



PV-WAVE *Advantage*[™]
PV-WAVE Command Language[™]



Programmer's Guide

Version 4.2 ••••
2861
June 1993



Visual Numerics[™]

IMSL and Precision Visuals are now one.

Visual Numerics, Inc.

Corporate Headquarters

Suite 400, 9990 Richmond Avenue
Houston, Texas 77042
United States of America
713/784-3131
FAX: 713/781-9260

Boulder, Colorado

6230 Lookout Road
Boulder, Colorado 80301
United States of America
303/530-9000
FAX: 303/530-9329

France

33-1-42-94-19-65
FAX: 33-1-42-94-94-22
33-1-34-51-26-26
FAX: 33-1-34-51-97-69

Germany

49-211-367-7122
FAX: 49-211-367-7100

Japan

81-3-5689-7550
FAX: 81-3-5689-7553

United Kingdom

44-0753-790-600
FAX: 44-0753-790-601

Copyright ©1993 by Visual Numerics, Inc.

The information contained in this document is subject to change without notice.

VISUAL NUMERICS, INC., MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Visual Numerics, Inc., shall not be liable for errors contained herein or for incidental, consequential, or other indirect damages in connection with the furnishing, performance, or use of this material.

All rights are reserved. No part of this document may be photocopied or reproduced without the prior written consent of Visual Numerics, Inc.

Restricted Rights Legend

Use, duplication or disclosure by the US Government is subject to restrictions as set forth in FAR 52.227-19, subparagraph (c)(1)(ii) of DOD FAR SUPP 252.227-7013, or the equivalent government clause for other agencies.

Restricted Rights Notice: The version of PV-WAVE described in this document is sold under a per-machine license agreement. Its use, duplication, and disclosure are subject to the restrictions in the license agreement.

Contents Summary

Preface* *xxi

***Chapter 1: Introduction to the PV-WAVE
Command Language* *1***

Chapter 2: Constants and Variables* *17

Chapter 3: Expressions and Operators* *31

Chapter 4: Statement Types* *51

Chapter 5: Using Subscripts and Matrices* *81

Chapter 6: Working with Structures* *101

Chapter 7: Working with Text* *121

Chapter 8: Working with Data Files* *135

Chapter 9: Writing Procedures and Functions* *233

Chapter 10: Programming with PV-WAVE* *257

Chapter 11: Tips for Efficient Programming* *275

Chapter 12: Accessing the Operating System	291
Chapter 13: Interapplication Communication	305
Chapter 14: Getting Session Information	395
Chapter 15: Using WAVE Widgets	405
Chapter 16: Using the Widget Toolbox	479
Appendix A: FORTRAN and C Format Strings	A-1
Appendix B: Motif and OLIT Widget Classes	B-1
Appendix C: Motif and OLIT Callback Parameters	C-1
Appendix D: Widget Toolbox Cursors	D-1
Multivolume Index	i

Table of Contents

Preface *xxi*

Contents of this Programmer's Guide *xxi*

Typographical Conventions *xxv*

Customer Support *xxvii*

Chapter 1: Introduction to the PV-WAVE Command Language *1*

Introduction to PV-WAVE *1*

 PV-WAVE Notation *2*

 PV-WAVE Libraries *2*

 Extensive Error Checking *3*

Data Types *3*

Constants and Variables *4*

 Numeric and String Constants *4*

 Variables *4*

Expressions and Operators *6*

 Expressions *6*

 Operators *6*

Statements *9*

 Assignment *9*

 IF *9*

 FOR *9*

 WHILE *10*

 REPEAT *10*

 CASE *10*

 GOTO *10*

 Block *10*

Procedure Call 11
Procedure Definition 11
Function Call 11
Function Definition 11
Common Block Definition 12

Input and Output 12

I/O Functions for Simplified Data Connection 13

Functions and Procedures 14

Definition Syntax 15
Usage Syntax 15
Where to Find PV-WAVE's Libraries 16

Chapter 2: Constants and Variables 17

Constants 17

Numeric Constants 18
String Constants 21

Variables 23

Attributes of Variables 23
Names of Variables 25
System Variables 26

Chapter 3: Expressions and Operators 31

Operator Precedence 32

Type and Structure of Expressions 34

Type Conversion Functions 36
Extracting Fields 37

Structure of Expressions 39

Relational Operators 41

Boolean Operators 43

PV-WAVE Command Language Operators 44

Chapter 4: Statement Types 51

Components of Statements	52
Statement Labels	52
Adding Comments	52
Assignment Statement	53
Form 1	53
Form 2	54
Form 3	55
Form 4	56
Associated Variables in Assignment Statements	59
Blocks of Statements	60
CASE Statement	62
Common Block Definition Statement	63
FOR Statement	65
Form 1: Implicit Increment	65
Form 2: Explicit Increment	67
Function Definition Statement	68
Automatic Compilation of Functions and Procedures	69
GOTO Statement	70
IF Statement	70
Definition of True in an IF Statement	71
Procedure Call Statement	73
Examples	74
Positional Parameters and Keyword Parameters	74
More On Parameters	75
Procedure Definition Statement	76
REPEAT Statement	77
WHILE Statement	78

Chapter 5: Using Subscripts and Matrices 81

Example of a Subscript Reference 82

"Extra" Dimensions 83

Subscripting Scalars 84

Subscript Ranges 84

Structure of Subarrays 86

Arrays as Subscripts to Other Arrays 88

Combining Array Subscripts with Others 89

Combining with Subscript Ranges 90

Combining with Other Subscript Arrays 90

Combining with Scalars 91

Storing Elements with Array Subscripts 91

Memory Order 93

Matrices 93

Reading and Printing Matrices Interactively 93

Reading a Matrix From a File 96

Printing a Matrix to a File 97

Subarrays 98

Matrix Expressions 99

Chapter 6: Working with Structures 101

Introduction to Structures 101

Defining and Deleting Structures 102

Example of Defining a Structure 103

Deleting a Structure Definition 104

Creating Unnamed Structures 105

Structure References 108

Subscripted Structure References 108

Examples of Structure References 109

Using INFO with Structures 110

Parameter Passing with Structures	111
Storing into Structure Array Fields	112
Creating Arrays of Structures	114
Examples of Arrays of Structures	115
Structure Input and Output	116
Formatted Input and Output with Structures	116
Unformatted Input and Output with Structures	117
String Input and Output	117
String Length Issues	118
Advanced Structure Usage	119
Example of Tag Indices	119

Chapter 7: Working with Text 121

Example String Array	121
Basic String Operations	122
Concatenating Strings	123
String Formatting	124
Using STRING with Byte Arguments	125
Converting Strings to Upper or Lower Case	127
Removing White Space From Strings	128
Determining the Length of Strings	130
Manipulating Substrings	131
Using Non-string and Non-scalar Arguments	133

Chapter 8: Working with Data Files 135

Simple Examples of Input and Output 135

Opening and Closing Files 138

Opening Files 138

Closing Files 140

Logical Unit Numbers (LUNs) 141

Reserved Logical Unit Numbers (-2, -1, 0) 141

Logical Unit Numbers for General Use (1...99) 143

Logical Unit Numbers Used by GET_LUN/FREE_LUN (100...128) 143

How is the Data File Organized? 144

Column-Oriented ASCII Data Files 144

Row-Oriented ASCII Data Files 146

How Long is a Record? 147

Types of Input and Output 151

Each Type of I/O has Pros and Cons 152

Functions for Simplified Data Connection 153

Binary I/O Routines 154

ASCII I/O Routines 155

Other I/O Related Routines 158

Free Format Input and Output 159

Free Format Input 159

Free Format Output 164

Explicitly Formatted Input and Output 165

Using FORTRAN or C Formats for Data Transfer 165

Transferring Date/Time Data 168

Reading, Sorting, and Printing Tables of Formatted Data 175

Reading Records Containing Multiple Array Elements 180

Using the STRING Function to Format Data 187

Input and Output of Binary Data 188

Input and Output of Image Data 189

READU and WRITEU 193

Transferring Data with READU and WRITEU	194
Binary Transfer of String Variables	197
Reading UNIX FORTRAN-Generated Binary Data	199
Reading VMS FORTRAN-Generated Binary Data	202
External Data Representation (XDR) Files	205
Opening XDR Files	205
Transferring Data To and From XDR Files	206
PV-WAVE XDR Conventions for Programmers	210
Associated Variable Input and Output	212
Advantages of Associated File Variables	212
Working with Associated File Variables	213
How Data is Transferred into Associated Variables	214
Using the Offset Parameter	216
Writing Associated Variable Data	217
Binary Data from UNIX FORTRAN Programs	218
Miscellaneous File Management Tasks	218
Locating Files	218
Flushing File Units	218
Positioning File Pointers	219
Testing for End-of-File	219
Getting Information About Files	220
Getting Input from the Keyboard	222
UNIX-Specific Information	223
Reading FORTRAN-Generated Binary Data with PV-WAVE	224
VMS-Specific Information	224
Organization of the File	224
Access Mode	225
Record Format	226
Record Attributes	226
File Attributes	227
Creating Indexed Files	228
Accessing Magnetic Tape	229

Chapter 9: Writing Procedures and Functions 233

Procedure and Function Parameters 234

- Correspondence Between Formal and Actual Parameters 235
- Copying Actual Parameters into Formal Parameters 236
- Number of Parameters Required in Call 236
- Example Using Keyword Parameters 238

Compiling Procedures and Functions 239

- Using .RUN with a Filename 239
- Compiling Automatically 239
- Compiling with Interactive Mode 240

System Limits and the Compiler 241

- Program Code Area Full 242
- Program Data Area Full 243

Using the ..LOCALS Compiler Directive 244

- Example 1 245
- Example 2 245
- Example 3 246

Parameter Passing Mechanism 246

Procedure or Function Calling Mechanism 248

- Recursion 248
- Example Using Variables in Common Blocks 249

Error Handling in Procedures 250

- Error Signaling 251
- “Disappearing Variables” 251

VMS Procedure Libraries 251

Creating VMS Procedure Libraries 253

The Users' Library 255

- Submitting Programs to the Users' Library 255
- Support for Users' Library Routines 256

Chapter 10: Programming with PV-WAVE 257

Description of Error Handling Routines 258

Default Error Handling Mechanism 258

Controlling Errors 258

Controlling Input and Output Errors 259

Error Signaling 261

Obtaining Traceback Information 262

Detection of Math Errors 263

Checking the Accumulated Math Error Status 263

Special Values for Undefined Results 263

Check the Validity of Operands 264

Check for Overflow in Integer Conversions 264

Trap Math Errors with the CHECK_MATH Function 265

Enable and Disable Math Traps 266

Examples Using the CHECK_MATH Function 267

Hardware-dependent Math Error Handling 268

Error Handling on a Sun-4 (SPARC) Running SunOS Version 4 268

DECstation Error Handling 268

VAX/VMS Error Handling 268

Checking for Parameters 269

Checking for Keywords 269

Checking for Positional Parameters 269

Checking for Number of Elements 270

Checking for Size and Type of Parameters 272

Example of Checking for Size and Type of Parameters 272

Using Program Control Routines 273

Executing One or More Statements 273

Example of Executing Multiple Statements in a Single Command 274

Chapter 11: Tips for Efficient Programming 275

- Increasing Program Speed 276**
- Avoid IF Statements for Faster Operation 276**
- Use Array Operations Whenever Possible 278**
- Use System Routines for Common Operations 280**
- Use Constants of the Correct Type 280**
- Remove Invariant Expressions from Loops 281**
- Access Large Arrays by Memory Order 281**
- Be Aware of Virtual Memory 283**
- Running Out of Virtual Memory? 284**
 - Controlling Virtual Memory System Parameters under UNIX 285
 - Controlling Virtual Memory System Parameters under VMS 286
 - Minimize the Virtual Memory Used 288
- Array Operations are Rewarded 289**

Chapter 12: Accessing the Operating System 291

- Manipulating UNIX Environment Variables 292**
 - SETENV: Adding a New Environment Variable 293
 - GETENV: Getting an Environment Variable's Equivalence String 294
 - ENVIRONMENT: Getting the Values of All Environment Variables 294
- Manipulating VMS Logicals and Symbols 295**
 - SETLOG: Defining a New Logical 295
 - TRNLOG: Getting a Logical's Equivalence String 295
 - DELLOG: Deleting a Logical 296
 - SET_SYMBOL: Defining a Symbol 296
 - GET_SYMBOL: Getting a Symbol's Value 296
 - DELETE_SYMBOL: Deleting a Symbol 297

Accessing the Operating System Using SPAWN 297

Using SPAWN to Issue Commands 297

Interactive Use of SPAWN 298

UNIX Shells 299

VMS Command Interpreter 299

Non-interactive Use of SPAWN 299

Avoiding the Shell under UNIX 300

Capturing Output 301

Changing the Current Working Directory 302

Using the CD Procedure 302

Using the PUSHD, POPD, and PRINTD Procedures 303

Chapter 13: Interapplication Communication 305

Methods of Interapplication Communication 305

Choosing the Best Method 307

Interapplication Communication Using SPAWN 309

Communicating with a Child Process 309

Example: Communicating with a Child Process Using SPAWN 310

Executing PV-WAVE Commands Externally 313

Compiling the External Program 313

Starting PV-WAVE from an External Program with waveinit 314

Sending Commands to PV-WAVE with wavecmd 315

Ending the Session with PV-WAVE with waveterm 315

Example: Calling PV-WAVE from a C Program 316

Example: Calling PV-WAVE from a FORTRAN Program 318

Using LINKNLOAD to Call External Programs 319

Calling PV-WAVE in a Statically Linked Program 333

cwavec: Calling PV-WAVE from a C Program 333

cwavefor: Calling PV-WAVE from a FORTRAN Program 340

How to Link Applications to PV-WAVE 346

Accessing Data in PV-WAVE Variables	350
Communication with Remote Procedure Calls	359
Remote Procedure Call (RPC) Technology	359
Synchronization of Client and Server Processes	360
Linking a Server or a Client with PV-WAVE	361
Using PV-WAVE as a Client: CALL_UNIX	362
Description of C Functions Used with CALL_UNIX	364
Example Server	369
Example Using CALL_UNIX	370
Using PV-WAVE as a Server: CALL_WAVE	371
Description of C Functions Used with PV-WAVE as Server	373
PV-WAVE Functions Used with PV-WAVE as Server	375
Examples Using PV-WAVE as a Server	376
Remote Procedure Call Examples	376

Chapter 14: Getting Session Information **395**

Using the INFO Procedure	395
Calling INFO with No Parameters	395
Calling INFO with Positional Parameters	398
Calling INFO with Keyword Parameters	398

Chapter 15: Using WAVE Widgets **405**

Methods of GUI Programming in PV-WAVE	406
Introduction to WAVE Widgets	410
Who Uses WAVE Widgets	410
WAVE Widgets are Standard Library Functions	411
Designing Your Own WAVE Widgets	411
WAVE Widgets are Portable	411
First Example and Basic Steps	412
First Example	412
The Basic Steps	413

Initializing WAVE Widgets	414
Example	415
Creating and Arranging WAVE Widgets	416
The Widget Hierarchy	416
Types of WAVE Widgets	418
Arranging Widgets in a Layout	419
Creating and Handling Menus	425
Menu Bar	426
Popup Menu	427
Option Menu	428
Menu Callbacks	428
Defining Menu Items with Unnamed Structures	429
Modifying Menu Items	431
Example	431
Creating a Button Box and a Tool Box	433
Button Box Example	434
Tool Box Example	434
Creating a Radio Box	436
Creating a Controls Box with Sliders	437
Creating a Drawing Area	439
Creating a Text Widget	443
Single-line Label (Read-only)	443
Single-line Editable Text Field	444
Multi-line Text Window	445
Creating a Scrolling List	446
Selection Mode	447
Scrolling List Callbacks	447
Creating Popup Messages	449
Blocking vs. Nonblocking Windows	449
Types of Message Windows (Motif only)	450

Default Message Box Buttons	450
Message Box Example	451
Creating Dialog Boxes	453
A Dialog is a Popup Widget	453
Blocking vs. Nonblocking Windows	453
Default Dialog Box Buttons	454
Dialog Box Example	454
Creating a File Selection Widget	456
A File Selection Widget is a Popup Widget	457
Blocking vs. Nonblocking Windows	457
File Tool Contents	457
File Selection Example	458
Creating a Command Widget	459
Example	460
Creating a Table Widget	462
Setting Colors and Fonts	463
Setting Colors	463
Setting Fonts	464
Using A Resource File to Set Colors and Fonts	464
Setting and Getting Widget Values	466
Passing and Retrieving User Data	467
Example	468
Managing Widgets	469
Showing/Hiding Widgets	469
Widget Sensitivity	470
Displaying Widgets and Processing Events	471
Programming Tips and Cautions	472
PV-WAVE Routines to Avoid	472
PV-WAVE Routines to Use with Caution	472
Application Example	473

Chapter 16: Using the Widget Toolbox 479

Introduction to the Widget Toolbox	479
Basic Steps in Creating the GUI	480
Combining WAVE Widgets and Widget Toolbox Functions	480
Specifying the Desired Toolkit	481
Initializing the Widget Toolbox	481
Creating Widgets	482
Setting and Getting Resources	483
Managing, Displaying, and Destroying Widgets	484
Adding Callbacks	485
Adding Event Handlers	487
Example	488
Adding Timers	489
Example	490
Adding Work Procedures	491
Adding Input Handler Procedures	492
Changing the Cursor	493
Creating Tables	493
Running an Application	494
Related Include Files	495
Example Widget Toolbox Application	495
Programming Tips and Cautions	498

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 — Importing Data with C and FORTRAN Format Strings A-2

Using Format Reversion **A-5**

Group Repeat Specifications **A-6**

FORTRAN Formats for Data Import and Export **A-8**

FORTRAN Format Specifiers A-8

FORTRAN Format Code Descriptions **A-11**

A Format Code A-11

: Format Code A-12

\$ Format Code A-12

F, D, E, and G Format Codes A-13

I, O, And Z Format Codes A-16

Q Format Code A-18

H Format Codes and Quoted Strings A-19

T Format Code A-20

TL Format Code A-20

TR And X Format Codes A-21

C Format Strings for Data Import and Export **A-22**

Using C Format Strings for Importing Data A-22

Using C Format Strings for Exporting Data A-24

Appendix B: Motif and OLIT Widget Classes **B-1**

Motif Widget Classes **B-1**

Convenience Widgets B-3

OLIT Widget Classes **B-5**

Appendix C: Motif and OLIT Callback Parameters C-1

Motif Callback Parameters C-1

Required Callback Parameters C-2

Additional Required Callback Parameters C-2

OLIT Callback Parameters C-6

Required Callback Parameters C-6

Additional Required Callback Parameters C-6

Appendix D: Widget Toolbox Cursors D-1

Standard X Cursors D-1

Custom Cursors D-5

Multivolume Index i

Preface

This programmer's guide is part of a larger set of documentation; the entire set is shown in Figure I. Start with the *PV-WAVE Tutorial*, and then refer to the user's guide and this programmer's guide for other fundamental information about how to use the high-level visualization features of *PV-WAVE Advantage* and *CL*. You will also want to refer to the *PV-WAVE User's Guide for Advantage and CL* and the *PV-WAVE Reference for Advantage and CL, Volumes I and II* for detailed information.

For your convenience, all of the documents shown in Figure I are available online, as well as being available in a hardcopy format. Additional copies of the hardcopy documentation can also be ordered from Visual Numerics, Inc., by calling 800/447-7147.

Contents of this Programmer's Guide

This programmer's guide contains the following chapters:

- **Preface** – Describes the contents of this guide, lists the typographical conventions used, explains how to use the *PV-WAVE* documentation set, and explains how to obtain customer support.

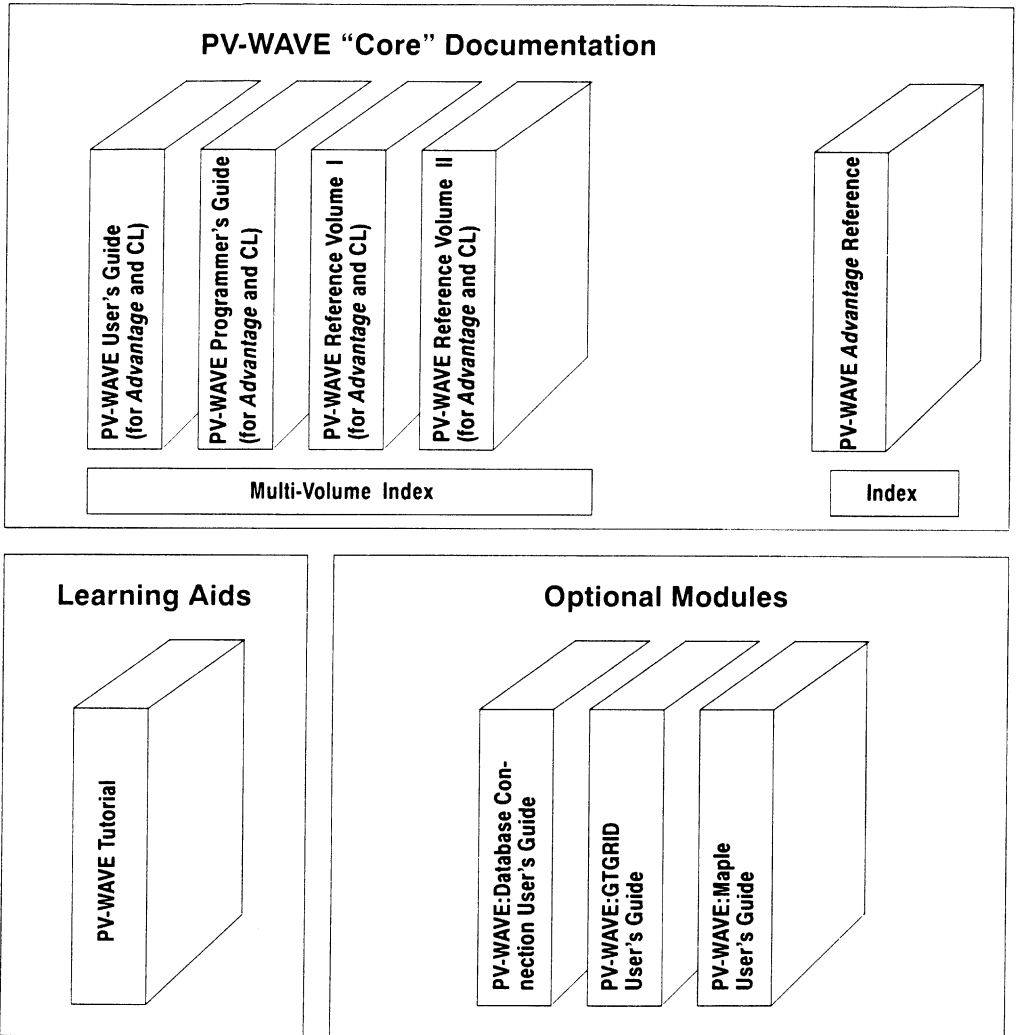


Figure I PV-WAVE documentation set; for more information about any one book you see shown here, refer to its preface, where the contents of each chapter are explained briefly. Be sure to avail yourself of our thorough product documentation and our excellent online learning aids. All documents are available both online and in a hardcopy format. Additional copies of the hardcopy documentation can be ordered by calling Visual Numerics, Inc., at 303/447-7147.

- **Chapter 1: Introduction to PV-WAVE** — 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** — 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 in PV-WAVE.
- **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: Accessing the Operating System** – Discusses the ways in which you can manipulate environment variables, logicals, and symbols from within PV-WAVE. In addition, the SPAWN command is introduced as a way to execute external programs from within PV-WAVE. Finally, ways to change the current directory are discussed.
- **Chapter 13: Interapplication Communication** – Discusses a variety of methods for interapplication communication. For example, PV-WAVE can execute external programs and exchange data with them. In addition, external programs can call PV-WAVE to perform graphics, data manipulation, and other functions.
- **Chapter 14: Getting Session Information** – Describes how to get information about the current PV-WAVE session.
- **Chapter 15: Using WAVE Widgets** – Discusses how to create a Motif or OPEN LOOK graphical user interface using the WAVE Widgets functions.
- **Chapter 16: Using Widget Toolbox** – Discusses how to create a Motif or OPEN LOOK graphical user interface using the Widget Toolbox functions.
- **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: Motif and OLIT Widget Classes** – Lists the widget classes available under Motif and OLIT.
- **Appendix C: Motif and OLIT Callback Parameters** – Lists the required callback parameters for widget routines under Motif and OLIT.
- **Appendix D: Widget Toolbox Cursors** – Lists the standard and custom cursors that are available for use with the WtCursor function.

- **Index** — A multivolume index that includes references to the *PV-WAVE User's Guide*, *PV-WAVE Programmer's Guide*, as well as both volumes of the *Reference*.

Typographical Conventions

The following typographical conventions are used in this guide:

- PV-WAVE code examples appear in this typeface. For example:

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

- Code comments are shown in this typeface, below the commands they describe. For example:

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

```
    This command plots air temperature data vs. sulphur dioxide concentration.
```

Comments are used often in this reference to explain code fragments and examples. Note that in actual PV-WAVE code, all comment lines must be preceded by a semicolon (;).

- PV-WAVE commands are not case sensitive. In this reference, 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 widget routines are shown in mixed case (WwMainMenu).
- A \$ at the end of a PV-WAVE line 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.

- The | symbol means “or” when used in a usage line. It is not to be typed. For example, in the following command:

```
result = QUERY_TABLE(table,  
' [Distinct] * | coli [alias] [, ..., coln [alias]] ...
```

the | means use either * or col_i [alias] [, ..., col_n [alias]], but not both.

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

Customer Support

If you have problems unlocking your software or running the license manager, you can talk to a Visual Numerics Customer Support Engineer. The Customer Support group researches and answers your questions about all Visual Numerics products.

Please be prepared to provide Customer Support with the following information when you call:

- The name and version number of the product. For example, PV-WAVE 4.2 or PV-WAVE P&C 2.0.
- Your license number, or reference number if you are an Evaluation site.
- The type of system on which the software is being run. For example, Sun-4, IBM RS/6000, HP 9000 Series 700.
- The operating system and version number. For example, SunOS 4.1.3.
- A detailed description of the problem.

The phone number for the Customer Support group is 303/530-5200.

Trademark Information

PostScript is a registered trademark of Adobe Systems, Inc.

QMS QUIC is a registered trademark of QMS, Inc.

HP Graphics Language, HP Printer Control Language, and HP LaserJet are trademarks of Hewlett-Packard Corporation.

Macintosh and PICT are registered trademarks of Apple Computer, Inc.

Open Windows and Sun Workstation are trademarks of Sun Microsystems, Inc.

TEKTRONIX 4510 Rasterizer is a registered trademark of Tektronix, Inc.

OPEN LOOK and UNIX are trademarks of UNIX System Laboratories, Inc.

PV-WAVE, PV-WAVE Command Language, PV-WAVE *Advantage*, and PV-WAVE P&C are trademarks of Visual Numerics, Inc.

OSF/Motif and Motif are trademarks of the Open Software Foundation, Inc.

X Window System is a trademark of the Massachusetts Institute of Technology

Introduction to the PV-WAVE Command Language

This chapter introduces the basic features of the PV-WAVE Command Language.

Introduction to PV-WAVE

PV-WAVE is compatible with the most up-to-date hardware available on the market today.

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).

PV-WAVE Notation

PV-WAVE provides a set of data types and operations to represent data with a natural notation and efficient representation. You can easily define and use structures containing aggregate. PV-WAVE variables, procedures, operators, and functions operate on scalar, vector, and array data with no change in notation or meaning.

PV-WAVE borrows much of its semantics from the programming language APL. The power and conciseness of PV-WAVE can be attributed to this APL influence. The main advantages over APL are syntax and control mechanisms plus visualization capabilities.

In the design of PV-WAVE, whenever there was a choice between brevity (and perhaps obscurity) and verbosity, the most readable alternative was selected.

Because scientists write their formulas using infix notation with parentheses, PV-WAVE has an expression syntax that resembles FORTRAN or BASIC, where operators are evaluated according to precedence and left-to-right sequence.

PV-WAVE Libraries

Libraries of procedures and functions written in PV-WAVE are already available. You can create new ones using a text editor. References to functions and procedures contained in library directories extract, compile, and execute the program unit without interruption. A number of procedures and functions written by PV-WAVE users are included in the Standard Library, which is tested, supported, and distributed with PV-WAVE. There are also additional unsupported, untested user routines in the Users' Library that may prove useful as you develop your applications.

Extensive Error Checking

As with any well-designed interactive language or system, extensive error checking and informative error messages are provided. The type of error and the associated variable are printed in an understandable format, without error codes or cryptic messages. You can stop a program that is running at any time and look at or change intermediate values. You can then resume the suspended program from the point of interruption.

Data Types

The following are the basic data types that PV-WAVE 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.
- **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 $\pm 10^{38}$, with 14 decimal places of significance.
- **Complex** — A real-imaginary pair of single-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.

Note 

Some other terms used in PV-WAVE include:

- *Scalars* are a single instance of one of the data types, excluding structures.
- *Arrays* contain multiple elements of the same data type.
- *Vectors* are one-dimensional arrays.
- *Elements* in an array are addressed using subscripts and subscript ranges. For more information about array subscripts, see Chapter 5, *Using Subscripts and Matrices*.

Constants and Variables

Constants and variables are the primary elements used with operators and expressions to mathematically manipulate data inside PV-WAVE programs.

Numeric and String Constants

A constant is a value that does not change during the execution of a program. PV-WAVE employs two types of constants: numeric and string. Numeric constants are defined by six data types: byte, integer, longword, single-precision floating-point, double-precision floating-point, and complex. For a complete description of numeric constants, see *Numeric Constants* on page 18.

String constants consist of characters enclosed by single quotes (') or by double quotes ("). The value of the constant is defined by the characters delimited by the single quotes or double quotes. See *String Constants* on page 21 for examples of valid and invalid string constants.

Variables

A variable is a named entity belonging to a data type that may assume any number of values. For example:

```
A = 6
B = 'This is a quote'
```

A is a variable that is assigned a numeric value of 6. B is a variable that represents the string value, "This is a quote". A variable can be either a scalar or an array of one of the seven data types. For a description of the basic features of variables, refer to *Attributes of Variables* on page 23 and *Names of Variables* on page 25.

Variable Declaration

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. In the examples above, A is automatically classed as an integer data type, and B is classed as a string data type.

System Variables

Besides variables assigned by you — user-defined variables — PV-WAVE also supports a special class of system variables. The names of these variables always begin with an exclamation point (!). For example, there is a system variable that contains the value for π , called !Pi. You can see its value by entering:

```
PRINT, !Pi  
3.14159
```

Variable names are not case sensitive. So !PI, !Pi, or !pi all refer to the same entity.

Each system variable belongs to a predefined data type which cannot be altered. Some system variables such as !Pi are read only, that is, their value cannot be changed. However, most system variables have a default value which you can change. You can also create your own system variables by using the PV-WAVE Standard Library routine DEFSYSV described in the *PV-WAVE Reference*. For more information on system variables, see *System Variables* on page 26.

Expressions and Operators

When variables and constants are combined using operators, expressions are created. Expressions and operators are briefly described in the following subsections.

Expressions

Expressions use operators to evaluate combinations of variables and constants. Expressions can also be combined with other expressions to create even more complex expressions. Unlike FORTRAN or BASIC expressions, PV-WAVE expressions work equally well on variables that contain scalar, one-, two-, or multi-dimensional data. Here are few examples of expressions using various operators:

```
C + 1
```

```
SIN(A * 4)
```

```
A / 2
```

```
Z > 9-2
```

```
(X LE 70) AND (X GE 35)
```

Note that expressions can also consist of functions (e.g., SIN) in addition to operators. Like constants and variables, expressions also have a data type and structure. See *Type and Structure of Expressions* on page 34 for more information.

Operators

PV-WAVE employs several types of operators within expressions.

Note



The presence of parentheses can affect the order in which numeric, relational, Boolean, and string operators are executed.

Numeric Operators

These operators, discussed in detail in *PV-WAVE Command Language Operators* on page 44, are summarized in the following table:

Table 1-1: Numeric Operators

Operator	Meaning
–	Unary minus
^	Exponentiation
*	Multiplication
#	Matrix multiplication
/	Division
+	Addition
–	Subtraction
MOD	Modulo operator

Boolean Operators

These operators, discussed in detail in *Boolean Operators* on page 43, are summarized in the following table:

Table 1-2: Boolean Operators

Operator	Meaning
NOT	Negation
AND	Union
OR	Intersection
XOR	Mutual exclusivity

Relational Operators

These operators, discussed in detail on *Relational Operators* on page 41, are summarized in the following table:

Table 1-3: 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

String Operators

These operators, discussed in detail in *Concatenating Strings* on page 123, are summarized in the following table:

Table 1-4: String Operators

Operator	Meaning
+	String concatenation
' ' or " "	String delineation

Array Concatenation Operator

The array operator, [], is discussed on *PV-WAVE Command Language Operators* on page 44.

Statements

The syntax of PV-WAVE statements is deliberately simple. The 13 types of statements should be familiar to anyone with a knowledge of FORTRAN or C.

The 13 statement types (with key words shown in capital letters) are summarized in the following subsections.

Assignment

var = expression

Assigns the value of the expression to the variable. This is the simplest form of the assignment statement. For more information, see *Assignment Statement* on page 53.

IF

IF expr THEN statement
IF expr THEN statement ELSE statement

Conditionally executes statements depending on the value of the expression. For more information, see *IF Statement* on page 70.

FOR

FOR var = init_expr, limit, [step] DO statement

Initializes the control variable to *init_expr*, repeatedly executes the *statement*, and increments the control variable until it is larger than the *limit* expression. The optional *step* value explicitly specifies an increment (either positive or negative) for the index. This is the general form of the FOR statement. For more details, see *FOR Statement* on page 65.

WHILE

WHILE *expr* DO *statement*

Repeatedly executes the *statement* while the value of the expression is true. For more information, see *WHILE Statement* on page 78.

REPEAT

REPEAT *statement* UNTIL *expr*

Repeatedly executes the *statement* until the expression is true. For more information, see *REPEAT Statement* on page 77.

CASE

CASE *expr* OF
expr: statement... END

Selects one of many statements for execution depending on the value of the expression. For details, see *CASE Statement* on page 62.

GOTO

GOTO, *label*

label:

Transfers control to the statement with the designated label. For more information, see *GOTO Statement* on page 70.

Block

BEGIN

statement

statement

END

A compound statement composed of one or more statements. For more information, see *Blocks of Statements* on page 60.

Procedure Call

procedure_name, parameter_list

Calls a system or user-written procedure with the given parameters. Unlike function calls, procedure parameters are not placed in parentheses. For more information, see *Procedure Call Statement* on page 73.

Procedure Definition

PRO *name, parameter_list*

Defines the start of a user-written procedure. For more information, see *Procedure Definition Statement* on page 76.

Function Call

result = function_name(parameters)

Calls a system or user-written function with a given parameter or parameters. For more information, see *Procedure or Function Calling Mechanism* on page 248.

Function Definition

FUNCTION *name, parameter_list*

Begins the definition of a user-written function. For details, see *Function Definition Statement* on page 68.

Common Block Definition

COMMON *name, variable_list*

Defines a group of variables that belong to a common block of variables that are globally recognized. For more information, see *Common Block Definition Statement* on page 63.

Input and Output

Almost any type of data, in either formatted or unformatted form, can be read or written by PV-WAVE. The command language allows most of the different forms of I/O supported by VAX/VMS. Sequential, relative, and indexed (ISAM) file organizations (with sequential, random access by key, and random access by file address methods) are available.

All of PV-WAVE's basic data types can be transferred in and out of files. Also, Date/Time formatted data can be transferred using the string data type.

Besides reading from standard input and output, PV-WAVE also can read from or write to a file. There are three basic steps for reading from or writing to a file:

- Open the file with an associated *logical unit number* (LUN).
- Read from or write to the file.
- Close the file.

Note



The first and final steps are unnecessary when using any I/O function that begins with the "DC" prefix. The DC functions are introduced in the section, *I/O Functions for Simplified Data Connection* on page 13.

Table 1-5 summarizes the procedures used to open and read files. For a complete discussion of how to transfer data, see Chapter 8,

Working with Data Files, which discusses PV-WAVE input/output and presents numerous examples.

Table 1-5: Input/Output Procedures

Procedure	Description
OPENR	Opens an existing file for reading.
OPENW	Opens a new file for writing.
OPENU	Opens an existing file for reading and writing.
READF	Reads data from a file into variables. If no Format keyword has been defined, PV-WAVE uses its default rules for formatting the data.
PRINTF	Writes data to a file. If no Format keyword has been defined, PV-WAVE uses its default rules for formatting the data.
READU	Reads unformatted binary data from a file into a variable.
WRITEU	Writes unformatted binary data from an expression into a file.

I/O Functions for Simplified Data Connection

Any PV-WAVE 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.

These functions are easy to use because they automatically handle many aspects of data transfer, such as opening and closing the data file. Also, they recognize C-style formatting tokens, whereas the

procedures listed in the previous table only recognize FORTRAN-style format codes. Here is a list of the DC functions:

Table 1-6: DC Functions

Function	Description
DC_WRITE_FIXED DC_READ_FIXED	Write (or read) formatted data to (or from) a file without having to explicitly choose a LUN. You use the Format keyword to provide the FORTRAN or C format string that is used to transfer the data.
DC_WRITE_FREE DC_READ_FREE	Write (or read) formatted data to (or from) a file without having to explicitly choose a LUN. You do not have to provide a FORTRAN or C format string to transfer the data.
DC_WRITE_8_BIT DC_READ_8_BIT	Write (or read) unformatted 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) unformatted 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.

For more information about the DC functions, see *Functions for Simplified Data Connection* on page 153 and the *PV-WAVE Reference*.

Functions and Procedures

Procedures and functions are programs that define specific tasks. You define procedures and functions in files using your text editor or create files directly at the WAVE> prompt. See *Creating and Running Programs Interactively* on page 25 of the *PV-WAVE User's Guide* and *Using a Text Editor to Create and Run Programs* on page 19 of the *PV-WAVE User's Guide* for details.

Definition Syntax

The syntax for creating a procedure is:

```
PRO procedure_name [Param1, Param2, Param3]  
. .  
END
```

The syntax for creating a function is:

```
FUNCTION function_name [Param1, Param2, Param3 ]  
. .  
RETURN, expression  
END
```

For more information about creating procedures and functions, see *Procedure and Function Parameters* on page 234.

For information on saving compiled procedures in files, see *Using PV-WAVE in Runtime Mode* on page 41 of the *PV-WAVE User's Guide*.

Usage Syntax

The syntax for using a procedure is:

```
procedure_name [, param1, param2, ..., paramn ]
```

The syntax for using a function is:

```
result = function_name(param1 [, param2, ..., paramn])
```

Where to Find PV-WAVE's Libraries

PV-WAVE's supported library routines are located in the directory `$WAVE_DIR/lib/std` (UNIX) or `WAVE_DIR:[LIB.STD]` (VMS). 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.) The standard library routines are summarized in Chapter 1, *Functional Summary of Routines*, in the *PV-WAVE Reference*.

The standard library contains the subdirectories `motif` and `olit`. The `motif` directory contains the standard WAVE Widgets routines for Motif, and the `olit` directory contains the OPEN LOOK WAVE Widgets. For information on WAVE Widgets, see Chapter 15, *Using WAVE Widgets*.

Another subdirectory of the Standard Library is the `guitools` directory. This directory 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*.

An unsupported user library (`$WAVE_DIR/lib/user` (UNIX) or `WAVE_DIR:[LIB.USER]` (VMS)) is also included in the PV-WAVE distribution. This library contains such entries as routines for compressing images, making pie charts, creating 2D/3D bar graphs, and displaying 3D scatterplots. For information on adding routines to the library, see *Submitting Programs to the Users' Library* on page 255. The routines in the user 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.

Constants and Variables

Constants and variables are the building blocks that 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 seven basic data types, each with its own form of constant:

- **Byte** – 8-bit unsigned integers.
- **Integer** – 16-bit signed integers.
- **Longword** – 32-bit signed integers.
- **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.
- **String** – Zero or more eight-bit characters which are interpreted as text.

In addition, structures are defined in terms of the seven 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 Table 2-1, where “*n*” represents one or more digits.

Table 2-1: 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>	"12
		' <i>n</i> 'O	'377'O

Table 2-1: Syntax of Integer Constants

Radix	Type	Form	Examples
	Long	"nL	"12L
		'n'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. Table 2-2 illustrates both examples of valid and invalid PV-WAVE constants.

Table 2-2: Examples of Integer Constants

Correct	Incorrect	Reason
255	256B	Too large, limit is 255
'123'L	'123L	Unbalanced 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.

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 Table 2-3. 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.

Table 2-3: 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, both of which are single-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.

Table 2-4 illustrates valid string constants and Table 2-5 illustrates invalid string constants.

In the last entry of Table 2-5, `" 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.

Table 2-4: 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'

Table 2-5: 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	' 'counte r' '	Double apostrophe is null string
129	"129"	Illegal octal constant

Representing Non-printable Characters

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 hexa-

decimal value. In order to construct a character string which actually contains a literal backslash character, it is necessary to enter two consecutive backslash characters. Table 2-6 gives examples of using octal or hexadecimal character notation.

Table 2-6: 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

Variables

Variables are named repositories where information is stored. A PV-WAVE variable may contain a scalar, vector, multidimensional array, or structure of virtually any size. Arrays may contain elements of any of the seven basic PV-WAVE data types plus structures. Variables may be used to store images, spectra, single quantities, names, tables, etc.

Attributes of Variables

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

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 aspect of PV-WAVE 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

PV-WAVE 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:

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.

Table 2-7: 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

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 DEF-SYSV and the ADDSYSVAR procedures.

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

The following table summarizes the system variables. For detailed information on each system variable, see Chapter 4, *System Variables*, in the *PV-WAVE Reference*.

Table 2-8: Summary of System Variables

Name	Type	Purpose
!C	long	Returns subscript of element found by MAX or MIN.
!D	structure	(Read only) Contains information on current plotting device.
!Date_Separator	string	Lets you change the default character (/) used to separate the parts of a date. Used for date to string conversion.
!Day_Names	string	An array of strings containing the names of the days of the week.
!Dir	string	Contains name of the main directory.
!Dpi	double	(Read only) Contains double-precision value of π .
!DT_Base	!DT	Contains the value of Julian Day 1 (September 14, 1752) as PV=WAVE Date/Time data.
!Dtor	float	(Read only) Converts degrees to radians ($\pi / 180$).

Table 2-8: Summary of System Variables

Name	Type	Purpose
!Edit_Input	integer	Enables/disables keyboard line editing.
!Err	long	Contains code of last I/O error message.
!Err_String	string	(Read only) Contains text of last I/O error message.
!Feature_Type	string	(Read only) Indicates whether PV-WAVE is in runtime or interactive mode.
!Holiday_List	!DT	Contains an array of up to 50 Date/Time structures that represent holidays created by the procedure CREATE_HOLIDAY. This system variable does not have a default value.
!Journal	long	(Read only) Logical unit number of journal output, or 0.
!Lang	string	Identifies the language currently being used. The default is "american".
!Month_Names	string	An array of strings containing the names of the months.
!Msg_Prefix	string	Contains prefix string for error messages.
!Order	long	Indicates direction of image transfer: bottom up (0), or top down (1).
!P	structure	Contains the plotting system variables.
!Path	string	Contains the search path for procedures and functions.
!PDT	structure	Contains the system variables for plotting Date/Time axes.
!Pi	float	(Read only) Contains the floating-point value of π .

