). Currently, when an error occurs in PV-WAVE, the inter-

preter recovers itself as best it can and then takes whatever action the user requested via the ON_ERROR and ON_IOERROR commands. To remain consistent with that model, an Option procedure or function needs to be able to tell the calling interpreter that the option procedure or function did not complete successfully. Then the interpreter will not continue interpreting any command that depends on the results of the Option procedure or function.

wave_error

Reports an error condition for the Option to PV-WAVE.

C Usage

```
void wave_error(error_number, string, print, trace, on_ioerror)
```

```
long error_number;
```

```
char *string;
```

```
long print, trace, on_ioerror;
```

FORTTRAN Usage

Not available.

Input Parameters

error_number — A value to use to set !Err.

string — A string to use to set !Err_String.

print — If nonzero the error message is printed.

trace — In nonzero trace information will be printed.

on_ioerror — If nonzero, the action taken after the error will be determined from the ON_IOERROR state of the currently active PV-WAVE procedure or function.

Discussion

The `wave_error` function allows the Option procedure or function to tell the PV-WAVE kernel that it should proceed as if the Option procedure or function failed. When the Option procedure or function returns to the calling interpreter, the interpretation of the current PV-WAVE command will be immediately terminated and

PV-WAVE will continue based on the ON_ERROR or ON_IOERROR state of the currently active PV-WAVE procedure or function.

NOTE The `wave_error` function should not be called if the system procedure or function is exiting because an OPI_DO_NOT_PROCEED status was returned from another OPI function. The OPI function will have already set the proper error conditions in the PV-WAVE kernel. Unsuccessful OPI functions return to the Option procedure or function for one reason only; to allow the Option procedure or function to clean up `malloc`'d memory before returning to the PV-WAVE kernel.

If the `wave_error` function is not called before an Option procedure or function returns, the PV-WAVE kernel will proceed as if there were no errors.

When the PV-WAVE interpreter detects an error condition, it normally prints an error message string, sets the `!Err` system variable to a number associated with the error, sets the `!Err_String` system variable to a text string associated with the error, prints a traceback message and then proceeds according to whether the error was an I/O error or regular error.

The Option procedure or function can control how PV-WAVE reacts to its errors with the arguments to the `wave_error` function, as follows:

- The `error_number` argument is the value to which `!Err` will be set.
- The `string` argument is the text string that will appear in the printed error message and in the `!Err_String` system variable.
- If the `print` argument is `FALSE`, the error message will not be printed; only the system variables will be set.
- If the `trace` argument is `FALSE`, the traceback message will not be printed.
- If the `on_ioerror` argument is `TRUE`, the action taken after the error will be determined from the ON_IOERROR state of the currently active PV-WAVE procedure or function. Otherwise, the action taken after the error will be determined from the ON_ERROR state of the currently active PV-WAVE procedure or function. The PV-WAVE `!Err` system variable will be set to the value of `error_number` and the `string` argument will be copied to the PV-WAVE `!Err_String` system variable. The PV-WAVE kernel actually copies the string and does not free it.
- If the `trace` argument is `TRUE`, the PV-WAVE kernel will print out the traceback information that commonly accompanies PV-WAVE errors.

- If the *print* argument is `TRUE`, the string will also be printed to `stderr` just as kernel error messages are printed.
- If *print* is `FALSE`, only the system variables will be updated. That is, no trace-back or error message will be printed.

If your Option needs to return any other status information to the PV-WAVE user, it should be written as a system function and return a PV-WAVE variable that contains the status information unique to your function.

wave_onerror

Sets the value of PV-WAVE error action.

C Usage

```
void wave_onerror(action)
```

```
long action;
```

FORTRAN Usage

Not available.

Input Parameters

action — A value of the PV-WAVE error action:

- 0 Stop at the statement in the procedure that caused the error. (This is the default action.)
- 1 Return all the way back to the main program level.
- 2 Return to the caller of the program unit which established the `ON_ERROR` condition.
- 3 Return to the program unit which established the `ON_ERROR` condition.

Discussion

The `wave_onerror` function sets the value of the action to take in the same way PV-WAVE command `ON_ERROR` does.

wave_is_onerror

Returns the current value of PV-WAVE error action.

C Usage

long wave_is_onerror()

FORTRAN Usage

Not available.

Discussion

Returns the value of the ON_ERROR action.

wave_onerror_continue

Sets the value of PV-WAVE error continue flag of the ON_ERROR condition.

C Usage

void wave_onerror_continue(*cont*)

long *cont*;

FORTRAN Usage

Not available.

Input Parameters

cont — A value of the PV-WAVE error continue flag of the ON_ERROR condition.

Discussion

The wave_onerror_continue sets the continue flag of the ON_ERROR condition. Valid values are TRUE or FALSE.

wave_is_onerror_continue

Returns the current value of PV-WAVE error continue flag of the ON_ERROR condition.

C Usage

long *wave_is_onerror_continue*()

FORTRAN Usage

Not available.

Discussion

Returns the current value of the continue flag of the ON_ERROR condition. Valid values are TRUE or FALSE.

FORTRAN and C Format Strings

This appendix discusses format strings that you can use to transfer data to and from PV-WAVE. Format strings specify the format in which data is transferred as well as the data conversion required.

Some PV-WAVE functions use format strings patterned after the ones used in the FORTRAN programming language. Other functions recognize both FORTRAN- or C-style format strings.

The rest of this appendix discusses the format specifications used in format strings. This appendix also discusses format reversion and the use of group repeat specifications.

What Are Format Strings?

A format string consists of one or more format specifications that tell PV-WAVE what types of data are being handled and how to format the data. For example, a C format string for importing data might look like this:

```
%3d %f
```

This string contains two format specifications. The first one (`%3d`) transfers signed integer data, with a maximum field width of three spaces. The second specification (`%f`) transfers floating-point data, with no specified field width. This format string might be used to read a data file containing two columns of numbers, one column containing integers and the other floating-point numbers.

When to Use Format Strings

All PV-WAVE functions that transfer or format data accept FORTRAN-style format strings. However, for a group of I/O commands that start with the letters “DC”, you have the choice of using either FORTRAN- or C-style format strings.

Use format strings to import or export data when you already know the type of data and its format. If you do not provide a format string, PV-WAVE uses its default rules for formatting the output. These rules are described in [Free Format Output on page 155](#).

TIP Another possibility, if you do not know exactly what your data looks like, use the DC_READ_FREE function, and let PV-WAVE help you with the interpretation of the data.

What to Do if the Data is Formatted Incorrectly

If asterisks appear in place of the data, then the values were formatted incorrectly. Possibly, the values to be transferred were larger than the format allows for, or the data type is not compatible with the format specification. If this happens, change the format to better accommodate the values.

NOTE The asterisks only appear if a FORTRAN format string is used. If the format was specified incorrectly using a C string, then incorrect data may be transferred.

Example — Using C and FORTRAN Format Strings

Below is shown part of a data file; this file contains data of many different types. For the purpose of this example, assume you are importing the data with one of PV-WAVE’s I/O routines. To specify a fixed format for importing this file, you have to know what kind of data it contains, and then create a format string that will import the data properly.

You can use C format strings only if you are using one of the DC routines (either DC_READ_FIXED or DC_WRITE_FIXED). These routines are introduced in [Functions for Simplified Data Connection on page 146](#). The detailed descriptions for these routines can be found in a separate volume, the *PV-WAVE Reference*.

Phone Data						
Date	Time	Minutes	Type	Ext	Cost	Number Called
901002	093200	21.40	1	311	5.78	2158597430
901002	093600	51.56	1	379	13.92	2149583711
901002	093700	61.39	2	435	16.58	9137485920

The first four lines of the phone data file are text — the title and column headings. Since these lines do not contain data, you may wish to filter them out. If you are using either `DC_READ_FREE` or `DC_READ_FIXED`, these lines can be skipped with the `Nskip` keyword.

The first two columns in the file, `date` and `time`, contain integer data. Since they appear to be fairly large integers, import them with the C conversion specification `%ld`. In FORTRAN, you need to specify the width of the field as well as the type. The FORTRAN specifier would be `I6`.

The next column, `minutes`, contains floating point numbers, which can be imported with `%f` for C or `F5.2` for FORTRAN.

If you have a data column that is not necessary for the analysis, such as the `type` column, import it as an ordinary character with `%c` in C or `A1` in FORTRAN. The data must be read when using a C format because assignment suppression is not allowed in `PV-WAVE`.

The `ext` column contains small integers, which you can import with `%i` (or `%d`) in C or `I3` in FORTRAN.

The `cost` column is best imported with `%f` in C or `F5.2` in FORTRAN.

In our example, the `Number Called` data is not needed for the analysis. This data can be skipped because it falls at the end of the row.

Based on this interpretation of the data, the C format string for reading this data file looks like this:

```
%ld %ld %f %c %i %f
```

The FORTRAN format string for reading this data file looks like this:

```
(I6, 1X, I6, 2X, F5.2, 4X, A1, 4X, I3, 2X, F5.2)
```

In FORTRAN, X is the specifier for blank space. It is used to account for the space between each column of values.

Another way to skip the type column would be to enter the following FORTRAN specification:

```
(I6, 1X, I6, 2X, F5.2, 9X, I3, 2X, F5.2)
```

This specification treats the type column as just another blank space.

NOTE Import `date` and `time` with a character format if you want to use the `STR_TO_DT` conversion utility to convert `date` and `time` into “true” (Julian) date/time data. This would change the C format to:

```
%s %s %f %c %i %f
```

and the FORTRAN format to:

```
(A6, 1X, A6, 2X, F5.2, 9X, I3, 2X, F5.2)
```

For more information on the `STR_TO_DT` conversion utility, refer to the description for `STR_TO_DT` in the *PV-WAVE Reference*, or refer to *Working with Date/Time Data* in the *PV-WAVE User’s Guide*. The chapter that describes date and time data also includes an example of how to handle date/time data that does not include the delimiters that the `STR_TO_DT` conversion utility expects.

Using Format Reversion

If the data is all of the same type, you can abbreviate the C and FORTRAN format strings using the technique of format reversion. Format reversion is a shorthand way of specifying a format string.

For more information on format reversion with FORTRAN format strings, refer to a FORTRAN 77 handbook.

Example — Using Format Reversion to Write Integer Data

This example writes data to a file using a single C format string:

```
var1 = INDGEN(20)
status = DC_WRITE_FIXED("simple.dat", var1, $
    Format="5%i")
```

Similarly, the entire file can be written with other PV-WAVE statements using the following FORTRAN format string:

```
OPENW, 1, "simple.dat"  
var1 = INDGEN(20) +1  
PRINTF, 1, var1, Format="(5(I4))"  
CLOSE, 1
```

The abbreviated format strings repeatedly writes the integer values in `var1` until all of the data has been transferred. The result is a data file, `simple.dat`, that contains 20 integer values:

```
 1    2    3    4    5  
 6    7    8    9   10  
11   12   13   14   15  
16   17   18   19   20
```

Example — Using Format Reversion to Read Floating-Point Data

The following data file, `tesla.dat`, contains only floating point numbers:

```
.8945 .5768 .3958 .3098 .8948 .8495  
.0938 .8749 .4798 .9204 .2479 .9485
```

This entire file can be read using a single C format string:

```
status = DC_READ_FIXED('tesla.dat', var1, $  
    Format="%f")
```

This abbreviated format string repeatedly reads or writes floating point numbers until all of the data are read or written.

Similarly, the entire file can be read with other PV-WAVE statements using the following FORTRAN format string:

```
OPENR, 1, "tesla.dat"  
var1 = FLTARR(12)  
READF, 1, var1, Format="(F5.4,2X)"  
CLOSE, 1
```

This FORTRAN format string example assumes that there are two spaces between each value, as represented by the `2X`.

Group Repeat Specifications

For data that is not all the same type, but follows a regularly-repeated pattern in the file, you can use a nested format specification enclosed in parentheses as part of the format string. This is called a *group specification*, and has the following form:

$$[n](q_1f_1s_1f_2s_2\dots f_nq_n)$$

A group specification consists of an optional repeat count n followed by a format specification enclosed in parentheses. The format specification inside the parentheses is reused n times before any more of the format string is processed.

Use of group specifications allows more compact format specifications to be written. For example, the format specification:

```
Format='("Result: ", "<", I5, ">", "<", $  
        I5, ">")'
```

can be written more concisely using a group specification:

```
Format='("Result: ", 2("<", I5, ">"))'
```

If the repeat count is one, or is not given, the parentheses serve only to group format codes for use in format reversion.