Table 2-8: Summary of System Variables

Name	Type	Purpose
!Product	string	(Read only) Identifies the product being run: either "cl" or "advantage".
!Prompt	string	Contains prompt string used by PV-WAVE.
!Quarter_Names	string	An array of four strings containing the following values by default: 'Q1', 'Q2', 'Q3', and 'Q4'.
!Quiet	integer	Suppresses compiler messages on screen.
!Radeg	float	(Read only) Converts radians to degrees ($180 / \pi$).
!Start	!DT	Contains the value of the date and time PV-WAVE was started.
!Time_Separator	string	Lets you change the default character (:) used to separate the parts of a time representation. Used for time to string conversion.
!Version	structure	(Read only) Contains the current version number of PV-WAVE, operating system, architecture, and platform.
!Weekend_List	long	An array of seven long integers. Element zero (0) represents Sunday, and so on. The value of an element is zero (0) if the day is a weekday, and one (1) if it is a weekend day or a holiday. The CREATE_WEEKEND procedure builds this variable. This system variable does not have a default value.
!X	structure	Axis variables for the X axis.
!Y	structure	Axis variables for the Y axis.
!Z	structure	Axis variables for the Z axis.

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 are 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 in PV-WAVE for data smoothing, shifting, transforming, evaluation of transcendental functions, etc. For a complete description of all PV-WAVE functions, 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

PV-WAVE 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:

Table 3-1: Operator Precedence

Priority	Operator
First (highest)	^ (exponentiation)
Second	* (multiplication) # (matrix multiplication) / (division) MOD (modulus)
Third	+ (addition) - (subtraction) < (minimum) > (maximum) NOT (Boolean negation)

Table 3-1: Operator Precedence

Priority	Operator
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 seven atomic data types, in decreasing order of complexity are:

- Complex 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.

Table 3-2: 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 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

Not all implementations of PV-WAVE check for overflow in type conversions. In particular, the Sun version of PV-WAVE, and other versions based on the Motorola 68000 computer do not. For example, in the Sun implementation, 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. See Chapter 10, *Programming with PV-WAVE*, for details on these error-checking functions.

The statement:

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

is perfectly legitimate in PV-WAVE; 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.

Table 3-3: Examples of Conversion Functions

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

Extracting Fields

When called with more than a single parameter, the BYTE, 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:

CONV_FUNCTION(*expression*, *offset* [, *dim*₁, ..., *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*₁, ..., *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:

Table 3-4: 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

PV-WAVE 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:

Table 3-5: Structure of Expressions

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

Seven functions exist to create arrays of the seven types: BYTARR, INTARR, LONARR, FLTARR, DBLARR, COMPLEXARR, 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.

Relational Operators

The six relational operators are:

Table 3-6: Relational Operators

Operator	Purpose
EQ	Equal to
NE	Not equal to
LE	Less than or equal to
LT	Less than
GE	Greater than or equal to
GT	Greater than

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.

In PV-WAVE, 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.

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:

Table 3-7: Action of AND, OR, XOR

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.

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.

PV-WAVE Command Language Operators

Operators are used to combine terms and expressions. The set of PV-WAVE operators is:

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 53.

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.

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.

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 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 PV-WAVE 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 in PV-WAVE 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.

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.
```

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.

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.

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 2π .

NE

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

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.

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.

Statement Types

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

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

Components of Statements

Statements in PV-WAVE 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 by PV-WAVE, 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.

Table 4-1 summarizes the four forms of assignment statements.

Table 4-1: 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	Scalar	Scalar	The scalar expression is stored in a single element of Var.
	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:

$$\text{DATA}([3, 5, 7, 9]) = 0$$

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:

$$\text{DATA}(\text{WHERE}(\text{DATA} \text{ LT } 0)) = -1$$

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.

$$\text{variable}(\text{subscript_range}) = \text{scalar_expression}$$

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 84.

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

`B (* , I) = 1`

Stores ones in the *i*th row.

`B (J , *) = 1`

Stores ones in the *j*th 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 long-word 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 ($T = ARR(I)$) 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 Table 4-2. 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.

Table 4-2: End Statements

End Statement	Usage
ENDCASE	CASE statement
ENDELSE	IF statement, ELSE clause
ENDFOR	FOR statement
ENDIF	IF statement, THEN clause
ENDREPEAT	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. In PV-WAVE, Common Block variables are matched variable to variable, unlike FORTRAN, where storage locations are matched. The third variable in a given PV-WAVE 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 = expr₁, expr₂ DO statement

Form 1: Increment of 1.

FOR var = expr₁, expr₂, expr₃ 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. In PV-WAVE, 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.

Caution

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.

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.

Note 

A `HISTOGRAM` function is provided by `PV-WAVE` in the Standard Library. For detailed information on the `HISTOGRAM` function, see the *PV-WAVE Reference*.

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

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.

Note 

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 Definition Statement

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

A function may be defined as a program unit containing one or more PV-WAVE statements which returns a value. Once a function has been defined, references to the function 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 74 and *More On Parameters* on page 75.

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 PV-WAVE search path defined by the system variable !Path. For more information setting the search path, see *WAVE_PATH: Setting Up a Search Path* on page 46 of the *PV-WAVE User's Guide*. For more information on system variables, see *System Variables* on page 26.
- 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.

Caution

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 *Using Executive Commands* on page 26 of the *PV-WAVE User's Guide*.

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 NE G 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 60.) 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, ..., pn

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 provided with PV-WAVE.
- User-written procedures written in PV-WAVE and compiled with the .RUN command.
- User-written procedures that are compiled automatically because they reside in directories in the search path. These procedures are compiled the first time they are used. See *Automatic Compilation of Functions and Procedures* on page 69.
- Standard Library procedures, written in PV-WAVE, contained in a directory in the search path, and provided 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 246.

Procedure Definition Statement

PRO *name*, *p*₁, *p*₂, ..., *p*_{*n*}

A sequence of one or more PV-WAVE 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 PV-WAVE search path (!Path) causes the procedure to be read from the disk, compiled, saved, and executed, without interrupting program execution. If you are running under VMS, see *VMS Procedure Libraries* on page 251 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 PV-WAVE's 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 and Matrices

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 53 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. A two-dimensional array, a matrix, with n columns and m rows, is addressed with a subscript of the form: (i, j) , where $0 \leq i < n$ and $0 \leq j < m$. The first subscript, i , is the column index, and the second subscript, j , is the row index.

Note 

The notion of columns and rows in PV-WAVE is reversed from that of linear algebra. The main reason for this is to allow the X subscript to appear first when subscripting images.

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.

Example of a Subscript Reference

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. In matrices and images, the first subscript denotes the column and the second subscript is the row. Like FORTRAN, but unlike linear algebra, the first subscript, which is the column number, varies the fastest.

If ARR is a 2-by-3 array, the elements are stored in memory as follows:

$$\begin{array}{l} \left[\begin{array}{ll} A_{0,0} & A_{1,0} \text{ Lowest memory address} \\ A_{0,1} & A_{1,1} \\ A_{0,2} & A_{1,2} \text{ Highest memory address} \end{array} \right. \end{array}$$

The elements are ordered in memory as: $A_{0,0}$, $A_{1,0}$, $A_{0,1}$, $A_{1,1}$, $A_{0,2}$, $A_{1,2}$.

Images are usually displayed with row zero at the bottom of the screen, matching the display's coordinate system, although this order may be reversed by setting the system variable !Order to a non-zero value.

Elements of arrays may also be specified using only one subscript, in which case the array is treated as a vector with the same number of points. In the above example, $A(2)$ is the same element as $A(0, 1)$, and $A(5)$ is the same element as $A(1, 2)$.

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 integer, a longword integer copy is made and used to evaluate the subscript.

“Extra” Dimensions

When creating arrays, PV-WAVE eliminates all “degenerate” trailing dimensions of size 1. Thus, the statements:

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

print the following:

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

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

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, the following statements print the value of 5, and then assign the scalar variable a value of 6.

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

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*.

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-column by 12-row array,

`ARR(*, 11)`

is the last row of ARR (i.e., a 10-element row vector), composed of elements

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

Similarly,

`ARR(0, *)`

is the first column of ARR,

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

and its dimensions are 1-column by 12-rows.

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

`ARR(*, 0 : 4)`

is made from all columns of rows 0 to 4 of ARR, or a 10-column, 5-row matrix.

Table 5-1 summarizes the possible forms of subscript ranges.

Table 5-1: Subscript Ranges

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

Structure of Subarrays

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

Degenerate dimensions (trailing dimensions whose size is equal to 1) are removed. This was illustrated in the above example by the expression `ARR(*, 1 1)` which resulted in a row 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 a column vector with dimensions of (1, 12) showing that the structure of columns is preserved 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-column by 12-row 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, *)

The fourth column of A, a 1-by-12 column vector. [A(3, 0), A(3, 1), ..., A(3, 11)].

A(*, 0)

The first row of A, a 10-element row vector. Note that the last dimension was removed because it was degenerate.

A(X - 1 : X + 1, Y - 1 : Y + 1)

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

$$\begin{bmatrix} a_{x-1,y-1} & a_{x,y-1} & a_{x+1,y-1} \\ a_{x-1,y} & a_{x,y} & a_{x+1,y} \\ a_{x-1,y+1} & a_{x,y+1} & a_{x+1,y+1} \end{bmatrix}$$

A(3 : 5, *)

Three columns of A, a 3-by-12 subarray:

$$\begin{bmatrix} a_{3,0} & a_{4,0} & a_{5,0} \\ a_{3,1} & a_{4,1} & a_{5,1} \\ \dots & & \\ \dots & & \\ a_{3,11} & a_{4,11} & a_{5,11} \end{bmatrix}$$

See the section *Assignment Statement* on page 53 for 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 which is an array, the process may be written as:

$$V(S) = \left\{ \begin{array}{l} V_{S_i} \text{ if } 0 \leq S_i < n \\ V_0 \text{ if } S_i < 0 \\ V_{N-1} \text{ if } S_i \geq n \end{array} \right\} \quad (\text{ for } 0 \leq i < m)$$

assuming that the vector V has n elements, and S has m elements. The result $V(S)$ has the same structure and number of elements as does the subscript vector S .

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

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.

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

$$A(\text{INDGEN}(10) * 11)$$

yields a 10-element vector equal to the diagonal elements of A. The subscripts of the diagonal elements, $A_{0,0}$, $A_{1,1}$, ..., $A_{n-1, n-1}$ are equal to 0, 11, 22, ..., 99, when subscripted with a single subscript.

The elements of the vector $\text{INDGEN}(10) * 11$ are also equal to 0, 11, 22, ..., 99. Applying the vector as a subscript selects the diagonal elements.

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

$$A(\text{WHERE}(A \text{ GT } 0))$$

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 with Subscript Ranges

When combining an array subscript with a subscript range, the result is an array of subscripts constructed by combining each element of the subscript array with each member of the subscript range. Combining an n -element array with an m -element subscript range yields an nm -element subscript. Each dimension of the result is equal to the number of elements in the corresponding subscript array or range.

For example, the expression $A([1, 3, 5], 7 : 9)$ is a 9-element, 3-by-3 array composed of the elements:

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

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

Example

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

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

Zeroes first and last rows.

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

Zeroes first and last columns.

Combining with Other Subscript Arrays

When combining two subscript arrays, each element of the first array is combined with the corresponding element of the other subscript array. The two subscript arrays must have the same number of elements. The resulting subscript array has the same number of elements as its constituents.

For example, the expression $A([1, 3], [5, 9])$ yields the elements $A_{1,5}$ and $A_{3,9}$.

Combining with Scalars

Combining an n -element subscript range or n -element subscript array with a scalar yields an n -element result. The value of the scalar is combined with each element of the range or array.

For example, the expression $A([1, 3, 5], 8)$ yields the 3-element vector composed of the elements $A_{1,8}$, $A_{3,8}$, and $A_{5,8}$. The second dimension of the result is 1 and is eliminated because it is degenerate. The expression $A(8, [1, 3, 5])$ is the 1-by-3 column vector: $A_{8,1}$, $A_{8,3}$, and $A_{8,5}$, illustrating that leading dimensions are not eliminated.

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.

The subscript array is converted to longword type before use if necessary. Regardless of structure, this subscript array is interpreted as a vector.

See *Assignment Statement* on page 53 for details and examples of storing with vector subscripts.

Example

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

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:

$$A = \text{FLTARR}(N, N)$$

$$A(\text{INDGEN}(N) * (N + 1)) = 1.0$$

The expression $\text{INDGEN}(N) * (N + 1)$ results in a vector containing the subscripts of the diagonal elements $[0, n + 1, 2n + 2, \dots, (n - 1)(n + 1)]$. Yet another way is to use two array subscripts:

$$A = \text{FLTARR}(N, N)$$

$$A(\text{INDGEN}(N), \text{INDGEN}(N)) = 1.0$$

which creates the array subscripts: $[(0, 0), (1, 1), \dots, (n - 1, n - 1)]$.

The statement:

$$A(\text{WHERE}(A \text{ LT } 0)) = -1$$

sets negative elements of A to minus 1. The statements:

$$A = \text{FLTARR}(10, 10)$$

$$A(\text{INDGEN}(10) * 11) = 1$$

create a 10-by-10 identity matrix.

Memory Order

In certain circumstances, such as computationally intensive operations on very large images or matrices, it is useful to know the memory order of the elements in the array. Given a 2×3 array created with the PV-WAVE statement

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

the elements of A are ordered in memory as $A_{0,0}$, $A_{1,0}$, $A_{0,1}$, $A_{1,1}$, $A_{0,2}$, $A_{1,2}$.

Knowledge of the memory order is also important when attempting to subscript multidimensional arrays with a single subscript. Elements of multidimensional arrays can be specified using only one subscript, in which case the array is treated as a vector with the same number of points. 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

Reading and Printing Matrices Interactively

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

```
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, INV(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 Advantage 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 Advantage 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, and the PV-WAVE Advantage INV function, 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 # INV(a)
```

```
AA-1 should be identity. Error due to roundoff.
```

```
1.00000      0.00000      0.00000
-2.98023e-07  1.00000      1.49012e-08
-9.53674e-07  0.00000      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) # (TRANSDPOSE(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 PV-WAVE 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 seven 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 referred to in PV-WAVE, structure definitions 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:

```
{ Structure_name, Tag_name1 : Tag_def1, ... : ...,  
  Tag_namen : Tagdefn }
```

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:

```
variable = { 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.

Note 

The variable created by the above statement and the structure definition *{Structure_name}* are separate entities. The variable is said to be of type *{Structure_name}*. The definition *{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 structure 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 PV-WAVE 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 \$ in PV-WAVE).

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 PV-WAVE 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

Variable_name(Subscripts) . Tag_name(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 112.

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 PV-WAVE 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 14, *Getting Session Information*, for information on these keywords.

Parameter Passing with Structures

As explained in *Parameter Passing Mechanism* on page 246, 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

`VAR.TAG = 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

`VAR.TAG = 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

$\text{VAR.TAG}(\textit{subscript}) = \textit{scalar_expr}$

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:

$\text{STAR.INTEN}(5) = 100$

Sets the 6th element of STAR.INTEN to 100.

$\text{STAR.INTEN}([2, 4, 6]) = 100.$

Sets elements 2, 4, and 6 to 100.

Rule 4

$\text{VAR.TAG}(\textit{subscript}) = \textit{array_expr}$

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:

$\text{STAR.INTEN}([2, 4, 6]) = [5, 7, 9]$

Sets elements 2, 4, and 6 to the values 5, 7, and 9, respectively.

Rule 5

$\text{VAR.TAG}(\textit{subscript_range}) = \textit{scalar_expr}$

The value of the scalar expression is stored into each element specified by the subscript range. Example:

$\text{STAR.INTEN}(8 : *) = 5$

Sets elements 8, 9, 10, and 11, to the value 5.

Rule 6

$\text{VAR.TAG}(\textit{subscript_range}) = \textit{array_expr}$

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 106 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 quite general and lead to a myriad of 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 165.

Unformatted Input and Output with 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 PV-WAVE 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 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 PV-WAVE 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 155 for details on data formatting.

As a simple example, the following PV-WAVE 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 165, 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.

Due to the way in which strings are implemented in PV-WAVE, 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 36.

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 = STRLOWCASE(string)
result = STRUPCASE(string)
```

where *string* is the string to be converted to lower or upper case.

The following PV-WAVE 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 PV-WAVE 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 PV-WAVE 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 ' + $
    'railing 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 PV-WAVE 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

PV-WAVE provides the STRPOS, STRPUT, and STRMID routines to 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 PV-WAVE statements use STRMID to print a table matching the number of each month with its three-letter abbreviation:

```
MONTHS = 'JANFEBMARAPRMAYJUNJULAUGSEP' + $
'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, PV-WAVE converts it to string type using the same default formatting rules that are used by the PRINT, or STRING routines. The function then operates on the converted result. Thus, the PV-WAVE 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*.

Working with Data Files

PV-WAVE provides many alternatives for working with data files. There are few restrictions imposed on data files by PV-WAVE, and there is no unique PV-WAVE format. This chapter describes PV-WAVE input and output methods and routines, and gives examples of programs that read and write data using PV-WAVE, C, and FORTRAN commands.

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 PV-WAVE 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 PV-WAVE window.

With two commands, the data is transferred from the file into the variable flow and displayed in a PV-WAVE window.

An alternate set of PV-WAVE 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 PV-WAVE window.

Example 2 — Output

Here's a simple example showing how you can transfer data from a PV-WAVE variable to a file:

```
ylow = [77, 63, 42, 56]
```

Define ylow to be a vector of integers.

```
status = DC_WRITE_FREE('data2.dat', ylow, $  
/Column)
```

DC_WRITE_FREE handles the opening and closing of the file. It takes the values in "ylow" 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 in PV-WAVE 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.



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 153.

Basic Commands for Opening Files

The three main OPEN commands are listed in Table 8-1:

Table 8-1: Procedures that Open Files

Procedure	Description
OPENR	Opens an existing file for input only.
OPENW	Opens a new file for input and output. Under UNIX, if the named file already exists, the previous contents are destroyed. Under VMS, 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.



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 141.
- **Simplified I/O Routines** — Any PV-WAVE 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 153.

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.



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 PV-WAVE 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 PV-WAVE statement:

```
PRINT, X
```

is equivalent to

```
PRINTF, -1, X
```

The following **PV-WAVE** 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 by **PV-WAVE**.

- **-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`.

VMS

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 com-

mands 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 PV-WAVE 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 PV-WAVE 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 PV-WAVE procedures and functions, be sure not to explicitly assign file unit numbers in the range {100...128}. If a PV-WAVE 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 — GE

T_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
  End of the procedure.
```

Note 

All PV-WAVE 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 PV-WAVE 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 146.

	Name: Hour Type: Integer Dimension 1: *			
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	1.3021	
FEB 4	29.9985	0.3262	1.5137	
FEB 6	33.8292	1.6744		
FEB 8	37.9902	0.4949		
FEB 10	39.7021	0.3274		
FEB 12	39.7			
FEB 14	3			
FEB 16				
FEB 18				

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.

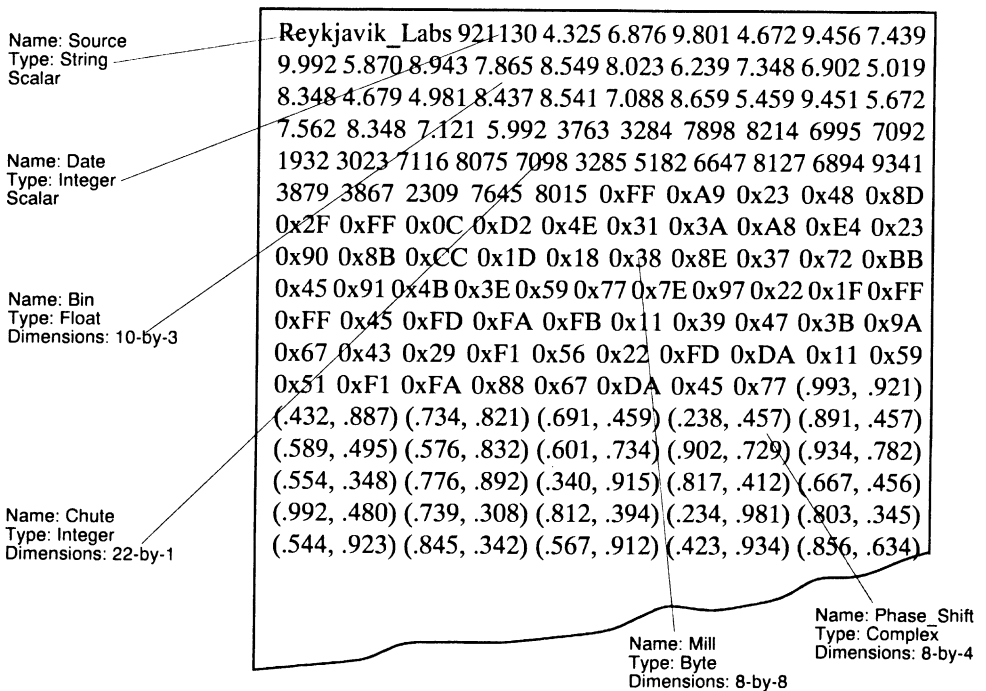


Figure 8-2 Typical file organization for a row-oriented ASCII data import file. Spaces are being used as the delimiter to separate adjacent data values. In this example, the first group of data is associated with a variable named Source, the second group with a variable named Date, the third group with 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. Differences between the UNIX and VMS 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 153.

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 a separate volume, *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.

For binary files, neither the concept of physical or logical records is relevant, although when using PV-WAVE in a VMS 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 a VMS environment, refer to *Record-Oriented I/O in VMS Binary Files* on page 149.

Note



For more information about how the operating system affects the transfer of binary data, refer to *Reading UNIX FORTRAN-Generated Binary Data* on page 199 and *Reading VMS FORTRAN-Generated Binary Data* on page 202.

Number of Records in a File

In the VMS 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 a VMS file, refer to *Creating Indexed Files* on page 228.

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 VMS Binary Files

All VMS 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 VMS files, refer to *VMS-Specific Information* on page 224; that section contains more information on how VMS handles files.

Example — Transferring Record-Oriented Data

When writing to VMS 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, consider the case of a file with 80 character records. This file would have been opened with the following statement:

```
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.

Note



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 Table 8-2 below.

Table 8-2: A Comparison of Binary and Human-Readable Input/Output (I/O)

	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 (Human-Readable) 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 Table 8-4 on page 156 and Table 8-5 on page 157.
- 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 PV-WAVE function to understand. The ASSOC function is discussed in *Associated Variable Input and Output* on page 212.
- Images stored in the TIFF format can be easily transferred using the DC_READ_TIFF and DC_WRITE_TIFF 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 189.
- 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 205.
- 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 159.

- 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 145.

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.



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 Table 8-6 on page 158.

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 Table 8-3:

Table 8-3: Routines for Transferring Binary Data

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 structure to a data file, providing efficient and convenient direct access to binary data.
GET_KBRD	Read single characters from the keyboard.

For more information about the routines shown in Table 8-3, refer to *Input and Output of Binary Data* on page 188, *Associated Variable Input and Output* on page 212, and *Getting Input from the Keyboard* on page 222.

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 Table 8-4 on page 156 and Table 8-5 on page 157.

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 PV-WAVE variables, 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 145.

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

The routines for freely formatted ASCII I/O are shown in Table 8-4:

Table 8-4: Routines for Transferring Freely Formatted ASCII Data

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 Table 8-4, 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 159.

ASCII I/O — Fixed Format

The routines for explicitly formatted ASCII I/O are shown in Table 8-5:

Table 8-5: Routines for Transferring Explicitly Formatted ASCII Data

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_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 Table 8-5, 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 165.

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 187.

Other I/O Related Routines

In addition to performing I/O to an open file, there are several PV-WAVE routines that provide other file management capabilities. These additional routines are shown in Table 8-6:

Table 8-6: Other I/O Related Commands

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_LUN	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.
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 Table 8-6, refer to a later section, *Miscellaneous File Management Tasks* on page 218.

Free Format Input and Output

PV-WAVE 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 Table 8-4 on page 156.

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.

Note

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:

```
'ello 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 PV=WAVE 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 seven of the basic PV=WAVE 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 Table 8-7 on page 164.

Importing Date/Time Data

The following PV-WAVE 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 Table 8-8 on page 169.

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 PV-WAVE 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 Chapter 7, *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. The formats used are summarized in Table 8-7:

Table 8-7: 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 Table 8-5 on page 157.

All data is handled in terms of basic PV-WAVE 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 PV-WAVE 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 a PV-WAVE 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 167.

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 vari-

ables), 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 PV-WAVE 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 PV-WAVE 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 into PV-WAVE 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 data/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 170.

- **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 173.

Method 2 utilizes the DC_READ routines. As discussed in *Functions for Simplified Data Connection* on page 153, 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 Table 8-8.

Table 8-8: 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 (,).

Note

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 Table 7-2 on page 230 and Table 7-3 on page 231 in the *PV-WAVE User's Guide*.

Example — Reading Date/Time Data into PV-WAVE

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 *PV-WAVE* 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 = ' ' & datels = locs & timels = 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, datels, $
        timels, 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 & datel(I) = datels & $
        timel(I) = timels & 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(datel, timel, $
    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 Tem-
    plate 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.

```

Note 

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 PV-WAVE 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 PV-WAVE's 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 Chapter 7, *Working with Date/Time Data*, in the *PV-WAVE User's Guide*.

Method 2 — Read the File with DC_READ_FIXED

The following PV-WAVE statements present another method for reading date/time data into PV-WAVE 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, $
  /Col, 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 Chapter 7, *Working with Date/Time Data*, in the *PV-WAVE User's Guide*.

Reading, Sorting, and Printing Tables of Formatted Data

PV-WAVE 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 PV-WAVE 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 PV-WAVE 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

Note

DC_READ_FIXED is not used in this method because the file, as it is shown on page 175, 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 some of 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 PV-WAVE 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 section Chapter 8, *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 informa-
  tion 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

Note 

For more information about PV-WAVE's functions for sorting and organizing table structures, and the keywords that can be used inside the QUERY_TABLE sort string, refer to Chapter 8, *Creating and Querying Tables*, in 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),
CVELO(100)
OPEN (UNIT=1, STATUS='NEW', FILE='aptv.dat')
.
.           Other commands go here.
.
WRITE (1, '(4(1x, G15.5))')
C(ALT(I), PRES(I), TEMP(I), VELO(I), I = 1,100)
END
```

PV-WAVE Read (Method 1)

The data is read into an array, the array is transposed storing each variable as a row, and each row is extracted and stored in a one-dimensional variable.

```
OPENR, 1, 'aptv.dat'
      Open file for input.

A = FLTARR(4,100)
      Define variable to hold 100 observations of data, 4 values per
      observation.

READF, 1, A
      Read the data.

A = TRANSPOSE(A)
      Transpose array so that columns become rows.
```

```
alt = A(*,0) & pres = A(*,1) &  
temp = A(*,2) & velo = A(*,3)
```

Extract the altitude, pressure, temperature, and velocity data from variable A.

```
CLOSE, 1
```

Close the file.

PV-WAVE Read (Method 2)

In this method, the data is read by calling `DC_READ_FIXED`, one of the DC routines for simplified I/O:

```
status = DC_READ_FIXED('aptv.dat', alt, $  
pres, temp, velo, /Column, Format="%f")
```

`DC_READ_FIXED` transfers the values in "aptv.dat" to the variables `alt`, `pres`, `temp`, and `velo`. One value from each record is transferred to each variable. `DC_READ_FIXED` creates the variables as floating-point vectors, with a length that matches the number of values transferred into the variables. `DC_READ_FIXED` handles the opening and closing of the file.

The variables could now be easily placed into a table structure with the following command:

```
aptv = BUILD_TABLE('alt, pres, temp, velo')
```

Create a table structure, with each column of information being an individual tag of the table.

Note

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 177, or refer to Chapter 8, *Creating and Querying Tables*, in 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),
CVELO(100)
OPEN (UNIT=1, STATUS='NEW', FILE='aptv.dat')
.
.           Other commands go here.
.
WRITE (1, '(4(1x,G15.5))') ALT, PRES, TEMP,
CVELO
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 167.

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),
CVELO(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),
CPRES(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 PV-WAVE 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.

Tip



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 Table 8-3 on page 154. All of the routines listed in Table 8-3 are described in this section except *ASSOC* and *GET_KBRD*; these important functions are discussed

in *Associated Variable Input and Output* on page 212 and *Getting Input from the Keyboard* on page 222.

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.

Images are treated in the same manner as any PV-WAVE variable. Images may be either square or rectangular. There is no restriction placed on the size of images by PV-WAVE; 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 PV-WAVE variable, and 24-bit images must be stored in a 3-dimensional PV-WAVE variable. For more information about how the RGB information in 24-bit image data is stored, refer to *Image Interleaving* on page 193.



Note

Your workstation or device must support 24-bit color mode if you intend to view 24-bit images with PV-WAVE. To find out if your device has this capability, refer to Appendix A, Output Devices and Window Systems, in the *PV-WAVE User's Guide*.

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 a separate volume, 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 193.

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 212.

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 212.

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.

Note 

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 RGBRGBRGBRGB RGBRGBRGBRGB	RRRRRRRRRRRR GGGGGGGGGGGG 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 189.

READU and WRITEU

READU and WRITEU provide PV-WAVE's 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 PV-WAVE 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 175.

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 PV-WAVE basic data type that does not have a fixed size is the string data type. A PV-WAVE 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.

Note

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 PV-WAVE 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 PV-WAVE 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 PV-WAVE 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 FOR-
    TRAN 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 FOR-
    TRAN 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.

Note 

For information about using similar commands to read a segmented record file created on a VMS system, refer to the example in the next section.

Reading VMS FORTRAN-Generated Binary Data

By default, VMS 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 VMS.

In segmented record files, a single segmented record is written as one or more actual VMS records. Each of the actual records has a two-byte control field prepended that allows FORTRAN to reconstruct the original record.

Example — VMS FORTRAN Program Writes, PV-WAVE Reads

VMS FORTRAN Write

The following VMS 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 VMS 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 VMS 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 VMS 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.

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. For example, a binary data file produced on an HP 9000 Series 700 workstation would differ from one produced on a Sun-4 or a DECstation 3100. 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” representation. All machines supporting XDR (such as Sun and DEC workstations) understand this standard representation, and have the ability to convert between it and their own internal representation.

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. However, XDR is widely available and can be easily moved to any UNIX system.

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 197.)
- 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.

PV-WAVE 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 Table 8-9.

Table 8-9: 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 structure of a PV-WAVE 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_structure* [, *offset*])

For examples showing how to use the *offset* parameter, refer to a later section, *Using the Offset Parameter* on page 216.

Note



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.

VMS 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:

- 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.

For these reasons, associated variables are the most efficient form of I/O.

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 PV-WAVE 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 structure 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 `PV-WAVE` 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 PV-WAVE 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 logical file unit, 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 variable 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.

Caution 

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 PV-WAVE variable by subscripting the file variable with a single subscript, and then access the individual array elements from the PV-WAVE 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 in UNIX

Under UNIX, 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 in VMS

Under VMS, 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 VMS 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 VMS files, refer to *VMS-Specific Information* on page 224; that section contains an overview of how VMS 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 199.

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 the current shell (as specified by the UNIX environment variable SHELL). For example, to determine the number of PV-WAVE procedure files that exist in the current directory:

```
PRINT, '# PV-WAVE CL.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
```



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 PV-WAVE 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 currently 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>

For more information about the INFO command, refer to Chapter 14, *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 if you are running PV-WAVE in a UNIX environment:

```
** 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
REC_LEN       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 PV-WAVE 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
```

This is the end of the read_data function.

Assuming that a file named `herc.dat` exists and contains 10 floating-point values, the following PV-WAVE 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 (PV-WAVE 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.
        ' ' : ..... ; Process letter A.
        ' ' : ..... ; 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
    End of procedure.
```

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 writing 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 with PV-WAVE

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 199.

VMS-Specific Information

VMS I/O is a relatively complex topic, involving a large number of formats and options. VMS 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 a VMS user must consider when transferring data in and out of PV-WAVE.

Organization of the File

A VMS 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 VMS file processing is bypassed).

Caution 

With some file organizations, VMS 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. VMS 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)

Note 

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 VMS 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 VMS Binary Files* on page 149.

VMS 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.



It is worth noting that VMS stream files are record oriented (and therefore, fail to provide much of the flexibility of UNIX stream files) although the VMS standard C library (upon which PV-WAVE is implemented) does a good job of concealing this limitation. Our experience indicates that I/O using VMS 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, VMS uses its carriage control attributes to determine how to output each line:

- **Explicit carriage control** — Specifies that VMS 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>.

- **FORTRAN 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 Table 8-10.

Note 

The default for PV-WAVE is Carriage Return carriage control.

Table 8-10: VMS 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.
36	\$	Prompt. A linefeed precedes the data, but no <Return> follows.
Other		Same as ASCII space character. Single-space 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 VMS 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 VMS CREATE/FDL command to create the file. FDL stands for File Definition Language, and is the standard method for specifying VMS file attributes. The options for creating indexed files are too numerous to cover in this document, but the *VMS File Definition Language Facility Manual* describes FDL in detail.



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"
  SEG0_LENGTH 32
  SEG0_POSITION 0
  TYPE string

KEY 1
  CHANGES yes
```

```

NAME "Salary"
SEG0_LENGTH 4
SEG0_POSITION 32
TYPE bin4

```

Assume that this description resides in a file named `wages.fdl`. The following PV-WAVE 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 VMS, 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 Table 8-11.

Table 8-11: 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.

Note 

To use the PV-WAVE magnetic tape procedures you must define a logical name "MTn" 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 PV-WAVE tape unit number 5, issue the following VMS 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 VMS before entering PV-WAVE. Then, within PV-WAVE, refer to the tape as unit number 5. The PV-WAVE 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 MTn 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
```


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.

A subdirectory `lib/user` exists in the main PV-WAVE directory. The `lib/user` subdirectory contains procedures and functions that can be accessed by all PV-WAVE 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 `lib/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 PV-WAVE Environment* on page 44 of the *PV-WAVE User's Guide*.

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 PV-WAVE 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.

Caution 

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 PV-WAVE compiler that a user-written program module follows. The syntax of these definition statements is:

PRO *Procedure_name*, *p*₁, *p*₂, ..., *p*_{*n*}

FUNCTION *Function_name*, *p*₁, *p*₂, ..., *p*_{*n*}

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.

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.

Note



Under VMS, 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.

Example of a Function

An example of a **PV-WAVE** function to compute the digital gradient of an image is shown. 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 **PV-WAVE** 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 = $\text{abs}(dz/dx) + \text{abs}(dz/dy)$ 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.

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 *Using Executive Commands* on page 26 of the *PV-WAVE User's Guide* 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 22 of the *PV-WAVE User's Guide*.

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 PV-WAVE 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 239, if the filename 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 PV-WAVE 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 subscripted variable, thereby avoiding this restriction in the case of library functions.

System Limits and the Compiler

When a PV-WAVE 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 the section *Using .SIZE* on page 31 of the *PV-WAVE User's Guide*.

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 the section *Using .SIZE* on page 31 of the *PV-WAVE User's Guide*.
- 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 the section *Using .SIZE* on page 31 of the *PV-WAVE User's Guide*.
- Use the .LOCALS executive command to increase the original size of the data area. The .LOCALS command is described in the section *Using .LOCALS* on page 31 of the *PV-WAVE User's Guide*.

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.



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 “inter-

preter" 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 PV-WAVE 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 subscripted 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 261 of the *PV-WAVE Programmer's Guide*.

“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.

VMS Procedure Libraries

Note

The information in this section applies to PV-WAVE running under VMS 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 VMS Procedure Libraries

Note

The information in this section applies to PV-WAVE running under VMS only.

PV-WAVE procedure libraries are simply standard VMS 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 VMS 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.

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 PV-WAVE Users' Library. The purpose of the library is to help PV-WAVE 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 submissions 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 the `lib/user` subdirectory of the main PV-WAVE directory. Procedures and functions in this subdirectory are automatically compiled when they are referenced from within PV-WAVE.

Note

If you are running PV-WAVE under VMS, you must load any new or modified Users' Library routines into the PV-WAVE text library. Text libraries are explained previously in this chapter.

Caution

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. Note that 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 your 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 `/lib/std` subdirectory.

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, 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 250, 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 `ON_ERROR` procedure determines the action taken when an error is detected inside a user-written procedure or function. The possible options for error recovery are given in Table 10-1.

Table 10-1: 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.

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, after debugging. This form of error recovery makes debugging a routine difficult because the routine is exited as soon as an error occurs. Therefore, it should be added once the code is completely tested.

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* keyword 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 PV-WAVE 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 in Chapter 2, *Function and Procedure Reference* 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 rou-
    tine 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 form:

```

    Called from: DIST <wave/lib/dist.pro (27)>

```

Programs can readily parse the traceback information to extract the source file name and line number.

Detection of Math Errors

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/VMS, 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 handles math errors on machines that implement the IEEE floating-point standard, but that *do not properly handle exceptions* by keeping 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 with this routine.

Special Values for Undefined Results

Machines which implement the IEEE standard for binary floating-point arithmetic, such as the Sun, have 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 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, $
    'verflow 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.

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 Table 10-2.

Caution 

Not all machines detect all errors.

Table 10-2: Error Status Code Values

Value	Condition
0	No errors detected since the last interactive prompt or call to <code>CHECK_MATH</code> .
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 <code>SQRT</code> or <code>ALOG</code> 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 mes-
    sages 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 sub-
    sequent 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.

DECstation 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/VMS 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.

Checking for Parameters

The informational routines, `N_ELEMENTS`, `SIZE`, `N_PARAMS`, 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 Keywords

The `KEYWORD_SET` function simply returns a 1 (true), if its parameter is defined and non-zero. Otherwise it returns 0 (false). For example, assume that a procedure is written which performs and returns the result of a computation. 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
```

Checking for 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.
```

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 Table 10-3.

Table 10-3: Type Codes Returned by the `SIZE` Function

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:

<code>b₀</code>	3	Three dimensions
<code>b₁</code>	3	First dimension
<code>b₂</code>	4	Second dimension
<code>b₃</code>	5	Third dimension
<code>b₄</code>	2	Integer type
<code>b₅</code>	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:

$$B = A * 16. / \text{MAX}(A)$$

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:

$$B = A * (16. / \text{MAX}(A))$$

Scale A from 0 to 16 using only one array operation.

or:

$$B = 16. / \text{MAX}(A) * A$$

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 PV-WAVE 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 PV-WAVE 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 the equivalent operation in PV-WAVE 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 PV-WAVE 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 PV-WAVE 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.

PV-WAVE 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 $8 \cdot 512^2$ 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 VMS

VMS, as it comes from DEC, is not tuned for image processing. To get the best performance from PV-WAVE, you should increase the VMS `SYSGEN` parameters, file sizes, and `AUTHORIZE` quotas which restrict the virtual memory system. This discussion is on the most elementary level and the appropriate VMS 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 VMS 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, DEC 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 sizes of these files.

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 PV-WAVE 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
    End of loop.
```

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 284.

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, PV-WAVE 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 PV-WAVE scalar operations, PV-WAVE array operations, and FORTRAN. The times for these operations are shown in the following table:

Table 11-1: 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.

Accessing the Operating System

This chapter discusses the methods for communicating with the operating system from PV-WAVE. The main topics include:

- Manipulating UNIX environment variables
- Manipulating VMS logicals and symbols
- Using SPAWN to access the operating system
- Changing the current directory

The utility routines used to communicate with specific operating systems are summarized in the following table:

Table 12-1: Routines for Accessing the Operating System

Method	UNIX	VMS	Use
SPAWN	Yes	Yes	Lets you spawn a child process to execute commands. The output generated by the commands can be captured in a variable for later processing by PV-WAVE.
GETENV, SETENV, ENVIRONMENT	Yes	No	Manipulate the UNIX environment.
SETLOG, TRNLOG, DELLOG	No	Yes	Manipulate VMS logical names.
SET_SYMBOL, GET_SYMBOL, DELETE_SYMBOL	No	Yes	Manipulate DCL interpreter symbols.
CD, PUSHD, POPD, PRINTD	Yes	Yes	Let you change the current working directory.

Manipulating UNIX Environment Variables

Every UNIX process has an “environment”. The environment consists of “environment variables”, each of which has a string value associated with it. Some environment variables always exist, such as `path`, which tells the shell where to look for programs, or `term` which specifies the kind of terminal being used. You may add other environment variables at any time from an interactive shell, or you can add them to the file, such as the `.login` file, that is executed when you log in.

When a process is created, it is given a copy of the environment from its parent process. PV-WAVE is no exception to this; when started, it inherits a copy of its parent’s environment. The parent

process to PV-WAVE is usually the interactive shell from which it was started. In turn, any child process created by PV-WAVE (such as those from the SPAWN procedure) inherits a copy of PV-WAVE's current environment.

Note 

It is important to realize that environment variables are not a PV-WAVE feature, they are part of every UNIX process. Although they can serve as a form of global memory, it is best to avoid using them in that way. Instead, PV-WAVE system variables and common blocks should be used in that role. This will make your PV-WAVE code portable to non-UNIX based PV-WAVE systems.

Environment variables should be used for communicating with child processes. For example, you can change the SHELL environment variable prior to calling SPAWN. SPAWN then uses the newly defined shell to run its process.

PV-WAVE provides the following procedures and functions for manipulating the UNIX environment. For more information on these procedures and functions, see the *PV-WAVE Reference*.

SETENV: Adding a New Environment Variable

The SETENV procedure adds a new environment variable, or changes the value of an existing environment variable, in the PV-WAVE process. It has the form:

```
SETENV, environment_expression
```

where *environment_expression* is a scalar string containing an environment expression to be added to the environment.

For example, you can change the shell used by SPAWN by changing the value of the SHELL environment variable. A PV-WAVE statement to change to the Bourne shell /bin/sh is:

```
SETENV, 'SHELL=/bin/sh'
```

GETENV: Getting an Environment Variable's Equivalence String

The GETENV function returns the value (equivalence string) of a specified environment variable. It has the form:

result = GETENV(*name*)

where *name* is the name of the environment variable for which the value is desired. If *name* does not exist in the environment, a null string is returned.

For example, to determine the type of terminal being used, you can enter the PV-WAVE statement:

```
PRINT, 'The terminal type is:', GETENV('TERM')
```

Executing this statement on a Sun workstation gives the result:

```
The terminal type is: sun
```

ENVIRONMENT: Getting the Values of All Environment Variables

The ENVIRONMENT function returns a string array containing the values (equivalence strings) of all the environment variables currently found in the PV-WAVE process environment.

The following PV-WAVE statements print the entire environment, one environment variable per line:

```
WAVE> env = ENVIRONMENT()  
Get a copy of the environment.
```

```
WAVE> FOR I = 0, N_ELEMENTS(env)-1 $  
DO PRINT, env(I)  
Print out, one variable per line.
```

Manipulating VMS Logicals and Symbols

PV-WAVE provides the following procedures and functions for manipulating VMS logicals and symbols. For more information on these procedures and functions, see the *PV-WAVE Reference*.

SETLOG: Defining a New Logical

The SETLOG procedure defines a logical name. It has the form:

SETLOG, *lognam*, *value*

where *lognam* is the scalar string containing the name of the logical to be defined and *value* is a string giving the value to which the logical will be set. If *value* is a string array, *lognam* is defined as a multi-valued logical where each element of *value* defines one of the equivalence strings.

TRNLOG: Getting a Logical's Equivalence String

The TRNLOG function searches the VMS name tables for a specified logical name and returns the equivalence string(s) in a PV-WAVE variable. TRNLOG returns the VMS status code associated with the translation as a longword value. As with all VMS status codes, success is indicated by an odd value (least significant bit is set) and failure by an even value. It has the form:

result = TRNLOG(*lognam*, *value*)

where *lognam* is a scalar string containing the name of the logical to be translated and *value* is a named variable into which the equivalence string is placed. If *lognam* has more than one equivalence string, the first one is used.