Example — Using Group Repeat Specifications to Read a Data File

Suppose you had a data file that contains data that needs to be read into three variables; the file is organized like the file shown below:

Bullwinkle	Boris	Natasha	Rocky		
10	11	10	11		
1000.0	9000.97	1100.0	0.0	0.0	2000.0
5000.0	3000.0	1000.12	3500.0	6000.0	900.0
400.0	500.0	1300.10	350.0	745.0	3000.0
200.0	100.0	100.0	50.0	60.0	0.25
950.0	1050.0	1350.0	410.0	797.0	200.36
2600.0	2000.0	1500.0	2000.0	1000.0	400.0
1000.0	9000.0	1100.0	0.0	0.0	2000.37
5000.0	3000.0	1000.01	3500.0	6000.0	900.12

The following statements read the data file shown above and place the data into three variables:

```
name = STRARR(4) & years = INTARR(4)  
salary = FLTARR(12, 4)  
; Create variables to hold the name, number of years, and monthly
```

```

; salaries.

status = DC_READ_FIXED('bullwinkle.wp', $
  name, years, salary, Format= "(4A16, " + $
  "/, I3, 3(10X,I3), /, 48(F7.2, 3X))", $
  Ignore=["$BLANK_LINES"])
; DC_READ_FIXED transfers the values in "bullwinkle.wp" to the
; variables in the variable list, working from left to right. Two slashes
; in the format string force DC_READ_FIXED to switch to a new
; record in the input file.

```

When reading row-oriented data with DC_READ_FIXED, each variable is “filled up” before any data is transferred to the next variable in the variable list. The four strings are transferred into the variable name, the four integers are transferred into the variable years, and the 48 floating-point values are transferred into the variable salary.

Because this example uses one of the “DC” functions, the data could also be read using C format specifiers:

```

status = DC_READ_FIXED('bullwinkle.wp', $
  name, years, salary, Ignore=$
  ["$BLANK_LINES"], Format="4%s 4%i 48%f")

```

NOTE The value of the *Ignore* keyword in the statements shown above insures that all blank lines are skipped instead of being interpreted as zeroes.

FORTRAN Formats for Data Import and Export

You can use FORTRAN format strings with any PV-WAVE function or procedure. FORTRAN format strings must be enclosed in parentheses, and the individual format specifiers must be separated by commas. This section discusses each of the format specifiers that can be used to produce a FORTRAN format string.

FORTRAN Format Specifiers

The following tables show the FORTRAN format specifiers that you can use in PV-WAVE.

The following table shows data conversion characters, which specify the type of data that is being transferred.

Data Transfer Format Codes

FORTTRAN Format Codes that Transfer Data

Conversion Character	How the Data is Transferred
[n]A[w]	Transfers character data. <i>n</i> is a number specifying the number of times to repeat the conversion. If <i>n</i> is not specified, the conversion is performed once. <i>w</i> is a number specifying the number of characters to transfer. If <i>w</i> is not specified, all the characters are transferred.
[n]D[w.d]	Transfers double-precision floating-point data. <i>n</i> is a number specifying the number of times to repeat the conversion. <i>w</i> specifies the number of characters in the external field, and <i>d</i> specifies the number of decimal positions.
[n]E[w.d]	Transfers single-precision floating-point data using scientific (exponential) notation. The options are the same as for the D conversion character.
[n]F[w.d]	Transfers single-precision floating-point data. The options are the same as for the D conversion character.
[n]G[w.d]	Uses E or F conversion, depending on the magnitude of the value being processed. The options are the same as for the D conversion character.
[n]I[w] or [n]I[w.m]	Transfers integer data. <i>n</i> is a number specifying the number of times to repeat the conversion. <i>w</i> specifies the width of the field in characters. <i>m</i> specifies the minimum number of non-blank digits required.
[n]O[w] or [n]O[w.m]	Transfers octal data. The options are the same as for the I conversion character.
[n]Z[w] or [n]Z[w.m]	Transfers hexadecimal data. The options are the same as for the I conversion character.
Q	Obtains the number of characters in the input record to be transferred during a read operation. This conversion character is used for input only. In an output statement, the Q format code has no effect except that the corresponding I/O list element is skipped.

NOTE Characters in square brackets [] are optional.

Data Appearance Format Codes

The following table shows specifiers that are used only to specify the appearance of data, such as the number of spaces between values in a file.

FORTRAN Format Codes that Do Not Transfer Data

Specifier	Appearance of Transferred Data
:	The colon format code in a format string terminates format processing if no more items remain in the argument list. It has no effect if variables still remain on the list.
\$	During input, the \$ format code is ignored. During output, if a \$ format code is placed anywhere in the format string, the newline implied by the closing parenthesis of the format string is suppressed.
<i>quoted string</i>	During input, quoted strings are ignored. During output, the contents of the string are written out.
nH	FORTRAN-style Hollerith string, where <i>n</i> is the number of characters. Hollerith strings are treated exactly like quoted strings.
nX	Skips <i>n</i> character positions.
Tn	Tab. Sets the absolute character position <i>n</i> in the current record.
TLn	Tab Left. Specifies that the next character to be transferred to or from the current record is the <i>n</i> th character to the left of the current position.
TRn	Tab Right. Specifies that the next character to be transferred to or from the current record is the <i>n</i> th character to the right of the current position.

NOTE For more information about the format codes shown in the previous two tables, refer to the detailed descriptions in the next section.

FORTRAN Format Code Descriptions

A Format Code

The A format code transfers character data.

Format

[n]A[w]

where:

- n — is an optional repeat count ($1 \leq n \leq 32767$) specifying the number of times the format code should be processed. If n is not specified, a repeat count of 1 is used.
- w — is an optional width ($1 \leq w \leq 256$), specifying the number of characters to be transferred. If w is not specified, the entire string is transferred. If w is greater than the length of the string, only the number of characters in the string is transferred. Since PV-WAVE strings have dynamic length, w specifies the resulting length of input string variables.

NOTE During input, if the Q FORTRAN format specifier is used, the number of characters in the input record can be queried and used as a “parameter” in a subsequent A FORTRAN format specifier.

Example

For example, the statement:

```
PRINT, Format='(A6)', '123456789'
```

generates the output:

```
123456
```

: Format Code

The colon format code terminates format processing if there are no more data remaining in the argument list.

Example

For example, the following statement:

```
PRINT, Format='(6(I1, :, ", "))', INDGEN(6)
```

outputs a comma separated list of integer values:

```
0, 1, 2, 3, 4, 5
```

The use of the colon format code prevents a comma from being output following the final item in the argument list.

\$ Format Code

When PV-WAVE completes output format processing, it normally issues a newline to terminate the output operation. However, if a \$ format code is found in the format specification, this default newline is not output.

NOTE The \$ format code is only used during output; it is ignored during input formatting.

Example

The most common use for the \$ format code is in prompting for user input. For example, the following statements:

```
PRINT, Format='($, "Enter Value: ")'  
    ; Prompt for input, suppressing any <Return>.  
  
READ, value  
    ; Read the response.
```

prompts for input without forcing the user's response to appear on a separate line from the prompt.

F, D, E, and G Format Codes

The F, D, E, and G, format codes are used to transfer floating-point values between memory and the specified file.

Format

[n]F[w.d]

[n]D[w.d]

[n]E[w.d] or [n]E[w.dEe]

[n]G[w.d] or [n]G[w.dEe]

where:

- n — is an optional repeat count ($1 \leq n \leq 32767$) specifying the number of times the format code should be processed. If n is not specified, a repeat count of 1 is used.

- $w.d$ — is an optional width specification ($0 \leq w \leq 256$, $1 \leq d < w$). w specifies the number of characters in the external field, and d specifies the number of decimal positions.
- e — is an optional width ($1 \leq e \leq 256$) specifying the width of exponent part of the field. PV-WAVE ignores this value, but it is allowed to maintain compatibility with FORTRAN.

During input, the F, D, E, and G format codes all transfer w characters from the external field and assign them as a real value to the corresponding entry in the I/O argument list.

The F and D format codes are used to output values using fixed-point notation. The value is rounded to d decimal positions and right-justified into an external field that is w characters wide. The value of w must be large enough to include a minus sign when necessary, at least one digit to the left of the decimal point, the decimal point, and d digits to the right of the decimal point. The code D is identical to F (except for its default values for w and d) and exists in PV-WAVE primarily to maintain compatibility with FORTRAN. The defaults for w , d , and e are shown in the following table:

Floating-point Format Defaults

Data Type	w	d	e
Float, Complex	15	7	2
Double	25	16	2
All Other Types	25	16	2

The E format code is used for scientific (exponential) notation. The value is rounded to d decimal positions and right justified into an external field that is w characters wide. The value of w must be large enough to include a minus sign when necessary, at least one digit to the left of the decimal point, the decimal point, d digits to the right of the decimal point, a plus or minus sign for the exponent, the character “e” or “E”, and at least two characters for the exponent.

The G format code is a compromise between these choices — it uses the F output style when reasonable and E for other values.

NOTE During output, if the field provided is not wide enough, it is filled with asterisks (*) to indicate the overflow condition. If w is zero, the “natural” width for the value is used — the value is output using a default format without any leading

or trailing white space, in the style of the C standard I/O library `printf(3S)` function.

If *w*, *d*, or *e* are omitted, the values listed in the previous table are used.

The case of the format code is ignored by PV-WAVE except during output. For output, the case of the E and G format codes determines the case used to output the exponent in scientific notation. The following table gives examples of several floating-point formats and the resulting output.

Examples of Floating Point Output

Format	Internal Value	Formatted Output
F	100.0	___100.0000000
F	100.0D	___100.000000000000000000
F10.0	100.0	___100.
F10.1	100.0	___100.0
F10.4	100.0	__100.0000
F2.1	100.0	**
e10.4	100.0	1.0000e+02
E10.4	100.0	1.0000E+02
g10.4	100.0	___100.0
g10.4	10000000.0	_1.000e+07

I, O, And Z Format Codes

The I, O, and Z, format codes are used to transfer integer values between memory and the specified file. The I format code is used for decimal values, O is used for octal values, and Z is used for hexadecimal values.

Format

[n]I[w] or [n]I[w.m]

[n]O[w] or [n]O[w.m]

[n]Z[w] or [n]Z[w.m]

where:

- n — is an optional repeat count ($1 \leq n \leq 32767$) specifying the number of times the format code should be processed. If n is not specified, a repeat count of 1 is used.
- w — is an optional integer value ($0 \leq w \leq 256$) specifying the width of the field in characters. The default values used if w is omitted are listed in the following table. If the field provided is not wide enough, it is filled with asterisks (*) to indicate the overflow condition.

NOTE If w is zero, the “natural” width for the value is used — the value is output using a default format without any leading or trailing white space, in the style of the C standard I/O library `printf(3S)` function.

Integer Format Defaults

Data Type	w
Byte, Integer	7
Long, Float	12
Double	23
All Other Types	12

- m — is the minimum number of non-blank digits required ($1 \leq m \leq 256$); this occurs only during output. The field is zero-filled on the left if necessary. If m is omitted or zero, the external field is blank filled.

The case of the format code is ignored by PV-WAVE, except during output. For output, the case of the Z format codes determines the case used to output the hexadecimal digits A-F. The following table gives examples of several integer formats and the resulting output.

Examples of Integer Output

Format	Internal Value	Formatted Output
I	3000	__3000
I6.5	3000	_03000
I5.6	3000	*****
I2	3000	**
O	3000	__5670

Examples of Integer Output (Continued)

Format	Internal Value	Formatted Output
I6.5	3000	_05670
O5.6	3000	*****
O2	3000	**
z	3000	___bb8
Z	3000	___BB8
Z6.5	3000	_00bb8
Z5.6	3000	*****
Z2	3000	**

Q Format Code

The Q format code returns the number of characters in the input record remaining to be transferred during the current read operation. It is ignored during output formatting.

Format

Q

Q is useful for determining how many characters have been read on a line. It can also be used to query the number of characters in the input record for later use as a “parameter” in the following A FORTRAN format specifier.

Example

The following statements count the number of characters in a file `demo.dat`:

```
OPENR, 1, "demo.dat"
    ; Open the file for reading.
n = 0L
    ; Create a longword integer to keep the count.
WHILE(not EOF(1)) DO BEGIN READF, 1, $
    cur, Format='(Q)' & n = n + cur &
    ; Count the characters.
END
PRINT, n, $
    Format='("Counted", I, "characters.")'
```

```
    ; Report the result.  
CLOSE, 1  
    ; Done with the file.
```

H Format Codes and Quoted Strings

Format

The format for a Hollerith constant is:

$$nHc_1c_2c_3\dots c_n$$

where:

n — is the number of characters in the constant ($1 \leq n \leq 255$).

c_i — represents the characters that make up the constant. The number of characters must agree with the value provided for n .

During output, any quoted strings or Hollerith constants are sent directly to the output. During input, they are ignored.

Example

For example, the statement

```
PRINT, Format='("Value: ", I0)', 23
```

results in

```
Value: 23
```

being output. Notice the use of single quotes around the entire format string and double quotes around the quoted string inside the format. This is necessary because we are including quotes inside a quoted string. It would have been equally correct to use double quotes around the entire format string and single quotes internally. Another way to specify the string is with a Hollerith constant:

```
PRINT, Format='(7HValue: , I0)', 23
```

NOTE The zero width of the integer format string (I) results in the “natural” width being used to output the value ‘23’.