The following PV-WAVE statements allow you to see these values:

```
WAVE> ret = TRNLOG("SYS$SYSROOT", trans, $
                /Full, /Issue_Error)
                Translate the logical.

WAVE> print, trans
```

DELLOG: Deleting a Logical

The DELLOG procedure deletes a logical name. It has the form:

DELLOG, *lognam*

where *lognam* is a scalar string containing the name of the logical to be deleted.

SET_SYMBOL: Defining a Symbol

The SET_SYMBOL procedure defines a DCL (Digital Command Language) interpreter symbol for the current process. It has the form:

SET_SYMBOL, *name*, *value*

where *name* is a scalar string containing the name of the symbol to be defined and *value* is a scalar string containing the value with which *name* will be defined.

GET_SYMBOL: Getting a Symbol's Value

The GET_SYMBOL function returns the value of a VMS DCL (Digital Command Language) interpreter symbol as a scalar string. If the symbol is undefined, the null string is returned. It has the form:

result = GET_SYMBOL(*name*)

where *name* is a scalar string containing the name of the symbol to be translated.

DELETE_SYMBOL: Deleting a Symbol

The `DELETE_SYMBOL` procedure deletes a DCL (Digital Command Language) interpreter symbol from the current process. It has the form:

`DELETE_SYMBOL, name`

where *name* is a scalar string containing the name of the symbol to be deleted.

Accessing the Operating System Using SPAWN

`SPAWN` is a very flexible command. It can create an interactive command interpreter (shell, for UNIX users) process, or simply issue a single command and return. Under UNIX, in this second case, the command can either be passed to a shell for processing, or it can be executed directly as a child process of PV-WAVE.

This section discusses `SPAWN` as it is used to issue commands and capture output. For detailed information on how to use `SPAWN` to execute and transfer data to and from a child process (external program), see *Interapplication Communication Using SPAWN* on page 309.

Using SPAWN to Issue Commands

The `SPAWN` procedure spawns a child process to execute a given command. It has the form:

`SPAWN [, command [, result]]`

The *command* parameter is a string containing the command to be issued. *command* can be either a scalar or an array. The way in which it is treated depends on whether the *Noshell* keyword is specified. For more information on *Noshell*, see *Avoiding the Shell under UNIX* on page 300.

If the *result* parameter is not present, the output from the child process simply goes to the standard output (usually the terminal). Otherwise, the output from the child process is placed into a string array (one line of output per array element) and assigned to *result*.

Interactive Use of SPAWN

If SPAWN is called without arguments, an interactive shell (UNIX) or command interpreter (VMS) process is started. You can enter one or more shell commands. While you use the shell or command interpreter process, PV-WAVE is suspended. When you exit the child process, control returns to PV-WAVE, which resumes at the point where it left off. The PV-WAVE session remains exactly as you left it.

The following statements demonstrate the interactive use of SPAWN.

An example of using SPAWN in a UNIX environment is:

```
WAVE> SPAWN
%date
Fri Aug 26 13:55:00 MDT 1988
%exit
WAVE>
```

An example of using SPAWN in a VMS environment is:

```
WAVE> SPAWN
$ SHOW TIME
29-JAN-1990 16:32:23
$ LOGOUT
WAVE>
```

Using SPAWN in this manner is equivalent to using the PV-WAVE \$ command. The difference between these two is that \$ can only be used interactively, while SPAWN can be used interactively or in PV-WAVE programs.

UNIX Shells

The most common UNIX shells are the Bourne shell (`/bin/sh`), the C shell (`/bin/csh`), and the Korn shell (`/bin/ksh`). Rather than force you to use a given shell, PV-WAVE follows the UNIX convention of using the shell specified by the UNIX environment variable `SHELL`. If `SHELL` does not exist, the Bourne shell is used. The UNIX environment is discussed in *Manipulating UNIX Environment Variables* on page 292 in this chapter.

Under UNIX, the interactive form of SPAWN is provided primarily for users of the Bourne shell and for compatibility with VMS. Shells that offer process suspension (e.g., `/bin/csh`) offer a more convenient and efficient way to get the same effect.

VMS Command Interpreter

Under VMS, the command interpreter is always DCL (Digital Command Language).

Non-interactive Use of SPAWN

If SPAWN is called with a single argument, that argument is taken as a command to be executed. In this case, PV-WAVE starts a child command interpreter process and passes the command to it. The argument should be a scalar string. The shell executes the command and exits, at which point PV-WAVE resumes operation. This form of operation is very convenient for executing single commands from PV-WAVE programs. For example, it is sometimes useful to delete a temporary scratch file. SPAWN can be used as shown in the following program fragment:

```
OPENW, Unit, 'scratch.dat', /Get_Lun
    Open the scratch file. Use the Get_Lun keyword to allocate a file
    unit.

. . .
    PV-WAVE commands that use the file go here.

FREE_LUN, Unit
    Deallocate the file unit and close the file.
```

```
if (!Version.os EQ 'vms') THEN $
  Cmd = 'DELETE' else Cmd = 'rm'
```

Use the !Version system variable to determine the proper file deletion command for the current operating system.

```
SPAWN, Cmd + 'scratch.dat'
```

Delete the file using SPAWN.

Actually, the *Delete* keyword to the OPEN procedure is a more efficient way to handle this job. The above example serves only to demonstrate the use of the SPAWN procedure.

Avoiding the Shell under UNIX

As mentioned above, SPAWN usually creates a shell process and passes the command to this shell, instead of simply creating a child process to directly execute the command. This default action is taken because the shell provides useful actions such as wildcard expansion and argument processing. Although this is usually desirable, creating a shell process has the drawback of being slower than necessary; it simply takes longer to start a shell. However, it is possible to avoid using the shell by using the *Noshell* keyword.

When SPAWN is called and *Noshell* is present and non-zero, the command is executed as a direct child process, avoiding the extra overhead of starting a shell. This is faster, but since there is no shell to break the command into separate arguments, you have to do it. Every UNIX program is called with a series of arguments. When you issue a shell command, you separate the arguments with white space (blanks and tabs). The shell then breaks up the command into an array of arguments, and calls the command (the first word of the command), passing it the array of arguments.

Thus, if you use *Noshell* to avoid using a shell, you have to break up the arguments yourself. In this case, the *command* argument should be a string array. The first element of the array is the name of the command to use, and the following elements contain the arguments.

For example, consider the command:

```
WAVE> SPAWN, 'ps ax'
```

which uses the UNIX `ps` command to show running processes on the computer. To issue this command without a shell, you write it as:

```
WAVE> SPAWN, ['ps', 'ax'], /Noshell
```

Capturing Output

By default, any output generated by the spawned command is sent to the standard output, which is usually the terminal. It is possible to capture this output in a PV-WAVE string array by calling SPAWN with a second argument. If this second argument, called *result*, is present, all output from the child process is put into a string array, one line of output per array element, and is assigned to *result*. For example, the following PV-WAVE statements can be used to give a simplistic count of the number of users logged onto the computer:

```
if (!Version.os EQ 'vms') THEN $
  Cmd = 'SHOW USERS' ELSE Cmd = 'who'
  Use the !Version system variable to determine the com-
  mand to use.

SPAWN, Cmd, Users
  Issue the command; catch the result in a string array.

N = N_ELEMENTS(Users)
  Count how many lines of output cue back. Under UNIX, this is
  the number of users logged in.

if (!Version.os EQ 'vms') THEN N = N - 5
  VMS outputs five extra header lines which are not actual users.

PRINT, 'There are ', N, ' users logged on.'
  Give the result.
```

Changing the Current Working Directory

Like every process, PV-WAVE has a current working directory. This is the default directory that is used whenever you specify a file without explicitly supplying the directory. The initial working directory is the directory you were in when you issued the command to start PV-WAVE.

Using the CD Procedure

You can use the CD command to change this working directory at any point during the PV-WAVE session. This new working directory affects the current session, and any child processes that you start from PV-WAVE, but it does not change the current directory of the process that started PV-WAVE. Therefore, when you exit PV-WAVE, you will find yourself back in the directory you were in when you started.

On a UNIX system, to change the current directory to /usr/stardata, enter:

```
WAVE> CD, '/usr/stardata'
```

On a VMS system, to change the current directory to SYS\$SYSDEVICE:[STARDATA], enter:

```
WAVE> CD, 'SYS$SYSDEVICE:[STARDATA]'
```

In order to change to your login directory, you can provide a null argument. In addition, the *Current* keyword can be used to save the current directory before any change is made. The following command saves the current directory and then changes it to your home directory:

```
WAVE> CD, '', Current=OLDDIR
```

Later, you can restore the current directory to its previous value with the command:

```
WAVE> CD, OLDDIR
```

Using the PUSH, POP, and PRINTD Procedures

The PUSH, POP, and PRINTD procedures are provided to make interactive use of CD more convenient by maintaining a stack of directories. PUSH saves the current directory on the top of the stack and then changes it to the directory given by its argument. POP sets the current directory to the directory at the top of the stack and removes that directory from the stack. PRINTD shows you the current entries on the stack. Using these user procedures, the example above could be written:

```
WAVE> PUSH, ''  
      Change to your home directory.  
  
  . . .  
      Execute some statements.  
  
WAVE> POP  
      Return to the original working directory.
```


Interapplication Communication

PV-WAVE provides a variety of methods for interapplication communication. For example:

- PV-WAVE can execute external programs and exchange data with them. Depending on the method used, the exchange of data can be *unidirectional* (one-way) or *bidirectional* (two-way).
- External programs can call PV-WAVE to perform graphics, data manipulation, and other functions. Again, depending on the method used, the communication can be unidirectional or bidirectional.

Methods of Interapplication Communication

The following table summarizes the methods of interapplication communication that can be used between PV-WAVE and other external applications. This table and the following section, *Choosing the Best Method* on page 307, can help you to determine the most appropriate method of interapplication communication to accomplish a desired task. Each method listed is described in detail later in this chapter.

Table 13-1: Methods of Interapplication Communication

Method	UNIX	VMS	Use
SPAWN	Yes	Yes	A PV-WAVE system routine that executes an external program from within PV-WAVE. Allows data to be transferred to and from PV-WAVE via bidirectional pipes and PV-WAVE's standard I/O facilities. See page 309.
waveinit, wavecmd, waveterm	Yes	No	Routines that allow a C or FORTRAN program to start PV-WAVE, execute commands, and exit PV-WAVE. No data is transferred back to the calling program. Available on UNIX systems only. See page 313.
LINKNLOAD (for use on: AIX, SunOS, HPUX, VMS, and Solaris only)	Yes *	Yes	A PV-WAVE system routine that allows PV-WAVE to call an external function via dynamic linking of shareable objects. It is the simplest method for attaching your own C code to PV-WAVE. Allows the transfer of binary data. See page 319.
cwavec	Yes	Yes	A routine that allows a statically linked C program to access PV-WAVE. Data is transferred between the C program and PV-WAVE via the wavevars routine. See page 333.
cwavefor	Yes	Yes	Works like cwavec, except from a statically linked FORTRAN program. See page 333.
CALL_UNIX	Yes	No	A PV-WAVE system routine that uses Remote Procedure Call (RPC) technology to allow PV-WAVE to call a separate application across a UNIX network. See page 376.
CALL_WAVE	Yes	No	A PV-WAVE system routine that uses Remote Procedure Call (RPC) technology to allow a separate application to call PV-WAVE across a UNIX network. See page 376.

* Not currently available for all versions of UNIX.

Choosing the Best Method

It is important to select the most appropriate method of interapplication communication for your particular needs. Choosing the wrong method often requires much more work than is necessary to accomplish a given task.

This section describes typical scenarios where some kind of interapplication communication is required. After each scenario is described (in italics), a suitable solution for interapplication communication is suggested.

I'm running PV-WAVE, and I want to execute an external program I've written. I'm not really concerned about returning anything to PV-WAVE.

This is the simplest case of interapplication communication. The SPAWN procedure is the best choice. SPAWN executes an external program, or an operating system command, from PV-WAVE. (Although it is not a requirement in this scenario, SPAWN can return data from the external application to PV-WAVE.)

For information on using SPAWN, see *Interapplication Communication Using SPAWN* on page 309.

I'm on a UNIX system, and I just want to call PV-WAVE from my C or FORTRAN program, execute some PV-WAVE commands, and exit PV-WAVE. It isn't necessary for my program to retrieve any data from PV-WAVE.

The routines `waveinit`, `wavecmd`, and `waveterm` can be used to accomplish this sort of task. These routines, which are only available under UNIX, allow one-way (unidirectional) communication from a C or FORTRAN application to PV-WAVE. They start PV-WAVE, execute specified commands, and exit PV-WAVE. No data is transferred back to the calling program.

For information on using `waveinit`, `wavecmd`, and `waveterm`, see *Executing PV-WAVE Commands Externally* on page 313.

I wrote a C program, and I want to be able to link it dynamically with PV-WAVE. My program needs to be able to access data directly from the data space of PV-WAVE. When my program is finished running, I want control returned back to PV-WAVE.

The LINKNLOAD procedure is the simplest method for attaching your own code to PV-WAVE. LINKNLOAD is a PV-WAVE system procedure that calls a function in an external sharable object. When used in conjunction with the `wavevars` function, data can be passed back and forth between the user-written routine and PV-WAVE.

For information on LINKNLOAD, see *Using LINKNLOAD to Call External Programs* on page 319. For information on the data transfer function `wavevars` see *Accessing Data in PV-WAVE Variables* on page 350.

I want to be able to call PV-WAVE from a C or FORTRAN program I've written. I want the program to be statically linked with PV-WAVE.

The `cwavec` function allows a statically linked C program to access PV-WAVE's data space. Data is transferred between the C program and PV-WAVE via the `wavevars` routine. In addition, the `cwavefor` function allows a statically linked FORTRAN program to access PV-WAVE's data space.

For information on `cwavec` and `cwavefor` see *Calling PV-WAVE in a Statically Linked Program* on page 333. For information on the data transfer function `wavevars` see *Accessing Data in PV-WAVE Variables* on page 350.

I have an application running across the UNIX network that I want my PV-WAVE program to communicate with.

Under UNIX, Remote Procedure Calls (RPCs) can be used to facilitate this kind of communication.

For information on interapplication communication routines that support RPCs, see *Remote Procedure Call Examples* on page 376.

Interapplication Communication Using SPAWN

This section explains how to use SPAWN to communicate with a child process (external program).

Communicating with a Child Process

In the previous chapter, the SPAWN procedure was used to start a child process and execute PV-WAVE commands. The PV-WAVE process waited until the child process was finished before continuing. The communication was one-way and only a single “transaction” was completed.

It is also possible to start a child process using SPAWN and continue the PV-WAVE process without waiting for the child process to finish. To do this, PV-WAVE attaches a bidirectional pipe to the standard input and output of the child process. This pipe appears in the PV-WAVE process as a normal logical file unit. Once a process has been started in this way, the normal PV-WAVE Input/Output facilities are used to communicate with it. The ability to use a child process in this manner allows you to solve specialized problems using other languages, and to take advantage of existing programs.

Starting the Child Process

In order to start such a process, the *Unit* keyword is used with SPAWN to specify a named variable into which the logical file unit number will be stored. Once the child process has done its work, the FREE_LUN procedure is used to close the pipe and delete the process.

When using a child process in this manner, it is important to understand the following points:

- The EOF function always returns false when applied to a pipe. This means that it is not possible to use this function to know when the child process is finished. As a result, the child process must be written in such a way that the controlling PV-WAVE procedure knows how much data to send and how much is coming back.
- A UNIX pipe is simply a buffer maintained by the operating system. It has a fixed length, and can therefore become completely filled. When this happens, the operating system puts the process that is filling the pipe to sleep until the process at the other end consumes the buffered data. The use of a bidirectional pipe can therefore lead to deadlock situations in which both processes are waiting for each other. This can happen if the parent and child processes do not synchronize their reading and writing activities.
- Most C programs use the input/output facilities provided by the Standard C Library `stdio`. In situations where PV-WAVE and the child process are carrying on a running dialog (as opposed to a single transaction), the normal buffering performed by `stdio` on the output file can cause communications to hang. We recommend calling the `stdio` function `setbuf()` as the first statement of the child program to eliminate such buffering:

```
(void) setbuf(stdout, (char *) 0);
```

It is important that this statement occur before any output operation is executed, otherwise it will have no effect.

Example: Communicating with a Child Process Using SPAWN

This example assumes you have a C program, `test_pipe.c`, that accepts floating point values from its standard input and returns their average on the standard output. The code for such a C program is shown next. An explanation of the program is given immediately after the listing.

Note

You can find the following example file in:

```
$WAVE_DIR/demo/interapp/spawn/test_pipe.c

#include <stdio.h>
/* System error number */
extern int errno;
/* System error messages */
extern char *sys_errlist[];
/* Length of sys_errlist*/
extern int sys_nerr;
main()
{
    float *data, total = 0.0;
    long i, n;
    /* Make sure the output is not buffered */
    setbuf(stdout, (char *) 0);
    /* Find out how many points */
    if (!fread(&n, sizeof(long), 1, stdin))
        goto error;
    /* Get memory for the array */
    if (!(data = (float *) malloc((unsigned)
        (n * sizeof(float)))) goto error;
    /* Read the data */
    if (!fread(data, sizeof(float), n, stdin))
        goto error;
    /* Calculate the average */
    for (i=0; i < n; i++) total += data[i];
    total /= (float) n;
    /* Return the answer */
    if (!fwrite(&total, sizeof(float), 1,
        stdout)) goto error;
    return;
error:
    fprintf(stderr, "test_pipe: %s\n",
        sys_errlist[errno]);
}
```

This C program returns a single-precision floating-point value, which is the average of the input values. The program also reads a long integer that tells how many data points to expect.

Since the amount of input and output for this program is explicitly known, and because it reads all of its input at the beginning and writes all of its results at the end, a deadlock situation as described in the previous section cannot occur.

Note

In actual practice, such a trivial program would never be used from PV-WAVE; it is simpler and more efficient to perform the calculation within PV-WAVE. It does, however, serve to illustrate the method by which significant programs can be called from PV-WAVE.

Using SPAWN to Access the C Program from PV-WAVE

The following PV-WAVE statements use `test_pipe` to determine the average of the values 0 through 9:

```
SPAWN, 'test_pipe', Unit=unit, /Noshell
```

Start `test_pipe`. The use of the `Noshell` keyword is not necessary, but speeds up the startup process.

```
WRITEU, unit, 10L, FINDGEN(10)
```

Send the number of points followed by the actual data.

```
answer = 0.0
```

```
READU, unit, answer
```

```
PRINT, "Average = ", answer
```

```
FREE_LUN, unit
```

Close the pipe, delete the child process, and deallocate the logical file unit.

Executing these statements gives the result:

```
Average = 4.50000
```

This mechanism provides you with a simple and efficient way to augment PV-WAVE with code written in other languages such as C or FORTRAN. In this case, however, it is not as efficient as writing the required operation entirely in PV-WAVE. The actual cost depends primarily on the amount of data being transferred. For

example, the above example can be performed entirely in PV-WAVE using a simple statement like:

```
PRINT, 'Average = ', TOTAL(FINDGEN(10))/10.0
```

The PV-WAVE calculation is always faster; however, the difference may only be significant when a large amount of data is transferred.

Executing PV-WAVE Commands Externally

The routines `waveinit`, `wavecmd`, and `waveterm` let you execute PV-WAVE from an external program, such as a C or FORTRAN program. The first routine, `waveinit`, starts PV-WAVE. The second routine, `wavecmd`, sends commands to PV-WAVE. The third routine, `waveterm`, ends the PV-WAVE session.

Note

These routines allow one-way (unidirectional) communication only. That is, PV-WAVE cannot pass data back to the calling program. If you require data to be passed back to the calling program (bidirectional transfer), then choose another method of interapplication communication.

Compiling the External Program

After these routines have been incorporated into an external program, the program must be compiled with the correct object module. The object module is named `callwave.o`, and its full pathname is the following, where *arch* is the machine architecture, such as `hps700` or `sun4`:

```
$WAVE_DIR/bin/bin.arch/callwave.o
```

If you are unsure about the architecture, you can get the information by typing the following command:

```
$WAVE_DIR/bin/arch
```

Once you know the architecture, you can compile your program. For example, if you were on a Sun-4, you would type one of the following commands:

```
cc -o myprog myprog.c \  
    $WAVE_DIR/bin/bin.sun4/callwave.o
```

or

```
f77 -o myprog myprog.f \  
    $WAVE_DIR/bin/bin.sun4/callwave.o
```

Note 

Compile and link options will vary by platform, and are sometimes site specific. Refer to the man page for your compiler for more detailed information on how to compile programs on your system.

Starting PV-WAVE from an External Program with waveinit

The initialization routine `waveinit` starts PV-WAVE. The routine first checks the environment variable `WAVE_DIR`. If `WAVE_DIR` is defined, the routine uses the path `$WAVE_DIR/bin/wave` to start PV-WAVE. When `WAVE_DIR` is not defined, the routine uses the path `/usr/local/bin/wave`. The last part of the path (`wave`) may be set to a symbolic link. When this path (`/usr/local/bin/wave`) is used, the path `/usr/local/lib/wave` must also be a valid path. The last part of this path may be set to a symbolic link that points to the main PV-WAVE directory.

The `waveinit` function has one output parameter, the name of a file to contain the PV-WAVE alphanumeric output (not the graphics output). For example, you can specify a character string denoting the filename or you can specify a null string, denoting that no alphanumeric output should be produced. Suppose you have a C program and you do not want to save the output, you can use:

```
waveinit("");
```


If you want the alphanumeric output written to a file (e.g., `wave.out`), use the following:

```
waveinit("wave.out");
```

To do the same thing from a FORTRAN program, you would use the following two commands:

```
CALL WAVEINIT('');
```

or

```
CALL WAVEINIT('wave.out');
```

Sending Commands to PV-WAVE with wavecmd

To send commands to PV-WAVE, you must use the `wavecmd` routine. The routine's single parameter is the string you want to send to PV-WAVE. For example, to plot a vector [1,2,3,4,5] from a C program you would use:

```
wavecmd("plot, [1,2,3,4,5]");
```

From a FORTRAN program:

```
CALL WAVECMD('PLOT, [1,2,3,4,5]')
```

The `wavecmd` routine can be called as many times as required.

Ending the Session with PV-WAVE with waveterm

When you are finished sending commands to PV-WAVE, you must call the routine `waveterm`. This routine ends the session with PV-WAVE and closes the necessary files. In a C program, type:

```
waveterm();
```

and in a FORTRAN program type:

```
CALL WAVETERM();
```

Example: Calling PV-WAVE from a C Program

The following C code sample shows how to pass a 5-element array to PV-WAVE, have PV-WAVE perform some calculations, and produce a surface plot.

Note

You can find the following listed file in:

```
$WAVE_DIR/demo/interapp/wavecmd/example.c
```

```
#include <stdio.h>
#include <string.h>
#define MAXSIZE 128

main()
{
/*
 * Variables for array passing
 */
    char buf[MAXSIZE];
    char temp[MAXSIZE];
/*
 * Variables for calculations
 */
    int i,x[5];
/*
 * Perform some calculations
 */
    for (i=0; i< 5; i++) x[i] = i * 4;
/*
 * Start PV-WAVE sending the alphanumeric
 * output to 'wave.save'.
 */
    waveinit("wave.save");
/*
 * Send the commands to PV-WAVE. First, we
 * need to send the array to PV-WAVE.
 */
```

```

    sprintf(buf, "a=[%d", x[0]);
    for (i=1; i< 5; i++) {
        sprintf(temp, "%d", x[i]);
        strcat(buf, temp);
    }
    strcat(buf, " ]");
    wavecmd(buf);
/*
 * Next, we perform a matrix multiplication.
 * Then we print the newly formed
 * two-dimensional array, as well as display
 * it as a surface.
 */
    wavecmd("b = a # a");
    wavecmd("print, b");
    wavecmd("surface, b");
/*
 * The following WAIT is needed so that the
 * surface will not be deleted as soon as it
 * has been drawn.
 */
    wavecmd("wait, 3.0");
/*
 * Since we are done sending commands to
 * PV-WAVE, we must call waveterm.
 */
    waveterm();
}

```

Example: Calling PV-WAVE from a FORTRAN Program

The following FORTRAN code fragment shows how you can use PV-WAVE to manipulate and plot data.

Note 

You can find the following listed file in:

```
$WAVE_DIR/demo/interapp/wavecmd/examplefor.f
```

```
PROGRAM TEST
C
C Start PV-WAVE and send the alphanumeric
C output to 'wavefor.save'.
C
CALL WAVEINIT('wavefor.save')
C
C Send my commands to PV-WAVE. First, we
C define a 5-element array. Next, we perform
C a matrix multiplication. Then we print the
C newly formed two-dimensional array and
C display it as a surface.
C
CALL WAVECMD('A =[110,90,27,48,60]')
CALL WAVECMD('B =A # A')
CALL WAVECMD('PRINT, B')
CALL WAVECMD('SURFACE, B')
C The following WAIT is needed so that the
C surface will not be deleted as soon as it
C has been drawn.
C
CALL WAVECMD('WAIT, 3.0')
C
C Since we are done sending commands to
C PV-WAVE, we must call waveterm.
C
CALL WAVETERM()
END
```

Using LINKNLOAD to Call External Programs

The LINKNLOAD function provides simplified access to external routines in shareable images. LINKNLOAD calls a function in an external sharable object and returns a scalar value. Parameters are passed through PV-WAVE to the specified external function by reference, thus allowing the external function to alter values of PV-WAVE variables. It is the simplest method for attaching your own C code to PV-WAVE. Unlike SPAWN, LINKNLOAD allows the sharing of binary data without duplication (transferal) overhead.

Usage

result = LINKNLOAD(*object*, *symbol* [, *param*₁, ..., *param*_{*n*}])

Parameters

object — A string specifying the filename, optionally including file path, of the sharable object file to be linked and loaded.

symbol — A string specifying the function name (symbol entry point) to be invoked in the shared object file.

*param*_{*i*} — The data to be passed as a parameter to the function.

For more detailed information on the LINKNLOAD parameters and optional keywords see the discussion of LINKNLOAD in the *PV-WAVE Reference*.

Discussion

LINKNLOAD lets you call a C function (or a FORTRAN function) from PV-WAVE almost as if you were calling a PV-WAVE function. The called function can obtain information from PV-WAVE through passed parameters or by accessing PV-WAVE's variables directly (see *Accessing Data in PV-WAVE Variables* on page 350).

Any PV-WAVE data type, except a structure, can be passed as a parameter to a C or FORTRAN routine. Parameters are always passed by reference (not by value), and thus it is up to the programmer's discretion whether or not the C or FORTRAN function alters the parameter's value. Parameters are passed in the traditional C fashion of *argc* and *argv*. The C function must know the type to expect for each parameter and must cast it to a C variable of the correct type.

For FORTRAN, since the parameters are passed as pointers, functions are provided to access their values. These functions are:

- `wlnl_getbyte`
- `wlnl_getshort`
- `wlnl_getlong`
- `wlnl_getfloat`
- `wlnl_getdouble`
- `wlnl_getcomplex`
- `wlnl_getstring`

For detailed information on these functions, see the file:

```
WAVE_DIR/util/variables/README
```

LINKNLOAD can be used only on operating systems that support dynamic linking. These operating systems currently include AIX, HPUNIX, SunOS, Solaris, and VMS 5.0.

Caution 

Make sure the number, type, and dimension of the parameters passed to the external function match what the external function expects (this can most easily be done from within PV-WAVE before calling LINKNLOAD). Furthermore, the length of string parameters must not be altered and multi-dimensional arrays are flattened to one-dimensional arrays.

Accessing the Data in PV-WAVE Variables

The `wavevars` function can be used to access the results generated by PV-WAVE in a user-written application called with LINKNLOAD. `wavevars` is a C function that can be invoked from code linked to PV-WAVE (either statically or dynamically) to obtain data from PV-WAVE's data variable space.

The `wavevars` calling sequence is:

```
result = wavevars(&argc, &argv, &argp);
```

For detailed information on `wavevars`, see *Accessing Data in PV-WAVE Variables* on page 350.

The `wavevars` function can only be used from a C function. It cannot be used from a FORTRAN function; however, a FORTRAN function could be used in conjunction with a C wrapper to accomplish the same task. See *Example 4* on page 330 for information on this technique.

Example 1: Calling a C Program

In this example, parameters are passed to the C external function using the conventional (`argc`, `argv`) UNIX strategy. `argc` indicates the number of data pointers which are passed from PV-WAVE within the array of pointers called `argv`. The pointers in `argv` can be cast to the desired type as the following program demonstrates.

Note

You can find the following listed file in:

```
$WAVE_DIR/demo/interapp/linknload/example.c
```

```
#include <stdio.h>
#include "wavevars.h"
long WaveParams(argc, argv)
    int argc;
    char *argv[];
{
    char *b;
    short *s;
```

```

long *l;
float *f;
double *d;
complex *c;
char **str;
if (argc != 7) {
    fprintf(stderr, "wrong # of parameters\n");
    return(0);
}

b = ((char **)argv)[0];
s = ((short **)argv)[1];
l = ((long **)argv)[2];
f = ((float **)argv)[3];
d = ((double **)argv)[4];
c = ((complex **)argv)[5];
str = ((char ***)argv)[6];
fprintf(stderr, "%d %d %ld %g %g <%g%gi>
    '%s'\n", (int) b[0], (int) s[0], (long) l[0],
    f[0], d[0], c[0].r, c[0].i, str[0]);
return(12345);
}

```

Compiling the Example C Routine

The following commands illustrate how to compile the example C routine to produce a sharable object on different platforms. Please refer to the appropriate operating system documentation for more information on these commands.

```

aix% cc example.c -o example.so -e \
    WaveParams -I$WAVE_DIR/util/variables

```

```

hpux% cc -c +z example.c \
    -I$WAVE_DIR/util/variables

```

```

hpux% ld -b -o example.sl example.o

```



```

sunos% cc -c -pic example.c \
        -I$WAVE_DIR/util/variables

sunos% ld -o example.so -assert pure-text \
        example.o

solaris% cc -c -pic example.c \
        -I$WAVE_DIR/util/variables

solaris% ld -G -o example.so example.o

vms$ CC EXAMPLE

vms$ DEFINE WAVE_IMAGE WAVE_DIR:[BIN]WAVE_X.EXE

vms$ LINK EXAMPLE, SYS$INPUT/OPT/SHARE
        WAVE_IMAGE/SHAREABLE UNIVERSAL=WAVEPARAMS

vms$ DEFINE EXAMPLE_SO -
        DISK$:[homedir]EXAMPLE.EXE

```

Caution 

Under AIX 3.2, the symbol entry point must be specified when the external sharable object is built, by using the “-e” flag, and thus the function symbol parameter to LINKNLOAD has no effect under AIX.

Accessing the External Function with LINKNLOAD

The following PV-WAVE code demonstrates how the C function defined in this example could be invoked.

```

ln = LINKNLOAD('example.so', 'WaveParams', $
        byte(1),2,long(3), float(4),double(5), $
        complex(6,7), 'eight')

```

The resulting output is:

```

1 2 3 4 5 <6,7i> 'eight'

```

Using the INFO command, you can see that LINKNLOAD returns the scalar value 12345 as expected.

```

INFO, ln
      LN      LONG      =      12345

```

The example program works with both scalars and arrays since the actual C program above only looks at the first element in the array and since PV-WAVE collapses multi-dimensional arrays to one-dimensional arrays:

```
ln = LINKNLOAD('example.so', 'WaveParams', $
  [byte(1)], [[2,3],[4,5]], [long(3)], $
  [float(4)], [double(5)], [complex(6,7)], $
  ['eight'])
```

The resulting output is:

```
1 2 3 4 5 <6,7i> 'eight'
```

Example 2: Calling a FORTRAN Program

In this example, parameters are passed from PV-WAVE to the FORTRAN external function. Variables from PV-WAVE are passed as pointers to the FORTRAN function. In the FORTRAN function, the `wlnl_get*` functions are used to retrieve the values of the variables. See *Discussion* on page 319 for more information on these functions.

Note

You can find the following listed file in:

```
$WAVE_DIR/demo/interapp/linknload/examplefor.f
```

```
INTEGER*4 FUNCTION WAVEPARAMS(ARGC,ARGP)
  INTEGER*4 ARGC, ARGP(*)

  BYTE B
  INTEGER*2 S
  INTEGER*4 L
  REAL*4 F
  DOUBLE PRECISION D
  STRUCTURE /CMLPX/
    REAL R,I
  END STRUCTURE
  RECORD /CMLPX/      C
```

```
INTEGER*4 RET
CHARACTER*80 STR
INTEGER*4 NCHAR
```

```
INTEGER*4          WLNL_GETLONG
INTEGER*4          WLNL_GETSTRING
INTEGER*4          WLNL_GETCOMPLEX
INTEGER*2          WLNL_GETSHORT
BYTE              WLNL_GETBYTE
REAL              WLNL_GETFLOAT
DOUBLE PRECISION  WLNL_GETDOUBLE
```

```
IF (LOC(ARGC) .NE. 7) THEN
PRINT *, 'Wrong # of parameters', LOC(ARGC)
WAVEPARAMS = 0
RETURN
ENDIF
```

```
B = WLNL_GETBYTE(%VAL(ARGP(1)))
S = WLNL_GETSHORT(%VAL(ARGP(2)))
L = WLNL_GETLONG(%VAL(ARGP(3)))
F = WLNL_GETFLOAT(%VAL(ARGP(4)))
D = WLNL_GETDOUBLE(%VAL(ARGP(5)))
RET = WLNL_GETCOMPLEX(%VAL(ARGP(6)), C.R, C.I)
NCHAR = WLNL_GETSTRING(%VAL(ARGP(7)), STR, 80)
```

```
PRINT 100, B, S, L, F
100  FORMAT(' BYTE=' , I3, ', SHORT=' , I6, ', LONG=' , I8, ', REAL=' , F10.5)
PRINT 200, D, C.R, C.I
200  FORMAT(' DOUBLE=' , D15.5, ', COMPLEX=[', F10.5, ', ', F10.5, ', ]')
PRINT *, 'STRING=' , STR, ', NCHAR=' , NCHAR
```

```
WAVEPARAMS = 1
RETURN
END
```

Compiling the Example FORTRAN Routine

The following commands illustrate how to compile the example FORTRAN routine to produce a sharable object on different platforms. Please refer to the appropriate operating system documentation for more information on these commands.

```
aix% xlf -c -Pv -Wp,-gv examplefor.f
```

```
aix% xlf examplefor.o -o examplefor.so \  
-e waveparams \  
-bI:$WAVE_DIR/util/variables/wavevars.imp
```

```
hpux% f77 -c +z examplefor.f
```

```
hpux% ld -b -o examplefor.sl examplefor.o -lcl \  
-lisamstub -lnlsstubs -lc
```

```
sunos% setenv LD_LIBRARY_PATH /usr/lang/SC0.0
```

NOTE: The correct setting of LD_LIBRARY_PATH may vary depending on how your operating system was installed. The recommended directory is /usr/lang. If necessary, see your system administrator to determine the correct directory.

```
sunos% f77 -c -pic examplefor.f
```

```
sunos% ld -o examplefor.so examplefor.o \  
-Bstatic -lF77 -lm -lc
```

```
solaris% setenv LD_LIBRARY_PATH /opt/SUNWspro/lib
```

NOTE: The correct setting of LD_LIBRARY_PATH may vary depending on how your operating system was installed. The recommended directory is /opt/SUNWspro/lib. If necessary, see your system administrator to determine the correct directory.

```
solaris% f77 -c -pic examplefor.f
```

```
solaris% ld -G -o examplefor.so examplefor.o \  
-lF77 -lm -lc
```

```

vms$ F77 EXAMPLEFOR

vms$ DEFINE WAVE_IMAGE WAVE_DIR:[BIN]WAVE_X.EXE

vms$ LINK EXAMPLEFOR, SYS$INPUT/OPT/SHARE
      WAVE_IMAGE/SHAREABLE UNIVERSAL=WAVEPARAMS

vms$ DEFINE EXAMPLEFOR_SO
      DISK$:[homedir]EXAMPLEFOR.EXE

```

Accessing the External Function with LINKNLOAD

The following PV-WAVE code demonstrates how the FORTRAN function defined in this example could be invoked.

```

ln = LINKNLOAD('examplefor.so', 'waveparams', $
              byte(1), 2, long(3), float(4), double(5), $
              complex(6, 7), 'eight')

```

The resulting output is:

```

1 2 3 4 5 <6,7i> 'eight'

```

Using the INFO command, you can see that LINKNLOAD returns the scalar value 12345 as expected.

```

INFO, ln
      LN      LONG      =      12345

```

The example program works with both scalars and arrays since the actual FORTRAN program above only looks at the first element in the array and since PV-WAVE collapses multi-dimensional arrays to one-dimensional arrays:

```

ln = LINKNLOAD('examplefor.so', 'waveparams', $
              [byte(1)], [[2, 3], [4, 5]], [long(3)], $
              [float(4)], [double(5)], [complex(6, 7)], $
              ['eight'])

```

The resulting output is:

```

1 2 3 4 5 <6,7i> 'eight'

```

Example 3

In this example, the PV-WAVE program calls the C function (via LINKNLOAD), passing it parameters. The C function modifies these parameters and also accesses PV-WAVE's variable data space directly. The C function then returns control to PV-WAVE, passing to PV-WAVE the result of the function.

This example contains two programs:

- `wave_to_c_main.pro` – A PV-WAVE main program that uses LINKNLOAD to call a C function. The PV-WAVE program passes parameters to the C function, which modifies the parameters' values. When control is returned to PV-WAVE, the values of the PV-WAVE variables which were passed as parameters are changed.
- `c_from_wave.c` – The C function to be called by PV-WAVE via LINKNLOAD.

On a UNIX system, the C and PV-WAVE code described in this example is available online in the directory:

```
$WAVE_DIR/demo/interapp/linknload
```

On a VMS system, the C and PV-WAVE code described in this example is available online in the directory:

```
WAVE_DIR:[DEMO.INTERAPP.LINKNLOAD]
```

The C function must be compiled as a shareable object, as explained below. It is because the C function is linked shareable and shares the same data space with PV-WAVE that the C function can access PV-WAVE variables directly.

Compiling the Example C Routine

The following commands illustrate how to compile the example C routine to produce a sharable object. Please refer to the appropriate operating system documentation for more information on these commands.

```

aix% cc c_from_wave.c -o c_from_wave.so \
      -e c_from_wave \
      -I$WAVE_DIR/util/variables
      -bI:$WAVE_DIR/util/variables/wavevars.imp

```

Under AIX, the file wavevars.imp must be bound to the object file, as demonstrated in the above commands.

```

hpux% cc -c +z c_from_wave.c \
      -I$WAVE_DIR/util/variables
hpux% ld -b -o c_from_wave.sl c_from_wave.o

```

```

sunos% cc -c -pic c_from_wave.c \
      -I$WAVE_DIR/util/variables
sunos% ld -o c_from_wave.so -assert pure-text \
      c_from_wave.o

```

```

solaris% cc -c -pic c_from_wave.c \
      -I$WAVE_DIR/util/variables
solaris% -G -o c_from_wave.so c_from_wave.o

```

```

VMS$ CC C_FROM_WAVE
      DEFINE WAVE_IMAGE -
      WAVE_DIR:[BIN]WAVE_X.EXE
VMS$ LINK C_FROM_WAVE, -
      SYS$INPUT/OPT/SHARE
      WAVE_IMAGE/SHAREABLE
      UNIVERSAL=C_FROM_WAVE
VMS$ DEFINE C_FROM_WAVE_SO -
      DISK$:[HOMEDIR]C_FROM_WAVE

```

Running the Example Program

Enter PV-WAVE and type the following at the WAVE> prompt:

```
.RUN wave_to_c_main
```

Note

Under Solaris, you need to do set the LD_LIBRARY_PATH environment variable to the directory containing the file c_from_wave.so. You must set this variable even if that file is in the current working directory. For example:

```
solaris% setenv LD_LIBRARY_PATH .
```

Example 4

In this example, the PV-WAVE program calls a C wrapper function, passing it PV-WAVE variables as parameters. The C wrapper calls the FORTRAN function, passing along the same parameters. The FORTRAN function modifies the values of the PV-WAVE variables it receives as parameters. The FORTRAN function then returns control to PV-WAVE, passing to PV-WAVE the result of the function. The new values, assigned to the PV-WAVE variables by the FORTRAN program, are accessible within PV-WAVE upon return from the FORTRAN program.

This example contains three programs:

- `wave_to_fort_main.pro` – A PV-WAVE main program that uses LINKNLOAD to call a FORTRAN function. The PV-WAVE program passes parameters to a C wrapper, which in turn calls the desired FORTRAN function to modify the PV-WAVE parameters. When control is returned to PV-WAVE, the values of the PV-WAVE variables which were passed as parameters are changed.
- `wave_to_fort_w.c` – A C function that is called by PV-WAVE, via LINKNLOAD. This routine is a C wrapper that allows a PV-WAVE program to pass parameters to and invoke a FORTRAN function. The FORTRAN function can modify the values of the PV-WAVE variables passed as parameters and the modified values will be accurately reflected upon return to PV-WAVE.
- `fort_from_wave.f` – The FORTRAN function to be called by the C wrapper.

On a UNIX system, the C and PV-WAVE code described in this example is available online in the directory:

```
$WAVE_DIR/demo/interapp/linknload
```

On a VMS system, the C and PV-WAVE code described in this example is available online in the directory:

```
WAVE_DIR: [ DEMO . INTERAPP . LINKNLOAD ]
```


Compiling and Linking the Programs

This section demonstrates how to compile the C wrapper function `wave_to_fort_w` and the FORTRAN function `fort_from_wave.f` to produce a shareable object on various platforms. Please refer to the appropriate documentation that came with your operating system for the meaning of these commands.

Note

See the description of LINKNLOAD in the *PV-WAVE Reference* for a list of notes and cautions that pertain to using LINKNLOAD.

```
aix% xlf -c fort_from_wave.f
aix% cc -c wave_to_fort_w.c \
-I$WAVE_DIR/util/variables
aix% xlf -o wave_to_fort_w.so \
wave_to_fort_w.o \
fort_from_wave.o \
-e wave_to_fort_w
```

```
hpux% f77 -c +z fort_from_wave.f
hpux% cc -c +z wave_to_fort_w.c \
-I$WAVE_DIR/util/variables
hpux% ld -o wave_to_fort_w.s \
-b wave_to_fort_w.o \
fort_from_wave.o \
-lcl -lisamstub -lnlsstubs -lc
```

```
sunos% setenv LD_LIBRARY_PATH /usr/lang/SC0.0
NOTE: The correct setting of LD_LIBRARY_PATH may vary
depending on how your operating system was installed. The
recommended directory is /usr/lang. If necessary, see your
system administrator to determine the correct directory.
```

```
sunos% cc -c -pic wave_to_fort_w.c \
-I$WAVE_DIR/util/variables
sunos% f77 -c -pic fort_from_wave.f
sunos% ld -o wave_to_fort_w.so \
wave_to_fort_w.o \
fort_from_wave.o \
-Bstatic -lF77 -lm
```

```

solaris% setenv LD_LIBRARY_PATH /opt/SUNWspro/lib
NOTE: The correct setting of LD_LIBRARY_PATH may vary
depending on how your operating system was installed. The
recommended directory is /opt/SUNWspro/lib. If necessary, see
your system administrator to determine the correct directory.

solaris% cc -c -pic wave_to_fort_w.c \
-I$WAVE_DIR/util/variables
solaris% f77 -c -pic fort_from_wave.f
solaris% ld -G -o wave_to_fort.so \
wave_to_fort_w.o \
fort_from_wave.o \
-lF77 -lm -lc

vms$ fort fort_from_wave
vms$ cc wave_to_fort_w
vms$ define wave_image wave_dir:[bin]wave_x exe
vms$ link c_from_wave, -
sys$input/opt/share
wave_image/shareable
universal=c_from_wave
vms$ define c_from_wave_so -
disk$:[homedir]c_from_wave

```

Running the Example Program

Enter PV-WAVE and type the following at the WAVE> prompt:

```
.RUN wave_to_fort_main
```



Under Solaris, you need to do set the LD_LIBRARY_PATH environment variable to the directory containing the file wave_to_fort.so. You must set this variable even if that file is in the current working directory.

Calling PV-WAVE in a Statically Linked Program

Under VMS or UNIX, an application written in C or FORTRAN can be linked directly (statically) with the PV-WAVE object libraries. The user application then passes PV-WAVE commands to the entry points `cwavec` (C application) or `cwavefor` (FORTRAN application) in the PV-WAVE shareable image.

cwavec: Calling PV-WAVE from a C Program

The routine `cwavec`, discussed in detail in this section, is the C application entry point to a PV-WAVE shareable image.

Usage

`istat = cwavec(action, numcmds, cmds)`

Parameters

action — Specifies how you wish PV-WAVE to execute. It can have one of the following values:

- *action*=1 — Run normally. You are interactively prompted for input and execution continues until you enter the end-of-file character or issue the EXIT command. At this point, `cwavec` returns with a value of 1. Once `cwavec` has been called in this mode, it should not be called again.
- *action*=2 — Execute the commands supplied by *cmds* array and return. The return value is the value of the !Error system variable. The `cwavec` routine can be called repeatedly in this mode.
- *action*=3 — It is necessary to wrap up the session by calling `cwavec` one last time with *action*=3. This performs any housekeeping required by PV-WAVE such as closing any open files. The return value for this mode is 1. Once `cwavec` has been called in this mode, it should not be called again.

numcmds — The number of elements supplied in *cmds*. This argument is ignored if *action*=3 or if *action*=1.

cmds — An array of pointers to strings. If *action*=2, *cmds* provides an array of PV-WAVE commands to execute. This argument is ignored if *action*=3 or if *action*=1.

Returned Value

istat — The returned value depends on the *action* selected, as explained previously.

Discussion

You can choose to communicate with PV-WAVE in either an interactive mode or by sending an array of commands. Both of these methods automatically initialize PV-WAVE.

The first parameter is the *action* parameter. The action parameter may have one of the following the values:

Value	Meaning
1	Run PV-WAVE interactively.
2	Execute a sequence of PV-WAVE commands and return to the C program.
3	Exit PV-WAVE and return to the C program.

The third parameter is the name of an array of pointers to strings (i.e., *char***) containing the PV-WAVE commands to be executed. The second parameter specifies the number of elements supplied in the third parameter. The second and third parameters are ignored if the value of the *action* parameter is 1 or 3.

The status value returned by *cwavec* depends on the value of the *action* parameter and in some cases on the value of the action performed. If the value of the *action* parameter is 1 or 3, *cwavec* will return 1 as the status. If the value of the *action* parameter is 2, *cwavec* will return the value of the PV-WAVE system variable *!Err* as the status.

Accessing the Data in PV-WAVE Variables

The `wavevars` function can be used to access the results generated by PV-WAVE in a user-written application called with `cwavec`. `wavevars` is a C function that can be invoked from code linked to PV-WAVE to obtain data from PV-WAVE's data variable space.

For detailed information on `wavevars`, see *Accessing Data in PV-WAVE Variables* on page 350.

The `wavevars` function can only be used from a C function. It cannot be used from a FORTRAN function; however, a FORTRAN function could be used in conjunction with a C wrapper to accomplish the same task. See *Example 4* on page 330 for information on this technique.

Ending the Session with PV-WAVE

If you are in interactive mode (`action=1`), enter EXIT at the WAVE> prompt to return to your C application. There is no need to call `cwavec` with `action=3` to end the session. However, if the application has accessed PV-WAVE in non-interactive mode (`action=2`), the session must be terminated by a final call to `cwavec` with `action=3`.

Running PV-WAVE from a C Program

To run PV-WAVE from a C program you must first link the C program with PV-WAVE. The C program may then invoke PV-WAVE via the entry point `cwavec` in the PV-WAVE shareable object. The C program must pass three parameters to the `cwavec` entry point. For details on linking the application to PV-WAVE, see *How to Link Applications to PV-WAVE* on page 346.

Example 1

In non-interactive mode, valid PV-WAVE commands are passed to `cwavec` as an array of strings. For example, to plot the vector [1, 2, 3, 4, 5] from a C application statically linked to PV-WAVE, the commands would be:

```
char *cmds[5];
.
.
.

cmds[0] = "a = indgen(5) + 1";
cmds[1] = "plot, a";
action=2;
status = cwavec(action, 2, cmds);
```

Example 2

This example shows how to pass a five-element array to PV-WAVE via `cwavec`, have PV-WAVE perform some calculations, and produce a plot.

Note



You can find the following listed file in:

```
$WAVE_DIR/demo/interapp/cwave/example.c
```

```
#include <stdio.h>
main()
{
  /* Variables for array calculations
  */
  int action, numcmds, istat, cwavec();
  char *cmds[5];
  /*
  * Access PV-WAVE in non-interactive mode
  */
  action=2;
  numcmds=5;
```

```

/*
 * Send the array of commands to PV-WAVE
 * Define the array A
 * Perform matrix multiplication
 * Print contents of B
 * Display B as a surface
 * Issue a wait command so you can view result
 * Call cwavec
 */
cmds[0] = "A = INDGEN(5) * 4";
cmds[1] = "B = A # A";
cmds[2] = "PRINT, B";
cmds[3] = "SURFACE, B";
cmds[4] = "WAIT, 3.0";
istat = cwavec(action, numcmds, cmds);
/*
 * Since we are done sending commands to
 * PV-WAVE, make a final call to cwavec
 * with action=3 to wrap up the session
 */
action=3;
istat = cwavec(action, 0, cmds);
}

```

Compiling and Linking the Example Program

You can use the following commands to compile the example program and link it to PV-WAVE on a UNIX system:

```

setenv arch '$WAVE_DIR/bin/arch'

cc -c example.c

make -f $WAVE_DIR/src/pub/quick.mk \
  link MAIN=example OBJ=example.o \
  TARGARCH=$arch

```

For more information on compiling programs and linking them using the makefile `quick.mk`, see *How to Link Applications to PV-WAVE* on page 346.

Example 3

In this example the C program passes commands to PV-WAVE to be executed and then accesses the results directly from PV-WAVE's variable data space via the `wavevars` routine.

This example uses one program:

- `wave_from_c.c` – The C function that calls PV-WAVE and accesses PV-WAVE variables directly.

On a UNIX system, this program is available online in the directory:

```
$WAVE_DIR/demo/interapp/cwave
```

On a VMS system, this program is available online in the directory:

```
WAVE_DIR: [ DEMO. INTERAPP. CWAVE ]
```

The C program must be compiled and linked with PV-WAVE to produce a single executable program, as explained in the next section. It is because your program is linked with PV-WAVE as a single executable that your program can “share” PV-WAVE variables.

Compiling and Linking the Example Program

You can use the following commands to compile the example program and link it to PV-WAVE on a UNIX system:

```
setenv arch '$WAVE_DIR/bin/arch'

cc -c wave_from_c.c

make -f $WAVE_DIR/src/pub/quick.mk \
    link MAIN=wave_from_c OBJ=wave_from_c.o \
    TARGARCH=$arch
```


For more information on compiling programs and linking them using the makefile `quick.mk`, see *How to Link Applications to PV-WAVE* on page 346.

Note 

The link operation for this example creates a large executable, because it links in all of PV-WAVE.

Running the Program

After the program is compiled and linked, it can be run by entering the name of the resulting executable file. For example, if the executable is called `wave_from_c`, enter:

```
wave_from_c
```

The output from this example is shown in Figure 13-1.

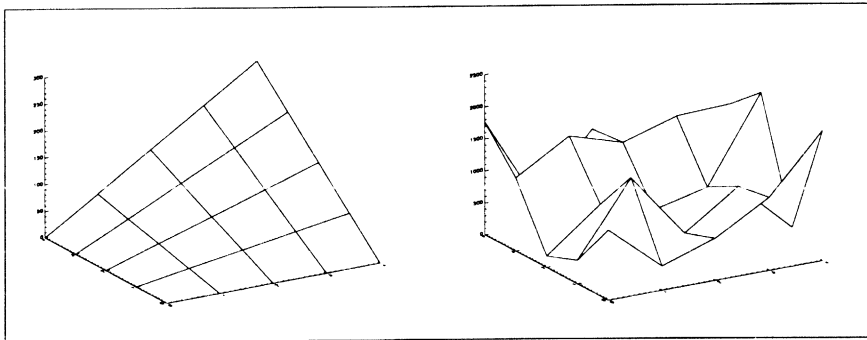


Figure 13-1 The first graphic produced by this example is shown on the left. The second graphic produced is on the right.

cwavefor: Calling PV-WAVE from a FORTRAN Program

The `cwavefor` routine is the FORTRAN application entry point to a PV-WAVE shareable image.

Usage

$istat = cwavefor(action, numcmds, cmds, cmdlen)$

Parameters

action — Specifies how you wish PV-WAVE to execute. It can have one of the following values:

- ***action=1*** — Run normally. You are interactively prompted for input and execution continues until you enter the end-of-file character or issue the EXIT command. At this point, `cwavefor` returns with a value of 1. Once `cwavefor` has been called in this mode, it should not be called again.
- ***action=2*** — Execute the commands supplied by `cmds` array and return. The return value is the value of the !Error system variable. The `cwavefor` routine can be called repeatedly in this mode.
- ***action=3*** — It is necessary to wrap up the session by calling `cwavefor` one last time with `action=3`. This performs any housekeeping required by PV-WAVE such as closing any open files. The return value for this mode is 1. Once `cwavefor` has been called in this mode, it should not be called again.

numcmds — The number of elements supplied in `cmds`. This argument is ignored if `action=3` or if `action=1`.

cmds — An array of strings. If `action=2`, `cmds` provides an array of PV-WAVE commands to execute. This argument is ignored if `action=3` or if `action=1`.

cmdlen — The declared length of each string element in the two-dimensional array.

Returned Value

istat — The returned value depends on the *action* selected, as explained previously.

Discussion

You can choose to communicate with PV-WAVE in either an interactive mode or by sending an array of commands. These methods automatically initialize PV-WAVE.

The first parameter is the *action* parameter. The action parameter may have one of the following the values:

Value	Meaning
1	Run PV-WAVE interactively.
2	Execute a sequence of PV-WAVE commands and return to the FORTRAN program.
3	Exit PV-WAVE and return to the FORTRAN program.

The third parameter is the name of an array of strings containing the PV-WAVE commands to be executed. The second parameter specifies the number of elements supplied in the third parameter. The second and third parameters are ignored if the value of the *action* parameter is 1 or 3.

The status value returned by `cwavefor` depends on the value of the *action* parameter and in some cases on the value of the action performed. If the value of the *action* parameter is 1 or 3, `cwavefor` will return 1 as the status. If the value of the *action* parameter is 2, `cwavefor` will return the value of the PV-WAVE system variable `!Error` as the status.

Ending the Session with PV-WAVE

If you are in interactive mode (`action=1`), enter EXIT at the WAVE> prompt to return to your FORTRAN application. There is no need to call `cwavefor` with `action=3` to end the session.

However, if the application has accessed PV-WAVE in non-interactive mode (`action=2`), the session must be terminated by a final call to `cwavefor` with `action=3`.

Running PV-WAVE from a FORTRAN Program

To run PV-WAVE from a FORTRAN program you must first link the FORTRAN program with PV-WAVE. The FORTRAN program can then invoke PV-WAVE via the entry point `cwavefor` in the PV-WAVE shareable object. The FORTRAN program must pass four parameters to the `cwavefor` entry point. For details on linking the application to PV-WAVE, see *How to Link Applications to PV-WAVE* on page 346.

Example 1

In non-interactive mode, valid PV-WAVE commands are passed to `cwavefor` as an array of strings. For example, to plot the vector [1, 2, 3, 4, 5] from a FORTRAN application statically linked to PV-WAVE, the commands would be:

```
character *50 cmds(5)
.
.
.

cmds(1) = 'a = INDGEN(5) + 1'
cmds(2) = 'plot, a'
action=2
call cwavefor(action, 2, cmds, 50)
```

Example 2

This example shows how to pass a five-element array to PV-WAVE via `cwavefor`, have PV-WAVE perform some calculations, and produce a plot.

Note



You can find the following listed file in:

```
$WAVE_DIR/demo/interapp/cwave/examplefor.f
```

```

PROGRAM EXAMPLE_175
C
C Variables for array calculations
C
    integer*4  action, numcmds, istat, cwavefor
    character  *30 cmds(5)
C
C In order to initialize stdin and stdout
C correctly so that output goes to your
C terminal, make the following call:
C This call for VMS only.
    call vaxc$ctrl_init
C
C Access PV=WAVE in non-interactive mode
C
    action=2
    numcmds = 5
C
C Send the array of commands to PV-WAVE
C Define the array A
C Perform matrix multiplication
C Print contents of B
C Display B as a surface
C Issue a wait command so user can view result
C Call cwavefor
C
    cmds(1) = 'A = INDGEN(5) *4'
    cmds(2) = 'B = A # A'
    cmds(3) = 'PRINT, B'
    cmds(4) = 'SURFACE, B'
    cmds(5) = 'WAIT, 3.0'
    istat = cwavefor(action, numcmds, cmds, 30)

```

```

C
C Since we are done sending commands to
C PV-WAVE, make a final call to cwavec
C with action=3 to wrap up the session.
C
    action=3
    istat = cwavefor(action, 0, cmds, 30)
    stop
    end

```

Compiling and Linking the Example Program

You can use the following commands to compile the example program and link it to PV-WAVE on a UNIX system:

```

setenv arch '$WAVE_DIR/bin/arch'

f77 -c examplefor.f

make -f $WAVE_DIR/src/pub/quick.mk \
    flink MAIN=examplefor OBJ=examplefor.o \
    TARGARCH=$arch

```

For more information on compiling programs and linking them using the makefile `quick.mk`, see *How to Link Applications to PV-WAVE* on page 346.

Example 3

In this example, the FORTRAN program passes commands to PV-WAVE to be executed and then accesses the results directly (via a C wrapper) from PV-WAVE's variable data space using the C function `wavevars`.

This example uses two functions:

- `wave_from_fort.f` – The FORTRAN function that calls PV-WAVE and accesses PV-WAVE variables directly.
- `wavevars_fl.c` – A C function (wrapper) that allows the FORTRAN program to retrieve and/or modify the values of floating-point arrays in PV-WAVE's variable data space. This

is accomplished via the `wavevars` function, which interacts directly with PV-WAVE's variable data space. (Direct interaction between a FORTRAN program and `wavevars` does not work because FORTRAN lacks the C language's ability to access a common data area by address.)

On a UNIX system, the C and FORTRAN code described in this example is available online in the directory:

```
$WAVE_DIR/demo/interapp/cwave
```

On a VMS system, the C and FORTRAN code described in this example is available online in the directory:

```
WAVE_DIR: [DEMO.INTERAPP.CWAVE]
```

The FORTRAN program must be compiled and linked with PV-WAVE and the C wrapper routine to produce a single executable program, as explained in the next section. It is because your program is linked with PV-WAVE as a single executable that your program can "share" PV-WAVE variables.

Compiling and Linking the Example Program

You can use the following commands to compile the example program and link it to PV-WAVE on a UNIX system:

```
setenv arch '$WAVE_DIR/bin/arch'
f77 -c wave_from_fort.f
cc -c wavevars_fl.c
make -f $WAVE_DIR/src/pub/quick.mk \
    flink MAIN=wave_from_fort \
    OBJ="wave_from_fort.o wavevars_fl.o" \
    TARGARCH=$arch
```

For more information on compiling programs and linking them using the makefile `quick.mk`, see *How to Link Applications to PV-WAVE* on page 346.

Note

The link operation for this example creates a large executable, because it links in all of PV-WAVE.

Running the Program

After the program is compiled and linked, it can be run by entering the name of the resulting executable file. For example if the executable is called `wave_from_fort`, enter:

```
wave_from_fort
```

The output from this example is shown in Figure 13-1 on page 339.

How to Link Applications to PV-WAVE

This section explains how to link C and FORTRAN applications to PV-WAVE on a UNIX workstation and how to link C applications to PV-WAVE under VMS.

Using the quick.mk Makefile on a UNIX System

The makefile `quick.mk` is provided to assist you in linking compiled C and FORTRAN programs to PV-WAVE.

On a UNIX system, this makefile is located in:

```
$WAVE_DIR/src/pub/quick.mk
```

For example, to link a compiled program called `mywavec` to PV-WAVE, you can use the following commands:

```
host> setenv arch '$WAVE_DIR/bin/arch'

host> make -f $WAVE_DIR/src/pub/quick.mk \
    link MAIN=mycwavec OBJ=mycwavec.o \
    TARGARCH=$arch
```

Note the backquotes must be entered exactly as shown.

`quick.mk` can be modified as desired to build a single PV-WAVE application or multiple applications. Refer to `$WAVE_DIR/src/pub/README` for more details.

Linking a C Application to PV-WAVE: UNIX

The following statements illustrate how to link an application written in C to PV-WAVE on a Sun-4 (UNIX) system. Note that the quotes and backquotes must be entered exactly as shown.

Note

The following commands are shown only as an example. It is possible that these link commands differ from the commands required to link an application under the current release of PV-WAVE. It is recommended that you use the makefile `quick.mk`, described in the previous section, to link applications to PV-WAVE.

```
setenv arch '$WAVE_DIR/bin/arch'

setenv BINARCH $WAVE_DIR/bin/bin.$sarch

setenv ARCHLIB \
'-Bstatic -lX11 -ldynamic -ltermcap -lm -
ldl -ltermcap -lm'

cc -o mycwavec mycwavec.o \
$BINARCH/wave.$sarch.a \
$VNI_DIR/cmathstat/bin/bin.$sarch/cmast.$sarch.a \
$VNI_DIR/dblink/bin/bin.$sarch/dbms.$sarch.a \
$BINARCH/dc.$sarch.a \
$BINARCH/nr.$sarch.a \
$BINARCH/render.$sarch.a \
$BINARCH/rpc.$sarch.a \
$BINARCH/table.$sarch.a \
$BINARCH/wt.$sarch.a \
$BINARCH/optionstubs.$sarch.a \
$BINARCH/wave.$sarch.a \
$ARCHLIB
```

Note

Link options as represented by ARCHLIB will vary by platform and are sometimes site specific. Refer to the macro LIBARCH in the files `$WAVE_DIR/src/*.mkcfg` (`sun4.mkcfg`, `hps700.mkcfg`, etc.) for suggested values of ARCHLIB.

Linking a FORTRAN Application: UNIX

The following statements illustrate how to link an application written in FORTRAN to PV-WAVE on a Sun-4 (UNIX) system. The backquotes must be entered exactly as shown.

Note

The following commands are shown only as an example. It is possible that these link commands differ from the commands required to link an application under the current release of PV-WAVE. It is recommended that you use the makefile `quick.mk`, described in the previous section, to link applications to PV-WAVE.

```
setenv arch '$WAVE_DIR/bin/arch'

setenv BINARCH $WAVE_DIR/bin/bin.$sarch

setenv ARCHLIB \
'-Bstatic -lX11 -Bdynamic -ltermcap -lm -
ldl -ltermcap -lm'

f77 -Bstatic -o mycwavefor mycwavefor.o \
$BINARCH/wave.$sarch.a \
$VNI_DIR/cmathstat/bin/bin.$sarch/cmast.$sarch.a \
$VNI_DIR/dblink/bin/bin.$sarch/dbms.$sarch.a \
$BINARCH/dc.$sarch.a \
$BINARCH/nr.$sarch.a \
$BINARCH/render.$sarch.a \
$BINARCH/rpc.$sarch.a \
$BINARCH/table.$sarch.a \
$BINARCH/wt.$sarch.a \
$BINARCH/optionstubs.$sarch.a \
$BINARCH/wave.$sarch.a \
$ARCHLIB
```

Linking a C Application: VMS

The following DCL procedure illustrates how to link an application written in C to PV-WAVE on a VMS system:

```
$! CLINK.COM --- link a C program with
$! PV-WAVE using the X driver
$!
$ cc my_c_app.c
$ define wave_image wave_dir:[bin]wave_x.exe
$ link my_c_app, sys$input/option/share
  wave_image/shareable
$!
```

Refer to the file:

```
WAVE_DIR:[SRC.PUB]README.VVMS
```

for more information on linking programs with PV-WAVE on a VMS system.

Accessing Data in PV-WAVE Variables

You can access PV-WAVE variables from a C program by calling the function `wavevars`. Once commands have been sent to PV-WAVE from an external application, you can use the `wavevars` function to access the results in the external application. `wavevars` is a C function that can be invoked from code linked to PV-WAVE either:

- statically via `cwavec`, or
- dynamically via `LINKNLOAD`.

Note

Direct interaction between a FORTRAN program and `wavevars` is not possible because FORTRAN lacks C's ability to access a common data area by address. Thus, to access PV-WAVE variables from a FORTRAN program, a C wrapper must be written that calls `wavevars`.

`wavevars` obtains data directly from PV-WAVE's variable data space.

Usage

```
int    argc;  
char  **argv;  
WaveVariable *argp;  
result = wavevars(&argc, &argv, &argp);
```

Parameters

argc — Set to the number of variables returned.

argv — Set to be an array of strings, sorted in lexicographic order, corresponding to variable names available at the current scope level of PV-WAVE.

argp — A type `WaveVariable` array of descriptors defining the type, structure, and dimension of the variables as well as providing

a pointer to their actual data. The `WaveVariable` structure is described in the *Discussion* section that follows.

Returned Value

result — A C int value which is nonzero if the routine executed successfully, and zero if an error (such as running out of memory) occurred.

Discussion

PV-WAVE variables can be accessed directly from a C function by calling the C function `wavevars` which is linked into PV-WAVE. The C function passes three parameters to the `wavevars` entry point.

The first parameter is the address of an integer variable into which `wavevars` will return the number of currently-defined PV-WAVE variables (including system variables). The second parameter is the address of an array of pointers to strings (i.e., `char**`) into which `wavevars` will return the names of currently-defined PV-WAVE variables. The third parameter is the address of an array of pointers to the C structure `WaveVariable` into which `wavevars` will return information regarding the type, structure, dimension, and data of each PV-WAVE variable (including a pointer to the current value of the variable).

`WaveVariable` is defined as follows in `$WAVE_DIR/util/variables/wavevars.h`. This header file must be included in any C function that calls `wavevars`.

```
typedef struct WaveVariable {
    int type;
    int read_only;
    int numdims;
    int dims[8];
    int numelems;
    int *data;
    char name[MAXIDLEN + 1];
} WaveVariable;
```

Caution 

Although `wavevars` returns pointers to the data associated with PV-WAVE's variables, keep in mind that the data pointer associated with a given variable can change after execution of certain PV-WAVE system commands. It's best to call `wavevars` immediately before it is needed to obtain information from the external program.

The `wavevars` function allocates space to store the information it returns to the caller. When the caller no longer needs the information returned by `wavevars`, then the `free_wavevars()` function should be called to free the space. The arguments to `free_wavevars()` should be identical to those used in the call to `wavevars` such as:

```
result = free_wavevars( &argc, &argv, &argp );
```

and `argc` must still contain the number of variables returned by the `wavevars` call.

The `WaveVariable` structure's fields are:

int type — The `type` field indicates the type of the variable. Valid PV-WAVE variable types, together with their C equivalents, are defined in `wavevars.h` as follows:

```
TYP_BYTE char;
TYP_INT short;
TYP_LONG long;
TYP_FLOAT float;
TYP_DOUBLE double;
TYP_COMPLEX struct { float r, i; } COMPLEX;
TYP_STRING char *;
```

In PV-WAVE, a structure is a collection of data where each field (tag) has a name. The C structure `WaveVariable` describes a PV-WAVE structure with a type of `TYP_STRUCT`, where each element of the structure is contained in a list of `WaveVariable` structures pointed to by the `data` field, which is described later in this section.

The constant `TYP_ARRAY` will be bitwise or-ed into the `type` field if the variable is in fact an array.

int read_only — Many PV-WAVE variables are read-only, and thus if this field is nonzero, it is not permissible to alter the actual variable data. This is often the case with system variables.

int numdims — PV-WAVE variables may be of dimension zero (scalar) to eight. The field `numdims` indicates the dimensionality of the variable.

int dims[8] — Indicates the size of each dimension of a variable if it is of type array.

int numelems — Corresponds to the total number of data values which are addressable from the `data` pointer.

int *data — Corresponds to the address of the actual variable data. The data is always stored as a one-dimensional C array regardless of the dimensionality of the PV-WAVE variable.

char name[MAXIDLEN + 1] — Only used when the variable being described is of type structure and represents the structure or tag field name (depending on context).

To access a specific PV-WAVE variable you must search the array of variable names returned by `wavevars` to find the index associated with that variable. Then use the index to access the correct PV-WAVE variable from the `WaveVariable` array. The `type` field in `WaveVariable` is used to determine a variable's type. To access the data associated with a PV-WAVE variable it is necessary to use the data pointer and cast it to the correct type. It is then possible to read and/or modify the actual data value(s).

Example 1

The following is a simple program that retrieves a list of all PV-WAVE variables and prints out their contents. The code fragment demonstrates several important concepts.

- The data pointer must be cast to appropriate type.
- The data is always stored as a flat one-dimensional array.
- PV-WAVE structures are stored recursively.

Note

You can find the following listed file in:

```
$WAVE_DIR/demo/interapp/wavevars/example.c
```

```
#include <stdio.h>
#include "wavevars.h"

printallvars() /* display names & value of all
               WAVE variables */
{
    int nvars, i;
    char **names;
    WaveVariable *vars;

    if (wavevars(&nvars,&names,&vars)) {
        for (i=0; i<nvars; i++) {
            fprintf(stderr,"%s\n",names[i]);
            printvar(& (vars[i]) );
            fprintf(stderr,"\n");
        }
        free_wavevars(&nvars, &names, &vars);
    }
}

printvar(v) /* print a WAVE variable on stderr
            */
WaveVariable *v;
{
```



```

    if ( v->name )
        fprintf(stderr, "\ttag: '%s'\n", v->name);
    else
        fprintf(stderr, "\ttag: \n");
    if (v->read_only)
        fprintf(stderr, "\tstat: READ_ONLY \n");
    else
        fprintf(stderr, "\tstat: READ/WRITE\n");
    fprintf(stderr, "\tnelems: %d\n",
        v->numelems);
    fprintf(stderr, "\tndims: %d\n", v->numdims);
    fprintf(stderr, "\tdims: %d %d %d %d %d %d %d
        %d\n",
        v->dims[0], v->dims[1], v->dims[2],
        v->dims[3],
        v->dims[4], v->dims[5], v->dims[6],
        v->dims[7]);
    printdata(v, v->numelems);
}

printdata(v, len)
    WaveVariable *v;
    int len;
{
    int i;

    if (v->type & TYP_ARRAY)
        fprintf(stderr, "\ttype: ARRAY OF ");
    else
        fprintf(stderr, "\ttype: ");

    switch(v->type & ~TYP_ARRAY) {
        case TYP_BYTE:
            {
                char *b = ((char *)v->data);
                fprintf(stderr, "BYTE\n");
            }
    }
}

```

```

        fprintf(stderr, "\tdata: ");
        for (i=0; i<len; i++)
            fprintf(stderr, "%d ",(int)b[i]);
    }
    break;
case TYP_INT:
    {
        short *b = ((short *)v->data);
        fprintf(stderr,"INTEGER\n");
        fprintf(stderr, "\tdata: ");
        for (i=0; i<len; i++)
            fprintf(stderr, "%d ",(int)b[i]);
    }
    break;
case TYP_LONG:
    {
        long *b = ((long *)v->data);
        fprintf(stderr,"LONG\n");
        fprintf(stderr, "\tdata: ");
        for (i=0; i<len; i++)
            fprintf(stderr, "%ld ",b[i]);
    }
    break;
case TYP_FLOAT:
    {
        float *b = ((float *)v->data);
        fprintf(stderr,"FLOAT\n");
        fprintf(stderr, "\tdata: ");
        for (i=0; i<len; i++)
            fprintf(stderr, "%g ",b[i]);
    }
    break;

```

```

case TYP_DOUBLE:
{
    double *b = ((double *)v->data);
    fprintf(stderr,"DOUBLE\n");
    fprintf(stderr, "\tdata: ");
    for (i=0; i<len; i++)
        fprintf(stderr, "%g ",b[i]);
}
break;
case TYP_COMPLEX:
{
    Complex *b = ((Complex *)v->data);
    fprintf(stderr,"COMPLEX\n");
    fprintf(stderr, "\tdata: ");
    for (i=0; i<len; i++)
        fprintf(stderr, "<%g,%g> ",b[i].r,
b[i].i);
}
break;
case TYP_STRING:
{
    char **b = ((char **)v->data);
    fprintf(stderr,"STRING\n");
    fprintf(stderr, "\tdata: ");
    for (i=0; i<len; i++)
        if ( b[i] )
            fprintf(stderr, "'%s' ",b[i]);
}
break;
case TYP_STRUCT:
{
    WaveVariable *b=((WaveVariable *)
v->data);
    fprintf(stderr,"STRUCTURE\n");
    fprintf(stderr, "\tdata:");
    for (i=0; i<len; i++) {

```

```

        fprintf(stderr, "\n");
        printvar(&(b[i]));
    }
}
break;
}
}

```

Example 2

For an example that shows how to call PV-WAVE and access its data space from a C program, see *Example 3* on page 338.

Example 3

For an example that shows how to call PV-WAVE and access its data space from a FORTRAN program, see *Example 3* on page 344.

Example 4

For an example that shows how to call a C program from within PV-WAVE and have the C program access PV-WAVE's variable data space, see *Example 3* on page 328.

Example 5

For an example that shows how to call a FORTRAN program from within PV-WAVE and have the FORTRAN program access PV-WAVE's variable data space (via a C wrapper), see *Example 4* on page 330.

Communication with Remote Procedure Calls

PV-WAVE can communicate with user-written applications by establishing a client/server relationship based upon Sun Remote Procedure Call (RPC) technology. PV-WAVE is able to act as either the client or server depending upon the requirements of the particular user-written application. This means that the user-written application can also act as either the client or server.

Remote Procedure Call (RPC) Technology

Remote procedure calls are a high-level communications paradigm which allow distributed applications to be developed using procedure calls which shield the programmer from knowing the details of the underlying network mechanisms. RPC implements a client-to-server communications system designed to support inter-application communication over a network. Data for RPC calls is transmitted using the External Data Representation (XDR) Standard.

PV-WAVE uses RPC calls to transmit data in XDR format between hosts across the network or between processes on the same host. The parameters used with the functions `CALL_UNIX` or `CALL_WAVE` (described later in this section) are “packed” into a proprietary variable structure, transmitted in an XDR format and then “unpacked” and made available for PV-WAVE and/or the user’s application.

For the purpose of our discussion, the following terms and definitions are used. A *client* initiates the remote procedure calls to a server. A *server* provides network access to a collection of one or more remote applications. A server may support multiple remote applications or multiple versions of the same remote application.

A client initiates the interapplication communication by sending a procedure call message to the server process and then waits (blocks) for a procedure reply message. The procedure call message contains the remote application’s parameters, among other things. The procedure reply message contains the results from the

remote application, among other things. Once the client has received the results through the procedure reply message, the results are made available to the client and program execution continues.

On the server side, the process is dormant while it awaits the arrival of a procedure call message. When this message arrives, the server process extracts the parameters passed from the client and passes them to the remote application. The remote application performs its tasks with the supplied parameters and returns the results to the server process. The server then sends a procedure reply message to the client and returns to a dormant state awaiting the next procedure call message from a client.

Synchronization of Client and Server Processes

Because client and server processes can be started and run independently of one another, it is possible for the client and server processes to be out of synchronization. In general, especially on relatively unloaded systems, synchronization is not a problem. However, it is good practice to ensure that the server process is in a state that can handle client requests before the client process is started.

If PV-WAVE is the server process, then it should be in a state to begin receiving client requests after the following message appears (assuming that the !Quiet system variable is zero):

```
Compiled - UNIX_LISTEN
```

If an external program that you wrote is used as the server process, you might want to include code in the program that lets PV-WAVE clients know when it is appropriate to begin RPC transactions.


Another alternative is simply to wait for a period of time before starting the client process; however, the amount of time required will vary greatly depending on the load of the client system and the load of the server system, which in some cases may be the same system.

Linking a Server or a Client with PV-WAVE

The PV-WAVE archive library contains all the routines used for RPC-based interapplication communication.

Linking a Server

The following commands describe the steps required to link a user's server code, `myserve.c` and a user's application code, `myapp.c`, with the PV-WAVE archive library.

Note  Compile and link options will vary by platform, and sometimes are site specific. Refer to the appropriate makefile in `$WAVE_DIR/util/rpc` for suggested compile and link options.

```
host> cc -c myserve.c
host> cc -c myapp.c
host> set arch=`$WAVE_DIR/bin/arch`
host> set ARCH=$WAVE_DIR/bin/bin.$arch
host> cc -o myserve myserve.o myapp.o \
    $ARCH/rpc.$arch.a
```

With the following command, the server is started in the background on the desired server host machine.

```
host> myserve &
```

The user can now access the remote application by using the PV-WAVE system function `CALL_UNIX` at the `WAVE>` prompt.

Linking a Client

In order to link a user application which is a client to communicate with PV-WAVE as a server, use the following commands:

```
host> cc -c myclient.c
host> cc -o myclient myclient.o \
    $ARCH/rpc.$arch.a
```

Once PV-WAVE has been started in the server state, the user can run his or her application client program.

```
host> myclient
```

Using PV-WAVE as a Client: CALL_UNIX

CALL_UNIX is designed to allow users to access applications on the same host machine running PV-WAVE or across a network to a remote host.

The usage for CALL_UNIX is:

```
CALL_UNIX(param [, ...])
```

where *param* is a variable parameter of any PV-WAVE type.

For a complete description of CALL_UNIX, see the *PV-WAVE Reference*.

In most cases where PV-WAVE is the client, a user desires to send data from PV-WAVE to an existing application to perform specific tasks and return the result to PV-WAVE for further analysis. In any case, the user must write a server process which acts as an interface between PV-WAVE and his or her application. The following figure shows schematically how PV-WAVE is used as a client.

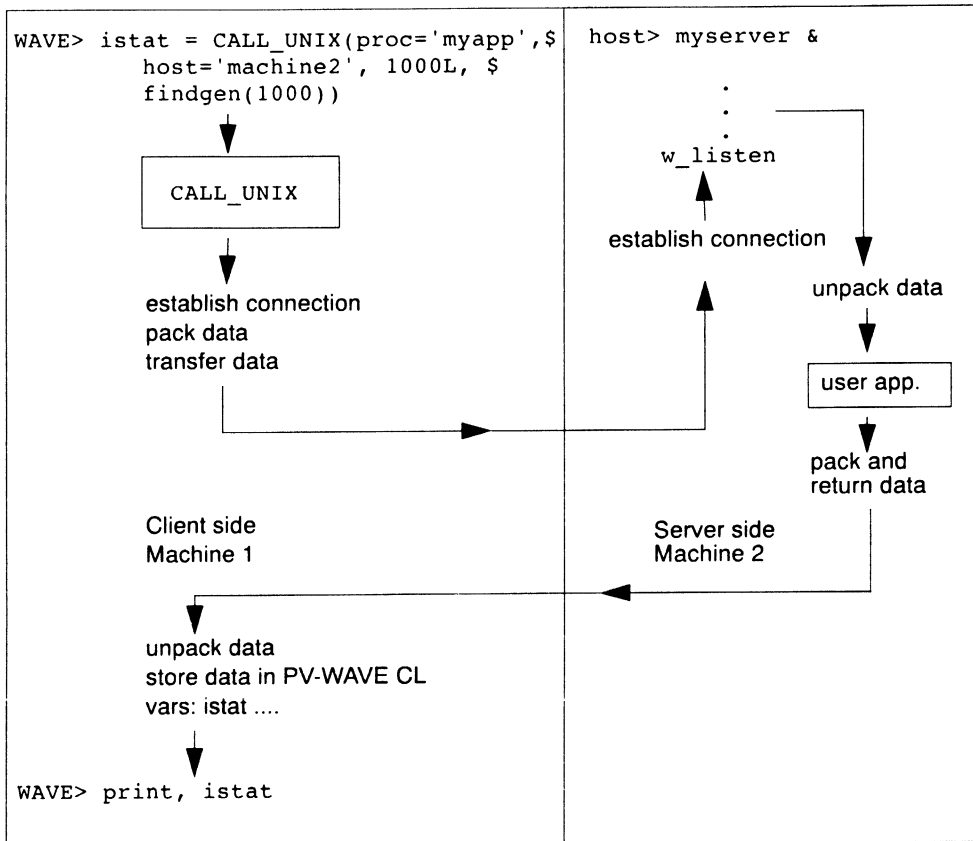


Figure 13-1 Interapplication Communication: PV-WAVE as Client

When using PV-WAVE as a client, users must first write a server to interface between PV-WAVE and their application. The server makes a call to the PV-WAVE RPC interface module `w_listen`, which is discussed in the section *Description of C Functions Used with CALL_UNIX* on page 364. When the server is started in the background, `w_listen` “sits and waits” for a procedure call message from PV-WAVE with the appropriate program number.

Once the server has been placed in this state, PV-WAVE as the client initiates the interapplication communication by calling

`CALL_UNIX` with the necessary parameters and keywords. `CALL_UNIX` opens a socket for communication, packs the parameters into XDR format and then transfers the data to the server. The data is then “unpacked” by using `w_get_par`, which is discussed in the section *Description of C Functions Used with CALL_UNIX* on page 364.

It is essential that argument parameters passed to the server be in the proper order and be of the expected type and structure. The type, number and structure of parameters can be checked within a PV-WAVE procedure before the call to the server is made. The unpacked parameters can then be used in a call to the remote application.

After execution of the remote application is complete, the results are passed back to PV-WAVE using one of the following three routines:

w_simpl_reply — Use this C routine when no passed parameters have been modified and there is only a single return value/array.

w_send_reply — Use this C routine if the input parameters have been modified and must be sent back to PV-WAVE in addition to some return value/array.

w_cmpnd_reply — Use this C routine to return multi-dimensional arrays.

Once the results have been returned to PV-WAVE, program control returns to PV-WAVE. The server can either exit or enter a new wait state.

Description of C Functions Used with CALL_UNIX

As explained previously, the `CALL_UNIX` function allows PV-WAVE to communicate with a user-written application. `CALL_UNIX` sends parameters to a server process which then calls the user-written application.

The server uses the following C routines, which are discussed in detail later in this section:

- **w_listen** — Connect with the process running PV-WAVE.
- **w_get_par** — Get the parameters.
- **w_send_reply**, **w_smpl_reply**, or **w_cmpnd_reply** — Send values and parameters back to PV-WAVE.

If an error occurs in a call to CALL_UNIX, it returns a -1. The function ON_IOERROR can also be used to catch CALL_UNIX errors.

w_listen

When placed in an external C source file, allows the routine to “sit and wait” until it is called.

Usage

```
int w_listen (int program_number, char **user, char
**procedure)
```

Discussion

When w_listen exits, the values of the parameters user, and procedure have been set. Use these strings in the external routine to control access to, and program flow in, the external routine.

Returned Value

w_listen returns a status of -1 if an error occurs, 1 otherwise.

Input Parameters

program_number — A unique identifier, set within the external routine that enables w_listen to determine which calls from CALL_UNIX are intended for this particular server. If the value of *program_number* is the same as the value of *program_number* in

another server, the communication with the first server will be lost. If communication with a server is lost, use the `kill -9` command.

user — A pointer to a string pointer. The parameter *user* is set by the keyword *User* in the `CALL_UNIX` call and set by `w_listen`. The intended use of *user* is to control access to the external routine. The memory that holds the string has already been allocated.

procedure — A pointer to a string pointer. The parameter *procedure* is set by the keyword *Procedure* in the `CALL_UNIX` call. The intended use of *procedure* is to control program flow in the external routine. The memory that holds the string has already been allocated.

w_get_par

Returns a specified parameter passed by `CALL_UNIX`.

Usage

```
char *w_get_par (int param_number, type);
```

Parameters

param_number — The index of the list of parameters. *param_number* is zero-based, so a value of zero will cause `w_get_par` to return a pointer to the first parameter, a value of one will cause `w_get_par` to return a pointer to the second parameter and so on.

type — The type of variable to be retrieved.

Discussion

Because of the way that functions are declared in C, `w_get_par` returns a pointer to a char, which is then cast to a pointer to the type of parameter desired. The space pointed to by the return value is allocated and freed by the RPC software. When the external routine replies to the client, the allocated space is freed. Thus, if there is a need to save a value returned by `w_get_par`, the value

pointed to by the return value of `w_get_par` must be copied into a variable. `w_get_par` returns NULL if there is no such parameter or the specified parameter is not of the specified type.

The types are defined in the file `$WAVE_DIR/util/rpc/wave_rpc_extern.h`. For example, `TYP_LONG` will cause `w_get_par` to return a pointer to a long. To specify an array, perform a bitwise OR of the type of the array with the constant `TYP_ARRAY`. For example, if the third parameter passed from `CALL_UNIX` to the server is an array of long, you would use the following code to get the array:

```
long *example_long_array;
example_long_array = (long *) w_get_par (2,
    TYP_LONG | TYP_ARRAY);
```

After the external routine has completed its processing, it will need to return information to the client. Three routines are supplied that are used to return the information. All three of these routines return a status value of `-1` to indicate an error and `0` to indicate success.

Note 

When one of these three routines is called, all the memory that the parameters occupied in the RPC software is freed. Thus, in the preceding example, the array that `example_long_array` points to will be freed.

w_smpl_reply and w_send_reply

- `w_smpl_reply` — Sends a single value array as the return value of `CALL_UNIX`.
- `w_send_reply` — Is used when it is necessary to modify and return parameters that have been passed in from `CALL_UNIX`.

Usage

```
int w_simpl_reply (type, long number_of_items, char  
*items);
```

```
int w_send_reply (type, long number_of_items, char
*items);
```

Parameters

type — The type of variable to be returned. The types are defined in the file:

```
$WAVE_DIR/util/rpc/wave_rpc_extern.h
```

To specify an array, perform a bitwise OR of the type of the array with the constant TYP_ARRAY.

number_of_items — The number of items to be returned by the server. If an array is being returned, *number_of_items* is the length of the array, otherwise, *number_of_items* is set to one.

items — A pointer to the item(s) being returned.

w_smpl_reply and *w_send_reply* expect pointers to char, so they may have to be type cast.

Discussion

w_smpl_reply can be thought of as giving the CALL_UNIX function a call by value parameter-passing mechanism. Only scalar values or single-dimension arrays (vectors) or scalars can be returned by *w_smpl_reply*.

With *w_send_reply*, data passed to a server cannot “grow”. If the server was passed FLTARR (100), the remote application cannot “grow” this to FLTARR (1000) and return it. Thus, *w_smpl_reply* can be thought of as giving the CALL_UNIX function a call by reference parameter-passing mechanism. Only single-dimension arrays or scalars can be returned by *w_send_reply*.

w_cmpnd_reply

Used when a multi-dimensional array must be passed back by the server.