T Format Code

The T format code specifies the absolute position in the current external record.

Format

T_n

where:

n — is the absolute character position within the external record to which the current position should be set ($1 \leq n \leq 255$).

NOTE T differs from the TL, TR, and X format codes primarily in that it requires an absolute position rather than an offset from the current position.

Example

For example:

```
PRINT, Format=$
      ('First', 20X, "Last", T10, "Middle")'
```

produces the following output:

```
      First      Middle      Last
```

TL Format Code

The TL format code moves the current position in the external record to the left.

Format

TL_n

where:

n — is the number of characters to move left from the current position ($1 \leq n \leq 255$). If the value of *n* is greater than the current position, the current position is moved to Column 1.

NOTE TL is used to move backwards in the current record. It can be used during input to read the same data twice, or during output to position the output nonsequentially.

Example

For example:

```
PRINT, Format='("First", 20X, "Last", TL15,$
              "Middle")'
```

produces the following output:

```
First           Middle           Last
```

TR and X Format Codes

The TR and X format codes move the current position in the external record to the right.

Format

```
TRn
```

```
nX
```

where:

n — is the number of characters to skip ($1 \leq n \leq 255$). During input, *n* characters in the current input record will be skipped. During output, the current output position is moved *n* characters to the right.

The TR or X format code can be used to leave blanks in the output record, or to skip over unwanted values while reading data.

Example

For example:

```
PRINT, Format=('First', 15X, 'Last')
```

or

```
PRINT, Format=('First', TR15, 'Last')
```

results in the output:

```
First           Last
```

These two format codes only differ in one way: using the X format code at the end of an output record will not cause any characters to be written unless it is followed by another format code that causes characters to be output. The TR format code always writes characters in this situation. Thus:

```
PRINT, Format=('First', 15X)'
```

does not leave 15 blanks at the end of the line, but the following statement does:

```
PRINT, Format=('First', 15TR)'
```

C Format Strings for Data Import and Export

You can use C format strings in PV-WAVE with any of the functions that begin with the two letters “DC”; this group of functions has been provided to simplify the process of getting your data in and out of PV-WAVE. This new group of I/O functions does not replace the READ, WRITE, and PRINT commands, but does provide an alternative for most I/O situations.

The FORTRAN strings, discussed in an earlier section of this appendix, are the same for either importing or exporting data. The C format strings, however, differ significantly for importing and exporting data. Thus, the following sections discuss C format strings for importing data, and then those for exporting data.

Using C Format Strings for Importing Data

The C format strings for importing data can be different than those for exporting data. The format strings for importing data are made up of conversion specifiers and literal characters (used for pattern matching), separated by a blank space. Each conversion specifier consists of a % followed by a conversion character. Between the % and the conversion character, you can place one of the following:

- An optional number specifying a *maximum* field width.
- An *h* if the imported integer is expected to be a short (16 bit) integer, or an *l* if the imported integer is expected to be a long (32 bit) integer.

NOTE Unlike the C programming language, PV-WAVE does not allow the use of an assignment suppression character, which is used to skip over an unwanted input field.

The following table shows the C conversion characters that can be used for importing data in PV-WAVE:

C-style Conversion Characters for Importing Data

Conversion Character	How the Data is Imported
c	Transfers character data, one character at a time.
e, f, g	Transfers double-precision floating-point data with optional sign, decimal point, and exponent. Precede with <i>l</i> for double-precision.

C-style Conversion Characters for Importing Data (Continued)

Conversion Character	How the Data is Imported
d or i	Transfers a signed integer. Precede with <i>l</i> for long integer.
o	Transfers octal data.
x	Transfers hexadecimal data.
u	Transfers unsigned integer data.
s	Transfers character strings.
%	Used in pattern matching to produce a literal % symbol. No conversion occurs.

Using C Format Strings for Exporting Data

The C format strings for exporting data can be different than those for importing data. The format string for exporting data is made up of ordinary characters and conversion specifications, which cause conversion and printing of the next value in the file. Each conversion specification consists of a % followed by a conversion character. Between the % and the conversion character, you may place, in order:

- A minus sign (–) to left justify the exported values.
- A number to specify the *minimum* field width.
- A period separating the field width number from the precision number.
- A number to specify the precision, or the maximum number of characters to be printed from a string, or the number of digits after the decimal point of a floating point value, or the minimum number of digits for an integer.
- An *h* if the exported integer is a short, or an *l* if the exported integer is a long.

The following table shows the C conversion characters that can be used for exporting data in PV-WAVE:

C-style Conversion Characters for Exporting Data

Conversion Character	How the Data is Exported
c	Transfers character data, one character at a time.

C-style Conversion Characters for Exporting Data

Conversion Character	How the Data is Exported
e or E	Transfers double-precision floating-point data using scientific (exponential) notation. Use the form $[-]m.ddddd e \pm xx$ or $[-]m.ddddd \pm Exx$, where the number of <i>d</i> 's is given by the precision.
f	Transfers double-precision floating point data in the form $[-]m.ddddd$, where the number of <i>d</i> 's is given by the precision. Precede with <i>l</i> for double-precision.
g or G	Uses <i>%e</i> or <i>%f</i> format, depending on the magnitude of the value being processed.
d or i	Transfer signed integer data.
o	Transfers octal data.
x or X	Transfers hexadecimal data.
u	Transfers unsigned integer data.
s	Transfers character strings.
%	Transfers a literal % symbol. No conversion occurs.

Modifying Your Environment

This Appendix discusses methods for modifying your PV-WAVE environment for UNIX, OpenVMS, and Windows.

Modifying Your PV-WAVE Environment (UNIX/OpenVMS Only)

Under UNIX, PV-WAVE uses environment variables to determine its initial state. Under OpenVMS, logical names are used for the same purpose. In either case the names and functions are the same. This section explains how to modify or customize environment variables and logicals.

NOTE Normally, you do not need to alter your environment. If PV-WAVE is installed properly, your environment will be already set up. The information in this section applies only if you wish to modify or customize your environment.

WAVE_DEVICE: Defining Your Terminal or Window System

In order to function properly, PV-WAVE must know the type of terminal or window system you wish to use. By default, it assumes X, the X Window System. If you wish, this default can be changed, as described below.

Changing the Default Device on a UNIX System

PV-WAVE reads the value of the environment variable `WAVE_DEVICE` when it starts. If `WAVE_DEVICE` is defined, PV-WAVE calls the procedure `SET_PLOT` with this string. For example, to use PV-WAVE with Tektronix terminals, include the following command in your `.login` (or `.profile`) file:

```
setenv WAVE_DEVICE tek
```

The device name can be entered in either upper or lower case. If `WAVE_DEVICE` is defined, it must contain the name of a valid PV-WAVE graphics device.

UNIX and OpenVMS USERS See the description of `SET_PLOT` in the *PV-WAVE Reference* for a complete list of device names.

Changing the Default Device on an OpenVMS System

PV-WAVE reads the value of the logical name `WAVE_DEVICE` when it starts. If `WAVE_DEVICE` is present, PV-WAVE calls the procedure `SET_PLOT` with this string. For example, to use PV-WAVE with Tektronix terminals, include the following command in your `LOGIN.COM` file:

```
$ DEFINE WAVE_DEVICE tek
```

WAVE_DIR: Ensuring Access to Required Files

`WAVE_DIR` is the root of the PV-WAVE directory structure. This environment variable is defined in `wvsetup`. All PV-WAVE files are located in subdirectories of `WAVE_DIR`.

Setting WAVE_DIR on a UNIX System

The `WAVE_DIR` environment variable must be correctly defined in order for PV-WAVE to run properly. If `WAVE_DIR` is not defined, PV-WAVE assumes a default of `/usr/local/lib/wave`.

`WAVE_DIR` is defined in the `wvsetup` file. To make sure that you have `WAVE_DIR` properly defined, enter the following command at the UNIX prompt:

```
source <maindir>/wave/bin/wvsetup
```

Setting WAVE_DIR on an OpenVMS System

The `WAVE_DIR` logical must be correctly defined in order for PV-WAVE to run properly. For example, if the PV-WAVE distribution is located in `DUA1: [WAVE]` on your system, enter the following DCL command:

```
$ DEFINE WAVE_DIR DUA1:[WAVE.]/trans= (conceal, term)
```

WAVE_DIR must be defined using the physical device name of the disk. Most sites use logical names to refer to disks. If you wish to define WAVE_DIR in terms of the disk's logical name, use the DCL F\$TRNLNM lexical function to translate the name.

For example, if the main PV-WAVE directory is DISKA: [WAVE]:

```
$ DEFINE WAVE_DIR 'F$TRNLNM("DISKA")' $  
    [WAVE.]/trans=(conceal, term)
```

WAVE_PATH: Setting Up a Search Path (UNIX, OpenVMS)

WAVE_PATH sets the function and procedure library directory search path. This environment variable is also defined in `wvsetup`. The search path is a list of locations to search if the procedure or function is not found in the current directory. The current directory is always searched first. PV-WAVE then looks for the function or procedure in the locations specified by the system variable !Path. The details of how !Path is initialized differ between UNIX and OpenVMS, although the overall concept is similar. For more information on system variables, see [System Variables on page 28](#).

Setting Up WAVE_PATH on a UNIX System

The environment variable WAVE_PATH is a colon-separated list of directories. If you do not explicitly define WAVE_PATH, and you use PV-WAVE's default startup command file, PV-WAVE starts its search in the current working directory, searches next in `$VNI_DIR/wave/lib`, and then searches numerous subdirectories of `$VNI_DIR/wave/demo`.

Each user may add directories to WAVE_PATH that contain PV-WAVE programs, procedures, functions, and "include" files. You may find it convenient to add to the value that is already defined in your `wvsetup` file. For example:

```
setenv WAVE_PATH $WAVE_PATH: "/user/mylib
```

This command adds the directory `/user/mylib` to the existing variable WAVE_PATH.

!Path is a colon-separated list of directories, similar to the PATH environment variable that UNIX uses to locate commands. When PV-WAVE starts, !Path is initialized from the environment variable WAVE_PATH. The value of !Path may be changed once you are running PV-WAVE. For example, the following command adds the directory `/usr2/home/wave_files` to the beginning of the search path:

```
WAVE> !Path = '/usr2/home/wave_files:' + !Path
```

Setting Up WAVE_PATH on an OpenVMS System

WAVE_PATH is comma-separated list of directories and library text files. Text libraries are distinguished by prepending an “@” character to their name. If you do not explicitly define WAVE_PATH, and you use PV-WAVE’s default startup command file, PV-WAVE starts its search in the current working directory, searches next in @WAVE_DIR: [LIB] USERLIB, and then searches numerous subdirectories of @WAVE_DIR: [DEMO].

Each user may assign WAVE_PATH to a unique combination of directories and text libraries that contain PV-WAVE programs, procedures, functions, and “include” files. You may find it convenient to set up this variable in your LOGIN.COM file. For example:

```
$ DEFINE WAVE_PATH $
    "DISKA:[USER.WAVELIB], $
    "@WAVE_DIR:[LIB]USERLIB.TLB"
```

causes PV-WAVE to search for programs first in the current directory, then in the directory DISKA:[USER.WAVELIB], and finally in the Standard library, which is supplied by Visual Numerics, Inc., as an OpenVMS text library. For more information on OpenVMS text libraries, see [OpenVMS Procedure Libraries on page 233](#).

WAVE_PATH can also be defined as a multi-valued logical name (for example a search list logical). Therefore, the above example can also be written as:

```
$ DEFINE WAVE_PATH DISKA:[USER.WAVELIB], $
    "@WAVE_DIR:[LIB]USERLIB.TLB"
```

PV-WAVE simply takes the various translations and concatenates them together into a comma separated list. Note that the quotes around the second translation in this example are necessary to keep DCL from seeing the “@” character as an invitation to execute a command file.

Under OpenVMS, !Path is a comma-separated list of directories and text libraries. Text libraries are distinguished by prepending an “@” character to their name.

When PV-WAVE starts, !Path is initialized from the logical name WAVE_PATH. The value may be changed once you are running PV-WAVE. For example, the following command adds the DISKA:[PROJECTLIB] directory to the beginning of the search path:

```
WAVE> !Path = 'DISKA:[PROJECTLIB], ' + !Path
```

WAVE_STARTUP: Using a Startup Command File

WAVE_STARTUP points to the name of a command file that is executed when PV-WAVE starts. Common uses are to compile frequently-used procedures or functions, to load data, and to perform other useful operations. It contains statements which are individually compiled and executed, in the same manner as command file execution. For more information on command files, see [Creating and Running a Command \(Batch\) File on page 7](#).