Usage

```
int w_cmpnd_reply (unsigned char reply_only, type,  
unsigned char number_dimensions, long  
dimensions[ ], char *items);
```

Parameters

reply_only — A flag that determines if the parameters sent by the client are modified and returned by the server. If *reply_only* has a value of one, then the parameters sent from the client are not returned by the server (call by value). Any other value of *reply_only* will cause the (possibly modified) parameters that were sent by the client to be sent by the server back to the client (call by reference).

type — The type of variable to be returned. The types are defined in the file `$WAVE_DIR/util/rpc/wave_rpc_extern.h`. To specify an array, perform a bitwise OR of the type of the array with the constant `TYP_ARRAY`.

number_dimensions — The size of the dimensions array. The maximum value is eight.

dimensions — An array containing the sizes of the each of the array dimensions. To specify a 4 x 5 x 6 array, set *number_dimensions* to 3, and *dimensions* to [4,5,6].

items — A pointer to the item(s) being returned.

`w_cmpnd_reply` expect pointers to char, so they may have to be type cast.

Example Server

An example server, `test_server.c`, is provided online in the directory `WAVE_DIR/util/rpc`. The file `test_server.c` contains three examples:

- **Example_1** — Accepts an array of long values. The array is passed to a function where each value is multiplied by two. The result is passed back and placed into the PV-WAVE variable `newarray`.

```
newarray = CALL_UNIX(proc='example_1', $
    1000L, lindgen(1000))
```

```
print, newarray
```

- **Example_2** – Accepts a string and returns a string. An array of strings is passed to the server and then printed. A new string is returned and placed in the PV-WAVE variable message.

```
strings=['one', 'two', 'three', 'four', $
    'five']
```

```
message=CALL_UNIX(proc='example_2', 5, $
    strings)
```

```
print, message
```

- **Example_3** – Accepts a long array. This array is then filled by pseudo random numbers and returned. This example shows how to modify a passed parameter rather than always returning a result to a new variable. All parameters passed to the server in this example must be variables, otherwise, when the server attempts to modify a parameter, PV-WAVE complains about not being able to store a value into a constant. old_array will contain an array of 1000 pseudo random numbers.

```
old_array=lonarr(1000)
```

```
old_arr_len=1000L
```

```
new_len=CALL_UNIX(old_arr_len, old_array, $
    proc='example_3')
```

```
print, new_len, old_array
```

Example Using CALL_UNIX

See *Example Procedure Using CALL_UNIX with PV-WAVE as Client* on page 377 for example code showing a PV-WAVE client procedure that uses CALL_UNIX.

And see *Example External C Routine as a Server* on page 380 for an example of the corresponding server.

Using PV-WAVE as a Server: CALL_WAVE

CALL_WAVE is designed to allow users to access PV-WAVE from existing or planned applications on the same host machine running PV-WAVE or across a network to a remote host. In most cases where PV-WAVE is the server, the user desires to send data to PV-WAVE either to access PV-WAVE's analytical routines or for plotting. In any case, the user must write a client process which acts as an interface between PV-WAVE and his or her application. See Figure 13-2 for an example of PV-WAVE as a server.

To run PV-WAVE as a server, start-up PV-WAVE and run a procedure which calls UNIX_LISTEN.



The procedure UNIX_LISTEN and several other procedures discussed in this section are located in the WAVE_DIR/util/rpc directory.

In addition, the file \$WAVE_DIR/util/rpc/RPC_README contains information on using RPCs and how PV-WAVE can be run as a server in the background, in another window, or on another host machine.

PV-WAVE is now in a "sit and wait" state. The user client application then initiates the interapplication communication by calling CALL_WAVE with the necessary parameters and keywords. CALL_WAVE opens a socket for communication, packs the arguments into XDR format and then transfers the data to PV-WAVE.

UNIX_LISTEN receives this procedure call message, unpacks the data from the client and then makes it available to PV-WAVE for some type of processing. UNIX_LISTEN can also return two strings to the PV-WAVE environment. These strings can contain specific PV-WAVE commands that specify the desired processing that should take place. Once this processing has been completed, PV-WAVE returns the results to the client application through UNIX_REPLY. When the results have been returned, program control returns to the client application while PV-WAVE can either exit or enter a new wait state.

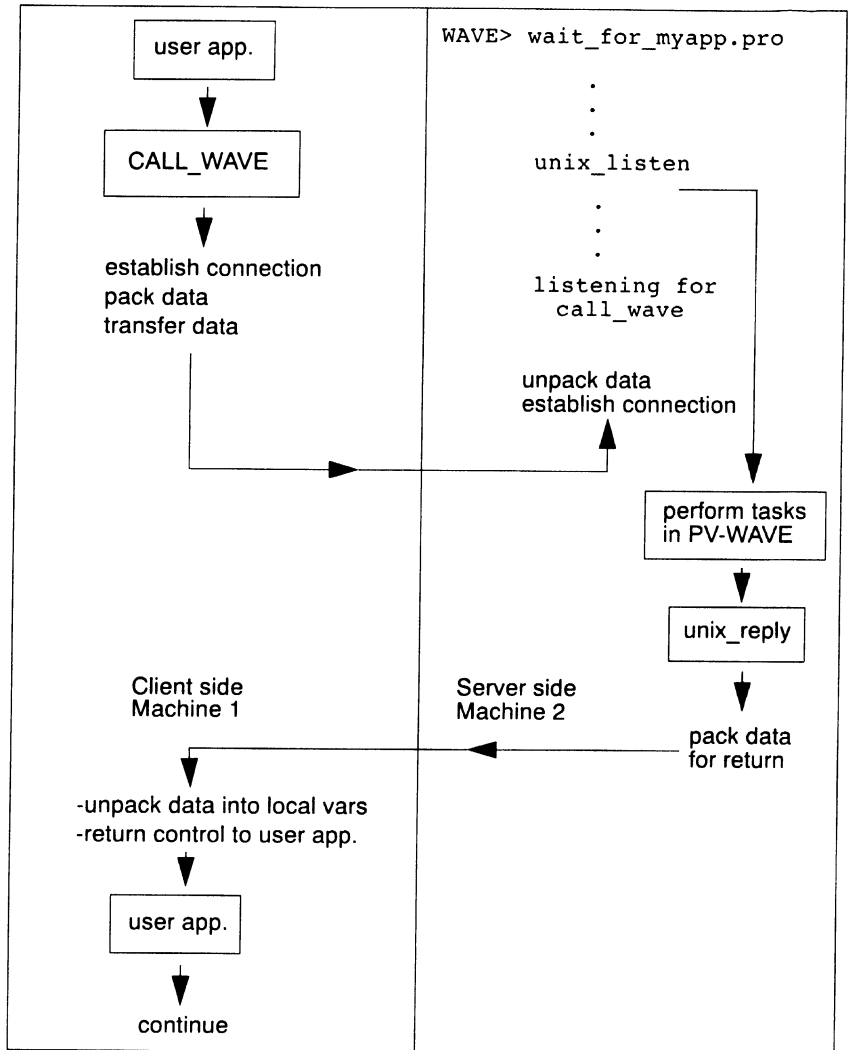


Figure 13-2 : Interapplication Communication: PV-WAVE as Server

Description of C Functions Used with PV-WAVE as Server

CALL_WAVE

Calls a server and returns a pointer to a UT_VAR after the server has completed its task.

Usage

```
UT_VAR *CALL_WAVE(int argc, UT_VAR *argv[ ],
char hostname [ ], int *unit, int close_unit, int program,
char *procedure, char *user, int time_out)
```

Parameters

argc – The number of values in *argv*. *argc* is also the number of parameters that will be passed to PV-WAVE.

argv – An array of pointers to UT_VAR. The UT_VARS contain the parameters that will be passed to PV-WAVE. It is up to the application to load these structures properly.

hostname – The name of the machine that the server (PV-WAVE) is running on. *hostname* is necessary because there could be several servers, each with the same program number, but running on different machines.

unit – A number that maps to an RPC socket. The purpose of *unit* is to allow the reuse of an open RPC socket, and thus avoid the overhead of re-opening the socket each time CALL_WAVE is called. However, based on experience, this overhead is not normally noticeable. If CALL_WAVE is called with NULL in the place of *unit*, no value is returned and a new socket is opened and closed after CALL_WAVE exits. If *unit is zero (i.e., unit == 0 and CALL_WAVE receives &unit), then a socket is opened and the socket number is returned in *unit. If *unit is greater than zero, socket number unit is used.

close_unit – A flag telling CALL_WAVE to close unit after CALL_WAVE exits. If *close_unit* is non-zero, the unit is closed.

program – A number identifying which server the application is calling. *program* is supplied in the C routine on the client. The value of *program* is the same value of the keyword *Program* in the UNIX_LISTEN call. If the UNIX_LISTEN call did not have the *Program* keyword in it, use zero for the value of *program*.

procedure – A string that is sent to the server. In the server, the UNIX_LISTEN keyword *Procedure* is used to retrieve the string. *user* is intended to be used for controlling program flow within the server.

user – A string that is sent to the server. In the server, the UNIX_LISTEN keyword *User* is used to retrieve the string. *procedure* is intended to be used for controlling access to the routines within the server.

time_out – The time, in seconds, that CALL_WAVE will wait for PV-WAVE to complete its task. There is no value for infinite time.

Discussion

The function CALL_WAVE calls a server and returns a pointer to a UT_VAR after the server has completed its task. UT_VAR is a structure for holding any valid type of PV-WAVE variable. This UT_VAR pointer points to the value sent back in the reply parameter of UNIX_REPLY. Upon receiving a pointer to a UT_VAR, the application must determine the type of variable it references. The definitions of UT_VAR and other types can be found in the header file:

```
$WAVE_DIR/util/rpc/wave_rpc_extern.h
```

For convenience, here is the definition of `UT_VAR`:

```
typedef struct
{
    unsigned chartype,
    unsigned charelement_size;
    unsigned charn_dim,;
    longdim[MAX_ARRAY_DIM];
    UT_TYPESvalue;
} UT_VAR;
```

PV-WAVE Functions Used with PV-WAVE as Server

The `UNIX_LISTEN` and `UNIX_REPLY` functions are used when `PV-WAVE` is acting as a server.

UNIX_LISTEN

The `UNIX_LISTEN` function allows `PV-WAVE` to be called by external routines written in C. `UNIX_LISTEN` waits until an external routine calls the function `CALL_WAVE`, and then returns the number of parameters with which it was called.

The parameters are accessed in the common block `UT_COMMON`, which is included in the server routine with the command `@UT_COMMON`. The first parameter is `ut_param0`, the second is `ut_param1`, and the thirtieth parameter is `ut_param29`. As well as being returned, the number of parameters is also contained in the variable `ut_num_params` in the `UT_COMMON` common block.

For more detailed information on `UNIX_LISTEN` and its keywords, see the *PV-WAVE Reference*.

UNIX_REPLY

When `PV-WAVE` has completed its processing and is ready to return information to the client, the function `UNIX_REPLY` is called. `UNIX_REPLY` has one parameter, `reply`, which is a variable that is sent back to the client. The `CALL_WAVE` function returns a pointer to a `UT_VAR` variable with the value of `reply` to

the client. A keyword, *Return_Params*, is used if the server must modify the incoming parameters and return them. The number of parameters that is sent back is the same as the number that came in. This number is tracked internally by PV-WAVE.

For more detailed information on UNIX_REPLY and its keywords, see the *PV-WAVE Reference*.

Examples Using PV-WAVE as a Server

See *Example Procedure Using UNIX_LISTEN and UNIX_REPLY with PV-WAVE as a Server* on page 392 for a code example that illustrates the interaction between a client application using CALL_WAVE and PV-WAVE as a server using the PV-WAVE functions UNIX_LISTEN and UNIX_REPLY.

An example client, `test_client.c`, is provided on the distribution tape in `WAVE_DIR/util/rpc`.


See also the code listings *Example C Routine with CALL_WAVE as a Client* on page 382 and *Example C Function to Load a UT_VAR* on page 387.

Remote Procedure Call Examples

This section contains the following RPC examples. Each example was referred to previously in this chapter:

- *Example Procedure Using CALL_UNIX with PV-WAVE as Client* on page 377.
- *Example External C Routine as a Server* on page 380.
- *Example C Routine with CALL_WAVE as a Client* on page 382.
- *Example C Function to Load a UT_VAR* on page 387.
- *Example Procedure Using UNIX_LISTEN and UNIX_REPLY with PV-WAVE as a Server* on page 392.

Example Procedure Using CALL_UNIX with PV-WAVE as Client

Note  You can find the following listed file in:

```
$WAVE_DIR/util/rpc/wave_client.pro

;This PV-WAVE procedure tests an array
;of samples from a single host or list of
;hosts. Each array will be a LONG array of
;length LENGTH. TIMES specifies the number
;of times to call each host.

PRO samples, length=length, times=times, $
    hostlist=hostlist

IF (N_ELEMENTS(times) eq 0) THEN times=100
IF (N_ELEMENTS(length) eq 0 ) THEN length = $
    1000

IF (N_ELEMENTS (hostlist) eq 0) THEN $
    hostlist= 'localhost'

;See if hostlist is a single host or a
;list of hosts

num_hosts = SIZE(hostlist)
IF(num_hosts(0) EQ 0) THEN BEGIN

;hostlist is a single hostname
;make it a string array of length 1

name = hostlist
hostlist = strarr(1)
hostlist(0) = name
num_hosts = 1

ENDIF ELSE BEGIN

;hostlist is an array of hosts
;num_hosts is now number of hosts

num_hosts = num_hosts(1)
ENDELSE
```

```

;connect is an array to store "unit" values
connect = LONARR(num_hosts)

;The server (external C routine) is expecting
;a LONG. It's important to pass a LONG, so
;that the SERVER will not "die", i.e.:
;      Segmentation Violation : (core dumped)

length = LONG(length)
xpos = 0
ypos = 0
xsize = 300
ysize = 300

;Open one window for each host to monitor
FOR i=0, num_hosts - 1 DO BEGIN
  hostname = hostlist(i)
  PRINT, 'Opening Window for '+hostname
  WINDOW, i, colors=128, title= $
  ' Data from '+hostname, $
  xpos=xpos, ypos=ypos, xsize=xsize, $
  ysize=ysize

  xpos = xpos + xsize + 20
  IF (xpos GT 800) THEN BEGIN
    xpos = 0
    ypos = ypos + ysize + 30
  END
ENDFOR

; Draw the plots with color
IF(!d.n_colors GT 2) THEN BEGIN
  !p.background = 12
  !p.color = 127
  LOADCT,4
ENDIF

;Loop through the server the correct number
;of times

```



```

FOR j=1, times - 1 DO BEGIN
;Call each host and save the connection "unit"
FOR i=0, num_hosts - 1 DO BEGIN
; plot in the correct window
WSET, i

;A temporary variable is needed because
;PV=WAVE is not able to store into an
;expression

unit = connect (i)

;This is the call to the external
;routine/server

array = CALL_UNIX (length, hostname= $
hostlist (i), unit=unit)

;After the first call, connect (i) stays
;the same

connect (i) = unit
PLOT, array

;Empty the graphics buffer

EMPTY
ENDFOR
ENDFOR

;This is the last time; close all units
FOR i=0, num_hosts - 1 DO BEGIN
WSET, i
unit = connect (i)

;connect(i) is closed after this call

array = CALL_UNIX (length, hostname= $
hostlist (i), unit=unit, /close)

```

```

        PLOT, array
        EMPTY
    ENDFOR
END

```

Example External C Routine as a Server

Note 

You can find the following listed file in:

```
$WAVE_DIR/util/rpc/samples.c
```

```

#include "wave_rpc_extern.h"

/*****
    This is an example of a simple server. It
    does little error checking and thus can
    core dump. If, however, PV-WAVE always
    sends the correct parameters, there is no
    problem.

    This program is intended to respond to the
    PV-WAVE statement:

    ARRAY = CALL_UNIX(num_samples)

    where num_samples is a LONG. The PV-WAVE
    procedure SAMPLES.PRO gives an example of
    how to access this server.
*****/

#define MAX_SAMPLES 100000

main (argc, argv)
int argc;
char *argv[];
{
    int i, id;
    char *user, *proc;
    long num_samples, samples[MAX_SAMPLES];

```

```

/* If no program number is specified, 0 is the
   default */

if (argv[1] != NULL) {
    id = atoi (argv[1]);
}

else {
    id = 0;
}

/* Forever or until "kill -9" */

while (1) {

/* Listen for PV-WAVE to call via
   "CALL_UNIX(params)" */

    w_listen (id, &user, &proc);

/* WAVE_LONG is a macro that evaluates to:
   *(long *)w_get_par(0, TYP_LONG);
   w_get_param will return NULL if the param-
   eter is not a long. The NULL pointer could
   cause a segmentation violation and cause a
   core dump. It is ok, however, as long as PV-
   WAVE passes expected values. */

    num_samples = WAVE_LONG;

/* Generate the samples. rand() returns a ran-
   dom number */

for (i = 0; i < num_samples; i++)
    samples[i] = rand();

/* Reply to PV-WAVE with the array "samples".
   Only "num_samples" are used. PV-WAVE will
   receive a LONG array num_samples in length.
   W_SMPL_REPLY() does not return the passed
   parameters to PV-WAVE. Since this routine
   didn't modify any of the parameters there
   is no need to return them */

```

```

        w_smpl_reply (TYP_LONG, num_samples,
                    samples);
    }
}

```

Example C Routine with CALL_WAVE as a Client



You can find the following listed file in:

```
$WAVE_DIR/util/rpc/test_client.c
```

```
#include "wave_rpc_extern.h"
```

```
/******
```

This is an example of a client C program that calls a PV=WAVE server. First, a LONG array is generated with "rand ()". The array is then sent to PV=WAVE along with a smoothing factor and a plot title. The following PV=WAVE procedure is used to accept the information, smooth the array, and return it as the result of "CALL_WAVE()". Optionally, PV=WAVE will plot the smoothed array.

This program can be started in two fashions:
(program is the name of the executable version of this program)

```
program
```

```
or
```

```
program program_number hostname
```

The program number is needed if the PV=WAVE SERVER was started with a program number other than the default value of 0. The hostname is needed if the server is not running on "localhost".

```
*****/
```

```
/* Length of the array to be smoothed */
```

```

#define ARRAY_LENGTH 200

/* Window to use for smoothing */

#define SMOOTH_FACTOR 5
main (argc, argv)
int argc;
char *argv[];

{
    int make_single_array ();
/* Code to create a UT_VAR; follows main() */
    int i, id, j, unit, status, times;

    char *title;

    UT_VAR *retval;

/* Variables are passed to WAVE using
   "UT_VAR"s; see wave_rpc_extern.h for defi-
   nition. While only 3 UT_VARS are used in
   this example, up to 30 variables may be
   passed. ut_arr[3] and ut_arr[4] are not
   used in this example and are not necessary
   and are there for convenience. */

static UT_VAR ut_arr[5];

/* "call_wave" expects an array of UT_VAR
   pointers */

static UT_VAR *argptr[] = {
    &ut_arr[0],
    &ut_arr[1],
    &ut_arr[2],
    &ut_arr[3],
    &ut_arr[4]};

/* While the actual user of "procedure" and
   "user" are actually determined by the
   SERVER, it is suggested that the "proce-
   dure" parameter be used to select a

```

```

        particular function inside the SERVER and
        the "user" parameter be used as site-spe-
        cific security. */

char *user, *procedure, *hostname;

long num_neighbors, long_arr[ARRAY_LENGTH];

/* If no pargram number was specified, use 0 */
if (argc >= 2) {
    id = atoi (argv[1]);
}
else {
    id = 0;
}

/* If no hostname was specified, use "local-
host" */
if (argc >= 3) {
    hostname = argv[2];
}
else {
    hostname = "localhost";
}

/* Fake user data; generally used for site-
specific security */

user = "Test Client User Data";

/* Function inside the server that this CLIENT
wishes to access */

procedure = "SMOOTH";

/* Title for plot */

title = "Smoothed Random Numbers";

/* Window to use for smoothing */

num_neighbors = SMOOTH_FACTOR;

/* Figure how many times to call the server */

```

```

printf("Number of times to call SERVER %s :",
      hostname);

scanf ("%d", &times);

for (j = 0; j < times; j++) {
/* Generate a random array */
for (i = 0; i < ARRAY_LENGTH; i++) {
    long_arr[i] = rand ();
}
/* Create a UT_VAR with smoothing factor */
status = make_single_array (argptr[0],
TYP_LONG, 1, &num_neighbors);
if (status < 0) {
    sprintf(stderr, "Error building UT_VAR\n");
    exit (1);
}
/* Create UT_VAR with long array */

status = make_single_array (argptr[1],
    TYP_LONG | TYP_ARRAY, ARRAY_LENGTH,
    long_arr);
if (status < 0) {
    sprintf (stderr, "Error building UT_VAR\n");
    exit (1);
}
/* Make UT_VAR with title */

status = make_single_array (argptr[2],
TYP_STRING, 1, &title);
if (status < 0) {
    sprintf(stderr, "Error building UT_VAR\n");
    exit (1);
}
}

```

```
/******
```

This is the call to the SERVER. The SERVER's response will be pointed to by "retval".

CALL_WAVE() is used in by :

```
RETURN_VALUE = CALL_WAVE(number of UT_VARS in
parameter array, parameter array (UT_VAR
*array[]), name of host to call, unit (NULL
means don't care) close (if 1, close spci-
fied unit), procedure name, user data,
timeout in secs (0 means use default))
```

The unit parameter provides a way to maintain a connection between the SERVER and CLIENT. If unit is non-null and points to an integer = 0, then once the connection is established it is left open. Further calls using that unit will use the existing connection instead of creating a new one. If close is 1, then the connection is closed after the call.

Using the unit parameter causes extra file descriptors to be left open inside the UNDERTOE CLIENT. Since there is a limit on the number of open file descriptors, don't leave too many units open. Using the unit parameter is useful when establishing the connection takes a long time. Tests on several machines have shown that this time is generally much less than a second. That delay is only a significant portion of the total transfer time if the total data transfer is < 10,000 bytes and the SERVER's computation time is negligible. So to avoid some added housekeeping, use NULL

```
*****/
```

```
retval = (UT_VAR *) call_wave (3, argptr,
hostname,
```

```
NULL, 0, id, procedure, user, 0);
```



```

/* Check the type of the returned parameter */
/* If it is not an error and it's an array,
   print it */

if (retval->type == (TYP_LONG | TYP_ARRAY))
{
    for (i = 0; i < ARRAY_LENGTH; i++) {
/* value.array is defined as (char *) so cast
   to the appropriate type. */
printf ("%d ", ((long *)
    retval->value.array)[i]);
    }
}

else {
printf("SERVER did not return \n");
printf("expected value\n");
printf ("Returned from call_wave \n");

printf ("retval type is %d \n",
    retval->type);
}
}
}

```

Example C Function to Load a UT_VAR



You can find the following listed file in:

```
$WAVE_DIR/util/rpc/test_client.c
```

```

int make_single_array (ut_ptr, type, length,
    array_ptr)

unsigned char type; /* type of UT_VAR */
long length; /* length of array */
char *array_ptr; /* pointer to data */
UT_VAR *ut_ptr; /* pointer to UT_VAR */

```

```

{
/* Sample routine to build UT_VARS. This routine will only build UT_VARS that are single dimensioned. To build a multi-dimensional array set n_dim and dim[], i.e.:

A three dimensional array that is 4 by 5 by 6 would be:
    ut_ptr->n_dim = 3;
    ut_ptr->dim[0] = 4;
    ut_ptr->dim[1] = 5;
    ut_ptr->dim[2] = 6;
*/

/* switch on type ignoring the TYP_ARRAY bit */

switch (type & SIMPLE_MASK) {

case TYP_BYTE:
    ut_ptr->type = TYP_BYTE;
    ut_ptr->element_size = sizeof (char);
    ut_ptr->n_dim = 1;
    ut_ptr->dim[0] = length;
    if (type & TYP_ARRAY) {
        ut_ptr->type = ut_ptr->type | TYP_ARRAY;
        ut_ptr->value.array = array_ptr;
    }

else {
/* array_ptr is a (char *), cast to proper type */
    ut_ptr->value.c = *(unsigned char *)
        array_ptr;
}

break;

```

```

case TYP_INT:
    ut_ptr->type = TYP_INT;
    ut_ptr->element_size = sizeof (short);
    ut_ptr->n_dim = 1;
    ut_ptr->dim[0] = length;
    if (type & TYP_ARRAY) {
        ut_ptr->type = ut_ptr->type | TYP_ARRAY;
        ut_ptr->value.array = array_ptr;
    }
else {
    /* array_ptr is a (char *), cast to proper
       type */
    ut_ptr->value.i = *(short *) array_ptr;
}
break;

case TYP_LONG:
    ut_ptr->type = TYP_LONG;
    ut_ptr->element_size = sizeof (long);
    ut_ptr->n_dim = 1;
    ut_ptr->dim[0] = length;
    if (type & TYP_ARRAY) {
        ut_ptr->type = ut_ptr->type | TYP_ARRAY;
        ut_ptr->value.array = array_ptr;
    }
else {
    /* array_ptr is a (char *), cast to proper
       type */
    ut_ptr->value.l = *(long *) array_ptr;
}
break;

```

```

case TYP_DOUBLE:
    ut_ptr->type = TYP_DOUBLE;
    ut_ptr->element_size = sizeof (double);
    ut_ptr->n_dim = 1;
    ut_ptr->dim[0] = length;
if (type & TYP_ARRAY) {
    ut_ptr->type = ut_ptr->type | TYP_ARRAY;
    ut_ptr->value.array = array_ptr;
}
else {
    /* array_ptr is a (char *), cast to proper
       type */
    ut_ptr->value.d = *(double *) array_ptr;
}
break;

case TYP_FLOAT:
    ut_ptr->type = TYP_FLOAT;
    ut_ptr->element_size = sizeof (float);
    ut_ptr->n_dim = 1;
    ut_ptr->dim[0] = length;
    if (type & TYP_ARRAY) {
        ut_ptr->type = ut_ptr->type | TYP_ARRAY;
        ut_ptr->value.array = array_ptr;
    }
else {
    /* array_ptr is a (char *), cast to proper
       type */
    ut_ptr->value.f = *(float *) array_ptr;
}
break;

```

```

case TYP_COMPLEX:
    ut_ptr->type = TYP_COMPLEX;
    ut_ptr->element_size = sizeof (COMPLEX);
    ut_ptr->n_dim = 1;
    ut_ptr->dim[0] = length;
    if (type & TYP_ARRAY) {
        ut_ptr->type = ut_ptr->type | TYP_ARRAY;
        ut_ptr->value.array = array_ptr;
    }
else {
    /* see wave_rpc_extern.h for definition of
    COMPLEX array_ptr is a
    * (char *), cast to proper type */
    ut_ptr->value.cmp.r = ((COMPLEX *)
        array_ptr)->r;
    ut_ptr->value.cmp.i = ((COMPLEX *)
        array_ptr)->i;
}
break;
case TYP_STRING:
    ut_ptr->type = TYP_STRING;
    ut_ptr->element_size = sizeof(char);
    ut_ptr->n_dim = 1;
    ut_ptr->dim[0] = length;
    if (type & TYP_ARRAY) {
        /* a string array is an array of (char *) */
        ut_ptr->element_size=sizeof (char*);
        ut_ptr->type = ut_ptr->type | TYP_ARRAY;
        ut_ptr->value.array = array_ptr;
    }
else {

```

```

/* array_ptr is a (char *), cast to proper
type */
    ut_ptr->value.string = *(char **)
        array_ptr;

}

break;

default:

/* unsupported types: at present
TYP_STRUCTURE is not supported */

return (-1);

break;

}

}

```

Example Procedure Using UNIX_LISTEN and UNIX_REPLY with PV-WAVE as a Server

Note 

You can find the following listed file in:

\$WAVE_DIR/util/rpc/example_server.pro

```

;This is an example of using PV-WAVE as the
;SERVER. Times is the number of times to loop
;through the SERVER. Show specifies that the
;resulting array should be plotted.

```

```

PRO example_server, Times=times, Show=show
@UT_COMMON

```

```

prog = 0

```

```

;Default is to loop through only once

```

```

IF (n_elements(times) EQ 0) then times = 1

```

```

WHILE(1) DO BEGIN

```

```

FOR i = 1, times DO BEGIN

```

```

;Listen for a call using the default program
;number zero. Return the received "procedure
;string into the variable "proc"

n = UNIX_LISTEN(program = prog, procedure = $
  proc)

;Make sure the proc is SMOOTH and we have the
;correct number of parameters

IF (n EQ 3) and (proc EQ 'SMOOTH') THEN BEGIN
;Smooth the array
array = SMOOTH(ut_param1, ut_param0)

;Return the array to the caller but do not
;return the parameters. To return the
;parameters, the call would be:
;
;   status = UNIX_REPLY(array, /RETURN)
status = UNIX_REPLY(array)

;Only plot when told to do so
IF (keyword_set(show)) THEN BEGIN
  !P.title = ut_param2
  PLOT, array
  EMPTY
  ENDIF
ENDIF ELSE BEGIN

;The server did not get the correct proc or
;did not get the correct number of parameters

```

```
PRINT, 'EXAMPLE_SERVER MISMATCH: ' + $
      'PROC = ',proc, ' N_PARAMS = ', n
      status = UNIX_REPLY(-1)
ENDELSE
ENDFOR
ENDWHILE
END
```


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:

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, on a Sun-4 computer, if you define a key with the statement:

```
DEFINE_KEY, /Terminate, 'R15', 'INFO, /Files'
```

the `INFO, /Keys` command produces output that includes the line:

```
R15 'INFO, /Files' <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. For more information on reviewing and reentering previously entered commands, see *Using Command Recall* on page 32 of 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 WAVE Widgets

This chapter explains how to use WAVE Widgets to create graphical user interface (GUI) applications in PV-WAVE. Several methods of GUI development are available to the PV-WAVE programmer. These include:

- WAVE Widgets
- Widget Toolbox
- GUI Builders
- C-based Applications

The first section of this chapter, *Methods of GUI Programming in PV-WAVE* on page 406, is designed to help you choose the most appropriate method for your application.

The rest of this chapter describes WAVE Widgets, a set of easy-to-use, high-level widget routines that allow you to develop Motif or OPEN LOOK GUIs with PV-WAVE. The Widget Toolbox is discussed in the next chapter.

Methods of GUI Programming in PV-WAVE

WAVE Widgets — An easy-to-use set of PV-WAVE functions for creating Motif or OPEN LOOK GUIs for PV-WAVE applications. Applications created with WAVE Widgets are completely portable between Motif and OPEN LOOK environments. WAVE Widgets are designed for developers who have little or no experience using either Motif or OLIT. See *Introduction to WAVE Widgets* on page 410 for more information.

Widget Toolbox — A set of highly flexible PV-WAVE functions used to create Motif or OPEN LOOK Graphical User Interfaces (GUIs) for PV-WAVE applications. The Widget Toolbox functions call Motif and OPEN LOOK Intrinsic Toolkit (OLIT) routines directly, and are designed primarily for developers who are already experienced using either Motif or OLIT. See Chapter 16, *Using the Widget Toolbox*, for detailed information on the Widget Toolbox functions.

Figure 15-1 shows how WAVE Widgets and the Widget Toolbox are layered on top of the Motif and OLIT toolkits, Xt Intrinsic, Xlib, and the operating system.

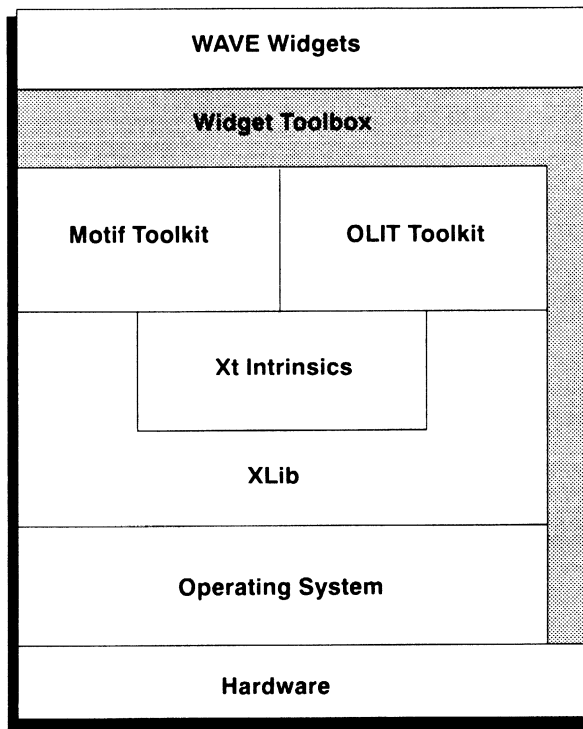


Figure 15-1 WAVE Widgets and the Widget Toolbox are built on top of the Motif and OLIT toolkits. The Widget Toolbox also communicates with Xt Intrinsic, XLib, and the Operating System.

Table 15-1 lists the advantages and disadvantages of all the methods available to PV-WAVE developers for creating GUI applications. These methods include:

- WAVE Widgets
- Widget Toolbox
- GUI Builders
- C-based Applications

Use Table 15-1 to help determine which method is best for you.

Table 15-1: GUI Development Methods: Advantages and Disadvantages

Method	Advantages	Disadvantages
Widget Toolbox	<ul style="list-style-type: none"> • Creates Motif or OPEN LOOK GUIs. • All programming done in PV-WAVE CL. • All Motif and OLIT widget classes are supported. • Allows rapid prototyping. • Highly flexible. • No knowledge of C required. 	<ul style="list-style-type: none"> • Application developer must be experienced using the Motif or OLIT widget toolkit. • Not portable between Motif and OLIT environments.
WAVE Widgets	<ul style="list-style-type: none"> • Easy to use. • Creates Motif or OPEN LOOK GUIs. • Completely portable between Motif and OPEN LOOK environments (no modification required). • All programming done in PV-WAVE CL. • No experience programming with the Motif or OLIT widget toolkit required. • No knowledge of C required. • WAVE Widgets functions can be modified, and new WAVE Widgets functions can be created by users. • Allows rapid prototyping. 	<ul style="list-style-type: none"> • Limited number of widget classes are supported. • Less overall flexibility in interface design.

Table 15-1: GUI Development Methods: Advantages and Disadvantages

Method	Advantages	Disadvantages
<p>GUI Builders:</p> <ul style="list-style-type: none"> • PV-WAVE:GUI/Maker • PV-WAVE:UIM/X <p>(Comprehensive GUI-building tools that allow you to interactively design, lay out, modify, and test the appearance and behavior of GUIs.)</p>	<ul style="list-style-type: none"> • Automatically generate the C code which creates the interface. • Can be used to develop extensive, professional applications with PV-WAVE CL and the Motif GUI. • PV-WAVE:UIM/X provides access to 100% of Motif. • PV-WAVE GUI/Maker provides access to 16 widgets via a pre-defined palette. 	<ul style="list-style-type: none"> • Knowledge of C required. • Motif only. OPEN LOOK not supported. • Must be purchased separately. • UNIX only.
<p>C-based applications that call PV-WAVE CL</p> <p>(The application interface can be developed in C, and, via PV-WAVE CL's inter-application communication functions, the C application can call PV-WAVE CL to perform data processing and display functions.)</p>	<ul style="list-style-type: none"> • Highly flexible. • Can be C code generated by any GUI builder. • Can be used to add Visual Data Analysis capability to an existing application. 	<ul style="list-style-type: none"> • Application developer must be experienced using the Motif or OLIT widget toolkit. • Most complicated method of application development. • Knowledge of C required. • Knowledge of interapplication communication required. • Not portable between Motif and OPEN LOOK environments.

Introduction to WAVE Widgets

WAVE Widgets provides an easy way for PV-WAVE application developers to create Motif or OPEN LOOK GUIs. You can think of a widget as a user-interface object, such as a dialog box, a button box, or a file selection box. WAVE Widgets is a set of PV-WAVE functions that create a number of different kinds of widgets. Widget characteristics such as text, color, and position are controlled using keywords. See page 416 for a complete list of the types of WAVE Widgets.

For example, Figure 15-1 shows a Motif-style dialog box created with the WAVE Widgets function `WwDialog`.

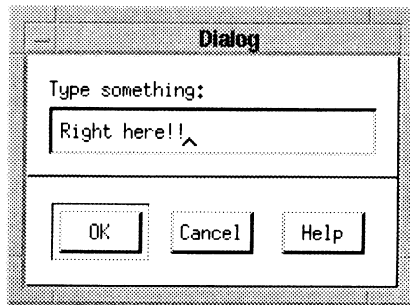


Figure 15-1 Motif-style dialog box, created with the `WwDialog` function.

Who Uses WAVE Widgets

WAVE Widgets are designed for PV-WAVE developers with little or no experience using the Motif or OLIT GUI toolkits. Developers who are experienced GUI programmers may want to use WAVE Widgets for the rapid development of application prototypes.

WAVE Widgets are Standard Library Functions

The WAVE Widgets functions are located in the PV-WAVE Standard Library in the subdirectories:

```
wave/lib/std/motif
```

and

```
wave/lib/std/olite
```

Designing Your Own WAVE Widgets

The implementation of WAVE Widgets is straightforward, making it relatively easy for developers to customize and invent their own WAVE Widgets routines. Developers who create their own WAVE Widgets routines may share them with other users by submitting them to the PV-WAVE Users' Library. For more information see *The Users' Library* on page 255.

WAVE Widgets are Portable

Applications developed with WAVE Widgets are completely portable between OPEN LOOK and Motif window systems. No matter which window system your application is destined to use, the WAVE Widgets programming interface is always the same. PV-WAVE automatically detects whether you are running Motif or OPEN LOOK, and executes the appropriate WAVE Widgets routines.

Specifying the Desired Toolkit

If you are running your application on a Sun workstation, you can use the environment variable `WAVE_GUI` to control whether the widgets are implemented using the Motif or the OPEN LOOK look-and-feel. To maximize user satisfaction, the look-and-feel you select should match the one used by other applications your users are accustomed to, and should be compatible with the window manager you expect your users to be using when they run your PV-WAVE application. On all other platforms that support

PV-WAVE, Motif is the only option, and thus setting WAVE_GUI on those platforms has no effect.

Note 

For detailed information on setting WAVE_GUI, see *WAVE_GUI: Selecting the GUI on Sun Workstations* on page 51 of the *PV-WAVE User's Guide*.

First Example and Basic Steps

This section briefly introduces the basic steps involved in creating a WAVE Widgets application.

First Example

The following example incorporates each of the basic steps described later in this section. To run this example, enter the callback procedure below in a file, and compile it with the .RUN command. Then enter the widget commands at the WAVE> prompt. A radio button box appears on the screen. Each time you click on a radio button, the callback procedure is executed and some information is printed in the main PV-WAVE window. The radio box is shown in Figure 15-2.

To dismiss the radio box widget, select the appropriate function (such as **C**lose) from the widget's window manager menu. You can run this example under Motif or OPEN LOOK.

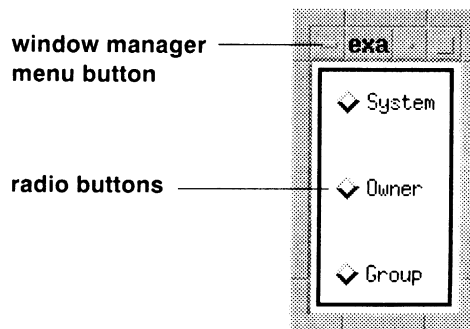


Figure 15-2 Radio button box (Motif style).

Callback Procedure



Note

Callback procedures are PV-WAVE routines that are executed in response to an event that occurs inside a widget, such a mouse click.

```
PRO RadioCB, wid, which
CASE which OF
  1: PRINT, 'First Toggle Selected'
  2: PRINT, 'Second Toggle Selected'
  3: PRINT, 'Third Toggle Selected'
ENDCASE
value = WwGetValue(wid)
PRINT, value
END
```

Widget Commands

```
top=WwInit('ww_ex1', 'Examples', layout)
labels=['System', 'Owner', 'Group']
rbox=WwRadioBox(layout, labels, 'RadioCB', $
  /Vertical, Border=2, Spacing=20)
status=WwSetValue(top, /Display)
WwLoop
```

The Basic Steps

To use WAVE Widgets in a PV-WAVE application, you always follow these basic steps. These steps are described in more detail in the remainder of this chapter.

- ❑ Create callback procedures.

Callback procedures are PV-WAVE routines that are executed in response to an event, such as clicking a button or dismissing a dialog box.

- ❑ Initialize WAVE Widgets with the WwInit function. For example:

```
top = WwInit('appl', 'Appl', layout)
```

- ❑ Create widgets by calling the appropriate WAVE Widgets functions. All WAVE Widgets function names begin with `Ww`. For example:

```
bbox=WwButtonBox(layout, labels, 'ButtonCB', $
    /Horizontal, Spacing=20)
```

- ❑ Display the top-level widget with the `WwSetValue` command and the *Display* keyword. For example:

```
status=WwSetValue(top, /Display)
```

- ❑ Execute the `WwLoop` function. This function executes the “event loop”, which handles events (such as mouse clicks) and dispatches callbacks (PV-WAVE routines that are executed in response to events).

Initializing WAVE Widgets

The WAVE Widgets function `WwInit` initializes WAVE Widgets. You must place a call to `WwInit` at the beginning of any application that uses WAVE Widgets. `WwInit` does the following:

- Establishes a connection to the X server.
- Initializes Xt Intrinsics.
- Initializes the Motif or OLIT Toolkit.
- Initializes WAVE Widgets.
- Creates a top-level shell.
- Creates a layout widget inside the top-level shell.

`WwInit` has the following form:

```
top = WwInit(name, class, workarea [, keywords])
```

The *name* parameter specifies the name of the application, and the *class* parameter indicates a more general category of application class. `WwInit` also creates the first top-level shell or root window and an initial layout widget. The ID of the layout widget is returned by the *workarea* parameter. The function returns the ID of the top-level shell. For example,

```
top=WwInit('simple_image', 'Examples', layout, $
          Background = 'Skyblue')
```

where `simple_image` is the name of the application, and `Examples` specifies a general class to which `simple_image` belongs. The `layout` parameter returns the ID of the layout widget that is created inside the top-level shell. Finally, a background color is specified with the *Background* keyword. The ID of the top-level shell is returned in the variable `top`. For more information on the top-level shell, see *The Widget Hierarchy* on page 416.

Note

One purpose of specifying a general class of application (the *class* parameter) is that resources can be shared, via a resource file, among elements of that class. In general, WAVE Widgets applications do not require a resource file. Some developers, however, may create applications that use both WAVE Widgets and Widget Toolbox calls to produce the GUI. In this case, a resource file can be shared by all of the widgets used in the application. Note, however, that if color keywords, such as *Background* and *Foreground*, are used in a WAVE Widgets call, the specified color(s) override color specifications made in a resource file.

For more information on layout widgets, see *Arranging Widgets in a Layout* on page 419.

Example

Here's a simple example showing the use of `WwInit` in the creation of a multi-line text widget. You can display the text widget by entering the commands as shown at the `WAVE>` prompt.

```
top=WwInit('ww_ex2', 'Examples', layout)
filename = getenv('WAVE_DIR')+ $
          '/Tips'
text=WwText(layout, 'TextCB', /Read, $
            File=filename, Cols=40, Rows=20)
status=WwSetValue(top, /Display)
WwLoop
```

Creating and Arranging WAVE Widgets

This section explains the widget hierarchy, lists the general types of widgets that are available, and explains how to arrange widgets in a layout.

The Widget Hierarchy

WAVE Widgets applications consist of a hierarchy of widgets. The widget hierarchy refers to the top-level or root widget and all of the widgets that are related to it. The relationship between widgets in a hierarchy is usually described as a “parent/child” relationship. Each time you create a new widget, you must specify its parent.

At the top of every widget hierarchy must be a special type of widget called the root or main window. This window is created by the `WwInit` function. The root window widget provides an interface between the widget hierarchy and the window manager. In addition, `WwInit` creates a “layout” widget, which is like a container in which other widgets are arranged.

Note 

You can create additional main windows with the `WwMainWindow` function.

Figure 15-3 shows a WAVE Widgets GUI that is composed of a root window, two layout widgets, and ten other widgets (w 1 – w 10). These widgets could be buttons, sliders, menu bars, etc.

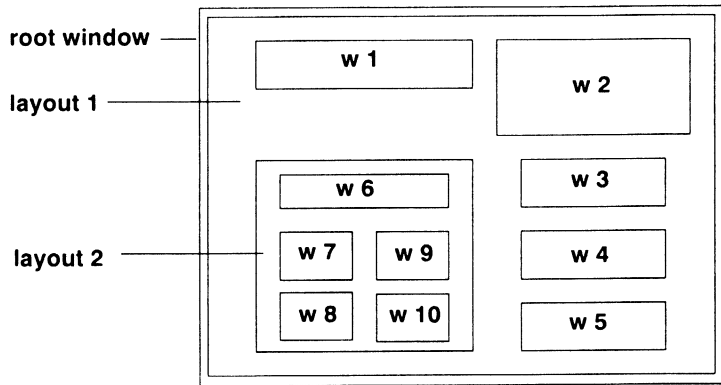


Figure 15-3 Schematic of WAVE Widgets GUI that contains two layout widgets and ten other widgets.

Figure 15-4 shows the hierarchical relationship between the widgets in the above GUI. The root window is the top-level window, and it is the parent of layout 1 (both the root window and the first layout widget are created by `WwInit`). Layout 1 is the parent of widgets `w 1`, `w 2`, `w 3`, `w 4`, `w 5`, and layout 2. Layout 2 is the parent of widgets `w 6`, `w 7`, `w 8`, `w 9`, and `w 10`.

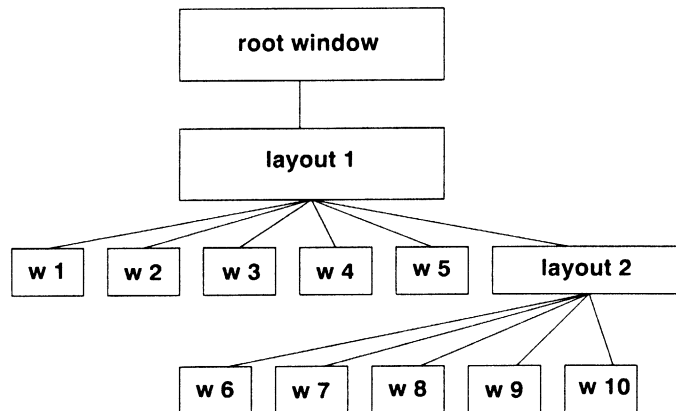


Figure 15-4 The widget hierarchy for the WAVE Widgets application shown in Figure 15-3.

Each separate widget is represented externally by a widget ID (a variable of type long), returned by the creation function, such as `WwButtonBox`.

Types of WAVE Widgets

The following is a list of the kinds of widgets you can create with WAVE Widgets. Examples are shown throughout this chapter.

- **Button Box** — A horizontally or vertically oriented box containing push buttons. See page 433 for more information.
- **Command Widget** — A widget used for command entry with a built-in command history mechanism. It includes an input text field, a label, and a command history list. See page 459 for more information.
- **Controls Box** — Horizontally or vertically oriented sliders, which can optionally contain text input fields for entering exact values. A slider allows the user to set or display the values of the variables that fall within a predefined range. See page 437 for more information.
- **Dialog Box** — A blocking (modal) or nonblocking (modeless) dialog box containing a text input field and button box with control buttons. See page 453 for more information.
- **Drawing Box** — A box that displays graphics generated by PV-WAVE. See page 439 for more information.
- **File Selection Box** — Displays the contents of directories and lets the user select files. See page 456 for more information.
- **Layout Widget** — A “container” used to hold other widgets in a specific arrangement. Types of arrangements include: row/column, form, and bulletin board. Keywords are used to select the type of layout and the orientation, spacing, and sizing of the widgets in the layout. By default, a layout widget is created when WAVE Widgets is initialized with the `WwInit` command. See page 416 for more information.
- **List Box** — A scrolling list that allows users to select one or more items from a group of choices using the mouse. An addi-

tional callback can be defined for the default action, activated with a double-click. See page 446 for more information.

- **Main Window** — A top-level (window manager) window and layout widget. By default, the `WwInit` function creates a top-level widget and a layout widget. `WwMainWindow` lets you create additional top-level and layout widgets. See the *PV-WAVE Reference* for more information.
- **Menu Bar** — A series of menu buttons. See page 425 for more information.
- **Message Box** — A popup message box containing a text message, which can be blocking or nonblocking. See page 449 for more information.
- **Option Menu** — A menu button that reflects the currently selected menu item. See page 425 for more information.
- **Popup Menu** — A menu that appears when the user presses the right mouse button over a parent widget. See page 425 for more information.
- **Radio Box** — A specified number of rows or columns of labeled toggle buttons. See page 436 for more information.
- **Table Widget** — An editable 2D array of cells similar to a spreadsheet. See page 462 for more information.
- **Text Area** — A static text label, a text entry field, or a full window editor. See page 443 for more information.
- **Tool Box** — An array of graphic buttons (icons) arranged in a specified number of columns or rows. See page 433 for more information.

Arranging Widgets in a Layout

A layout widget is like the canvas on which other widgets are drawn in whatever arrangement you specify. All types of widgets, except the root window and popup widgets, must be related to a layout widget.

The `WwInit` function creates a main window and one layout widget by default. Additional layout widgets can be created with the `WwLayout` function.

A layout widget allows three types of basic arrangements:

- Row/column layout
- Bulletin Board layout
- Form layout

In addition, layout widgets can be embedded within other layout widgets to create more complex GUIs.

Row/Column Layout

Row/column is the default layout. A row/column layout consists of widgets arranged either horizontally or vertically depending on the keywords used. The following commands create a row/column layout, where the widgets are horizontally aligned, with five pixels of space between each widget and widget borders three pixels wide. The result is shown in Figure 15-5.

```
top=WwInit('ww_ex3', 'Examples', layout, $
  /Horizontal, Spacing=5, Border=3)
btn1=WwButtonBox(layout, 'Button 1', 'CB')
btn2=WwButtonBox(layout, 'Button 2', 'CB')
btn3=WwButtonBox(layout, 'Button 3', 'CB')

status=WwSetValue(top, /Display)

WwLoop
```

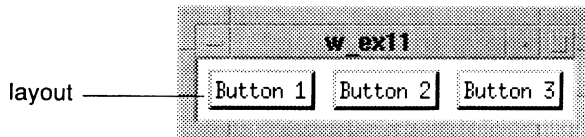


Figure 15-5 Horizontally aligned layout with three buttons (Motif style).

Bulletin Board Layout

To create a bulletin board layout, the `WwInit` or `WwLayout` function is used with the *Board* keyword.

Each widget is positioned on the bulletin board with the *Position* keyword, which specifies *x* and *y* coordinates. By default, widgets are placed in the upper-left corner of the bulletin board (coordinates *x*=0, *y*=0).

For example, the following calls position button widgets on the bulletin board called `bboard`. Note that the positions of the buttons are specified with the *Position* keyword. The result is shown in Figure 15-6.

```
top=WwInit('ww_ex4', 'Examples', bboard, $
/Board)

btn1=WwButtonBox(bboard,'Button 1', 'CB', $
Position=[0,0])

btn2=WwButtonBox(bboard,'Button 2', 'CB', $
Position=[0,50])

btn3=WwButtonBox(bboard,'Button 3', 'CB', $
Position=[0,100])

status=WwSetValue(top, /Display)

WwLoop
```

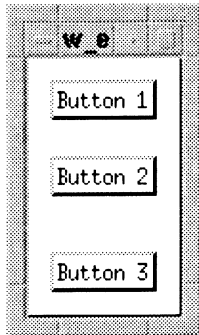


Figure 15-6 Bulletin board layout (Motif style).

Form Layout: Attachments

On a form layout, widgets are “attached” to one another. These attachments are specified with the keywords *Top*, *Bottom*, *Right*, and *Left*. You can specify widget attachments in relation to the parent widget or in relation to other child widgets. Many combinations of attachments are possible, and it is best to experiment with the attachment keywords to produce the desired effect.

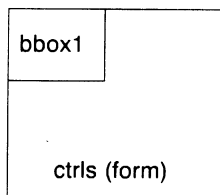
For example, the following call creates a form layout called `ctrls`:

```
ctrls=WwLayout(top, /Form)
```

If no attachment keyword is specified, a child widget `bbox1` is placed in the upper left corner of the layout. For example:

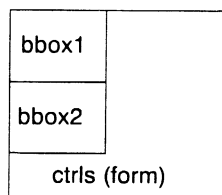
```
bbox1=WwButtonBox(ctrls, 'Click Here', $  
    'buttonCB')
```

The result is illustrated in the following figure:



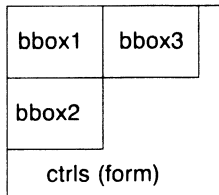
The next call places another button in the layout. This attachment places the top of `bbox2` on the bottom of `bbox1`, as shown in the following figure.

```
bbox2=WwButtonBox(ctrls, 'Click Here', $  
    'buttonCB', Top=bbox1)
```



The third button is attached to the right edge of the first button, bbox1, as shown in the following figure.

```
bbox3=WwButtonBox(ctrls, 'Click Here', $  
    'buttonCB', Left=bbox1)
```



Finally, a fourth button is attached to bottom of the third button and to the left of the second button, as shown in the following figure.

```
bbox4=WwButtonBox(ctrls, 'Click Here', $  
    'buttonCB', Left=Box2, Top=bbox3)
```

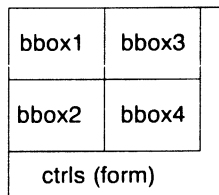


Figure 15-7 summarizes the general effect of widget attachments specified for the widget Child_2 in relation to the widget Child_1.

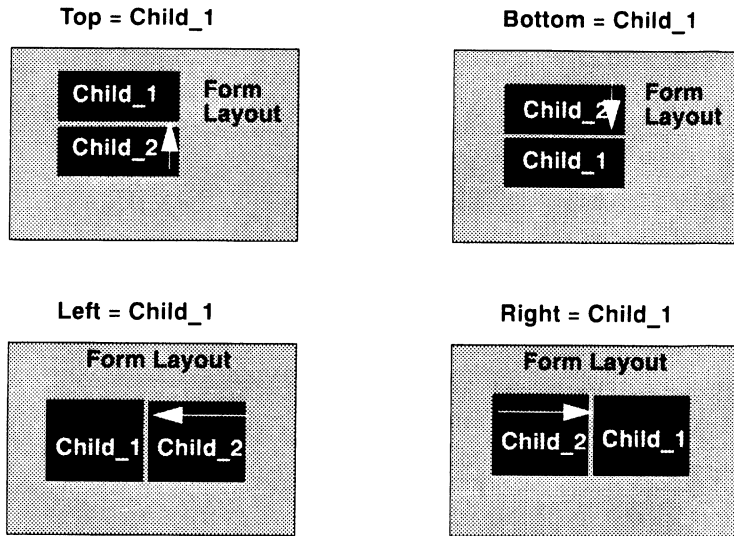


Figure 15-7 Effects of attachment keywords specifying attachments in relation to child widgets.

You can also specify attachment keywords with a value of one (*Keyword = 1*). This positions widgets in relation to the parent layout. For example:

```
bbox1=WwButtonBox(layout, 'Click Here', $
                'buttonCB', /Bottom)
```

This call attaches the widget `bbox1` to the bottom of the parent layout widget `layout`. These default attachments are always specified in relation to the parent widget.

Figure 15-8 summarizes the effect of some basic attachment keyword defaults (*Keyword = 1*).

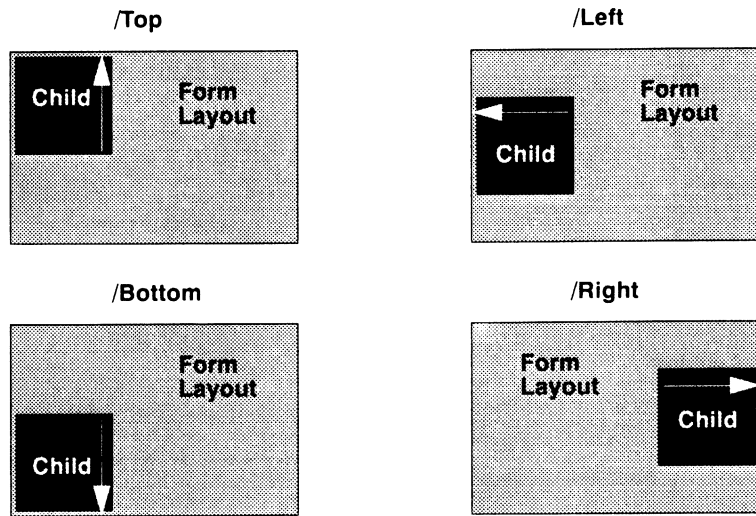


Figure 15-8 Effects of the default attachments for a single child widget.

Creating and Handling Menus

This section discusses the three basic types of WAVE Widgets menus:

- Menu bar
- Popup menu
- Option menu

Menu Bar

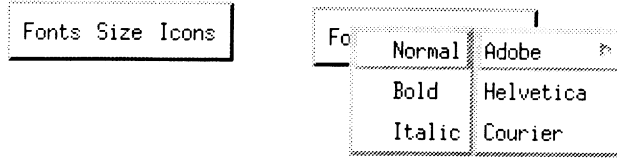


Figure 15-9 A menu bar. On the left no menu items are selected. On the right, the Fonts menu is selected, and an additional pullright menu is displayed.

A menu bar is a set of buttons that activate menus. When you select a menu button with the mouse, a pulldown menu appears with a set of menu items. A menu item can itself activate another menu, called a pullright menu, as shown in Figure 15-9.

The `WwMenuBar` function is used to create a menu bar. This function takes two parameters:

$$menubar = \text{WwMenuBar}(\text{parent}, \text{items})$$

The returned value, *menubar*, is the ID of the newly created menu bar widget. The *parent* parameter is the widget ID of the parent widget, often the ID of the layout widget. The *items* parameter is an unnamed structure containing all of the menu information. For detailed information on the *items* parameter, see *Defining Menu Items with Unnamed Structures* on page 429.

Popup Menu

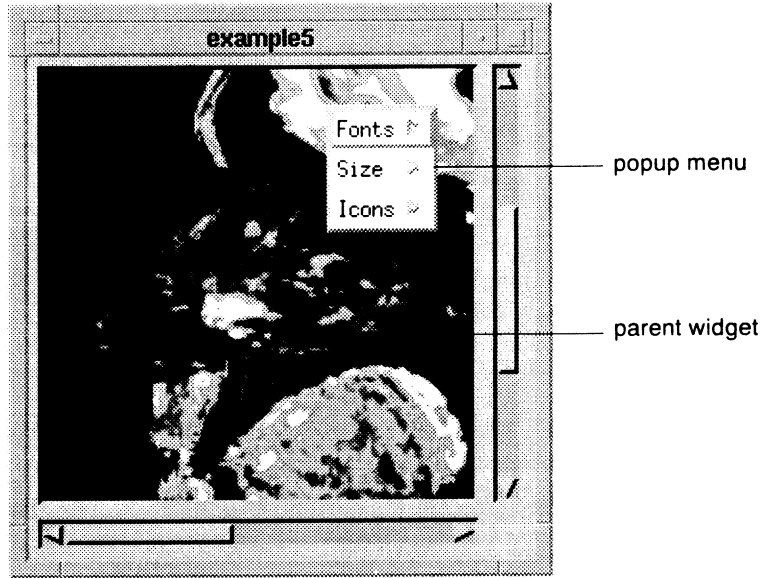


Figure 15-10 Simple popup menu, activated over a drawing area parent widget (Motif style).

A popup menu, shown in Figure 15-10, is a menu that appears when you press the MENU mouse button (usually the right button) while the cursor is positioned inside the menu's parent widget.

The `WwPopupMenu` function is used to create a popup menu. This function takes two parameters:

```
menubar = WwPopupMenu(parent, items)
```

The *parent* parameter is the widget ID of the parent widget. The *items* parameter is an unnamed structure containing all of the menu information. For detailed information on the *items* parameter, see *Defining Menu Items with Unnamed Structures* on page 429.

Option Menu

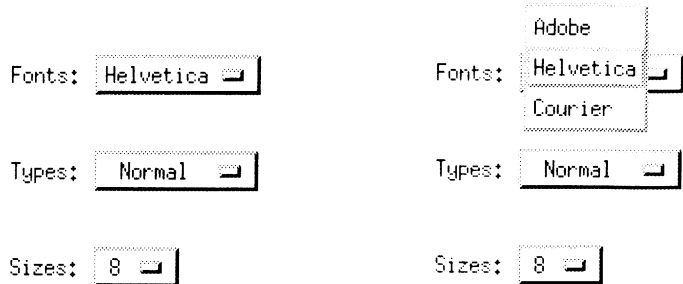


Figure 15-11 On the left are three option menus, Fonts, Types, and Sizes. Each one displays the current selection. On the right, the Fonts menu is selected.

An option menu, shown in Figure 15-11, is a menu button that displays the current selection. When the user presses the appropriate mouse button (usually the left button for Motif and the right for OPEN LOOK) over an option menu button, a menu appears. After the user makes a selection, the option button's text changes to reflect the current selection.

The `WwOptionMenu` function is used to create an option menu. This function takes three parameters:

options = `WwOptionMenu(parent, label, items)`

The *parent* parameter is the widget ID of the parent widget. The *label* parameter is a text string containing a label for the menu. The *items* parameter is an unnamed structure containing all of the menu information. For details on the *items* parameter, see *Defining Menu Items with Unnamed Structures* on page 429.

Menu Callbacks

When an active menu item is selected, the menu callback is called with the menu's widget ID as the first parameter and the menu item index (1...*n*) as the second parameter.

For example, here is a very simple callback routine called MenuCB, which prints a message depending on which menu button is selected.

```
PRO MenuCB, wid, which
CASE which OF
  1: PRINT, 'First Button Selected'
  2: PRINT, 'Second Button Selected'
  3: PRINT, 'Third Button Selected'
ENDCASE
END
```

Defining Menu Items with Unnamed Structures

Each of the menus discussed above takes a parameter (*items*) that contains the menu information. This parameter is defined as an unnamed structure.

An unnamed structure has the following general definition:

$$x = \{, tag_name_1: tag_def_1, tag_name_n: tag_def_n\}$$

For detailed information on unnamed structures, see *Creating Unnamed Structures* on page 105.

The following tag names and tag definitions can be used in the unnamed structure used to define menu items. For an example, see the next section.

- `title: 'name'` — (optional) Specifies a title for the menu.
- `pushpin: TRUE/FALSE` — (optional) Specifies if a pushpin is displayed. By default, a pushpin is not displayed. Pushpins are an OPEN LOOK feature only.
- `callback: 'cbkname'` — The name of the callback routine to be executed when an active menu button is selected. Always place the callback name before the active button's definition. The active items, which are described below, include `button`, `icon`, `toggle`, and `menubutton`.

- `button: 'labelname'` — The name of a pushbutton on the menu. A pushbutton is a button that calls a callback when selected.
- `icon: 'bitmap_filename'` — Creates an iconic (graphic) pushbutton. The bitmap filename is the full name of a file containing the icon's bitmap.
- `toggle: 'labelname'` — Creates a “toggle” type of button. A toggle button contains a small box to the left of the label. When the button is selected, the box is highlighted. When the button is not selected, the box is not highlighted.
- `menubutton: 'labelname'` — The name of a menu button on the menu bar, or in the menu for pullright menus.
- `menu: structure` — An unnamed structure that defines the contents of menus. For a simple pulldown menu, the structure has only one level. Include pullright menus by embedding additional structures in the top-level structure.
- `separator: value` — Separates the previous from the next menu item. Possible values are:

Values for Motif	Separator
0	No line
1	Single line
2	Double line

Values for OLIT	Separator
0	5 pixels of separation
1	10 pixels of separation
2	15 pixels of separation

- `current: index (1 – n)` — Used only for the option menu. Specifies the item to be selected as current when the option menu is created. This tag must be the last one in the list of tags in the structure definition.

Modifying Menu Items

WwMenuItem lets you dynamically update menus that have already been created. All menu items are placed in a parent menu pane, and the widget ID of the appropriate menu pane can be acquired using the *Menus* keyword of the WwMenuBar, WwPopupMenu, or WwOptionsMenu function. For details on the WwMenuItem function, see *WwMenuItem Function* on page 451 of the *PV-WAVE Reference, Volume II*.

Example

The following example shows the PV-WAVE code used to create the items for the menus shown in Figure 15-12. The menu bar contains three menus: **Fonts**, **Size**, and **Icons**. The contents of these menus are defined by the unnamed structure, *menus*, which is then passed to the WwMenuBar function. Note that the first menu, **Fonts**, contains a pullright menu called **Adobe**. This is defined as a separate unnamed structure embedded in the top-level structure.

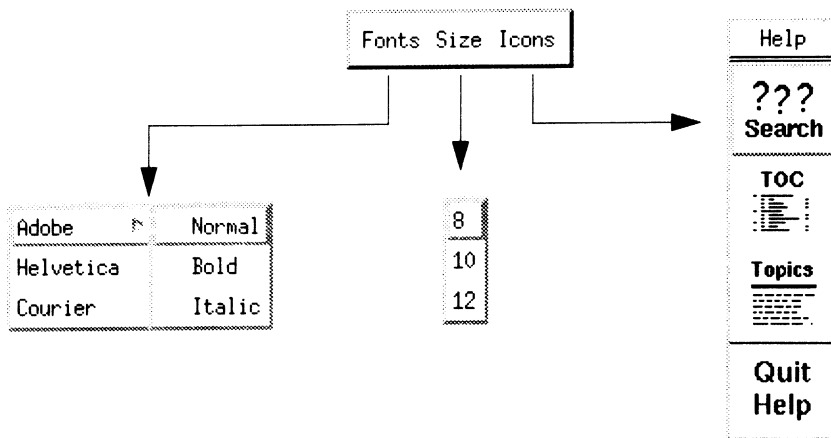


Figure 15-12 Menu bar containing three menus. Menus are displayed when the user presses the SELECT mouse button (usually the left button) over the menu name. The menus for each button are shown below the menu bar.

```

top=WwInit('ww_ex5', 'Examples', layout)
menus = {, callback: 'MenuCB', $
    menubutton: 'Fonts', $
        Create menu button "Fonts" on the menu bar.
    menu: {, callback: 'MenuCB', $
        menubutton: 'Adobe', $
            A new menu is created by embedding another unnamed struc-
            ture in the top-level structure. The menubutton tag creates a
            pullright menu called "Adobe". The pullright menu contains the
            toggle buttons "Normal", "Bold", and "Italic".
        menu: {, callback: 'MenuCB', $
            toggle: 'Normal', $
            toggle: 'Bold', $
            toggle: 'Italic', $
        button: 'Helvetica', $
        button: 'Courier', $
            Create two more pushbuttons on the Fonts menu.
        menubutton: 'Size', $
            Create a second menu button "Size" on the menu bar.
        menu: {, callback: 'MenuCB', $
            button: '8', $
            button: '10', $
            button: '12', $
            Create three pushbuttons on the "Size" menu.
        menubutton: 'Icons', $
            Create a third menu button "Icons" on the menu bar.
        menu: {, callback: 'MenuCB', $
            pushpin: 0, $
                Do not display a pushpin.
            title: 'Help', $
                Create a title for the Icons menu.
            icon: getenv('WAVE_DIR') + $
                '/xres/wxbm_btn_help_search', $
            icon: getenv('WAVE_DIR') + $
                '/xres/wxbm_btn_help_toc', $

```

```

icon:getenv('WAVE_DIR')+ $
    '/xres/wxbm_btn_help_topics',$
    Place four icon buttons on the Icons menu. Full pathnames to
    bitmap images are given to produce the icon pictures.
separator:1,$
    Insert a single-line separator.
icon:getenv('WAVE_DIR')+ $
    '/xres/wxbm_btn_help_quit'}}
```

Place the last icon below the line.

```
bar = WwMenuBar(layout, menus)
```

The call to `WwMenuBar` creates the menu bar. The 'layout' parameter is the widget ID of the parent layout widget. For example: `layout = WwLayout(top, /Horizontal, /Spacing=5)`. The value of the 'menus' parameter is the multi-level unnamed structure defined above.

```
status=WwSetValue(top, /Display)
WwLoop
```

Creating a Button Box and a Tool Box

A button box and a tool box serve similar functions — both trigger specific actions when the user clicks on an item. A button box contains an array of labeled buttons, organized in rows or columns. A tool box contains an array of icons (graphical buttons), also arranged in rows or columns.

Buttons and icons are usually used to apply changes, confirm decisions, display new windows, or start new applications. When the user clicks on a button or icon, its three-dimensional appearance inverts, so that it looks like the button has been depressed. When the button or icon is released, it returns to its normal appearance.

Button Box Example



Figure 15-13 A button box containing five buttons.

A button box is created with the `WwButtonBox` function. In the following example layout is the parent widget. `ButtonCB` is the name of the procedure that is executed when a button is selected. The buttons are arranged horizontally and spaced 20 pixels apart. `WwButtonBox` is passed two parameters — the button box widget ID and a value that corresponds to the selected button. The result is shown in Figure 15-13.

```
top=WwInit('ww_ex6', 'Examples', layout)

labels = ['Quit', 'Dialog', 'Message', $
         'FileSelection', 'Command']

bbox=WwButtonBox(layout, labels, 'ButtonCB', $
                /Horizontal, Spacing=20)

status=WwSetValue(top, /Display)
WwLoop
```

Tool Box Example

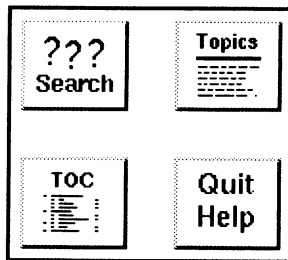


Figure 15-14 A toolbox containing four icons.

A tool box is created with the `WwToolBox` function. This example creates a tool box containing four icons, arranged in two columns, which are specified in the string array `pixmap`s. `DrawnCB` is the name of the callback routine (not shown) that is executed when a button is selected. The result is shown in Figure 15-14.

```
top=WwInit('ww_ex7', 'Examples', layout)

pixmap = [getenv('WAVE_DIR')+'/xres/wxbm_btn_help_search', $
          getenv('WAVE_DIR')+'/xres/wxbm_btn_help_toc', $
          getenv('WAVE_DIR')+'/xres/wxbm_btn_help_topics', $
          getenv('WAVE_DIR')+'/xres/wxbm_btn_help_quit']
          Create a variable "pixmap" that contains the paths to the graphics files used for the icons.

dbox=WwToolBox(layout, pixmap, 'DrawnCB', /Vertical, $
              Spacing=20, Measure=2)
              The tool box is created with four buttons, spaced 20 pixels apart, and arranged in two columns.

status=WwSetValue(top, /Display)

WwLoop
```

Creating a Radio Box

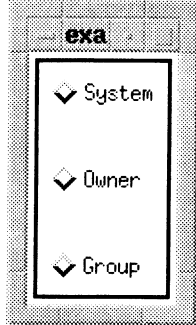


Figure 15-15 A radio box (Motif style).

A radio box contains a set of buttons, usually used to set or display the state of a variable, process, or action. Radio box buttons can be either exclusive or nonexclusive. If the buttons are exclusive, only one can be selected at a time (like the buttons on a car radio). If the buttons are set to nonexclusive, any number of them can be selected at once. When a button is selected, its appearance changes to reflect this. Radio box buttons can also be labeled.

Example

A radio box is created with the `WwRadioBox` function. In this example, the keyword `NoMany` specifies that the buttons are to be nonexclusive. The `Border` keyword specifies the thickness, in pixels, of the border around the buttons. The `Spacing` keyword specifies the space, in pixels, between buttons. `RadioCB` is the name of a callback routine that is executed when a button is selected. The result is shown in Figure 15-15.

```
top=WwInit('ww_ex8', 'Examples', layout)

labels=['System', 'Owner', 'Group']
Create three labels for the radio buttons.
```

```

rbox=WwRadioBox(layout, labels, 'RadioCB', $
    /Vertical, Border=2, Spacing=20, /Nofmany)
    Create the radio button box.

status=WwSetValue(top, /Display)
WwLoop

```

Creating a Controls Box with Sliders

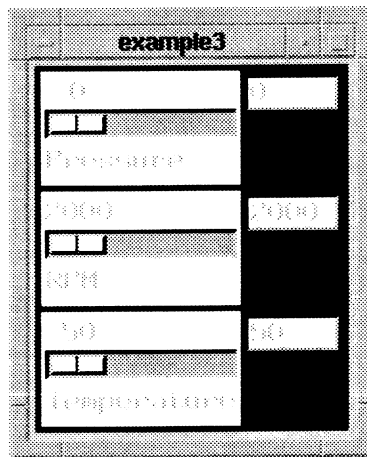


Figure 15-16 A controls box with three sliders (Motif style).

A controls box is a box containing “sliders”. A slider can be used to change numerical values interactively by positioning the pointer on the slider, pressing the left mouse button, and dragging. A controls box can have any number of sliders oriented either horizontally or vertically in a specified number of rows or columns. In addition, a slider can have an input text field in which the user can enter an exact value.

A controls box is created with the `WwControlsBox` function.

Example

In this example, a controls box containing three sliders, labeled Pressure, RPM, and Temperature, is created. They are arranged vertically. The keyword *Text* creates a text input field for each slider. Whenever a slider is moved, the callback routine named *SliderCB* is executed. The result is shown in Figure 15-16.

```
top=WwInit('ww_ex9', 'Examples', layout)

labels=[ 'Pressure', 'RPM', 'Temperature' ]
        Create the slider labels.

ranges=[ 0,100,2000,4000,50,150 ]
        Specify the ranges of each slider.

controls = WwControlsBox(layout, labels,$
        ranges, 'SliderCB', /Vertical, /Text, $
        Foreground='red', Background='yellow',$
        Basecolor='blue' )
        Create the controls box.

status=WwSetValue(top, /Display)

WwLoop
```

Creating a Drawing Area



Figure 15-17 A drawing area (Motif style).

A drawing area is a window in which an application can display plots or images generated by PV-WAVE. Both horizontal and vertical scroll bars are attached to the drawing area. The scroll bars can be used to pan across a drawing that is too large to fit in the window. A drawing area is created with the `WwDrawing` function.

Whenever a drawing area widget is created, `WwDrawing` automatically associates the drawing area with a PV-WAVE window index (see the `WINDOW` command in the *PV-WAVE Reference* for information on the window index). Then, the drawing area callback is executed and the graphics are displayed. The callback is also executed on systems that do not provide backing store for when the drawing window is obscured and then redisplayed.

Example 1: Basic Drawing Area

In this example, a drawing area window is created. The parent widget is called `layout`, the window index of the PV-WAVE window used is 1. Note that the size of the drawing area is 256 x 256, while the total size of the drawing is 512 x 512 (the image is larger than the actual drawing area). Scroll bars are provided for moving the image inside the drawing area. The result is shown in Figure 15-17.

To run this example, enter the callback procedure in a file and compile it with the `.RUN` command. Then enter the widget commands at the `WAVE>` prompt.

Callback Procedure

```
PRO DrawCB, wid, data
  COMMON draw, img
  PRINT, 'Draw'
  TV, img
END
```

Widget Commands

```
top=WwInit('ww_ex10', 'Examples', layout)
COMMON draw, img
LOADCT, 5, /Silent
img=BYTARR(512,512)
Openr,1,'$WAVE_DIR/data/head.img'
  Note: Under VMS, enter: WAVE_DIR:[DATA]head.img
READU,1,img
CLOSE, 1
draw=WwDrawing(layout, 1, 'DrawCB', $
  [256,256], [512,512])
status=WwSetValue(top, /Display)
WwLoop
```

Example 2: User Resizes the Drawing Area

This example demonstrates how the drawing area can be dynamically resized when the user resizes the main window.

To run this example, enter the code into a procedure file, and execute it with the `.RUN` command.

```

        Draw contents of the drawing area

PRO DrawCB, wid, index
    COMMON Trdraw, draw, area, widx
    print, 'Draw', !D.x_vsize, !D.y_vsize, $
        index, widx

    x = indgen(100)
    plot, x

END

PRO Testresize
    COMMON Trdraw, draw, area, widx
        Initialize toolkit, create form layout
    top = WwInit('testresize', 'Test', layout, $
        Background='Green', /Form)
    Create drawing area
    widx = -1
    draw = WwDrawing(layout, widx, 'DrawCB', $
        [256, 256], [512, 512], /Noscroll, $
        Area = area, /Right, /Left, /Top, $
        /Bottom)
    status = WwSetValue(top, /Display)
    WtLoop

END
```

Example 3: Resizing the Drawing Area Programatically

This example demonstrates how to resize the drawing area programatically using `WwSetValue`.

To run this example, enter the code into a procedure file, and execute it with the `.RUN` command.

```
PRO ButtonCB, wid, which
  COMMON Test, draw, top, size
  CASE which OF
    1: BEGIN
      size = 600
      status = WwSetValue(draw, [600, 600])
    END
    2: BEGIN
      erase, 0
      tvscl, dist(size)
    END
  ENDCASE
END
```

Draw the image.

```
PRO DrawingCB, wid, which
  COMMON Test, draw, top, size
  tvscl, dist(size)
END
```

```
PRO Testdraw
  COMMON Test, draw, top, size
  size = 400
  top = WwInit('testdraw', 'Testdraw', $
    layout, /Vertical, $
    Title = 'Test Drawing', $
    Position = position)
  Create Pushbutton.
```



```

b = ['Resize', 'Redraw']
pushb = WwButtonBox(layout, b, 'ButtonCB', $
    /Vertical, Border = 0, $
    Spacing = 10, Position = [10, 400])
    Create Drawing.

drawing_surf = WwDrawing(layout, 1, $
    'DrawingCB', [400, 400], [400, 400], $
    Area = draw, Position = [0, 0])

status = WwSetValue(top, /Display)

WwLoop
    Waiting for callbacks.

END

```

Creating a Text Widget

The WwText function can be used to create three kinds of text widgets:

- A single-line read-only label.
- A single-line editable text field, used for user input.
- A multi-line text window that can be read-only or editable.

Horizontal and vertical scroll bars allow the user to scroll through the multi-line text window.

Single-line Label (Read-only)

To create a single-line read-only text label, use the WwText function with the *Label* keyword. For an example, see *Static Label and Editable Text Field Example* below.

Single-line Editable Text Field

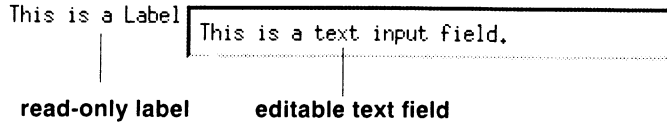


Figure 15-18 An editable text field with a label.

To create a single-line editable text field, use the `WwText` function. This text widget is used primarily for applications that require the user to enter a value or a string. `WwText` returns a string, which can be passed to the callback for processing.

Static Label and Editable Text Field Example

This example creates an editable text field with a label. The left edge of the text field widget is attached to the right edge of the label widget. The result is shown in Figure 15-18.

```
top=WwInit('ww_ex11', 'Examples', layout, $
    /Form)
    Initialize WAVE Widgets and create the form layout widget.
label=WwText(layout, /Label, $
    Text='This is Label')
    Create the label widget.
text=WwText(layout, 'TextCB', Cols=40, $
    left=label)
    Create the single-line text field widget, attaching it to the
    right edge of the label widget.
status=WwSetValue(top, /Display)
WwLoop
```

Multi-line Text Window

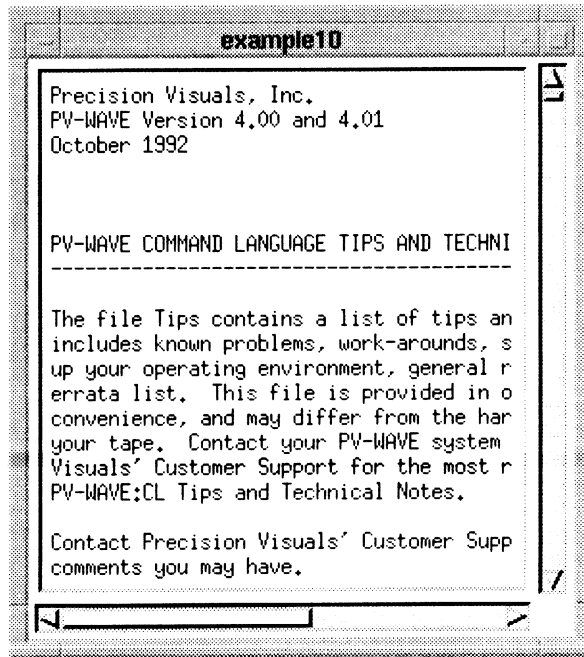


Figure 15-19 A multi-line text window (Motif style).

To create a multi-line text window, use the `WwText` function with the `Col` and `Rows` keywords to specify the height and width of the text area. If you use the `Read` keyword, the text area is read-only. If this keyword is not used, then the user can edit the text using the standard editing keys on the keyboard or the mouse (for example cut and paste).

Example

This example creates a multi-line, read-only text window 40 columns by 20 rows, and displays the text from the file `Tips`. The result is shown in Figure 15-19.

```
top=WwInit('ww_ex12', 'Examples', layout)
filename = getenv('WAVE_DIR')+ $
```

```
    '/Tips'  
    text=WwText(layout, 'TextCB', /Read, $  
        File=filename, Cols=40, Rows=20)  
    status=WwSetValue(top, /Display)  
WwLoop
```

Creating a Scrolling List

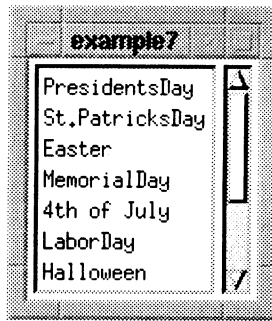


Figure 15-20 A scrolling list (Motif style).

A scrolling list allows the user to select one or more items from a list of items. The “items” in a scrolling list are text strings, defined in a string array. Scroll bars are provided for lists that are too long to display in the scrolling list window.

Use the `WwList` function to create a scrolling list.

Selection Mode

You can create a scrolling list in one of two selection modes: single or multiple selection.

Single Selection Mode

This is the default selection mode. Single selection means that the user can select one item at a time. To select an item, the user positions the pointer over the item and clicks the SELECT mouse button, usually the left button. If the user selects another item, the first item is deselected. This then executes the *selectCallback* routine.

If the user double-clicks on an item, the *defaultCallback* routine is executed.

Multiple Selection Mode

In multiple selection mode, the user can select more than one item from the list. The first item is selected with left mouse button. Additional items can then be selected using left mouse button. To deselect an item, the user clicks the left mouse button on it.

Scrolling List Callbacks

Two callbacks are used with the *WwList* function. The first one, *selectCallback*, is called whenever the user selects an item (single-clicks on it). The second callback, *defaultCallback*, is called when the user double-clicks on an item. This callback is called “default” because it usually executes a default action. For example, the Motif file selection widget operates in this manner. When the user clicks on a file name, the text is placed in a text input field. When the user double-clicks on a file name, the file is selected, the selection widget is dismissed, and the *defaultCallback* is called. For an illustration of the Motif file selection widget, see page 456.

Example

The following example creates a scrolling list containing the names of holidays. The *Visible* keyword specifies the number of items that are displayed in the scrolling list at a time. The *Multi* keyword sets the scrolling list to the multiple selection mode. The result is shown in Figure 15-20.