The default startup file for UNIX is called wavestartup and is located in `<path>/wave/bin`. For OpenVMS the default file is wavestartup.dat and is located in:

```
WAVE_DIR: [000000.BIN]
```

The wavestartup file turns off the compiler messages, sets up the WAVE> prompt, and then calls the Standard library routine setdemo.pro. This routine sets up the default key bindings for the function keys and displays their definitions upon entering PV-WAVE. For more information about the SETDEMO command, see the *PV-WAVE Reference*.

Using a Startup File Under UNIX

To use a startup file under UNIX, set the environment variable WAVE_STARTUP to the name of the file to be executed. For example, assume the startup file named startfile contains the following statements:

```
.RUN add.pro
.RUN square.pro
INFO
```

Set the environment variable with the setenv command:

```
setenv WAVE_STARTUP startfile
```

When you start PV-WAVE by entering wave at the UNIX prompt, you get the following display:

```
PV-WAVE. Version ...
.
.
% Compiled module: ADD.
% Compiled module: SQUARE.
% At $MAIN$ .
Code area used: 0.00% (0/16384), Data area used: 0.05% (2/4096)
# local variables: 0, # parameters: 0
Saved Procedures:
```

```
ADD
Saved Functions:
  SQUARE
WAVE>
```

The startup file compiles the ADD procedure and the SQUARE function, and displays general information about the current status of PV-WAVE before displaying the WAVE> prompt.

Using a Startup File Under OpenVMS

To use a startup file under OpenVMS, assign the OpenVMS logical name WAVE_STARTUP to the name of the file to be executed.

The procedure search path, !Path, is used to search for the file if it is not in the current directory.

To define a startup file named `startfile`, enter:

```
DEFINE WAVE_STARTUP startfile
```

When you enter `wave` at the operating system prompt, the file is executed.

WAVE_FEATURE_TYPE: Setting the Default Operating Mode

The environment variable WAVE_FEATURE_TYPE lets you set the default operating mode to “runtime”. When this environment variable is set to RT, compiled PV-WAVE applications can be executed directly from the operating system prompt without using the `-r` option. For example:

```
% setenv WAVE_FEATURE_TYPE RT
    Set the environment variable.

% wave somerset
    Run a compiled, saved PV-WAVE application called somerset.
```

WAVE_RT_STARTUP: Using a Startup Procedure in Runtime Mode

WAVE_RT_STARTUP points to the name of a compiled procedure file that is executed when PV-WAVE initializes in runtime mode. The startup file may contain PV-WAVE routines that are executed each time PV-WAVE is started in runtime mode.

UNIX and OpenVMS USERS For more information on saving and using compiled routines, see *Runtime Mode for UNIX and OpenVMS* on page 12.

On a UNIX system, the default startup file for runtime mode is:

```
$WAVE_DIR/lib/std/rtwavestartup.cpr
```

On an OpenVMS system, the default startup file for runtime mode is:

```
WAVE_DIR: [000000.LIB.STD]RTWAVESTARTUP.CPR
```

WAVE_INIT_CODESIZE: Setting Initial Size of the Code Area

The environment variable `WAVE_INIT_CODESIZE` lets you set the initial size of the code area for PV-WAVE. For example:

```
% setenv WAVE_INIT_CODESIZE 2000000
    Set the initial size of the code area to 2 MB.
```

WAVE_INIT_LVARS: Setting Initial Value for Number of Local Variables

The environment variable `WAVE_INIT_LVARS` lets you set the initial number of local variables for PV-WAVE. For example:

```
% setenv WAVE_INIT_LVARS 400
    Set the initial number of local variables to 400.
```

Changing the PV-WAVE Prompt

The text string PV-WAVE uses to prompt you for input is specified by the system variable `!Prompt`. You can change the prompt by setting this system variable to the new prompt string. The prompt is currently defined in the file `wavestartup` for UNIX and `WAVESTARTUP.DAT` for OpenVMS.

Here's an example showing how to tailor your prompt to display text:

```
!Prompt = 'Hello World!> '
```

Here's another example that causes PV-WAVE to ring the bell on the terminal without echoing visible text when prompting:

```
!Prompt = '\007'
```

The ASCII code for the bell is 7. It does not have a printable representation, so it is specified using the octal escape sequence `\007`.

UNIX and OpenVMS USERS See *Representing Nonprintable Characters with UNIX/OpenVMS* on page 24.

TIP You can also place a prompt definition in your `WAVE_STARTUP` file, as described on *WAVE_STARTUP: Using a Startup Command File* on page B-5.

For an alternate way to modify the prompt, see the description of the `PROMPT` procedure in the *PV-WAVE Reference*.

Defining Keyboard Shortcuts

Function keys may be equated to a character string using the `DEFINE_KEY` procedure. For example, the `<R4>` key on a Sun-style keyboard or the `<PF4>` key on a Digital keyboard, can be equated to the string `PLOT`, as shown in the example below. This allows frequently used strings and commands to be entered with a single key stroke.

```
SETUP_KEYS
    ; Load predefined function key definitions.

DEFINE_KEY, 'F11', 'PLOT'
    ; Enter the text "PLOT" when the F11 function key is pressed.
```

For detailed information on `DEFINE_KEY`, see its description in the *PV-WAVE Reference*.

The command `INFO, /Keys` displays the current definition of all function keys.

TIP A natural place to put your key definitions is in the startup file so that the function keys are defined when `PV-WAVE` is initialized. The defaults for the key definitions are set up with the `setdemo.pro` procedure in the `wavestartup` file. See *WAVE_STARTUP: Using a Startup Command File* on page B-5.

Using PV-WAVE with X Windows

A brief explanation of how to set up the X Windows system to work with `PV-WAVE` is provided in this section.

TIP The interface to the X Windows system is described in detail in the *PV-WAVE Reference*.

If You Are Running Under X Windows

Little or no customizing is required to use PV-WAVE with the X Windows system. You can control the number of colors used by PV-WAVE, if and how windows are repainted, and the type of color system (visual class).

Be sure that your system is properly set up to display X graphics. For UNIX systems under the C shell, you may need to enter:

```
% setenv DISPLAY hostname:0.0
% xhost hostname
```

For OpenVMS systems, you may need to enter:

```
$ SET DISPLAY /CREATE /NODE=nodename -
    /SCREEN=0.0 /TRANSPORT=transport_type
```

where *hostname* or *nodename* is the name of the system on which you want graphics to be displayed.

Modifying Your PV-WAVE Environment (Windows)

Under Windows, PV-WAVE obtains the information it needs to determine its initial state from environment variables. Windows NT and Windows 95 allow environment variables to be specified in two ways:

- The Registry (Windows NT and Windows 95)
- The System window (launched from the Windows Control Panel — Windows NT only)
- The AUTOEXEC.BAT file (in Windows 95 only)

This section discusses ways to customize PV-WAVE using system variables and environment variables. In addition, several important environment variables are discussed in detail.

NOTE As long as PV-WAVE is installed properly, your environment will be already set up. Much of the information in this section applies only if you wish to modify or customize your environment.

Adding a Procedure Library to the Search Path

This section explains how to add your own procedure library to the default PV-WAVE path. This default path is used by PV-WAVE to locate procedure librar-

ies (directories containing .pro and .cpr files), such as the Standard Library and the Users' Library.

The best way to add a procedure library to the default path is to use the !Path system variable.

The system variable !Path stores a list of directories, similar to the PATH environment variable that Windows uses to locate commands. When PV-WAVE starts, !Path is initialized with all of the directory paths necessary to run PV-WAVE and any PV-WAVE Companion Technologies or options that have been installed.

You can modify !Path while PV-WAVE is running, or you can modify it in your PV-WAVE startup file. If you modify !Path during a PV-WAVE session, the change only takes effect for that session. If you modify the startup file, the change takes effect every time you start PV-WAVE. (For information on the startup file, see [WAVE_STARTUP: Using a Startup Command File](#) on page B-5.)

For example, the following command adds the directory

D:\myra\results\wave to the beginning of the search path:

```
!Path = 'D:\myra\results\wave;' + !Path
```

Again, if this line is typed at the WAVE> prompt, the change takes effect only for the current session. If you add this line to a PV-WAVE startup file, the change takes effect every time you start PV-WAVE.

TIP When looking for a function or procedure file, PV-WAVE searches current working directory first. PV-WAVE then looks for the function or procedure in the locations specified by !Path.

For more information on system variables, see [System Variables](#) on page 28.

Environment Variables

PV-WAVE relies on the user's environment for configuration and customization information. The following environment variables tell PV-WAVE where it is installed, where to find important files, and how it is to behave when started.

PV-WAVE Environment Variables

IMSLERRPATH	WAVE_DEMO	WAVE_HELPDIR
IMSLSERRPATH	WAVE_GALL	WAVE_HELP_PATH
VNI_DIR	WAVE_GALL2	WAVE_USER

WAVE_DIR	WAVE_GALL3	WAVE_PATH
WAVE_STARTUP	WAVE_ARL	WAVE_VERSION
WAVE_APPL	WAVE_RAY	WAVE_BIN
WAVE_CODEBOOK	WAVE_LANG	LM_LICENSE_FILE
WAVE_DATA	WAVE_LIB	

Support for Environment Variables in Windows

Environment Variable Support in Windows NT

Windows NT was designed to support applications ported from UNIX and has strong support for environment variables. You can set environment variables for an individual user and for a particular computer by using the **System** icon in the Windows Control Panel. This environment is picked up by GUI applications launched from the Program Manager as well as Console applications. Environment variables can take up as much space as needed and can be easily changed on a system-wide basis without restarting Windows.

Environment Variable Support in Windows 95

Windows 95 does not offer the same strong degree of support for environment variables as does Windows NT. Windows 95 Consoles are designed to support MS-DOS applications and offer only a limited amount of space for environment variables; the amount of space is configurable, but this is difficult to do for a particular Console without requiring the user to manually change the settings.

Under Windows 95, the system environment as seen by new Console windows and by applications launched from shortcuts and the Start menu, must be set in the AUTOEXEC .BAT file at the time the system is started. If you want to change these environment settings you must modify AUTOEXEC .BAT and reboot the system.

In general, it is impractical to store all of the environment variable information needed by your system's applications in the AUTOEXEC .BAT file.

For this reason, PV-WAVE makes use of a new Windows feature, the Registry, to extend the way in which environment variables are obtained.

What is the Registry?

The Registry is a repository provided by Windows NT and Windows 95 to allow applications and the operating system to store and manipulate configuration and

status information; the Registry is designed to replace the use of environment variables and system configuration (.INI) files. The Registry is essentially a hierarchical database. See *The Windows Interface Guidelines for Software Design* for more information.

WIN32 applications are expected to store their configuration information at a particular location in the Registry. PV-WAVE follows this convention and stores the information previously contained in environment variables at the following keys.

This first key is the location where PV-WAVE environment variables are defined upon product installation:

```
HKEY_LOCAL_MACHINE\Software\Visual Numerics\PV-WAVE\
6.0\Environment
```

The environment variables defined in this location are:

- VNI_DIR
- WAVE_DIR
- LM_LICENSE_FILE

You can define your own environment variables in the following Registry location. Any variables defined in this user area take precedence over variables defined in the PV-WAVE Registry location.

```
HKEY_CURRENT_USER\Software\Visual Numerics\PV-WAVE\ 6.0\Environment
```

For information on how to modify the Registry, see [Modifying the Registry on page 13](#).

The information at the keys is stored in name-value pairs, where the name is the name of the environment variable and the value is the variable's value.

How the PV-WAVE Environment is Set

At initialization, PV-WAVE looks for each of the environment variables listed in the Table on [page B-10](#) in the current environment. If the variable is found, PV-WAVE uses its value and does no further processing for that variable.

If the variable is not found, PV-WAVE looks for an entry in the HKEY_CURRENT_USER key in the Registry, and finally looks in the HKEY_LOCAL_MACHINE Registry key. This means that you can override the Registry entries in a local shell and that any batch or startup files that you are currently using will continue to work — PV-WAVE only uses the Registry if it cannot find the information it needs in the local environment.

TIP The environment variables are processed in the order listed in listed in the Table on [page B-10](#). This means that you can use the values of earlier environment variables when setting later ones: since `VNI_DIR` is listed before `WAVE_DIR`, you can define `WAVE_DIR` as `%VNI_DIR%/wave`.

Modifying the Registry

Normally, the PV-WAVE Registry entries are only modified by the setup program when PV-WAVE is installed and it should not be necessary to change the entries there by hand. But, if this does become necessary, you can use the Windows Registry Editor utility to modify PV-WAVE registry entries.

On Windows NT this program is `regedt32.exe`, and on Windows 95 it is `regedit.exe`. You must have Administrator privileges to modify entries under the `HKEY_LOCAL_MACHINE` key on Windows NT.

CAUTION Be extremely careful when modifying the Registry. In general, changing entries under the PV-WAVE keys described above is safe but changing values in other parts of the Registry can cause serious system problems. If you are unsure about whether you are changing the proper value, consult with your System Administrator.