```
top=WwInit('ww_ex13', 'Examples', layout)

items = ['PresidentsDay', 'St.PatricksDay', $
        'Easter', 'MemorialDay', '4th of July', $
        'LaborDay', 'Halloween', 'Thanksgiving', $
        'Hanukkah', 'Christmas', 'New Years Eve']
        Define the list of items.

list=WwList(layout, items, 'ListCB', $
           'DefaultCB', Visible=7, /Multi, Left=rbox, $
           Top=controls)
           Create a scrolling list widget that displays the list of holi-
           days.

status=WwSetValue(top, /Display)
WwLoop
```

Creating Popup Messages

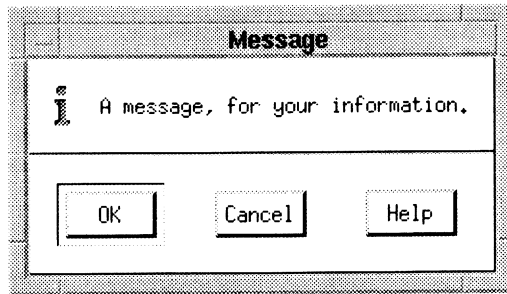


Figure 15-21 A pop-up message (Motif-style).

A popup message is a window that contains some text. Usually it informs the user of a condition — a warning or the confirmation of a choice, for instance — then is dismissed when the user clicks on the OK (Motif) or Confirm (OPEN LOOK) button. No other interaction is required.

The `WwMessage` function is used to create message windows.

Note 

Message windows are *popup widgets*. This means that they must have an intermediate widget, such as a button, as their parent. The popup widget appears after the user selects the intermediate button. See *Message Box Example* on page 451 for more information.

Blocking vs. Nonblocking Windows

A message window can be blocking or nonblocking. Blocking means that no other user action can occur until the message or dialog is confirmed — user clicks on OK (Motif) or Confirm (OPEN LOOK) — or the message box is dismissed. The blocking window “blocks” the user from performing other actions as long as the window remains on the screen.

A nonblocking message window can remain on the screen while the user performs other actions. The nonblocking dialog or message does not “block” the user from performing other actions.

The `WwMessage` function has a *Block* and a *Nonblock* keyword. Use the *Block* keyword to create a blocking window and the *Nonblock* keyword to create a nonblocking window.

Types of Message Windows (Motif only)

You can create four types of message boxes. These types are available only with the Motif message widget. They are shown in Figure 15-22.

- Information message — Specified with the *Info* keyword.
- Working message — Specified with the *Working* keyword.
- Warning message — Specified with the *Warning* keyword.
- Question message — Specified with the *Question* keyword.

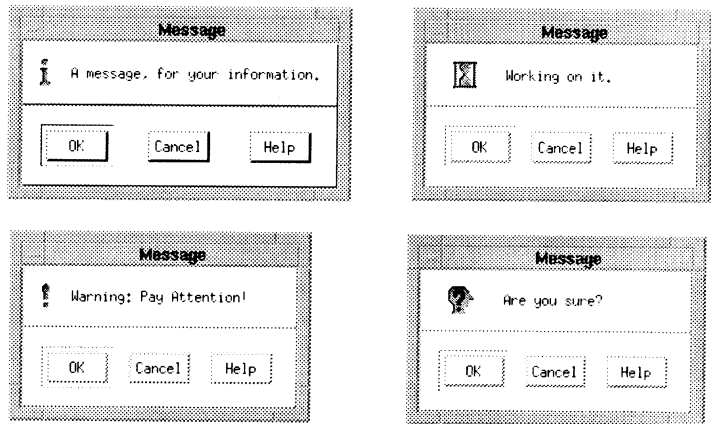


Figure 15-22 The four types of message windows (Motif only). Clockwise from upper-left: Information, Working, Question, and Warning.

Default Message Box Buttons

By default, two (for OPEN LOOK) or three (for Motif) buttons appear along the bottom edge of message windows. These buttons are used to confirm or cancel the message.

Motif Buttons

- OK — Confirms the message.
- Cancel — Cancels the message.
- Help — Not supported.

OPEN LOOK Buttons

- Confirm — Confirms the message.
- Cancel — Cancels the message.

Message Box Example

This example creates a row of buttons you can click on to display the four different types of Motif message boxes. (Under OPEN LOOK, the message boxes will look the same.) To run the example, enter the callback procedures in a file and compile them with the .RUN command. Then enter the widget commands at the WAVE> prompt. The result is shown in Figure 15-21.

Callback Routines

```
PRO MessageOK, wid, data
    print, 'Message OK'
END

PRO MessageCancel, wid, data
    print, 'Message Cancel'
END

PRO MbuttonCB, wid, data

case data of
1: message=WwMessage(wid, $
    'This is a Test Message', 'MessageOK', $
    'MessageCancel', TITLE='Information')
2: message=WwMessage(wid, $
    'This is a Test Message', 'MessageOK', $
```

```

        'MessageCancel', /Working, $
        TITLE='Working')
3: message=WwMessage(wid, $
    'This is a Test Message', 'MessageOK', $
    'MessageCancel', /Warning, $
    TITLE='Warning')
4: message=WwMessage(wid, $
    'This is a Test Message', 'MessageOK', $
    'MessageCancel', /Question, $
    TITLE='Question')
ENDCASE
END

```

Widget Commands

```

top=WwInit('ww_ex14', 'Examples', layout)
button=WwButtonBox(layout, ['Information', $
    'Working', 'Warning', 'Question'], $
    'MbuttonCB')

status=WwSetValue(top, /Display)
WwLoop

```

Creating Dialog Boxes

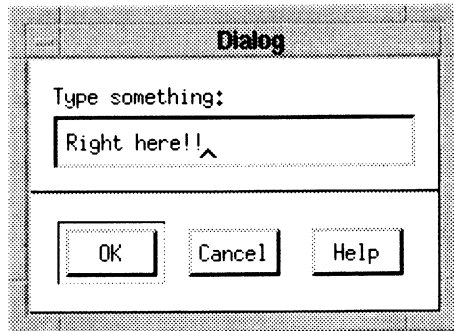


Figure 15-23 A dialog box (Motif-style).

A dialog box requires user interaction. For instance, the users may enter some text in the dialog box, then “accept” the entry by clicking on a button. If the users do not wish to apply the change to the dialog box, they can click on another button to dismiss the dialog box.

The `WwDialog` function creates dialog boxes.

A Dialog is a Popup Widget

Dialog boxes are *popup widgets*. This means that they must have an intermediate widget, such as a button, as their parent. The popup widget appears after the user selects this intermediate button. See *Dialog Box Example* on page 454 for more information.

Blocking vs. Nonblocking Windows

A dialog window can be blocking or nonblocking. Blocking means that no other user action can occur until the dialog is confirmed — the user clicks on OK (Motif) or Confirm (OPEN LOOK) — or until the dialog box is dismissed. The blocking window “blocks” the user from performing other actions as long as the window remains on the screen.

A nonblocking dialog can remain on the screen while the user performs other actions. The nonblocking dialog does not “block” the user from performing other actions.

The `WwDialog` function has a *Block* and a *Nonblock* keyword. Use the *Block* keyword to create a blocking window and the *Nonblock* keyword to create a nonblocking window.

Default Dialog Box Buttons

By default, two (for OPEN LOOK) or three (for Motif) buttons appear along the bottom edge of dialog boxes. These buttons are used to confirm or cancel the dialog box.

Motif Buttons

OK — Confirms the input in the dialog.

Cancel — Cancels the dialog.

Help — Not supported.

OPEN LOOK Buttons

Apply — Accepts the input in the dialog.

Cancel — Cancels the dialog.

Dialog Box Example

The following example creates a button that you can click on to display a dialog box. To run the example, enter the callback procedures in a file and compile them with the `.RUN` command. Then enter the widget commands at the `WAVE>` prompt.

The “Type something” string is a label for the text input field. The `DialogOK` parameter is the name of a callback that is executed when the user clicks on the OK button. The `DialogCancel` parameter is the name of a callback that is executed when the Cancel button is selected. The result is shown in Figure 15-23.

Callback Procedures

```
PRO DbuttonCB, wid, data
    select=WwDialog(wid,'Type something:',$
        'DialogOK', 'DialogCancel', $
        Title='Type')
END

PRO DialogOK, wid, text
    PRINT,'Dialog OK'
    value = WwGetValue(text)
    PRINT, value
END

PRO DialogCancel, wid, data
    PRINT,'Dialog Cancel'
END
```

Widget Commands

```
top=WwInit('ww_ex15', 'Examples', layout)
button=WwButtonBox(layout, 'Dialog Box', $
    'DbuttonCB')

status=WwSetValue(top, /Display)
WwLoop
```

Creating a File Selection Widget

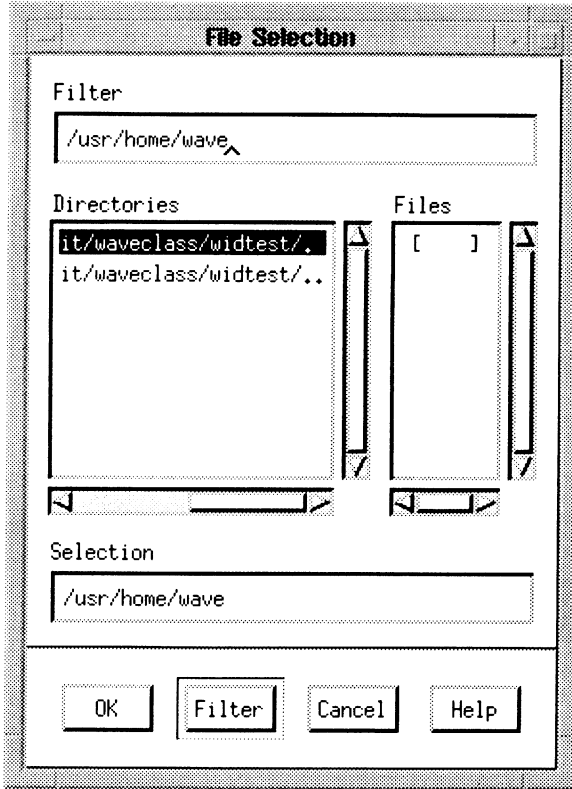


Figure 15-24 File selection widget (Motif style).

A file selection widget lets the user move through directories and select files. File selection widgets are created with the `WwFileSelection` function.

A File Selection Widget is a Popup Widget

File selection widgets are *popup widgets*. This means that they must have an intermediate widget, such as a button, as their parent. The popup widget appears after the user selects this intermediate button. See *File Selection Example* on page 458 for more information.

Blocking vs. Nonblocking Windows

A file selection window can be blocking or nonblocking. Blocking means that no other user action can occur until the message or dialog is confirmed (user clicks on OK, Apply, Cancel, or some other confirming button). The blocking window “blocks” the user from performing other actions as long as the window remains on the screen.

A nonblocking file selection window can remain on the screen while the user performs other actions. The nonblocking window does not “block” the user from performing other actions.

The `WwFileSelection` function has a *Block* and a *Nonblock* keyword. Use the *Block* keyword to create a blocking window and the *Nonblock* keyword to create a nonblocking window.

File Tool Contents

The file tool lets the user move through directories, view their contents, and select files. It consists of:

- A text field where the user enters a directory name to display subdirectories and files.
- A list of directories and files.
- A text input field for displaying or editing a filename.
- The buttons: OK, Filter, Cancel, and Help. (The Help button does not appear if you are running OPEN LOOK.)

See Figure 15-24 for an example Motif file selection box.

File Selection Example

The following example creates a button that you can click on to display a file selection widget. To run the example, enter the callback procedures in a file and compile them with the `.RUN` command. Then enter the widget commands at the `WAVE>` prompt.

`FileOK` and `FbuttonCB` are names of callback routines. The `Title` keyword specifies a name for the search tool. The result is shown in Figure 15-24.

Callback Procedures

```
PRO FbuttonCB, wid, data
    file = WwFileSelection(wid, 'FileOK', $
        'FileCancel', Title='Search')
END

PRO FileOK, wid, shell
    value = WwGetValue(wid)
    PRINT, value
    status = WwSetValue(shell, /Close)
END
```

Widget Commands

```
top=WwInit('ww_ex16', 'Examples', layout)
button=WwButtonBox(layout, 'File Tool', $
    'FbuttonCB')

status=WwSetValue(top, /Display)
WwLoop
```

Creating a Command Widget

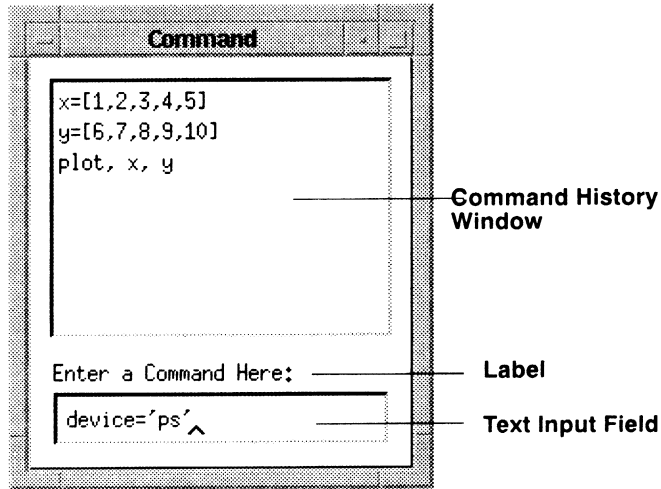


Figure 15-25 A command widget (Motif style).

A command widget is used for command entry and provides a built-in command history mechanism. The command widget includes a text input field, a label for the text input field, and a command history window.

The `WwCommand` function is used to create a command widget. Use `WwGetValue` in the callback routine to obtain the text strings entered from the command widget.

The user types a command in the text input field and presses <Return> to execute the command. The command is then added to the end of the command history window. When the user clicks on a command in the command history window, the command is displayed in the text entry field, ready to be executed. The user can double-click on a command in the command history list to execute it directly. This also adds the command to the end of the list.

Note

Command widgets are *popup widgets*. This means that they must have an intermediate widget, such as a button, as their parent. The

popup widget appears after the user selects this intermediate button. See the following *Example* section for more information.

Example

The following example shows a simple WwCommand call. To run the example, enter the callback procedures in a file and compile them with the .RUN command. Then enter the widget commands at the WAVE> prompt.

CommandOK is a callback that is executed when the user enters the command and presses <Return>, or double-clicks the command from the history list. CommandDone is a callback that is executed when the user quits the command window. The *Title* keyword specifies a name for the command window, and *Position* specifies its location on the screen. The result is shown in Figure 15-25.

Callback Procedures

```
PRO CbuttonCB, wid, data
    command = WwCommand(wid, 'CommandOK', $
        'CommandDone', Title= $
        'Command Entry Window', $
        Position=[300,300])
END

PRO CommandOK, wid, shell
    value = WwGetValue(wid)
    Obtain the string entered in the text input field.
    PRINT, value
END

PRO CommandDone, wid, shell
    status = WwSetValue(shell, /Close)
END
```

Widget Commands

```
top=WwInit('ww_ex17', 'Examples', layout)
button=WwButtonBox(layout, 'Command', $
    'CbuttonCB')
status=WwSetValue(top, /Display)
WwLoop
```

Creating a Table Widget

	DATE	TIME	DUR	INIT
0	901002	93200	21.4000	TAC
1	901002	94700	1.05000	BMD
2	901002	94700	17.4400	EDH
3	901002	94800	16.2300	TIN
4	901002	94800	1.31000	RLD
5	901003	91500	2.53000	DLH
6	901003	91600	2.33000	JHT
7	901003	91600	0.350000	CCW
8	901003	91600	1.53000	SRB
9	901003	91600	0.450000	HLI

Figure 15-26 A table widget.

A table widget is used for displaying and editing a 2D array of cells. The example table shown in Figure 15-26 contains scroll bars which can be used to display cells that are currently hidden.

The `WwTable` function is used to create a table widget. A variety of selection and editing methods are available within a table created with `WwTable`. For detailed information on the many keywords and options of `WwTable` and the code used to produce Figure 15-26, see *WwTable Function* on page 472 of the *PV-WAVE Reference, Volume II*.

Setting Colors and Fonts

Colors and fonts are common attributes of all WAVE Widgets. Keywords are provided for setting these attributes.

Setting Colors

Many WAVE Widgets provide keywords for setting colors. The keywords include: *Basecolor*, *Background*, and *Foreground*.

Basecolor — The color of the “container” or “box” for the following widgets: *ButtonBox*, *RadioBox*, *ToolBox*, and *ControlsBox*.

Background — The color of a button.

Foreground — The color of the text on a button.

Figure 15-27 illustrates the use of each color keyword.

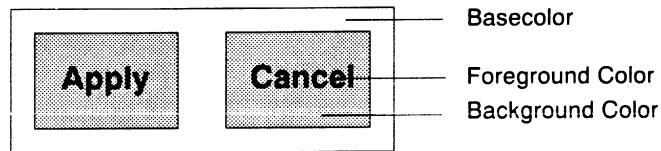


Figure 15-27 How the color keywords are applied.

The color keywords have the form:

Keyword = '*colorname*'

where *colorname* is the name of an X library color. For example:

Background = 'Skyblue'

The names of colors can be found in:

- On most UNIX systems: /usr/lib/X11/rgb.txt
- On most VMS systems: SYS\$MANAGER:DECW\$RGB.COM

If you cannot find these color files on your system, see your System Administrator.

Setting Fonts

The *Font* keyword is used to set the font used to create text in a widget. This keyword has the form:

```
Font = 'fontname'
```

where *fontname* is the name of a font available on your system. For example:

```
Font = '-b&h-lucida-bold-r-normal-sans-14-*
```



Use the command `xlsfonts` to obtain a list of fonts available on your X server.

If you inadvertently specify a font that cannot be found by the X server, a different font will be substituted.

Using A Resource File to Set Colors and Fonts

Characteristics such as foreground color, background color, and font are known as *resources*, and these characteristics can be initialized using a *resource file*. WAVE Widgets provides the ability to specify these resources with the keywords *Foreground*, *Background*, *Basecolor*, and *Font*, as discussed in the previous two sections.

When the Resource File is Checked

If widget resources are not specified with the keywords *Foreground*, *Background*, *Basecolor*, and *Font*, PV-WAVE will query the X resource database and use the values defined there.



Using a resource file for WAVE Widgets is optional.

Adding resources to the resource database is a two step process.

- The first step is to create a resource file. The following is a sample resource file for setting application-wide resources for foreground color, background color, and font of a WAVE Widgets application called `my_gui`:

```
my_gui*foreground:Black
my_gui*background:Cyan
my_gui*font:         fixed
```

The application name `my_gui` is defined in the `WwInit` command. For example:

```
top = WwInit('my_gui', 'Examples', $
            layout, /Vertical)
```

You can also put the resource names in your `.Xdefaults` file, if you only need to customize your own version of the software. Either way, the new values of the resources will not take effect until the next time `PV-WAVE` is started.

- The second step is to install the resource file into the X resource database. The simplest way to install resources is to merge your resource file with X resource database using the following command:

```
xrdb -merge resource_filename
```

where *resource_filename* is the name of your resource file.



It is also possible to set the resources for particular widgets or groups of widgets in your application. To do this, you will have to know the names of all or some of the widgets in the widget hierarchy. (A widget's name is determined by its placement in the hierarchy.) The more specific the widget hierarchy you provide, the fewer the number of widgets that are affected by the change.

For more information about how widgets are related to one another, see *The Widget Hierarchy* on page 416.

To find out widget names of particular `WAVE Widgets`, refer to the appropriate `WAVE Widgets .pro` files located in the standard library directories (`$WAVE_DIR/lib/std/olit` and `$WAVE_DIR/lib/std/motif`).

For more information about how to write and install X Window System resource files, refer to Volume 4 of the *X Toolkit Intrinsics Programming Manual*, O'Reilly & Associates, Inc., Sebastopol, CA, 1990.

Setting and Getting Widget Values

Most of the widgets have values associated with them that are set when the widget is created.

After a widget is created, these values can be changed or obtained using the `WwGetValue` or `WwSetValue` routines.

The values that are set or obtained with `WwSetValue` and `WwGetValue` differ from one WAVE Widgets routine to another. These values are listed in the routine descriptions in Chapter 4.

For example, when `WwSetValue` is passed a widget ID from a widget created with `WwList`, the value that is set is an array of strings to replace the current items in the list. If `WwGetValue` is passed the widget ID from a widget created with `WwList`, it returns a string array containing the selected items in the scrolling list.

The following callback uses `WwGetValue` to obtain the list of selected items from a scrolling list. Then, `WwSetValue` is used to replace the selected items with new strings. To run this example, enter the callback routine in a file and compile it with the `.RUN` command. Then enter the widget commands at the `WAVE>` prompt.

Callback Procedure

```
PRO ListCB, wid, data
  print, 'Item Selected'
  value = WwGetValue(wid)
    Obtain the values of the selected items in the scrolling list.
  print, value

  print, 'Setting..'
  status=WwSetValue(wid, ['First', 'Second', $
    'Third'])
    Set the value of the scrolling list widget whose ID is wid to
    the strings First, Second, and Third.
END
```


Widget Commands

```
top=WwInit('ww_ex18', 'Examples', layout)

items = ['Presidents Day', 'St.Patricks Day', $
'Easter', 'Memorial Day', '4th of July', $
'Labor Day', 'Halloween', 'Thanksgiving', $
'Hanukkah', 'Christmas', 'New Years Eve']
    Define the list of items.

list=WwList(layout, items, 'ListCB', $
'DefaultCB', Visible=7, /Multi, Left=rbox, $
Top=controls)
    Create a scrolling list widget that displays the list of holi-
days.

status=WwSetValue(top, /Display)
WwLoop
```

Passing and Retrieving User Data

All WAVE Widgets can carry the user-specified value of a PV-WAVE variable. This allows the developer to store the copy of the variable with the widget in one PV-WAVE routine and retrieve it in another routine. Any value can be stored and retrieved; it is up to the discretion of the programmer.

This feature is useful for passing values between routines without using Common Block variables.

To store the value 111 with a widget, use the following command:

```
status = WwSetValue(widgetID, Userdata=111)
```

To retrieve the value of Userdata from the widget, use the command:

```
value = WwGetValue(widgetID, /Userdata)
```

Example

The following example shows a practical use for passing a value with the *Userdata* keyword to close an application when the user clicks on a specified button. The value of the top-level widget, *top*, is passed from the widget creation procedure *myap* to the callback procedure *ButtonCB* via the *Userdata* keyword. This value is then used in the *WwSetValue* function to close the application by destroying the top-level widget when the user clicks on *Quit*. You can run this example by typing the callback procedures into a file and compiling them with the *.RUN* command. Then enter the application procedure in a file and run it.

Callback Procedure

```
PRO ButtonCB, wid, data
CASE data OF
  1: BEGIN
      top=WwGetValue(wid, /Userdata)
      Get the value of the top-level widget.
      PRINT, top
      status=WwSetValue(top, /Close)
      END
  2: PRINT, 'Dialog Selected'
  3: PRINT, 'Message Selected'
ENDCASE
END

PRO RadioCB, wid, which
CASE which OF
  1: PRINT, 'First Toggle Selected'
  2: PRINT, 'Second Toggle Selected'
  3: PRINT, 'Third Toggle Selected'
ENDCASE
value = WwGetValue(wid)
print, value
END
```

Application Procedure

```
PRO myap
top=WwInit('ww_ex19', 'Examples', layout,$
  /Vertical, Spacing=30, Border=10)
blabels = ['Quit','Dialog','Message']

bbox=WwButtonBox(layout, blabels, $
  'ButtonCB', /Horizontal, Spacing=20)

status=WwSetValue(bbox, Userdata=top)
  Store the value of the top-level widget with the Userdata key-
  word.

rlabels=['System','Owner','Group']

rbox=WwRadioBox(layout,rlabels, 'RadioCB', $
  /Vertical, Border=2, Spacing=20, $
  Top=controls)

status=WwSetValue(top, /Display)
WwLoop
END
```

Managing Widgets

Besides colors, fonts, and userdata values, two additional widget attributes can be managed by the developer: widget visibility and sensitivity.

Showing/Hiding Widgets

The *Show* and *Hide* keywords to the `WwSetValue` function control whether a widget is visible or not. By default, all widgets are shown when they are created.

To hide a widget, use the *Hide* keyword:

```
status = WwSetValue(widget, /Hide)
```

To show a hidden widget, use the *Show* keyword:

```
status = WwSetValue(widget, /Show)
```

In these functions, the *widget* parameter is the widget ID of the widget you want to show or hide.

You can also test to determine if a widget is shown or hidden, using the *Shown* keyword in the *WwGetValue* function.

```
shown = WwGetValue(widget, /Shown)
```

If the widget is shown, the function returns 1; if hidden, 0 is returned.

Widget Sensitivity

If a widget is sensitive, some action will occur when the user selects the widget. For example, if the user clicks on a button that is sensitive, an action occurs. By default, all widgets are sensitive when they are created.

You can set a widget to be nonsensitive using the *Nonsensitive* keyword in the *WwSetValue* function. When a widget is set to nonsensitive, its foreground color is grayed-out, and the widget cannot accept input from the user.

```
status = WwSetValue(widget, /Nonsensitive)
```

To change a widget from nonsensitive to sensitive, use the */Sensitive* keyword:

```
status = WwSetValue(widget, /Sensitive)
```

To determine if a widget is sensitive, use the *Sensitive* keyword in the *WwGetValue* function.

```
shown = WwGetValue(widget, /Sensitive)
```

If the widget is sensitive, the function returns 1; if nonsensitive, 0 is returned.

Displaying Widgets and Processing Events

After all of the widgets in a widget hierarchy have been created, they are displayed when the top-level or “root” window is displayed. The following command accomplishes this:

```
status = WwSetValue(top, /Display)
```

For more information on the widget hierarchy, see *The Widget Hierarchy* on page 416.

Next, control must be transferred to the main event loop, which handles events (mouse clicks, for example) and executes callbacks. To do this, simply call the `WwLoop` function:

```
WwLoop
```

The application remains in the main loop until the top-level window is closed with the following call:

```
status = WwSetValue(top, /Close)
```

or until the top-level window is closed from the window manager menu. Under Motif, the window manager menu usually contains a **Close** button, and under OPEN LOOK there is usually a **Quit** button.

While the loop is running, any callback procedures that have been defined are executed whenever the appropriate events occur.

The use of `WwSetValue` to display and close widgets is demonstrated in examples throughout this chapter. See for example *File Selection Example* on page 458.

Programming Tips and Cautions

PV-WAVE CL Routines to Avoid

Avoid using the following PV-WAVE routines in applications developed with WAVE Widgets or the Widget Toolbox. These routines wait for keyboard input and thus block the GUI. Where possible, alternative methods are suggested.

Standard Library Routines

- GET_KBRD – Try using a text field widget instead.
- HAK – Try using a non-blocking message widget instead.
- MOVIE – Try using WgMovieTool instead.

User Library Routines

- ANMENU – Try using WAVE Widgets menus instead.
- UNCMPRS_IMAGES
- XANIMATE – Try using WgAnimateTool instead.

PV-WAVE CL Routines to Use with Caution

Use the following routines with caution in applications developed with WAVE Widgets or the Widget Toolbox. All of these routines block the user from interacting with the GUI.

Standard Library Routines

- ADJCT – Try using WgCtTool or WgCbarTool instead.
- C_EDIT – Try using WgCeditTool instead.
- COLOR_EDIT – Try using WgCeditTool instead.
- CURSOR – Try using the Widget Toolbox event handler instead.
- DEFROI

- PALETTE — Try using WgCeditTool instead.
- PROFILES
- RDPIX
- TVMENU — Try using a menu bar widget instead.
- WAIT — Try using the WtTimer function instead.
- WMENU — Try using a menu bar widget instead.
- ZOOM

Application Example

The following example program uses WAVE Widgets to create an image processing application that includes a drawing area for displaying the image and a menu containing image processing functions. Figure 15-28 on page 478 shows the main window of the application with an image displayed in the drawing area.

You can find this program file in the following location:

Under UNIX

```
$WAVE_DIR/demo/wavewidgets/simple_image.pro
```

Under VMS

```
WAVE_DIR:[DEMO.WAVEWIDGETS]SIMPLE_IMAGE.PRO
```

```

;
; Example of WAVE Widgets. It displays an image, and allows you to
; change color table, and do some basic Image Processing.
;

PRO FileOK, wid, shell      ; File Selection is done, let's load it
common widgets, top, slider
common images, orig, image, draw, size
common rotate, sliderval

file = WwGetValue(wid)      ; Get the file name
file = FINDFILE(file)      ; Find the file

if N_ELEMENTS(file) lt 1 then begin ; File not found display warning
message=WwMessage(wid,'File Not Found!','/WARNING,TITLE='File Error')
endif else begin ; Got a file lets load it
OPENR, /GET_LUN, unit, file(0)
status=FSTAT(unit) ; Get the size and hope it is square image
size=LONG(SQRT(status.SIZE)) ; Calculate height,width
image=MAKE_ARRAY(size,size,/BYTE); make new image array
READU,unit,image ; Read image, close the unit
FREE_LUN, unit
orig=image ; Store original for reload
sliderval = 0 ; Reste slider value
status=WwSetValue(slidebar,0) ; Reset rotate slider
status=WwSetValue(draw,[size,size]) ; Set new draw area value
DrawCB, draw, 1 ; Redraw the image
endelse

status = WwSetValue(shell,/CLOSE) ; Close the file selection window
END

PRO FileCancel, wid, shell ; File selection canceled
status = WwSetValue(shell,/CLOSE) ; Close the file selection window
END

PRO FileCB, wid, which ; File handling
common widgets, top, slider
common images, orig, image, draw, size

case which of
1: begin ; Reload the image
image=orig
status=WwSetValue(slidebar,0) ; Reset rotate slider

```



```

        DrawCB,draw,1
        end

2: begin          ; Display file selection window
    if !version.platform eq 'vms' then $
        dir = getenv('WAVE_DIR')+'[data]' $
    else $
        dir = getenv('WAVE_DIR')+'/data/'
        file = WwFileSelection(wid,'FileOK','FileCancel',$
            POSITION=[200,200],$
            TITLE='Load Image',DIR=dir,PATTERN='*.img')
    end

3: status=WwSetValue(top,/Close)    ; Close the application,Bye,bye.
endcase
END

PRO ImageCB, wid, which; Modifying image
common images, orig, image, draw, size
case which of
1: image=ERODE(image,[[0,1,0],[1,1,1],[0,1,0]],/Gray) ; Erode
2: image=DILATE(image,[[0,1,0],[1,1,1],[0,1,0]],/Gray) ; Dilate
3: image=SHIFT(ALOG(ABS(FFT(image,-1))),size/2) ; FFT
4: image=HIST_EQUAL(image) ; Histogram
5: image=ROBERTS(image) ; Roberts
6: image=SMOOTH(image,5) ; Smooth
7: image=SOBEL(image) ; Sobel
endcase
DrawCB, draw, 1 ; Redraw the image
END

PRO ColorCB, wid, which ; Loading new colortable
LOADCT,which-1,/SILENT
END

PRO DrawCB, wid, windex; Drawing the image
common images, orig, image, draw, size
TVSCL, image
END

PRO SliderCB, wid, which ; Lets rotate the image
common images, orig, image, draw, size
common rotate, sliderval

```

```

value=WwGetValue(wid)
if sliderval ne value then begin
    image=ROT(image,value)
    DrawCB, draw, 1          ; Redraw the image
    sliderval = value
endif
END

PRO simple_image

common widgets, top, slider
common images, orig, image, draw, size
common rotate, sliderval
image = Bytarr(512,512)
size=512
sliderval=0

; Loading first image
if !version.platform eq 'vms' then $
    filename=getenv('WAVE_DIR')+'[data]head.img' $
else $
    filename='$WAVE_DIR/data/head.img'

test=FINDFILE(filename)
if N_ELEMENTS(test) lt 1 then begin
    if test(0) eq '' then begin
        message,'Data subdirectory not available'
        EXIT
    endif
endif

    Openr,1,filename
    readu,1,image
    close, 1
    orig=image

top=WwInit('simple_image','Examples',layout,BACKGR='SkyBlue',$
    POSITION=[100,100], /VERTICAL, SPACING=5)

; Main menu bar for File, Image Processing, Color Tables
menus=,{,$
    menubutton:'File',$
    menu:,{,callback:'FileCB',title:'File',$

```

```

        button:'Reload Image',$
        button:'Load New Image',$
        button:'Exit'},$
menubutton:'Image Processing',$
    menu:{,callback:'ImageCB',title:'Image',$
        button:'Erode',$
        button:'Dilate',$
        button:'FFT',$
        button:'Histogram',$
        button:'Roberts',$
        button:'Smooth',$
        button:'Sobel'},$
menubutton:'Color Tables',$
    menu:{,callback:'ColorCB',title:'Color',$
        button:'Black/White Linear',$
        button:'Blue/White',$
        button:'Green/Red/Blue/White',$
        button:'Red Temperature',$
        button:'Blue/Green/Red/Yellow',$
        button:'Standard Gamma-II',$
        button:'Prism',$
        button:'Red/Purple',$
        button:'Green/White Linear',$
        button:'Green/White Exponential',$
        button:'Green/Pink',$
        button:'Blue/Red',$
        button:'16 Level',$
        button:'16 Level II',$
        button:'Steps',$
        button:'PV WAVE'}$
    }
bar=WwMenuBar(layout, menus); Let's create menu bar

draw=WwDrawing(layout,1,'DrawCB',[400,400],[512,512]) ;Creating Draw Area

; Creating Slider for Rotation
slider = WwControlsBox(layout,'Rotate',[0,360],'SliderCB',/VERTICAL,$
/TEXT, FOREGROUND='red',BACKGROUND='yellow',WIDTH=300)

status=WwSetValue(top,/DISPLAY); Displaying widget hierarchy

WwLoop                ; Waiting for callbacks

END

```

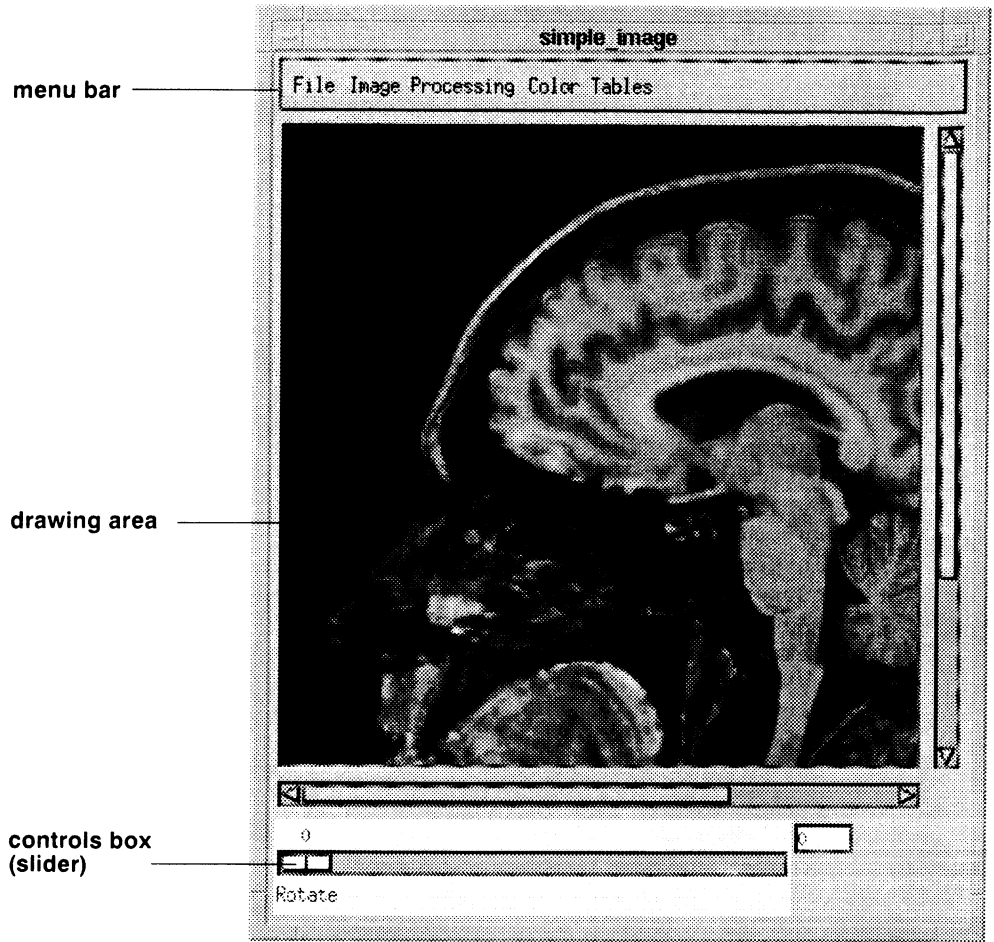


Figure 15-28 The main window of the image processing application `simple_image.pro` (Motif style).

Using the Widget Toolbox

Introduction to the Widget Toolbox

The Widget Toolbox is an application programmer's interface (API) that is built on top of the OLIT and OSF/Motif GUI toolkits. The Widget Toolbox provides a high-level method of creating and manipulating the GUI, while using the flexibility and power of PV-WAVE to process and display data. The Widget Toolbox consists of PV-WAVE system routines that give you access to all the widget types supported by the Motif and OLIT toolkits.

This chapter describes the Widget Toolbox and the basic steps in incorporating widgets into PV-WAVE applications. The topics include:

- An overview of the Widget Toolbox
- How to use the Widget Toolbox
- A brief description of include files that are used with the Widget Toolbox
- An example PV-WAVE application using the Widget Toolbox

Note 

The Widget Toolbox routines are primarily for PV-WAVE developers who are familiar with Xt Intrinsics and either the Motif or

OLIT widget toolkit. If you are not familiar with Motif or OLIT programming, then use the WAVE Widgets functions described in Chapter 15, *Using WAVE Widgets*.

Basic Steps in Creating the GUI

The basic steps involved in creating an application GUI with the Widget Toolbox are:

- Initialize the Widget Toolbox with a call to the `WtInit` function.
- Create widgets with the `WtCreate` function, and set resources that control the appearance and other characteristics of the widgets.
- Manage, display, and destroy widgets with the `WtSet` function.
- Add callbacks, event handlers, and timers.
- Run the application.
- Close the Widget Toolbox.

Combining WAVE Widgets and Widget Toolbox Functions

It is possible to combine WAVE Widgets and Widget Toolbox functions in the same application.

Widget IDs returned by WAVE Widget routines can be used in Widget Toolbox routines, and Widget Toolbox widget IDs can be passed to WAVE Widget routines.

The basic steps for creating an application that combines the two kinds of widget functions do not change. You must initialize either WAVE Widgets or the Widget Toolbox (`WwInit` or `WtInit` function), create the widgets, display or “realize” the widgets, and execute the main loop with either the `WwLoop` or `WtLoop` function. For information on WAVE Widgets, Chapter 15, *Using WAVE Widgets*.

Specifying the Desired Toolkit

If you are running your application on a Sun workstation, you can use the environment variable `WAVE_GUI` to control whether the widgets are implemented using the Motif or the OPEN LOOK look-and-feel. To maximize user satisfaction, the look-and-feel you select should match the one used by other applications your users are accustomed to, and should be compatible with the window manager you expect your users to be using when they run your PV-WAVE application. On all other platforms that support PV-WAVE, Motif is the only option, and thus setting `WAVE_GUI` on those platforms has no effect.



For detailed information on setting `WAVE_GUI`, see *WAVE_GUI: Selecting the GUI on Sun Workstations* on page 51 of the *PV-WAVE User's Guide*.

Initializing the Widget Toolbox

The PV-WAVE system function `WtInit` initializes the Widget Toolbox. You must execute `WwInit` before any other Widget Toolbox functions. `WtInit` does the following:

- Establishes a connection to the X server.
- Initializes Xt Intrinsics.
- Initializes the Motif or OLIT Toolkit.
- Initializes the Widget Toolbox.

For example:

```
top = WtInit(name,class)
```

The *name* parameter specifies the name of the application, and the *class* parameter indicates a more general category of application class. For example, in the example program at the end of this chapter, the call to `WtInit` is:

```
war(0) = WtInit('example', 'Examples')
```

where `example` is the name of the application, and `Examples` specifies a general class to which `example` belongs. One purpose of the general class (`Examples`) is that resources in a single resource file can be shared among elements of that class.

See also the example Widget Toolbox application at the end of this chapter.

Creating Widgets

Rather than dealing directly with X library windows, applications using Xt Intrinsic-based toolkits (Motif and OLIT) use widgets. A widget is a complex data structure containing interface-related data and set of procedures that perform actions on that data.

Each widget is represented externally by a widget ID. Widgets form hierarchies known as widget trees. The root of every widget hierarchy is a special type of widget called a shell. The shell widget provides an interface between the child widget and the window manager.

The `WtCreate` function provides the general mechanism for creating all PV-WAVE widgets. For example:

```
result = WtCreate(name, class, parent, args)
```

The *name* parameter is a string that identifies the widget. The *class* parameter is a widget class ID that specifies the type of widget to be created. For example, `xmFormWidgetClass` is a Motif widget class and `checkBoxWidgetClass` is an OLIT widget class. Widget class IDs are long values defined in the PV-WAVE Standard Library files `wtxmclass.pro` and `wtolclass.pro`. The widget class IDs for Motif are also listed in Appendix B, *Motif and OLIT Widget Classes*.

The *parent* parameter must be a widget ID of a widget that already exists. This can be a shell or any other type of widget that can have child widgets. The *args* parameter specifies values for resources used by the widget. See the next section for information on setting resources.

See also the example Widget Toolbox application at the end of this chapter.

Setting and Getting Resources

You can change the way a widget appears or behaves by specifying values for resources used by the widget.

Resources are specified using a PV-WAVE unnamed structure with each tag name representing the *resource name* and each tag definition representing the *resource value*.

An unnamed structure has the following general definition:

$$x = \{, tag_name_1: tag_def_1, tag_name_n: tag_def_n\}$$

For detailed information on unnamed structures, see *Creating Unnamed Structures* on page 105.

A resource name is a string indicating the type of resource. The resource name of a Widget Toolbox widget is derived directly from Motif or OLIT resource names. These resource names are listed in the *OSF/Motif Programmer's Guide* and *OLIT Widget Set Programmer's and Reference Manuals*. If you have a Motif toolkit, remove the XmN prefix from the Motif resource names. If you have an OLIT toolkit, remove the XtN prefix from the OLIT resource names. For example:

Motif Resource	OLIT Resource	Widget Toolbox Resource
XmNx	XtNx	x
XmNy	XtNy	y
XmNlabel	XtNlabel	label
XmNforeground	XtNforeground	foreground

The data type of a resource's value depends on the type of the resource.

Example

```
args={,x: 30, y: 50, label:'Done', $  
      foreground: 'red'}
```

Create an unnamed structure containing resource names and values.

```
wid=wtcreate('button', $  
            xmPushButtonWidgetClass, parent, args)
```

Create a pushbutton widget and use the args structure to specify its resources.

See also the example at the end of this chapter.

Managing, Displaying, and Destroying Widgets

Except for top-level shell widgets, all widgets must be “managed” by a parent widget. A widget’s parent manages the widget’s size and location, determines whether or not the widget is mapped (associated with an X window), and also controls the input focus of the widget.

By default all PV-WAVE widgets are managed when created. To unmanage a widget after creation use the `WtSet` function with the *Unmanage* keyword. For example:

```
status=WtSet(wid, /Unmanage)
```

To display a widget hierarchy, “realize” the shell widget of a hierarchy using the `WtSet` function with the *Realize* keyword. For example:

```
status=WtSet(shellid, /Realize)
```

To undisplay an individual widget, “unmanage” it using `WtSet` with the *Unmanage* keyword. For example:

```
status=WtSet(wid, /Unmanage)
```

To undisplay a whole widget hierarchy, unmanage the shell widget using `WtSet` with the *Unmanage* keyword. For example:

```
status=WtSet(shellid, /Unmanage)
```

To destroy a widget use `WtSet` with the *Destroy* keyword. For example:

```
status=WtSet(wid, /Destroy)
```

To destroy and close a whole widget hierarchy, use the `WtClose` function. For example:

```
status=WtClose(shellid)
```

For more information on `WtClose` see the *PV-WAVE Reference*.

See also the example Widget Toolbox application at the end of this chapter.

Adding Callbacks

Most widgets provide “hooks” that call particular procedures when a predefined condition occurs. These hooks are known as callback lists and the procedures are called callbacks. To add a callback to a widget’s callback list, use the `WtAddCallback` function:

```
status = WtAddCallback(wid, reason, procedure,  
[client_data])
```

The *wid* parameter is the ID of the widget to add the callback to. The *reason* parameter is a string that specifies the callback list to which the callback routine (PV-WAVE function or procedure) is to be added. The reason name is derived from the Motif or OLIT reason names. If you have a Motif toolkit, remove the `XmN` prefix from the Motif reason names. If you have an OLIT toolkit, remove the `XtN` prefix from the OLIT reason names. For example:

Motif Reason	OLIT Reason	Widget Toolbox Reason
<code>XmNactivateCallback</code>	<code>XtNactivateCallback</code>	<code>activateCallback</code>

The application can optionally use the *client_data* parameter to specify some application-defined data to be passed to the callback procedure when the callback is invoked. If *client_data* is a local

variable (defined only in the current procedure), a copy of that variable is created and passed (passed by value). If the *client_data* is a global variable (defined in a Common Block), it is passed by reference.

For more information on `WtAddCallback`, see the *PV-WAVE Reference*.

The form of every Widget Toolbox callback procedure is:

```
PRO CallbackProc, widget, client_data, $
    nparams, [p1, p2, ... pn]
```

where:

widget — The widget ID.

client_data — The *client_data* passed to `WtAddCallback`.

nparams — The number of callback-specific parameters after *nparams*. Under Motif, two additional parameters are required: event and reason. For information on these additional parameters, see Appendix C, *Motif and OLIT Callback Parameters*.

p_i — The optional callback-specific parameters. For additional information on these parameters, see Appendix C, *Motif and OLIT Callback Parameters*.

Example

This example adds a callback called `quitit` to the callback list for the widget `warr(2)`. The callback reason is `activateCallback`.