Backing Up the Registry

On Windows 95, it is a good idea to make a backup copy of the Registry before modifying Registry values. The Registry is stored in the file `SYSTEM.DAT` in the `%windir%` folder and you can make a backup by copying this file to another folder with the Explorer or via the `xcopy` command from a Console.

See the article *Backing Up the Registry or Other Critical Files, Id: Q132332*, in the Microsoft Knowledge Base for more information.

WAVE_PATH: Setting Up a Search Path (Windows)

`WAVE_PATH` sets the function and procedure library directory search path. This environment variable is also defined in `wvsetup`. The search path is a list of locations to search if the procedure or function is not found in the current directory. The current directory is always searched first. PV-WAVE then looks for the function or procedure in the locations specified by the system variable `!Path`. The details of how `!Path` is initialized differ between UNIX and OpenVMS, although the overall

concept is similar. For more information on system variables, see [System Variables on page 28](#).

Setting Up WAVE_PATH on a UNIX System

The environment variable `WAVE_PATH` is a colon-separated list of directories. If you do not explicitly define `WAVE_PATH`, and you use **PV-WAVE**'s default startup command file, **PV-WAVE** starts its search in the current working directory, searches next in `VNI_DIR\wave\lib`, and then searches numerous subdirectories of `VNI_DIR\wave\demo`.

Each user may add directories to `WAVE_PATH` that contain **PV-WAVE** programs, procedures, functions, and “include” files. To add the `WAVE_PATH` environment variable:

- Step 1** In the Control Panel, double-click the System icon.
- Step 2** In the System Properties dialog box, click the Advanced tab, then the Environment Variable button (or click the Environment tab).
- Step 3** Create the new system variable `WAVE_PATH` and add the paths to the directories you want included in **PV-WAVE**'s path at startup.

VNI_DIR and WAVE_DIR: Ensuring Access to Required Files

The `VNI_DIR` environment variable must be correctly defined in order for **PV-WAVE** to run properly; `VNI_DIR` is defined during the installation process.

All **PV-WAVE** files are placed in a subdirectory of `VNI_DIR` that becomes the top-level directory for **PV-WAVE**; the path to this directory is stored in the environment variable `WAVE_DIR`. Look in the file system where you have installed Visual Numerics software and you will see a subdirectory named `wave`. This is where **PV-WAVE** has been installed, the directory to which the environment variable `WAVE_DIR` points, and where **PV-WAVE** expects to find its required files.

To see what the value of `WAVE_DIR` is on your computer, enter the following command at the prompt of an MS-DOS command prompt window:

```
echo %WAVE_DIR%
```

NOTE Remember that `VNI_DIR` and `WAVE_DIR` are only defined automatically on the machine where **PV-WAVE** was installed, and only for the user who performed the installation. If you wish to run **PV-WAVE** on a different machine or for

different users, you must explicitly set the value of these variables using the Control Panel's System window (on Windows NT) or in the AUTOEXEC .BAT file (Windows 95) before you run PV-WAVE on that machine for the first time. Or, you can rerun the installation program (the setup program). For detailed information on rerunning the setup program, see the *Installation Guide*. You may also need to use the File Manager to connect the disk that contains the PV-WAVE files. For instructions on connecting another disk, consult your Windows documentation.

When Are PV-WAVE's Environment Variables Defined?

VNI_DIR and WAVE_DIR are the only environment variables that get defined during the installation process. Other PV-WAVE environment variables get defined dynamically as PV-WAVE is started. However, the value of any environment variable that has been explicitly defined prior to PV-WAVE startup is left intact instead of being redefined during startup. For information about other PV-WAVE environment variables, refer to subsequent sections in this discussion of the PV-WAVE environment.

Method of Starting PV-WAVE Can Affect Your Environment

When you use System window (launched from the Control Panel on Windows NT only) to set the value of environment variables, the value applies only to MS-DOS command prompt windows that you open subsequent to making those changes. A session of PV-WAVE that you start from a new MS-DOS command prompt window will be aware of your changes and will honor them.

However, the Windows program group to which PV-WAVE belongs will be unaware of the changes you made. Consequently, if you start PV-WAVE by clicking on an icon in a program group, that session of PV-WAVE will also be unaware of the changes you made. For the program group to be aware of the changes you made, you must log off and then log back on to your computer.

WAVE_DATA: Retrieving Data Files Directly Where They Reside

You can store your data in a different area, e.g., a disk that actually resides on a different computer, and still easily access those files, by defining a value for WAVE_DATA prior to starting PV-WAVE. This way you can leave your data files intact in the area where you first stored them, and not have to specify a long path to them or copy them into your current working directory.

NOTE The default `WAVE_DATA` path is used by the PV-WAVE Gallery and other demonstration programs. If you reset `WAVE_DATA`, these demonstration systems will not work.

Using WAVE_DATA in PV-WAVE Function and Procedure Calls

`WAVE_DATA` specifies a single path, and this path is used to initialize the PV-WAVE system variable `!Data_Dir`. You can then use this system variable as a shorthand notation for pointing to data files, as shown in the following sample PV-WAVE statements:

```
OPENR, 1, !Data_Dir+'latest.dat'
```

or

```
status = DC_READ_DIB(!Data_Dir+'ztee.bmp', $  
    zt, Imagewidth=xsize, Imagelength=ysize)
```

PV-WAVE does not use a search path when accessing, opening, and closing data files; it uses only the path that you specify, which is the current working directory if you have not specified any other path.

NOTE By default, `!Data_Dir` is initialized during PV-WAVE startup by a call to `setdemo.pro`. If the startup command file you are using does not include a call to `setdemo.pro`, several system variables, including `!Data_Dir`, may not be initialized properly. For more details, refer to [WAVE_RT_STARTUP: Using a Startup Procedure in Runtime Mode](#) on page B-19.

WAVE_DEVICE: Defining Your Terminal or Window System

PV-WAVE must know the type of terminal or window system that you are using. By default, it assumes `win32` (Windows, 32 bit). If you wish, this default can be changed, as described below.

Changing the Default Device

PV-WAVE sets the value of the environment variable `WAVE_DEVICE` to `win32` when it starts. But if you prefer `WAVE_DEVICE` to have a different value, such as `PS` (PostScript), you can enter the following command at the prompt of an MS-DOS command prompt window:

```
set WAVE_DEVICE=PS
```

In this situation, PV-WAVE will start as expected, but will send graphics output to a file using the specified format (PostScript) and the default filename of `wave.ps`. You will not be able to display graphics windows on the screen of your computer until you enter the command:

```
SET_PLOT, 'win32'
```

The device name can be entered in either upper or lower case. If `WAVE_DEVICE` is defined, it must contain the name of a valid PV-WAVE graphics device. For details, see the list of valid output devices in the *PV-WAVE Reference*.

TIP Use the `DEVICE` command to reconfigure the behavior of any of PV-WAVE's drivers, including the PostScript driver mentioned in this section. For example, you could use the *Filename* keyword to choose a different output filename, or you could use the *Epsi* keyword to enable encapsulated PostScript interchange format.

WAVE_STARTUP: Using a Startup Command File

`WAVE_STARTUP` points to the name of a command file that is executed by PV-WAVE on initialization. The startup file contains a series of PV-WAVE statements and is executed each time PV-WAVE is started. Common uses are to compile frequently-used procedures or functions, to load data, and to perform other useful operations. It contains PV-WAVE statements which are individually compiled and executed, in the same manner as command file execution. For more information on command files, see [Creating and Running a Command \(Batch\) File on page 7](#).

The default startup filename is `wavestartup` and is located in `%WAVE_DIR%\bin`. The `wavestartup` file turns off the compiler messages, defines the `WAVE>` prompt, and then calls the Standard library routine `setdemo.pro`. This routine sets up the default key bindings for the function keys and displays their definitions upon entering PV-WAVE. For more information about the `SETDEMO` command, see the *PV-WAVE Reference*.

To use a different PV-WAVE startup file:

- ❑ Create a file containing the commands you want to be executed every time you start PV-WAVE. For example, assume the startup file named `startfile.txt` contains the following statements:

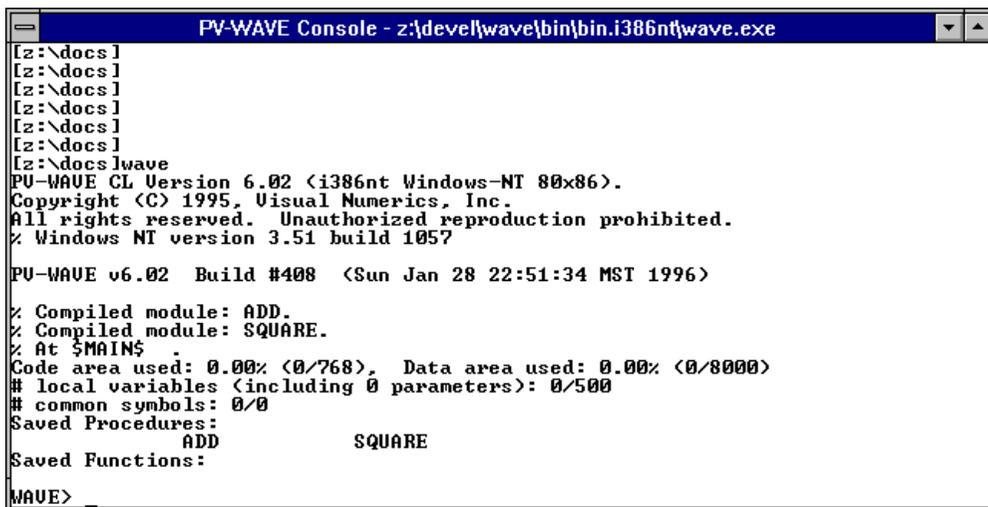
```
.RUN add.pro
.RUN square.pro
INFO
```

- ❑ In an MS-DOS command prompt window, enter this command to set the environment variable `WAVE_STARTUP` to the name of the file to be executed:

```
set WAVE_STARTUP=startfile.txt
```

The startup file compiles `ADD` and `SQUARE` and displays general information about the current status of `PV-WAVE` before displaying the normal `WAVE>` prompt. The messages you see when you start `PV-WAVE` this way are shown in [Figure B-1](#) below.

To use this startup file every time you start `PV-WAVE` use the **System** icon in the Windows Control Panel (Windows NT) or modify the `AUTOEXEC.BAT` file (Windows 95).



```
PV-WAVE Console - z:\devel\wave\bin\bin.i386nt\wave.exe
[z:\docs]
[z:\docs]
[z:\docs]
[z:\docs]
[z:\docs]
[z:\docs]
[z:\docs]wave
PV-WAVE CL Version 6.02 (i386nt Windows-NT 80x86).
Copyright (C) 1995, Visual Numerics, Inc.
All rights reserved. Unauthorized reproduction prohibited.
% Windows NT version 3.51 build 1057

PV-WAVE v6.02 Build #408 (Sun Jan 28 22:51:34 MST 1996)

% Compiled module: ADD.
% Compiled module: SQUARE.
% At $MAIN$ .
Code area used: 0.00% (0/768), Data area used: 0.00% (0/8000)
# local variables (including 0 parameters): 0/500
# common symbols: 0/0
Saved Procedures:
      ADD          SQUARE
Saved Functions:
WAVE>
```

Figure B-1 The appearance of a `PV-WAVE` Console window when a startup file is being used. The messages that get printed to the screen vary depending on the exact contents of the startup file.

TIP When you create a new startup file, start with a copy of `PV-WAVE`'s default startup file, `wavestartup`. This way, you will still define a default `WAVE>` prompt and default behavior for function keys. Also, you will not inadvertently disable the setups for a `PV-WAVE` feature, e.g., the `PV-WAVE` gallery, that you might want to use later.

WAVE_FEATURE_TYPE: Setting the Default Operating

Mode

The environment variable `WAVE_FEATURE_TYPE` lets you set the default operating mode of PV-WAVE to “runtime”. When this environment variable is set to RT, compiled PV-WAVE applications can be executed directly from the operating system prompt without using the `-r` option. For example:

```
C:\ set WAVE_FEATURE_TYPE=RT
```

Set the environment variable.

```
C:\ wave somerset
```

Run a compiled, saved PV-WAVE application called somerset.

WAVE_RT_STARTUP: Using a Startup Procedure in Runtime Mode

`WAVE_RT_STARTUP` points to the name of a compiled procedure file that is executed when PV-WAVE initializes in runtime mode. The startup file may contain saved, compiled PV-WAVE routines that are executed each time PV-WAVE is started in runtime mode.

Windows USERS For more information on saving and using compiled routines, see [Runtime Mode for Windows on page 15](#).

The default startup filename that PV-WAVE looks for when it is running in runtime mode is:

```
%WAVE_DIR%\LIB\STD\rtwavestartup.cpr
```

WAVE_INIT_CODESIZE: Setting Initial Size of the Code Area

The environment variable `WAVE_INIT_CODESIZE` lets you set the initial size of the code area for PV-WAVE. For example:

```
C:\ set WAVE_INIT_CODESIZE 2000000
```

Set the initial size of the code area to 2 MB.

WAVE_INIT_LVARS: Setting Initial Value for Number of Local Variables

The environment variable `WAVE_INIT_LVARS` lets you set the initial number of local variables for PV-WAVE. For example:

```
C:\ set WAVE_INIT_LVARS 400
```

Set the initial number of local variables to 400.

Changing the PV-WAVE Prompt

The text string PV-WAVE uses to prompt you for input is specified by the system variable !Prompt. You can change the prompt by setting this system variable to the new prompt string. The prompt is currently defined in the file wavestartup.

Here's an example showing how to tailor your prompt to display text:

```
!Prompt = 'Hello World!> '
```

Here's another example that causes PV-WAVE to echo the text string, plus ring the bell on the terminal when prompting:

```
!Prompt = 'Hello World!> ' + STRING(7B)
```

The ASCII code for the bell is 7. It does not have a printable representation, so it is specified using the value 7.

Windows USERS See [Representing Nonprintable Characters with Windows on page 24](#).

TIP You can also place a prompt definition in your WAVE_STARTUP file; this file is described in [WAVE_STARTUP: Using a Startup Command File](#) on page B-17 of this guide.

For an alternate way to modify the prompt, see the description of the PROMPT procedure in the *PV-WAVE Reference*.

Programmer's Guide Index

A

- ABS function 22
- absolute value 22
- access mode, for VMS 205
- actual parameters 218–220
- addition operator 38
 - See also operators
- AND operator 41
 - See also operators
- annotation
 - See also string processing; strings; fonts
- arithmetic errors, checking for 245
- arithmetic operations, overflow condition 21
- arrays
 - accessing 260
 - assignment statements 50–51
 - associative 104
 - * (asterisk) in notation 75
 - columns in 40
 - combining 78–81
 - concatenation operators 40
 - degenerate dimensions 76
 - eliminating unnecessary dimensions 73
 - extracting fields from 34
 - functions to create from various data types 36
 - indexed by rows and columns 71
 - lists 104
 - masking 45
 - multidimensional subscripts 72
 - non-existent element, referencing 72
 - of structures 99
 - reading
 - data into 168, 169
 - rows in 40
 - storing elements with array subscripts 81
 - structures with 97–99
 - subarrays 76
 - subscripts
 - multidimensional 72
 - of other arrays 76–78
 - storing elements 81
 - subsetting with relational operators 44
 - transposition of 40
- ASCII
 - See also input/output
 - fixed format I/O 148, 149
 - formatted files 139
 - free format 148–149
 - row-oriented files 140
- assignment operator 38
 - See also operators
- assignment statements 48–53
- ASSOC function 145, 147
- associated variables
 - advantages 195
 - description of 26
 - efficiency of 194
 - example 195–196
 - exporting image data with 176
 - in assignment statements 53
 - in relation to UNIX FORTRAN programs 199
 - writing into 198
- associative arrays, defining 104
- attributes
 - of an expression 251
 - of variables 26
 - VMS record 206–207

B

batch files. See command files

BEGIN statement [53](#)
 binary data
 See also input/output
 advantages and disadvantages of [144](#)
 comparison with ASCII, input/output [145](#)
 efficiency of [174](#)
 procedures, input/output [147](#), [174](#)
 reading FORTRAN files [204](#)
 record length [142](#)
 record-oriented [143](#)
 strings, transferring [182](#)
 UNIX vs. FORTRAN [183](#)
 VMS [143](#)
 block-mode file access (VMS) [205](#)
 blocks of statements [53–55](#), [64](#)
 breakpoints
 clearing in debugger [282](#)
 setting in debugger [281](#)
 byte
 See also BYTE function; data types
 a basic data type [19](#), [25](#)
 converting characters to [33](#), [116](#)
 reading from XDR files [190](#)
 BYTE function [34](#), [117](#), [183](#)

C

carriage control
 VMS FORTRAN, table of [207](#)
 VMS output [206](#)
 case folding [117](#)
 CASE statement [55](#)
 changing the PV-WAVE prompt [B-20](#)
 characters
 See also annotation; fonts; strings; string
 processing
 extracting [120](#)
 in regular expressions [125](#)
 non-printable, specifying [24](#), [116](#)
 non-string arguments to string routines
 [122](#)
 CHECK_MATH function [33](#), [244](#), [245](#)
 clipboard
 command line functions [211](#)
 menu controls [211](#)
 used to exchange image data [210](#)
 closing
 files [135](#)
 code area, PV-WAVE [224](#)

code size
 setting at startup [17](#)
 color
 images [175](#)
 pseudo [175](#)
 represented by shades of gray [175](#)
 TIFF files [178](#)
 color tables
 saving in TIFF file [178](#)
 column-oriented data
 ASCII files [139](#)
 FORTRAN write [168](#)
 in arrays [40](#)
 transposing with rows [40](#)
 command files
 creating and running [7](#), [8](#)
 differ from main programs [7](#)
 executing at startup [B-5](#), [B-17](#)
 command line editing [271](#)
 command recall
 with INFO command [271](#)
 comment, adding a [48](#)
 common block
 COMMON statement [56](#), [230](#)
 creating common variables for
 procedures [229](#)
 use of variables [56](#)
 compiled files. *See* runtime mode
 compiling
 See also executive commands;
 programming; runtime mode
 automatically [5](#), [61](#), [218](#), [223](#)
 interactively [223](#)
 procedures and functions [222](#)
 .RUN with filename [17](#), [222](#)
 run-time [253](#)
 saving compiled procedure [13](#), [15](#)
 system limits [224](#)
 with EXECUTE function [253](#)
 complex
 constants [22](#)
 data type [19](#), [25](#), [141](#)
 double-precision type [19](#), [25](#)
 numbers [22](#)
 variables, importing data into [152](#)
 COMPLEX function [34](#)
 concatenating strings [114](#)
 concatenation operators [40](#)
 See also operators

- constants
 - types of [20](#)
 - using correct data types [259](#)
- conversion character. *See* format strings
- conversion functions [32](#)
- converting
 - data to
 - string type [115](#)
 - mixed data types in expressions [259](#)
 - radians to degrees [28](#)
 - strings, case of [117](#)
- cpr files. *See* runtime mode
- customizing PV-WAVE
 - prompt string [B-20](#)
 - your environment (Windows) [B-9](#)

D

- data
 - See also* data types; extracting data; files; input/output; reading; writing
 - CSV format [147](#)
 - drop-outs [52](#)
 - files
 - pointing to with WAVE_DATA [B-15](#)
 - where to store [B-15](#)
 - fixed format I/O [146](#), [155](#), [163](#)
 - floating-point, with format reversion [A-5](#)
 - formatting
 - into strings [115](#)
 - strings [A-1–A-2](#)
 - with STRING function [183](#)
 - free format
 - ASCII I/O [151](#), [153](#), [155](#)
 - output [155](#)
 - logical records [142](#)
 - record-oriented [144](#)
 - row-oriented [140](#)
 - sorting tables of formatted data [163](#)
 - types of [19](#)
 - unformatted [179](#)
- data area, PV-WAVE [225](#)
- data types
 - combining in arrays [36](#)
 - constants [19](#)
 - eight basic [31](#)
 - mixed, in
 - expressions [32](#)
 - relational operators [43](#)
 - of variable, determining [251](#)
- !Data_Dir system variable [B-16](#)
- date/time data
 - reading into PV-WAVE [160](#)
 - transfer with DC_READ_FIXED [163](#)
 - transferring with templates [158–159](#)
- DC (Data Connection) functions
 - advantages of [146](#)
 - for simplified data connection [146](#)
 - opening, closing files [135](#)
- DC_READ_FIXED function [150](#), [163–171](#), [173](#), [A-7](#)
- DC_READ_FREE function [132](#), [149](#)
- DC_READ_TIFF function [147](#), [177](#)
- DC_READ_24_BIT function [147](#)
- DC_READ_8_BIT function [147](#)
- DC_WRITE_FIXED function [150](#)
- DC_WRITE_FREE function [132](#), [149](#)
- DC_WRITE_TIFF function [147](#), [177](#)
- DC_WRITE_24_BIT function [147](#)
- DC_WRITE_8_BIT function [147](#), [177](#)
- deallocating
 - file units [202](#)
 - LUNs [135](#), [138](#)
- debugger [275](#)
- DECLARE FUNC statement [60](#)
- declaring functions [60](#)
- DECStation 3100, error handling [247](#)
- degrees
 - convert from radians [28](#)
- DELSTRUCT procedure [92](#)
- DELVAR procedure [92](#)
- device drivers
 - getting information about [269](#)
- DIB data
 - input/output [211](#)
- digital gradient function [220](#)
- dimensions of expressions, determining [251](#)
- directory path
 - !Path [9](#)
- disappearing variables [231](#), [238](#)
- \$ (dollar sign) in command files [8](#)
- double complex
 - data type [19](#), [25](#)
- double-precision
 - complex data type [19](#)
 - constants [21](#)
 - data type [19](#)

dynamic
 data types [32](#)
 definition of environment variables
 (Windows) [B-15](#)
 structures [32](#)

E

editor, in debugger [281](#)
8-bit image data
 how stored [175](#)
ELSE
 in CASE statement [55](#)
 in IF statement [63](#)
EMF files
 input and output [211](#)
END statement [53](#)
ENDCASE, end statement [55](#)
ENDIF, end statement [63](#)
ENDREPEAT statement [55](#)
ENDWHILE statement [55](#)
environment
 modifying (UNIX/OpenVMS) [B-1](#)
 modifying (Windows) [B-9](#)
environment variables
 changing [B-1](#)
 changing (Windows) [B-9](#)
 don't seem to be defined properly
 (Windows) [B-15](#)
 VNI_DIR (Windows) [B-14](#)
 WAVE_DATA (Windows) [B-15–B-16](#)
 WAVE_DEVICE [B-1–B-2](#)
 WAVE_DEVICE (Windows) [B-16](#)
 WAVE_DIR [B-2](#)
 WAVE_DIR (Windows) [B-14](#)
 WAVE_FEATURE_TYPE [B-6](#)
 WAVE_INIT_CODESIZE [B-7, B-19](#)
 WAVE_INIT_LVARS [B-7, B-19](#)
 WAVE_PATH [B-3–B-4, B-13–B-14](#)
 WAVE_RT_STARTUP [B-6](#)
 WAVE_STARTUP [B-5–B-6](#)
 WAVE_STARTUP (Windows) [B-17, B-19](#)
 Windows [B-11](#)
EOF function [160, 200](#)
EQ operator [45](#)
 See also operators
equality operator [45](#)
 See also operators
error handling

See also debugger; math errors
accumulated math error status [244](#)
arithmetic [243](#)
categories of [237](#)
DECStation 3100 [247](#)
during program execution [238](#)
enabling/disabling math traps [246](#)
floating-point [243](#)
input/output [135, 240](#)
list of procedures [237](#)
machine dependent handling [247](#)
mechanism in procedures and
 functions [230](#)
message
 issuing [231, 241–242](#)
OpenVMS [248](#)
options for recovery [238](#)
overflow [245](#)
recovery from [230](#)
running SunOS Version 4 [247](#)
SGI IRIX 5.3 workstations [248](#)
Sun-3 and Sun-4 [247](#)
escape sequences, creating [24](#)
Excel. *See* Microsoft Excel
exclusive OR operator [42](#)
 See also operators
executing
 application in debugger [277, 283](#)
 command files [8](#)
 commands from a function [226, 253](#)
 interactive programs [3](#)
 main programs [6](#)
executive commands
 .LOCALS [226](#)
 ..LOCALS compiler directive [226](#)
 .RUN
 compiling functions and proce-
 dures [222](#)
 syntax [222](#)
 .SIZE [225](#)
explicitly formatted. *See* fixed format
exponentiation operator [39](#)
 See also operators
exporting data. *See* data; writing
expressions
 efficiency of evaluation [258](#)
 evaluation of [32](#)
 finding attributes of [251](#)
 forcing to specific data types [33](#)

- general description [29](#)
- invariant [260](#)
- structure of [35–37](#)
- type and structure of [31, 35](#)
- with mixed data types [259](#)

extracting data from variables [34–35](#)

F

files

- access mode, VMS [205](#)
- allocating units [134, 138](#)
- closing [135](#)
- column-oriented ASCII [139](#)
- CSV format [147](#)
- deallocating LUNs [135, 138](#)
- fixed format I/O [155](#)
- formatting, codes for [A-8–A-21](#)
- free format ASCII I/O [148](#)
- getting information about [200, 270](#)
- indexed files for VMS [207](#)
- I/O with structures [100](#)
- locating on disk [199](#)
- locating on disk (Windows) [B-15](#)
- logical records, changing size of [142](#)
- logical units [134](#)
- opening [134–135](#)
- organization of VMS [204](#)
- organization options [139](#)
- pointer, positioning [200](#)
- portable binary [188](#)
- record-oriented binary files [143](#)
- row-oriented [140](#)
- standard input, output and error [135](#)
- startup command (Windows) [B-17](#)
- testing for end of file [200](#)
- UNIX [203](#)
- VMS [198, 205–207](#)
- XDR [189](#)

FINDFILE function [199](#)

FINITE function [33, 244](#)

FIX function [33–34](#)

fixed format

- ASCII I/O [148, 149](#)
- choosing [148](#)
- comparison with free format [146](#)
- data, examples of reading [164](#)
- I/O [155, 163](#)

fixed-length record format, VMS [198, 205](#)

FLOAT function [34](#)

floating-point

- constants [21](#)
- data type [19, 25](#)
- errors [243](#)

FLUSH procedure [150, 199](#)

flushing

- file units [199](#)

FOR statement [57–60](#)

force fields

- examples [58](#)
- increment parameter [58](#)
- simple [58](#)

formal parameters

- copying actual parameters into [219](#)
- definition of [218](#)

format reversion [A-4](#)

format strings

- C conversion characters [A-19–A-20](#)
- C format codes [A-19–A-21](#)
- definition of [A-1](#)
- discussion of [155–158](#)
- for data transfer [156](#)
- FORTRAN format codes [171, A-8–A-18](#)
- group repeat specifications [A-6](#)
- how to use [A-2](#)
- interpreting [156](#)
- reading multiple array elements [171](#)
- reversion [157, A-4](#)
- when to use [156](#)

formatted data

- rules for [151, 156](#)
- strings [115](#)
- structures for I/O [101](#)
- using STRING function [173](#)

FORTRAN programming language

- binary data, reading with PV-WAVE [204](#)
- format strings [171, A-8–A-18](#)
- on UNIX [183](#)

free format

- ASCII I/O [148–149](#)
- input [151–152](#)
- output [155](#)

FREE_LUN procedure [138](#)

FSTAT function [201–202](#)

FUNCTION definition statement [218](#)

function keys

- equating to character strings [B-8](#)
- getting information about [271](#)
- functions
 - compiling with .RUN and filename [17](#)
 - copying actual parameters into formal parameters [219](#)
 - creating
 - interactively [4](#)
 - with a text editor [5](#)
 - declaring [60](#)
 - defining [218](#)
 - discussion of [217](#)
 - keyword parameters [61](#)
 - libraries of VMS [233](#)
 - parameters [218](#)
 - actual [218](#)
 - passing mechanism [228](#)
 - positional [61](#)
 - search path [B-3–B-6](#), [B-13–??](#)
 - type conversion [32](#)
 - user-defined [224](#)

G

- GE operator [45](#)
 - See also* operators
- GET_KBRD function [148](#), [203](#)
- GET_LUN procedure [138](#)
- GOTO statement [48](#), [62](#)
- graphics
 - device driver, changing [B-16](#)
- greater than [30](#)
 - See also* operators
- greater than or equal [30](#)
 - See also* operators
- group repeat specifications [A-6](#)
 - See also* format strings
- GT operator [45](#)
 - See also* operators

H

- help, online
 - file information [200](#)
 - information, on a session [267](#)

I

- IF statement
 - avoiding [256](#)
 - discussion of [62–64](#)
- images
 - 8-bit format [175](#)
 - how stored [175](#)
 - interleaving [176](#), [178](#)
 - output [176](#)
 - reading
 - associated variable method [195](#)
 - block of data from tape [210](#)
 - storing data [175](#)
 - 24-bit format [175](#)
- IMAGINARY function [22](#)
- inclusive OR operator [42](#)
 - See also* operators
- indexed files, VMS [207](#)
- INFO procedure
 - detailed examples [267](#)
- input/output
 - See also* error handling; format strings; reading; writing
 - ASCII, pros and cons of [145](#)
 - associated variables [194](#)
 - binary data
 - portability [189](#)
 - procedures [174](#)
 - pros and cons of [145](#)
 - record-oriented files [143](#)
 - string variables [182](#)
 - choosing a method [145](#)
 - DIB data [211](#)
 - EMF data [211](#)
 - fixed formats [155](#)
 - fixed vs. free format data [146](#)
 - free format [151](#)
 - image data [175](#)
 - portable binary [189](#)
 - PV-WAVE and C program [179](#)
 - record-oriented binary files [143](#)
 - strings with structures [102](#)
 - structures [100](#)
 - unformatted
 - associated variable [194](#)
 - discussion [151](#)
 - in structures [101](#)
 - string variables [152](#), [181](#)

- VMS binary files [143](#)
- when to open a file [135](#)
- XDR files [188](#)
- integer
 - constants [20](#)
 - conversions, errors in [245](#)
 - data type [19, 25](#)
 - data, shown in figure [141](#)
 - output, for format codes [A-14](#)
 - syntax of constants [20](#)
 - writing with format reversion [A-4–A-5](#)
- interapplication communication
 - C programs
 - creating XDR files [191](#)
 - reading files with [181](#)
 - writing to PV-WAVE [179](#)
- interleaving
 - description of [176](#)
 - image data [176, 178](#)
 - pixels [178](#)
- invariant expressions, removing from loops [260](#)

J

- journaling
 - description of [10](#)
 - examples [11](#)
 - in relation to PRINTF [11](#)

K

- keyboard
 - accelerators [B-8](#)
 - bindings used in debugger [281](#)
 - defining keys [B-8](#)
 - getting input from [203](#)
 - show current bindings [271](#)
- keywords
 - checking for presence of [250](#)
 - definition of [219](#)
 - examples [221](#)
 - functions, using with [61](#)
 - passing of [219](#)
- KEYWORD_SET function [249](#)

L

- LE operator [30, 45](#)
 - See also* operators
- less than [30, 45](#)
 - See also* operators
- less than or equal [30, 45](#)
 - See also* operators
- libraries
 - creating and revising for VMS [234–235](#)
 - Standard (std) [231](#)
 - Users' [231–233](#)
 - VMS procedure, searching [233](#)
- linear algebra
 - rules [39](#)
- lists, defining [104](#)
- local variables
 - setting number of at start up [17](#)
- local variables, definition of [221](#)
- .LOCALS [226](#)
- ..LOCALS [226](#)
- logical operators [43](#)
 - See also* operators
- logical records [142](#)
- logical unit number. *See* LUNs
- longword
 - data type [19](#)
 - integer, on Digital UNIX [188](#)
- loops. *See* statements
- LT operator [30, 45](#)
 - See also* operators
- LUNs
 - getting information using FSTAT [201](#)
 - operating system dependencies [137](#)
 - reserved [135–136](#)
 - to open and close files [134](#)
 - use of [136–138](#)

M

- magnetic tape
 - accessing under VMS [208](#)
- main
 - programs, executing [6](#)
 - program, definition of [6](#)
 - program, differs from command file [7](#)
 - program, re-using [7](#)

- Visual Numerics directory (Windows) [B-14](#)
- masking
 - arrays [45](#)
- math errors
 - See also* error handling
 - accumulated math error status [244](#)
 - detection of [243](#)
 - hardware-dependent [247](#)
 - procedures for controlling [237](#)
 - traps, enabling/disabling [246](#)
- matrix
 - expressions [88](#)
 - multiplication [39](#), [84](#)
 - printing
 - interactively [84](#)
 - reading
 - from a file [86](#)
 - interactively [84](#)
 - subarrays [87](#)
 - subscripts [72](#)
- maximum operator [44](#)
 - See also* operators
- memory
 - See also* virtual memory
 - allocation [226](#)
 - order and arrays [260](#)
 - physical [261](#)
- message
 - error [231](#), [241](#)–[242](#)
- Microsoft Excel
 - importing data from PV-WAVE [213](#)
 - transferring spreadsheet data to PV-WAVE [213](#)
- minimum operator [44](#)
 - See also* operators
- modulo operator [40](#)
 - See also* operators

N

- NE operator [45](#)
 - See also* operators
- nested procedures, getting information on [273](#)
- non-printable characters [24](#)
- not equal [30](#)
 - See also* operators
- NOT operator [42](#)
 - See also* operators
- N_ELEMENTS function [250](#)

- N_PARAMS function [249](#)
- N_TAGS function [103](#)

O

- ON_ERROR procedure [231](#), [237](#)
- ON_IOERROR procedure [237](#), [240](#)
- opening files. *See* files
- OPENR procedure [134](#)
- OPENU procedure [134](#)
- OPENW procedure [134](#)
- operands, checking validity of [244](#)
- operating system
 - See also* interapplication communication; UNIX operating system; VMS operating system; Windows operating system
- operators
 - Boolean [41](#)
 - general description [29](#)
 - grouping of expressions [38](#)
 - hierarchy of [30](#)
 - list of [37](#)
 - precedence [30](#)
 - relational [43](#)
 - using with arrays [44](#)
- OPI applications
 - bindings [307](#)
 - creating [295](#)
 - error handling [345](#)
 - example [293](#)
 - keyword processing [302](#)
 - license management [303](#)
 - loading [288](#)
 - required files [291](#)
 - runtime mode [16](#)
 - summary of variable-handling routines [311](#)
 - unloading [288](#)
 - variable handling [311](#)
- Option Programming Interface. *See* OPI applications
- OR operator [42](#)
 - See also* operators
- overflow
 - See also* error handling
 - checking for in integer conversion [245](#)
 - in type conversions [33](#)

with integer arithmetic 21

P

padding

bytes 101

page faults 261

page file quota 265

parameters

See also keywords

actual 218

checking for

number of 249

copying 219

formal to actual correspondence 219

number required 220

passing

by reference 228

by value 228

mechanism 97, 218, 228

positional 219

PARAM_PRESENT function 249, 251

!Path system variable B-10

Pgflquo 264

pixels

interleaving 176, 178

plotting

!P system variable 28

POINT_LUN procedure 200, 205

portable data, XDR 188–189

PRINT procedure 149

PRINTF procedure 11, 149

printing

formatted data 173

PRO statement 218

significance of 6

Procedure Call statement 64

procedures

actual parameters 218

automatic compiling, conditions for 61

compiling 222

compiling with .RUN and filename 17

creating with editor 5

discussion of 217

libraries of VMS 233

parameters 218

sources of 64

program

call stack 242

code area full 224

control routines 237, 252

creating 5

data area full 225

declaring functions 60

determining number of parameters
249

file search method B-4, B-10

formal parameters 218

format of 6, 7

increasing speed of 255

information on a 272

list of compiled 272

main PV-WAVE 6

maximum size of code 268

nested, information on 273

number of

parameters 249

required parameters 220

required components 220

running as batch 8

search path 9, B-3, B-13

submitting to Users' Library 232

user-written 61

programming

See also debugger; error handling;

WAVE Widgets; Widget Toolbox

accessing large arrays 260

code size 268

commenting programs 48

efficiency 256

format strings A-1–A-2

tips 255–266

virtual memory

minimize allocation 265

running out of 262

!Prompt system variable B-20

prompt, changing B-7, B-20

pseudo-color

images 175

PV-WAVE session

customizing (Windows) B-9

how affected by environment

variables (Windows) B-15

recording 10

R

radians, converting to degrees 28

random file access [205](#)
 READF procedure [149](#), [164](#), [169](#), [172](#)
 reading
 binary files [180](#)
 binary files between different systems [188](#)
 byte data from an XDR file [190](#)
 C-generated XDR data [191](#)
 CSV data [147](#), [213](#)
 date/time data [153](#), [160](#)
 DC_READ routines [A-7](#)
 DIB data [211](#)
 EMF data [211](#)
 files, using C programs [181](#)
 fixed-format ASCII data [163](#)
 freely-formatted ASCII data [149](#)
 from magnetic tapes [208](#)
 into complex variables [152](#)
 into structures [153](#)
 keyboard input [203](#)
 multiple array elements [168](#)
 multiple array elements with FORTRAN
 format string [171](#)
 records with multiple array elements [168–171](#)
 row-oriented FORTRAN written data [170](#)
 tables of formatted data [163](#)
 unformatted data [179](#)
 word-processing data [164](#)
 XDR files [192](#)
 8-bit image data [175](#)
 READU procedure [147](#)
 record attributes of VMS files [206](#)
 record-oriented, binary files [144](#)
 records
 definition of [142](#)
 extracting fields from [34](#)
 fixed length format (VMS) [198](#), [205](#)
 length of [142](#)
 multiple array elements [168–172](#)
 recovering from errors [238](#)
 registry [B-13](#)
 registry, Windows [B-11](#)
 regular expressions [123–129](#)
 relational operators [43](#)
 See also operators
 REPEAT statement [67](#)
 REPLICATE function [99](#), [180](#)
 reserved LUNs. *See* LUNs
 reserved words [27](#)

RETALL procedure [231](#)
 RETURN procedure [61](#), [220](#), [231](#)
 reversion, format [157](#), [A-4](#)
 REWIND procedure [208](#)
 RGB
 triplets [178](#)
 RMS files, reading images [198](#)
 row-oriented
 ASCII data [140](#)
 FORTRAN write [170](#)
 .RUN [222](#)
 See also executive commands
 running. *See* executing
 runtime mode
 compilation of statements [12](#)
 compiling procedures for [13](#)
 definition of [12](#)
 developing applications [14](#)
 OPI applications [16](#)
 search path [14](#)
 starting PV-WAVE in [13](#), [15](#)

S

saving
 TIFF data [177](#)
 scalars
 arrays, relation to [266](#)
 combining with subscript arrays or
 ranges [80](#)
 definition of [26](#)
 subscripting [74](#)
 searching
 for VMS libraries [233](#)
 semicolon after @ symbol [9](#)
 set command
 for WAVE_DEVICE [B-16](#)
 for WAVE_FEATURE_TYPE [B-19](#)
 for WAVE_STARTUP [15](#), [B-18](#)
 setenv UNIX command
 for WAVE_DEVICE [B-2](#)
 for WAVE_PATH [B-3](#)
 for WAVE_STARTUP [B-5](#)
 setup program, for PV-WAVE (Windows)
 [B-15](#)
 Silicon Graphics IRIX 5.3, error handling
 [248](#)
 .SIZE [225](#)
 SIZE function [251–252](#)

- SKIPF procedure [208](#)
- sorting
 - tables [167–168](#)
- source file. *See* debugger
- spheres
 - See also* rendering
- standard error output [135](#)
- standard input [135](#)
- Standard Library
 - location of [3](#), [232](#)
 - suggestions for writing routines [232](#)
- standard output [135](#)
- starting PV-WAVE
 - flag for code size [17](#)
 - flag for number of local variables [17](#)
 - from program group [B-15](#)
 - not working as expected [B-15](#)
- startup file
 - for UNIX [B-5](#)
 - for VMS [B-6](#)
 - for Windows [B-17](#)
- statements
 - assignment [48–49](#)
 - blocks of [53](#)
 - CASE [55](#)
 - changing data types [50](#)
 - COMMON block [56](#)
 - compiling and executing with EXECUTE
 - function [253](#)
 - components of [47](#)
 - GOTO [62](#)
 - IF [62](#), [63](#)
 - labels [48](#)
 - list of 12 types [47](#)
 - procedure calls and definition [64](#)
 - REPEAT [67](#)
 - runtime compilation [12](#), [13](#)
 - spaces in [48](#)
 - tabs in [48](#)
 - types of [47](#)
- stream mode files, VMS [206](#)
- STRING function [34](#), [173](#)
- string processing
 - extracting substrings [120](#)
 - inserting substrings [114](#), [120](#)
 - locating substrings [120–121](#)
 - non-string arguments [122](#)
 - obtaining length of strings [119](#)
 - removing white space [114](#), [118](#)
 - working with text [113](#)
- strings
 - See also* annotation; fonts; string processing
 - basic data type [25](#)
 - binary transfer of [181](#)
 - concatenating [114](#)
 - constants [22](#)
 - converted from byte [116](#)
 - converting to byte [183](#)
 - definition of [113](#)
 - determining length of [114](#)
 - examples of string constants [22](#)
 - formatting [114](#)
 - FORTTRAN and C formats [156](#)
 - importing with free format [152](#)
 - initializing to a known length [183](#)
 - input/output with structures [102](#)
 - length issues with structures [102](#)
 - operations supported [113–114](#)
 - substrings [120](#)
 - used in structures [102](#)
 - writing to a file [155](#)
- STRLEN function [114](#)
- STRLOWCASE function [117](#)
- STRMID function [121](#)
- STRPOS function [120](#)
- STRPUT procedure [121](#)
- STRTRIM function [118–119](#)
- STRUCTREF function [91](#)
- structures
 - advanced usage [103](#)
 - arrays [97](#), [99](#)
 - associative arrays [104](#)
 - data type [25](#)
 - defining [90](#)
 - deleting [90–91](#)
 - formatted input/output [101](#)
 - getting information about [96](#), [272](#)
 - importing data into [153](#)
 - input and output [100](#)
 - lists [104](#)
 - number of tags in [103](#)
 - passing as parameters [97](#)
 - references [94–95](#)
 - replicating [99](#)
 - scope of named and unnamed [92](#)
 - string
 - input/output [102](#)

- length issues [102](#)
- subscripted references [94](#)
- tag names
 - reference to a field [94](#)
- unformatted input/output [101](#)
- unnamed, creating [92](#)
- writing [101](#)
- STR_TO_DT function [161](#)
- subarrays
 - See also* arrays; subsetting
 - structure of [76](#)
 - subscript range use [87](#)
- submatrices
 - See also* matrix; subsetting
 - subscript range use [87](#)
- subscripts
 - arrays of [52](#)
 - * (asterisk) operator [75](#)
 - combining arrays with [79–81](#)
 - matrices, use in [72](#)
 - multidimensional arrays [72](#)
 - notation for columns and rows [71](#)
 - ranges [51](#)
 - list of 4 types [74–75](#)
 - summary table of [75](#)
 - to select subarray [74](#)
 - reference syntax of [71](#)
 - scalars, use with [74](#)
 - storing elements with [81](#)
 - structure references [94](#)
 - subscript arrays [49](#)
- subsetting
 - relational operators, use of [44](#)
 - subarrays, selecting ranges of [74–78](#)
- subtraction operator. *See* operators
- SunOS Version 4, error handling [247](#)
- swap area. *See* virtual memory
- SYSGEN parameters
 - VIRTUALPAGECNT [264](#)
 - WSMAX [264](#)
- system variables
 - definition of [28](#)
 - getting information about [273](#)
 - passing [228–229](#)
 - values of [273](#)

T

tables

- examples [167](#)
- sorting [167–168](#)
- tabs in statements. *See* statements
- tag
 - See also* data types; unnamed structures
 - definition of [89](#)
 - field reference in structure [94](#)
 - names in a structure [103](#)
 - numbers of [103](#)
- TAG_NAMES function [103](#)
- tape, magnetic
 - accessing under VMS [208](#)
 - end of file mark [208](#)
 - mounting a tape drive (VMS) [209](#)
 - reading from [208–210](#)
 - rewinding [208](#)
 - skipping records or files [208, 209](#)
 - writing to [208](#)
- TAPRD procedure [208](#)
- TAPWRT procedure [208](#)
- TeX documents. *See* LaTeX documents
- text. *See* annotation; characters; fonts; string processing; strings
- TIFF
 - compression of files [177](#)
 - conformance levels [178](#)
 - reading a file in [177](#)
 - saving data in TIFF format [177](#)
- TOTAL function [40](#)
- traceback information [242](#)
- transferring data. *See* input/output; reading; writing
- TRANSPOSE function [40](#)
- transposing
 - rows and columns [40](#)
- 24-bit image data
 - storing [175](#)
- types
 - See also* data types [31, 35](#)

U

- unformatted data
 - See also* input/output
 - advantages/disadvantages of [144](#)
 - input/output [101, 151, 152, 181, 194](#)
 - reading

- associated variable method [194](#)
 - FORTRAN generated in UNIX [183](#)
 - FORTRAN generated in VMS [186](#)
 - problems in [189](#)
 - routines for transferring [149](#)
- UNIX operating system
 - description of files in [203–204](#)
 - environment variable
 - WAVE_DIR [B-2](#)
 - FORTRAN programs with ASSOC [199](#)
 - reading data in relation to FORTRAN [183](#)
 - reserved LUNs [137](#)
 - virtual memory [263](#)
 - writing FORTRAN programs to PV-WAVE [184](#)
- unnamed structures
 - creating [92–94](#)
 - uses for [92](#)
- updating a file, using OPENU [133](#)
- Users' Library
 - documentation for [233](#)
 - location of [231](#)
 - submitting programs to [232](#)
 - support for routines in [233](#)

V

- variable length record format files, VMS [206](#)
- variables
 - associated [26](#), [53](#), [198](#)
 - attributes [25](#)
 - checking for undefined [250](#)
 - complex, importing data [152](#)
 - data types [26](#)
 - definition of [24](#)
 - determining
 - data type of [251](#)
 - number of elements in [250](#)
 - disappearing [231](#), [238](#)
 - examining in debugger [283](#)
 - forcing to specific data type [33](#)
 - in common blocks [56](#)
 - local, definition of [221](#)
 - naming conventions [26](#)
 - size and type information [26](#), [251](#)
 - structure of [26](#)
 - system. *See* system variables
 - types of [26](#)
- vector

- definition of [26](#)
 - subscripts [76](#)
 - using as subscripts to other arrays [78](#)
- vertex lists
 - See also* polygons
- virtual memory
 - description of [261](#)
 - in relation to PV-WAVE [261](#)
 - in relation to swap area [263](#)
 - minimizing use of [265](#)
 - swap area, increasing [263](#)
 - UNIX [263](#)
 - variable assignments [262](#)
 - VMS [263](#)
- virtual page count parameter [264](#)
- VMS operating system
 - access mode [205](#)
 - accessing magnetic tape [208](#)
 - binary data [143](#)
 - data files, information on [204](#)
 - error handling [248](#)
 - files [204–207](#)
 - formal parameters for procedures and functions [220](#)
 - FORTRAN programs, writing [186](#)
 - libraries [233–234](#)
 - mounting a tape drive [209](#)
 - offset parameter [198](#)
 - record attributes [206](#)
 - record-oriented data, transferring [144](#)
 - reserved LUNs [137](#)
 - RMS files, reading images [198](#)
 - setting WAVE_DIR [B-2](#)
 - stream mode files [206](#)
 - variable length format [206](#)
 - virtual memory [263](#)
 - working set size [264](#)
- VNI_DIR
 - description of [B-15](#)
 - UNIX system [B-3](#)
 - Windows system [B-14](#)

W

- wavestartup file [B-5](#)
- wavestartup file (Windows) [B-17](#)
- wave_assign_num [317](#)

- wave_assign_string [317](#)
- wave_assign_struct [317](#)
- wave_compile [314](#)
- WAVE_DATA [B-15–B-16](#)
- WAVE_DEVICE [B-1–B-2](#), [B-16](#)
- WAVE_DIR [B-2](#), [B-14](#), [B-15](#)
 - OpenVMS system [B-2](#)
 - UNIX system [B-2](#)
 - Windows system [B-14](#)
- wave_execute [313](#)
- wave_free_WCH [317](#)
- wave_free_wsdh [337](#)
- wave_free_WVH [323](#)
- wave_get_unWVH [322](#)
- wave_get_WVH [321](#)
- wave_interp [316](#)
- WAVE_PATH [B-3–B-4](#), [B-13–B-14](#)
- WAVE_STARTUP [B-5–B-6](#), [B-17](#)
- wave_type_sizeof [330](#)
- wave_wsdh_from_name [336](#)
- wave_wsdh_from_wvh [335](#)
- WEOF procedure [208](#)
- WHERE function [50](#), [52](#), [257](#)
- WHILE statement [68](#)
- wildcards, vs. regular expressions [128](#)
- window systems
 - defining with WAVE_DEVICE [B-1–B-2](#)
 - Windows [B-9–B-20](#)
 - X Windows [B-8](#)
- Windows
 - autoexec.bat [B-11](#)
 - environment variables [B-9](#), [B-11](#), [B-15](#)
 - program group [B-15](#)
 - registry [B-9](#), [B-11](#)
- word-processing application, reading data from [163](#)
- working set
 - description of [261](#)
 - page maximum parameter [264](#)
 - quota [265](#)
- WRITEU procedure [147](#), [180](#)
- writing
 - binary data file [180](#)
 - CSV data [147](#), [213](#)
 - DIB data [211](#)
 - EMF data [211](#)
 - flushing buffers [199](#)
 - free format data [155](#)
 - integer data, using format reversion [A-4–](#)

- [A-5](#)
 - strings to a file [155](#), [182](#)
 - to tape (VMS) [208](#)
 - unformatted data [179](#)
 - with UNIX FORTRAN programs [184](#)
 - with VMS FORTRAN [186](#)
- wsdh_element [342](#)
- wsdh_name [338](#)
- wsdh_ntags [339](#)
- wsdh_offset [341](#)
- wsdh_sizeofdata [341](#)
- wsdh_tagname [339](#)
- Wsquo quota [264](#)
- wvh_dataptr [334](#)
- wvh_dimensions [329](#)
- wvh_is_constant [333](#)
- wvh_is_scalar [332](#)
- wvh_name [324](#)
- wvh_ndims [327](#)
- wvh_nelems [328](#)
- wvh_sizeofdata [330](#)
- wvh_type [326](#)

X

- X Window System
 - using with PV-WAVE [B-8](#)
- XDR data, reading and writing [188–193](#)
- XOR operator [42](#)
 - See also operators*

Z