```
status=WtAddCallback(warr(2), $
    'activateCallback', 'quitit')
```

See also the example Widget Toolbox application at the end of this chapter.

Adding Event Handlers

An event handler is a PV-WAVE procedure that is executed when a specific type of event occurs within a widget. Some, all, or no X events can be handled using one or more event handlers. To register an event handler for events that occur in a widget use the system function `WtAddHandler`:

```
status = WtAddHandler(wid, eventmask, handler,  
[client_data])
```

This function registers a callback procedure specified by the *handler* parameter (a string) as an event handler for the events specified by the *eventmask* parameter. The *eventmask* parameter must be one of the standard event masks defined in the file `wtxlib.pro` in the PV-WAVE Standard Library. The *wid* parameter is the ID of the widget to add the handler to.

You can register event handlers for multiple event masks using the OR operator. For example, `ButtonUp` and `ButtonDown` are combined with `ButtonUp OR ButtonDown`, causing the event to be triggered when the mouse button is pressed and when it is released.

The application can optionally use the *client_data* parameter to specify some application-defined data to be passed to the event handler procedure when the callback is invoked. If *client_data* is a local variable (defined only in the current procedure), a copy of that variable is created and passed (by value). If *client_data* is a global variable (defined in a Common Block), it is passed by reference.

For more information on `WtAddHandler`, see the *PV-WAVE Reference*.

The form of Widget Toolbox event handlers is:

```
PRO EventHandlerProc, widget, client_data, $  
nparams, eventmask, event
```

where:

widget — The widget ID.

client_data — The *client_data* passed to WtAddHandler.

nparams — The number of event handler-specific parameters after *nparams*. (This number is always 2, for the *eventmask* and *event* parameters.)

eventmask — One of the standard event masks defined in the file `wtxlib.pro` in the Standard Library. For a description of event masks, see Appendix E, “Event Reference”, of the *Xlib Reference Manual, Volume 2*, (O’Reilly & Associates, Inc., 1989).

event — PV-WAVE structure containing all the fields as defined for the Xlib XEvent structure. For a description of event structures, see Appendix E, “Event Reference”, of the *Xlib Reference Manual, Volume 2*, (O’Reilly & Associates, Inc., 1989).

Example

```
.
.
pane=WtCreate('menu', PopupMenuWidget, $
    parent)

status = WtAddHandler(pane, ButtonPressMask, $
    'PostMenu', parent)

.
.
PRO PostMenu, wid, parent, nparams, mask, $
    event

@wtxlib

status=WtPointer("GetLocation", wid, state)

if (Button3Mask AND state(6)) ne 0 then $
    status=WtSet(pane, POPUP=event)

END
```

X Event Handler Procedure Example

```
PRO handler, widget, data, nparams, mask, $
    event
    ...
END
```

Adding Timers

A timer (the Xt Intrinsics term is `TimeOut`) is a PV-WAVE procedure that is invoked when a specified time interval has elapsed. This function can be used to add or remove a timer. To register a timer routine for a specified interval, use the `WtTimer` function in the application with the `ADD` parameter.

```
timerid = WtTimer("ADD", interval, timer, [client_data])
```

The *interval* parameter specifies the time interval, in milliseconds, until the PV-WAVE procedure timer callback *timer* is invoked. The `WtTimer` routine, unlike the Xt `TimeOut` function, restarts itself when the *timer* callback procedure is called. To stop the timer, use the following command in the timer callback:

```
status=WtTimer("REMOVE", timerid)
```

The application can optionally use the *client_data* procedure to specify some application-defined data to be passed to the timer callback procedure when the callback is invoked. If *client_data* is a local variable (defined only in the current procedure), a copy of that variable is created and passed (passed by value). If *client_data* is a global variable (defined in a Common Block), it is passed by reference.

For more information on `WtTimer`, see the *PV-WAVE Reference*.

The form of Widget Toolbox timer procedures is:

```
PRO TimerProc, widget, client_data, nparams, $
    timerid, time
```

where:

widget – The top application shell widget ID.

client_data – The *client_data* passed to WtTimer.

nparams – The number of timer-specific parameters after *nparams*. (This number is always 2, for *timerid* and *time*.)

timerid – The unique timer ID.

time – The time interval in milliseconds.

Note 

This timer routine is not related to the TIMER procedure in the PV-WAVE Users' Library.

Example

```
common timer, tid
.
.
id=WtTimer("ADD", 100, 'TimerCallback', $
my_data)

PRO TimerCallback, wid, client_data, $
timer_id, interval
COMMON timer, tid
tid = timer_id
.
.
END
```

Adding Work Procedures

Most applications spend most of their time waiting for events to occur. You can register a work procedure that will be called when the toolkit is idle (waiting for an event). The work procedure is the only means offered by the Xt Toolkit for performing background processing. A work procedure is useful if you need to execute a time-consuming operation from a callback procedure. While a callback procedure normally blocks events until the operation is finished, a work procedure lets you execute a callback while the toolkit is idle.

To register a work procedure, use the system function `WtWorkProc`:

status = `WtWorkProc(function, parameters)`

The *function* parameter is an *Add* or *Remove* operation. *Add* registers a named work procedure. The *parameters* used depend on whether an *Add* or *Remove* operation is specified.

For more information on `WtWorkProc` and an example of its use, see the *PV-WAVE Reference*.

Adding Input Handler Procedures

While most GUI applications are driven only by events, some applications need to incorporate other sources of input into the X Toolkit event handling mechanism. `WtInput` supports input or output gathering from files. The application registers an input source handler procedure and a file with the X Toolkit. When input is pending on the file, the registered handler is invoked. Note that a “file” in this context should be loosely interpreted to mean any sink (destination of output) or pipe (source of data).

To register an input handler procedure, use the system function `WtInput`:

```
status = WtInput(function [, parameters])
```

The *function* parameter is an *Add* or *Remove* operation. *Add* registers an input handler procedure. The *parameters* used depend on whether an *Add* or *Remove* operation is specified.

The form of a Widget Toolbox input handler procedure is:

```
PRO InputHandlerProc, widget, client_data, $
    nparams, inputid, lun, source
```

where:

widget — The top application shell widget ID.

client_data — The client data passed to `WtInput`.

nparams — The number of input handler-specific parameters after *nparams*. This number is always three, for the *inputid*, *lun*, and *source* parameters.

inputid — A unique input handler ID.

lun — The logical unit number of the source (could be a file or a pipe) generating the event.

source — The file descriptor of the source (could be a file or a pipe) generating the event.

For more information on `WtInput`, see *WtInput Function* on page 382 of the *PV-WAVE Reference, Volume II*.

Changing the Cursor

The `WtCursor` function lets you change or set the cursor. For instance, when a long file is read into memory, you can display a wait cursor. You can select from a large number of cursors listed in Appendix D, *Widget Toolbox Cursors*.

A call to `WtCursor` has the following form:

```
status = WtCursor(function, widget [, index])
```

The *function* parameter specifies the type of cursor: `default`, `system`, `wait`, or `set`. If `set` is specified, then you must specify a cursor index from the list in Appendix D, *Widget Toolbox Cursors*.

For more information on `WtCursor` and an example, see the description of `WtCursor` in the *PV-WAVE Reference*.

Creating Tables

The `WtTable` function lets you modify tables of data created with the `XbaeMatrix` widget. The `XbaeMatrix` widget is an editable 2D array of string data (cells) similar to a spreadsheet.

The Motif version of the `XbaeMatrix` widget was originally developed by Andrew Wason of Bellcore. This widget was enhanced and an OLIT version was developed by Visual Numerics, Inc.

On a UNIX system, complete documentation for the `XbaeMatrix` widget is available in `$WAVE_DIR/docs/widgets`:

- `matrix_motif.ps` — A PostScript file containing documentation on the Motif version of `XbaeMatrix` widget. You can print this file on any PostScript printer.
- `matrix_olit.ps` — A PostScript file containing documentation on the OLIT version of `XbaeMatrix` widget. You can print this file on any PostScript printer.

On a VMS system, these files are in the directory:
`WAVE_DIR: [DOCS.WIDGETS]`.

Refer to these documents for detailed information on the MbaeMatrix widget's resources and callbacks.

For more information on creating tables and using WtTable, see the description of WtTable in the *PV-WAVE Reference*.

Running an Application

When all the widgets to be displayed are created and managed, and callbacks, handlers, and timers are defined, you must then realize (display) the root widget in the widget hierarchy and initiate the event loop:

```
top = WtInit('appl', 'Appl')
.
.
    Create the widgets.
.
.
status = WtSet(top, /Realize)
    Display the widget hierarchy.
WtLoop
    Initiate the event loop.
```

This causes the PV-WAVE routine to loop indefinitely, processing the events and dispatching callbacks, handlers, and timers. The WtLoop procedure can be stopped by destroying and closing the shell widget by calling WtClose:

```
status=WtClose(top)
```

See also the example Widget Toolbox application at the end of this chapter.

Related Include Files

The following files are located in PV-WAVE Standard Library (`wave/lib/std`). They are used in Widget Toolbox applications as include files. To include a file in a PV-WAVE program, use the `@` command. For example:

```
@wtxmclasses
```

- **wtxmclasses.pro** — Contains definitions of Motif widget classes. Include this routine in every procedure that creates or handles Motif widgets.
- **wtxmconsts.pro** — Contains definitions of Motif-related constants (for example, `Xm...`). Include this routine in each procedure using Motif-related constants.
- **wtolclasses.pro** — Contains definitions of OLIT widget classes. Include this routine in every procedure that creates or handles OLIT widgets.
- **wtolconsts.pro** — Contains definitions of OLIT-related constants (for example, `OL_...`). Include this routine in each procedure using OLIT-related constants.
- **wtxlib.pro** — Contains X Event mask and type definitions, and other Xlib constants for X Event handling. Include this file whenever you need to use these constants.
- **wtcursor.pro** — Contains the indexes for standard and custom cursors.

Example Widget Toolbox Application

The following program demonstrates how to use Widget Toolbox functions to create a simple Motif GUI. This program displays a form containing a single button. When you click on the button, the button's label changes. The following figure shows the Motif version of the example program's output:

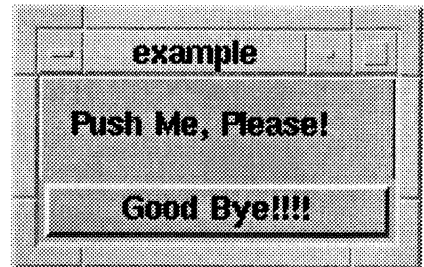
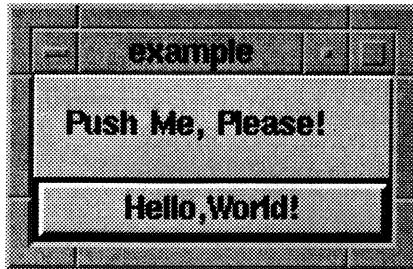


Figure 16-1 Motif GUI created by the example program.

```
PRO QUITIT, wid, data, npars, reason, event, $
    count
```

The QUITIT procedure is a callback routine that is executed when the "Hello World" button is selected.

```
COMMON widgets, warr, pushed
```

If the button is selected, replace the button label with "Good Bye!!!". Next time the button is selected, close the top-level window to quit the application.

```
IF pushed EQ 0 THEN BEGIN
```

```
    args={,labelString:'Good Bye!!!!'}
```

args is defined as an unnamed structure.

```
    status=WtSet(wid,args)
```

```
    pushed=1
```

```
ENDIF ELSE BEGIN
```

```
    PRINT, 'Quitting...'
```

```
    status=WtClose(warr(0))
```

ENDELSE

END

PRO EXAMPLE

This procedure creates a pushbutton with the QUITIT routine as
a the callback.

@wtxmclasses

@wtxmconsts

COMMON widgets, warr, pushed

warr = LONARR(3)

pushed=0

Initialize the application.

warr(0)=WtInit('example', 'Examples')

Create a "container" widget.

warr(1) = WtCreate('form', xmFormWidgetClass, \$
warr(0))

args={,x:10,y:10}

Create a label.

warr(2)=WtCreate('Push Me, Please!', \$
xmLabelWidgetClass, warr(1),args)

args={,y:40,width:140,height:25,\$
recomputeSize:FALSE}

Create the "Hello World" push button.

warr(3)=WtCreate('Hello,World!', \$
xmPushButtonWidgetClass, warr(1),args)

Add the button callback.

status=WtAddCallback(warr(3), \$
'activateCallback', 'quitit')

Display the top-level shell.

status=WtSet(warr(0), /Realize)

Process events. Call callbacks.

WtLoop

END

Programming Tips and Cautions

For information on PV-WAVE routines to avoid or use with caution when you are developing widgets applications, see *Programming Tips and Cautions* on page 472.

FORTRAN and C Format Strings

This appendix discusses the 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 Table 8-7.



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 — Importing Data with 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 153. 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 Chapter 7, *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)
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 is read.

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 PV-WAVE 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 PV-WAVE 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. Table A-1 shows data conversion characters, which specify the type of data that is being transferred. Table A-2 shows specifiers that are used only to specify the appearance of data, such as the number of spaces between values in a file.

**Table A-1: FORTRAN 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.

Table A-2: 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 Table A-1 and Table A-2, 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 PV-WAVE 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 PV-WAVE 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 PV-WAVE 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 Table A-3:

Table A-3: 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 Table A-3 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. Table A-4 gives examples of several floating-point formats and the resulting output.

Table A-4: Examples of Floating Point Output

Format	Internal Value	Formatted Output
F	100.0	___100.0000000
F	100.0D	___100.0000000000000000000
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 Table A-5. 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.

Table A-5: 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.

Table A-6: Examples of Integer Output

Format	Internal Value	Formatted Output
I	3000	__3000
I6.5	3000	_03000
I5.6	3000	*****
I2	3000	**
O	3000	__5670
O6.5	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 PV-WAVE 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 PV-WAVE 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

Note that 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

Tn

where:

n — is the absolute character position within the external record to which the current position should be set ($1 \leq n \leq 32767$).

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

TLn

where:

n — is the number of characters to move left from the current position ($1 \leq n \leq 32767$). 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 ($1 \leq n \leq 32767$). 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:

Table A-7: 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 / for double-precision.
d or i	Transfers a signed integer. Precede with / 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:

Table A-8: C-style Conversion Characters for Exporting Data

Conversion Character	How the Data is Exported
c	Transfers character data, one character at a time.
e or E	Transfers double-precision floating-point data using scientific (exponential) notation. Use the form $[-]m.d\text{d}\text{d}\text{d}\text{d}\text{d} e \pm xx$ or $[-]m.d\text{d}\text{d}\text{d}\text{d}\text{d} \pm E\text{xx}$, where the number of <i>d</i> 's is given by the precision.
f	Transfers double-precision floating point data in the form $[-]m.d\text{d}\text{d}\text{d}\text{d}\text{d}$, 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.

Motif and OLIT Widget Classes

This appendix lists the Motif and OLIT widget classes.

Motif Widget Classes

The following widget classes are defined in the file `wtxmclasses.pro` in the PV-WAVE Standard Library. They are used as a parameter to the `WtCreate` function. For example:

```
widget=WtCreate('Done', $  
xmPushButtonWidgetClass, top)
```



Widget classes are only used with Widget Toolbox functions. WAVE Widgets users do not need to use this appendix.

```
overrideShellWidgetClass  
vendorShellWidgetClass  
transientShellWidgetClass  
topLevelShellWidgetClass  
applicationShellWidgetClass
```

xmArrowButtonWidgetClass
xmBulletinBoardWidgetClass
xmCascadeButtonWidgetClass
xmCommandWidgetClass
xmDialogShellWidgetClass
xmDrawingAreaWidgetClass
xmDrawnButtonWidgetClass
xmFileSelectionBoxWidgetClass
xmFormWidgetClass
xmFrameWidgetClass
xmLabelWidgetClass
xmListWidgetClass
xmMainWindowWidgetClass
xmMenuShellWidgetClass
xmMessageBoxWidgetClass
xmPanedWindowWidgetClass
xmPushButtonWidgetClass
xmRowColumnWidgetClass
xmScaleWidgetClass
xmScrollBarWidgetClass
xmScrolledWindowWidgetClass
xmSelectionBoxWidgetClass
xmSeparatorWidgetClass
xmTextWidgetClass
xmTextFieldWidgetClass
xmToggleButtonWidgetClass

xmArrowButtonGadgetClass
xmCascadeButtonGadgetClass
xmLabelGadgetClass
xmPushButtonGadgetClass
xmSeparatorGadgetClass
xmToggleButtonGadgetClass
xbaeMatrixWidgetClass

Convenience Widgets

Convenience widgets allow easy manipulation of predefined groups of widgets. Convenience widgets are unmanaged when created. To make them visible use `WtGet(. . . , /Manage)` to manage them. For more information on convenience widgets, see the *OSF/Motif Programmer's Guide*.



The convenience widgets listed here are not related to WAVE Widgets.

BulletinBoardDialogWidget
ErrorDialogWidget
FileSelectionDialogWidget
FormDialogWidget
InformationDialogWidget
MenuBarWidget
MessageDialogWidget
OptionsMenuWidget
PopupMenuWidget
PromptDialogWidget
PulldownMenuWidget

QuestionDialogWidget
RadioBoxWidget
ScrolledListWidget
ScrolledTextWidget
SelectionDialogWidget
SimpleCheckBoxWidget
SimpleMenuBarWidget
SimpleOptionMenuWidget
SimplePopupMenuWidget
SimplePulldownMenuWidget
SimpleRadioBoxWidget
WarningDialogWidget
WorkAreaWidget
WorkingDialogWidget

OLIT Widget Classes

The following widget classes are defined in the file `wtxolclasses.pro` in the PV-WAVE Standard Library. They are used as a parameter to the `WtCreate` function. For example:

```
widget=WtCreate('Done', $
               flatButtonWidgetClass, top)
```



Widget classes are only used with Widget Toolbox functions. WAVE Widgets users do not need to use this appendix.

`overrideShellWidgetClass`
`vendorShellWidgetClass`
`transientShellWidgetClass`
`topLevelShellWidgetClass`
`applicationShellWidgetClass`
`abbrevMenuButtonWidgetClass`
`arrowtidgetClass`
`bulletinBoardWidgetClass`
`captionWidgetClass`
`checkBoxWidgetClass`
`controlAreaWidgetClass`
`drawAreaWidgetClass`
`dropTargetWidgetClass`
`exclusivesWidgetClass`
`flatButtonWidgetClass`
`flatCheckBoxWidgetClass`
`flatColorsWidgetClass`
`flatExclusivesWidgetClass`
`flatNonexclusivesWidgetClass`

flatWidgetClass
footerPanelWidgetClass
formWidgetClass
gaugeWidgetClass
helpWidgetClass
menuShellWidgetClass
menuButtonWidgetClass
menuButtonGadgetClass
nonexclusivesWidgetClass
noticeShellWidgetClass
oblongButtonWidgetClass
oblongButtonGadgetClass
popupWindowShellWidgetClass
rectButtonWidgetClass
rubberTileWidgetClass
scrollbarWidgetClass
scrolledWindowWidgetClass
scrollingListWidgetClass
sliderWidgetClass
staticTextWidgetClass
stubWidgetClass
textEditWidgetClass
textFieldWidgetClass
menuButtonGadgetClass
oblongButtonGadgetClass
xbaeMatrixWidgetClass

Motif and OLIT Callback Parameters

This appendix describes the parameters that are required in callback routines for Widget Toolbox functions under Motif and OLIT.

Motif Callback Parameters

For any Widget Toolbox callback under Motif, the first five parameters are always the same; however, some widget classes require additional callback parameters. These required and additional parameters are described below.

Note 

The callback parameters described below are only used with Widget Toolbox functions. WAVE Widgets users do not need to use this appendix.

Required Callback Parameters

The callback routines for all Motif widgets must have the following five parameters. The only exception to this is when the *Noparams* keyword is used in the *WtCreate* function. When this keyword is used, callbacks need only two parameters, the *widget* and *data*. All callback parameters are read-only.

- *widget* — The widget ID.
- *data* — Client data passed to *WtAddCallback*.
- *nparams* — The number of callback parameters after *nparams*. Two are required: *reason* and *event*.
- *reason* — Callback reason (For example: *Xm_CR...*).
- *event* — *XEvent* structure. This structure contains the tag fields documented in Appendix E, “Event Reference”, of the *Xlib Reference Manual, Volume 2*, (O’Reilly & Associates, Inc., 1989) and an event tag, which contains the address of the *XEvent* structure.

See the *OSF/Motif Programmer’s Reference* for more information on callback parameters.

Additional Required Callback Parameters

This section lists additional required callback parameters for the Motif widget classes.

Note

Callbacks for the *XbaeMatrix* (table) widget are discussed in the *XbaeMatrix* documentation, which you can find in the following file:

- `$WAVE_DIR/docs/widgets/matrix_motif.ps` — A PostScript file containing information on the Motif version of *XbaeMatrix* widget. You can print this file on any Post-Script printer.

ArrowButton

- *click_count* — Number of clicks. Only specified if the callback reason is XmCR_ACTIVATE. See the *OSF/Motif Programmer's Reference* for more information.

Command

- *command* — A string containing the last command entered.
- *length* — Length of the command.

DrawingArea

- *window* — Window ID of the drawing area.

DrawnButton

- *window* — Window ID of the drawn button.
- *click_count* — Number of clicks. Only specified if the callback reason is XmCR_ACTIVATE. See the *OSF/Motif Programmer's Reference* for more information.

FileSelection

- *filespec* — A string containing the file specification.
- *filespeclen* — File specification length.
- *mask* — A string containing the directory/file mask.
- *masklen* — The directory/file mask length.
- *dir* — A string containing the directory.
- *dirlen* — Directory length.
- *pattern* — A string containing the file search pattern.
- *patternlen* — The file search pattern length.

List

- *item* – A string containing the last selected item.
- *itemlen* – The last selected item length.
- *position* – Position of the last selected item (for callback reasons XmCR_MULTIPLE_SELECT and XmCR_EXTENDED_SELECT).
- *selected_items* – An array of strings containing selected items.
- *count* – Number of selected items.
- *positions* – An array of long integers specifying positions of the selected items.
- *selectiontype* – Type of selection (XmINITIAL, XmMODIFICATION, XmADDITION).

PushButton

- *click_count* – Number of clicks. Only specified if the callback reason is XmCR_ACTIVATE. See the *OSF/Motif Programmer's Reference* for more information.

RowColumn

The following parameters are only used only if the callback reason is XmCR_ACTIVATE.

- *widget* – Widget ID of the selected RowColumn item.
- *button* – Button number of the selected button.
- *click_count* – Number of clicks. For more information, see the *OSF/Motif Programmer's Reference*.

Scale

- *value* – New slider value.

ScrollBar

- *value* – New slider value.
- *pixel* – *X*-coordinate of the mouse button for horizontal direction; *y*-coordinate of the mouse button for vertical direction. Used only for the XmCR_TO_BOTTOM and Xm_CR_TO_TOP callback reasons.

Selection

- *selection* – A string containing the selection.
- *selectionlen* – Selected string length.

Text, TextField

For the callback reasons:

XmCR_LOSING_FOCUS,
XmCR_MODIFYING_TEXT_VALUE,
XmCR_MOVING_INSERT_CURSOR

- *doit* – Indicates if action is performed.
- *currInsert* – Current position of insert cursor.
- *newInsert* – New position of insert cursor.

For the callback reasons:

XmCR_MODIFYING_TEXT_VALUE,
XmCR_MOVING_INSERT_CURSOR

- *startPos* – Starting position of the text to modify.
- *endPos* – Ending position of the text to modify.

For the callback reason:

XmCR_MODIFYING_TEXT_VALUE


- *text* – A string containing the text to be inserted.
- *textlen* – Text length.
- *format* – Format of the text.

ToggleButton

- *value* – The toggle button’s current state.

OLIT Callback Parameters

For any Widget Toolbox callback under OLIT, the first two parameters are always the same; however, some widget classes require additional callback parameters. All callback parameters are read-only.

Note  The callback parameters described below are only used with Widget Toolbox functions. WAVE Widgets users do not need to use this appendix.

Required Callback Parameters

The callback routines for all OLIT widgets must have the following two parameters.

- *widget* – The widget ID.
- *data* – Client data passed to `WtAddCallback`.

Note  All OLIT callback parameters are read-only. See the *OPENLOOK Intrinsic Toolkit 3.0 Widget Set Reference Manual* for more information.

Additional Required Callback Parameters

This section lists additional parameters required for some widget classes. The only exception to this is when the *Noparams* keyword is used in the `WtCreate` function. When this keyword is used, all callbacks need only two parameters, *widget* and *data*.

If you use any of the additional parameters described below, the third parameter (after *widget* and *data*) must be *nparams*:

- *nparams* — Number of additional callback parameters after *widget* and *data*.

Note 

Callbacks for the XbaeMatrix (table) widget are discussed in the XbaeMatrix documentation, which you can find in the following file:

- `$WAVE_DIR/docs/widgets/matrix_olite.ps` — A PostScript file containing information on the OLIT version of XbaeMatrix widget. You can print this file on any PostScript printer.

drawArea

For the callback reasons:

XtNexposeCallback
XtNgraphicsExposeCallback
XtNresizeCallback

- *reason* — Callback reason:
OICR_EXPOSE
OICR_GRAPHICS_EXPOSE
OICR_RESIZE
- *event* — XEvent structure. This structure contains the tag fields documented in Appendix E, “Event Reference”, of the *Xlib Reference Manual, Volume 2*, (O’Reilly & Associates, Inc., 1989), and an event tag, which contains the address of the XEvent structure.
- *x* — X-coordinate of the upper-left corner of the drawing area.
- *y* — Y-coordinate of the upper-left corner of the drawing area.
- *width* — Draw area width.
- *height* — Draw area height.

flatCheckBox, flatExclusives, flatNonexclusives

For the callback reasons:

XtNselectProc
XtNunselectProc

- *item_index* — Sub-object initiating callback.
- *items* — Sub-object list.
- *num_items* — Number of sub-objects.
- *item_fields* — Key of fields for the list.
- *num_item_fields* — Number of item fields.
- *num_fields* — Number of fields.

popupWindow

For the callback reason:

XtNverify

- *verify* — Verify popdown (always True).

scrollbar

For the callback reason:

XtNsliderMoved

- *new_location* — Location of attempted scroll.
- *new_page* — Used to set page number.
- *ok* — Allow/disallow scroll.
- *slidermin* — Same as XtNsliderMin.
- *slidermax* — Same as XtNsliderMax.
- *delta* — New to old location.
- *more_cb_pending* — True if more callbacks are pending.

scrolledWindow

For the callback reasons:

XtNhSliderMoved
XtNvSliderMoved

(Same as scrollbar.)

scrollingList

For the callback reason:

XtNdefaultAction — Double click selection (not supported in OLIT).

XtNuserMakeCurrent — Single click selection.

- ***action*** — One of the following constant values is returned in this parameter:

LIST_STATE_NORMAL — Waiting for any click.

LIST_STATE_ADJUSTING — Middle click occurred.

LIST_STATE_DRAG_MOTION — Middle drag occurred.

LIST_STATE_EXTENDED — Shift left occurred, extend selection.

LIST_STATE_WAIT_FOR_DOUBLE — Left click occurred, waiting for second.

- ***selected_items_count*** — A long integer specifying the number of selected items.
- ***selected_items*** — A string array containing the list of selected items.
- ***selected_items_positions*** — An array of long integers specifying the positions of selected items.

Note

This implementation of the `scrollingList` widget class is built on top of the OLIT `scrollingList` widget class and allows single or multiple (drag) selection and double click (`action`).

slider

For the callback reason:

XtNsliderMoved

more_cb_pending – More callbacks will be called/used while the slider is being dragged.

value – Current value of slider.

stub – Only XtNdestroy, XtNexpose, XtNinitialize, XtNrealize, XtNresize reasons are supported.

textEdit

For the callback reasons:

XtNbuttons

XtNkeys

- ***consumed*** – See the *OLIT Widget Set Reference Manual*.
- ***event*** – XEvent structure. This structure contains the tag fields documented in Appendix E, “Event Reference”, of the *Xlib Reference Manual, Volume 2*, (O’Reilly & Associates, Inc., 1989) and an event tag, which contains the address of the XEvent structure.
- ***keysym*** – Current key symbol.
- ***buffer*** – Text buffer handle.
- ***ol_event*** – For information on this parameter, see the *OLIT Programmer’s Reference*.

For the callback reason:

XtNmargin

- ***hint*** – OL_MARGIN_EXPOSED, OL_MARGIN_CALCULATED
- ***x*** – X-coordinate of exposed rectangle.
- ***y*** – Y-coordinate of exposed rectangle.
- ***width*** – Width of exposed rectangle.

- *height* – Height of exposed rectangle.

For the callback reasons:

XtNmodifyVerification
XtNmotionVerification

- *ok* – True if update should occur.
- *current_cursor* – Current cursor value.
- *select_start* – Selection start.
- *select_end* – Selection end.

For the callback reason:

XtNmodifyVerification

- *new_cursor* – New cursor value requested.
- *new_select_start* – New selection start.
- *new_select_end* – New selection end.
- *text* – Modified text.
- *length* – Length of the text.

For the callback reason:

XtNpostModifyNotification

- *requestor* – See the *OLIT Widget Set Reference Manual*.
- *new_cursor* – New cursor value requested.
- *new_select_start* – New selection start.
- *new_select_end* – New selection end.
- *inserted* – Inserted text.
- *deleted* – Deleted text.
- *delete_start* – High 16 bits: line number; low 16 bits: offset.
- *delete_start_buffer* – Delete start buffer handle.
- *delete_end* – High 16 bits: line number; low 16 bits: offset.
- *delete_end_buffer* – Delete end buffer handle.
- *insert_start* – High 16 bits: line number; low 16 bits: offset.

- *insert_start_buffer* – Insert start buffer handle.
- *insert_end* – High 16 bits: line number; low 16 bits: offset.
- *insert_end_buffer* – Insert end buffer handle.
- *cursor_position* – Cursor position.

textField

For the callback reason:

XtNverification

- *string* – Text string of the text field.
- *ok* – Not used.

Widget Toolbox Cursors

This appendix lists the standard and custom cursors that are available for use with the `WtCursor` function.

Standard X Cursors

For detailed information on the following cursors, see Appendix I of the *Xlib Reference Manual, Volume 2*, O'Reilly & Associates, Inc.

`XC_X_cursor`

`XC_arrow`

`XC_based_arrow_down`

`XC_based_arrow_up`

`XC_boat`

`XC_bogosity`

`XC_bottom_left_corner`

`XC_bottom_right_corner`

`XC_bottom_side`

XC_bottom_tee
XC_box_spiral
XC_center_ptr
XC_circle
XC_clock
XC_coffee_mug
XC_cross
XC_cross_reverse
XC_crosshair
XC_diamond_cross
XC_dot
XC_dotbox
XC_double_arrow
XC_draft_large
XC_draft_small
XC_draped_box
XC_exchange
XC_fleur
XC_gobbler
XC_gumby
XC_hand1
XC_hand2
XC_heart
XC_icon
XC_iron_cross
XC_left_ptr

XC_left_side
XC_left_tee
XC_leftbutton
XC_ll_angle
XC_lr_angle
XC_man
XC_middlebutton
XC_mouse
XC_pencil
XC_pirate
XC_plus
XC_question_arrow
XC_right_ptr
XC_right_side
XC_right_tee
XC_rightbutton
XC_rtl_logo
XC_sailboat
XC_sb_down_arrow
XC_sb_h_double_arrow
XC_sb_left_arrow
XC_sb_right_arrow
XC_sb_up_arrow
XC_sb_v_double_arrow
XC_shuttle
XC_sizing

XC_spider
XC_spraycan
XC_star
XC_target
XC_tcross
XC_top_left_arrow
XC_top_left_corner
XC_top_right_corner
XC_top_side
XC_top_tee
XC_trek
XC_ul_angle
XC_umbrella
XC_ur_angle
XC_watch
XC_xterm

Custom Cursors

The following custom cursors were developed by Visual Numerics and are available for use with WtCursor:

WIDGET_WAIT_CURSOR
WIDGET_DEFAULT_CURSOR
WIDGET_CROSSHAIR_CURSOR
WIDGET_X_CURSOR
WIDGET_ARROW_CURSOR
WIDGET_VCOLORBAR_CURSOR
WIDGET_HCOLORBAR_CURSOR
WIDGET_MACRO_PLAY_CURSOR
WIDGET_MACRO_RECORD_CURSOR
WIDGET_MAGNIFIER_CURSOR
WIDGET_PREV_FIELD_CURSOR
WIDGET_PREV_HEADER_CURSOR
WIDGET_PREV_RECORD_CURSOR
WIDGET_PROF_COL_CURSOR
WIDGET_PROF_END_CURSOR
WIDGET_PROF_ROW_CURSOR
WIDGET_VIEWFINDER_CURSOR
WIDGET_CAUTION_CURSOR

Multivolume Index

This is a multivolume index that includes references to the *PV-WAVE User's Guide* (UG), *PV-WAVE Programmer's Guide* (PG), and both volumes of the *PV-WAVE Reference* (R1 and R2).

Symbols

! character UG-34
operator PG-46
\$ character UG-34
& character UG-35
' character UG-34
" character UG-35
() operator PG-44
* character UG-36
* operator PG-45, PG-85
*, in array notation PG-85
+ operator PG-45, PG-123
- operator PG-45
. character UG-35
/ operator PG-45
: character UG-36
; character UG-34
 after @ symbol UG-20
< operator PG-46
= operator PG-44
> operator PG-46
@ character UG-36
^ operator PG-46

A

A format code PG-A-9, PG-A-11
aborting
 plots UG-62
 PV-WAVE UG-15–UG-16
ABS function R1-30, PG-21
absolute value R1-30, PG-21
access mode, for VMS PG-225
accessing data in PV-WAVE PG-350
accumulated math error status R1-90,
 PG-263
ACOS function R1-31
actual parameters PG-234, PG-236
ADD_EXEC_ON_SELECT procedure
 R1-33
addition operator (+)
 examples PG-45
 for concatenating strings PG-123
 operator precedence PG-32
ADDSYSVAR procedure R1-34
ADJCT procedure R1-36, UG-331
allocating colors UG-312
ALOG function R1-38
ALOG10 function R1-40
ambient component of color UG-199
ampersand
 in command files UG-21
 separating multiple statements
 UG-35
AND operator
 description of PG-48
 operator precedence PG-33
 truth table PG-43
animation
 controlling the pace R2-280,
 R2-315
 cycling through images R2-280
 double-buffering on 24-bit displays
 UG-A-79
 of data in pixmaps R2-278, R2-316

- of images R2-277, R2-315
- of images using pixmaps UG-A-85
- annotation
 - alignment of the text R2-497
 - axes tick marks R2-527, R2-531, R2-535, R2-559
 - centering of text R2-497
 - character size R2-501, R2-518, R2-543, R2-554
 - color of R2-504
 - contour plots R2-499
 - controlling tick marks of UG-78
 - display without data R2-511
 - margin for R2-525, R2-530, R2-534
 - on axes R2-524, R2-529, R2-533
 - plots R2-490, UG-68–UG-69
 - positioning text, example UG-91
 - three-dimensional orientation of text R2-521
 - width of text string R2-524
 - with hardware fonts UG-291
 - with software fonts UG-294, UG-302
 - X axis R2-520, R2-527
 - Y axis R2-531
 - Z axis R2-535
- application programming
 - create special effects with
 - graphics functions UG-A-94
 - the write mask UG-A-94
 - look-and-feel of GUI PG-411, UG-51
 - not enough colors available UG-A-81
 - providing a GUI UG-A-70
 - setting resources PG-465
 - unexpected colors UG-A-96
 - WAVE Widgets PG-412, PG-414
 - Widget Toolbox PG-479, PG-494
- arc-cosine R1-31
- arcsine R1-41
- arctangent R1-45
- area of polygon R1-522
- arithmetic errors, checking for PG-263, PG-265
- arithmetic operations, overflow condition PG-19
- arrays
 - * in notation PG-85
 - accessing
 - by memory order PG-281, PG-282
 - large arrays PG-281
 - arbitrary type, creating R1-449
 - assignment statements PG-55–PG-56
 - average of R1-46
 - building tables from R1-63
 - calculating determinant of R1-248
 - columns in PG-47
 - combining PG-88–PG-91
 - concatenation operators PG-8, PG-47
 - contouring 2D UG-94
 - correlation coefficient of R1-133
 - creating
 - dynamically R1-449
 - from unique elements R2-230
 - with arbitrary initialization R2-31
 - day of the year for each date in a date/time variable R1-155
 - definition of PG-4
 - degenerate dimensions PG-86
 - double-precision type R1-156
 - elements, number of R1-469
 - eliminating unnecessary dimensions PG-83
 - extracting fields from PG-37
 - finding the maximum value R1-452
 - floating type R1-340, R1-343
 - functions to create from various data types PG-39
 - in structures PG-112, PG-114
 - indexed by rows and columns PG-81
 - integer type, creating R1-409, R1-413
 - linear log scaling R1-497
 - locating non-zero elements in R2-354

- log-linear scaling R1-497
- log-log scaling R1-497
- longword type, creating R1-442
- masking PG-49
- mean of R2-136
- median value of R1-454
- minimum value of R1-462
- number of elements R1-469
- reading
 - data into PG-180, PG-181
 - into from display R2-223, UG-142
 - with READF PG-181
- referencing a non-existent element PG-83
- reformatting R2-24
- replicating values of R2-31
- resizing of R1-125, R2-21
- rotating R2-48
- rows in PG-47
- scaling to byte values R1-73
- setting default range of R2-81
- shifting elements of R2-99
- size and type of R2-114
- smoothing of R2-119
- sorting contents of R2-125
- standard deviation of R2-136
- storing elements with array subscripts PG-91
- string array with names of days of the month R1-465
- string array with names of days of the week R1-153
- string, creating R2-111, R2-139
- subarrays PG-86
- subscripts
 - of other arrays PG-87–PG-88
 - of points inside polygon region R1-542, R1-550
 - storing elements PG-91
- subsetting PG-42
- sum elements of R2-200
- transposition of R2-204
- transposition of rows and columns PG-47

- ASCII
 - files
 - formatted PG-144
 - reading R1-161, R1-177
 - row-oriented PG-146
 - fixed format PG-155, PG-157
 - free format PG-155–PG-156
 - free format data, writing to a file R1-203, R1-212
 - input/output
 - comparison with binary PG-151
 - procedures PG-155
- ASIN function R1-41
- assignment operator (=) PG-44
- assignment statements
 - description of PG-9
 - examples PG-53–PG-54
 - form 4 PG-56
 - four forms of PG-53
 - using array subscripts with
 - form 2 PG-55
 - form 4 PG-57
 - with associated variables PG-59
- ASSOC function R1-43
 - description PG-154
 - example of PG-213, PG-214
 - exporting image data with PG-191
 - general form PG-212
 - in relation to UNIX FORTRAN programs PG-218
 - offset parameter PG-212, PG-216
 - use of PG-152
- associated variables R1-43
 - See also ASSOC function
 - description PG-24
 - in assignment statements PG-59
- asterisk, for subscript ranges UG-36
- at (@) character UG-36
- ATAN function R1-45
- attachments of widgets PG-422–PG-424
- attributes
 - See also resource file
 - of an expression PG-272

- of variables PG-24
- VMS record PG-226–PG-227
- average, boxcar R2-119
- AVG function R1-46
- axes
 - See also AXIS procedure
 - See also tick marks
 - adding to plots UG-87
 - additional R1-48
 - annotation of tick marks R2-527, R2-531, R2-535, R2-559
 - box style R2-526, R2-531, R2-534
 - color of R2-504
 - controlling
 - appearance R2-554
 - scaling R2-554
 - converting between data and normalized coordinates R2-557
 - coordinate systems for UG-59, UG-87, UG-89
 - data range of R2-526, R2-530, R2-534
 - data values of tick marks R2-560
 - date/time UG-221, UG-245
 - drawing and annotating UG-87
 - drawn without data R2-511
 - endpoints R2-560
 - exchange of UG-126
 - inhibiting drawing of UG-88
 - intervals R2-525
 - length of tick marks R2-522–R2-535, R2-559
 - linear R2-560
 - logarithmic R2-529, R2-532, R2-536
 - scaling R2-560, UG-84
 - multiple R1-48, UG-68, UG-87
 - output axis range R2-554
 - outward pointing tick marks R2-522
 - positioning of UG-85
 - range
 - input range R2-556
 - max and min of last plot R2-554
 - specification R2-526–R2-534, UG-65
 - specifying data to plot R2-556
 - saving the scaling parameters R2-517
 - scaling of R2-557, UG-63, UG-84
 - setting
 - style R2-557
 - thickness R2-558
 - size of annotation R2-524, R2-529, R2-533
 - styles of UG-64
 - suppressing of R2-526
 - title of R2-529, R2-532, R2-535, R2-560
- AXIS procedure R1-48, UG-87–UG-89

B

- background color
 - LJ-250 output UG-A-23
 - PCL output UG-A-27
 - plotting background color R2-498
 - PostScript output UG-A-40
 - setting R2-543
 - SIXEL output UG-A-61
- backing store, for windows UG-A-7
- back-substitution for linear equations R2-177
- bandpass filters R1-250, UG-164
- bar charts UG-76–UG-79
- bar, menu PG-426
- base 10 logarithm R1-40
- basis function, example of R1-137
- batch files
 - See command files
- BEGIN statement PG-60
- BESELI function R1-51
- BESELJ function R1-53
- BESELY function R1-55
- BILINEAR function R1-57
- bilinear interpolation R1-57, UG-141, UG-170

- binary
 - data PG-152
 - advantages and disadvantages of PG-151
 - input/output procedures PG-188
 - input/output, comparison with ASCII PG-151
 - reading FORTRAN files PG-224
 - transferring PG-154
 - transferring strings PG-197
 - UNIX vs. FORTRAN PG-199
 - files
 - comparison to human-readable files PG-151
 - efficiency of PG-188
 - record length PG-148
 - record-oriented PG-149
 - VMS PG-149
 - input/output
 - advantages and disadvantages of PG-151
 - procedures PG-154
 - routines PG-154
 - transfer of string variables PG-197
- BINDGEN function R1-60
- bit shifting operation R1-418
- bitmap, used for cursor pattern UG-A-72
- blank, removing from strings R2-140
- block mode file access (VMS) PG-225
- blocking window
 - dialog box PG-453
 - file selection box PG-457
 - popup message PG-449
- blocks of statements
 - BEGIN and END identifiers PG-60
 - definition PG-10
 - description of PG-60
 - END identifier PG-61
 - with IF statements PG-72
- Boolean operators
 - AND PG-43, PG-48
 - applying to integers PG-43
 - applying to non-integers PG-43
 - NOT PG-43, PG-50
 - operator precedence PG-32
 - OR PG-43, PG-50
 - table of PG-7, PG-43
 - XOR PG-50
- Bourne shell PG-293, PG-299
- box style axes R2-526, R2-531, R2-534
- BREAKPOINT procedure R1-61
- buffer, flushing output R1-294, R1-344
- BUILD_TABLE function R1-63, UG-267
- bulletin board widget PG-421
- Butterworth filters UG-163–UG-164
- button box widget R2-406, PG-433
- buttons (widgets)
 - active PG-429
 - iconic PG-430, PG-433
 - in dialogs PG-454
 - in messages PG-451
 - menu toggle PG-430
 - radio PG-436
 - sensitivity of PG-470
 - toolbox PG-433
- BYTARR function R1-66
- byte
 - See also BYTE function
 - arrays, creating R1-66
 - data
 - a basic data type PG-3, PG-17
 - converting to characters PG-37, PG-125
 - reading from XDR files PG-206–PG-207
 - scaling R1-73
 - shown in figure PG-146
 - type
 - converting to R1-68
 - extracting from R1-68
- BYTE function
 - description R1-68
 - for type conversion PG-37
 - using with STRING function PG-199
 - with single string argument PG-126
- BYTEORDER procedure R1-71
- BYTSCL function R1-73, UG-154

C

C format strings

- conversion characters PG-A-23
- for data export PG-A-24
- for data import PG-A-22
- when to use PG-167

C functions

- CALL_WAVE PG-373
- loading a UT_VAR PG-387, PG-392
- used with CALL_UNIX PG-364
- w_cmpnd_reply PG-368
- w_get_par PG-366
- w_listen PG-365
- w_send_reply PG-367
- w_smpl_reply PG-367
- wavevars PG-321, PG-335

C programs

- accessing
 - from PV-WAVE with SPAWN PG-312
 - PV-WAVE's variable data space from PG-350—PG-351
- as a client PG-382
- as a server PG-380
- CALL_WAVE as a client PG-382
- calling
 - from within PV-WAVE PG-310
 - PV-WAVE (cwavec) PG-337
 - PV-WAVE (wavecmd) PG-316
 - using LINKNLOAD function R1-429
- communicating with R2-232
- creating XDR files PG-207—PG-208
- ending PV-WAVE with waveterm command PG-315
- linking to PV-WAVE PG-316
 - under UNIX PG-347
 - under VMS PG-349
- reading files with PG-196
- running PV-WAVE from PG-335, PG-342
- sending commands to PV-WAVE PG-315

- SETBUF function PG-310—PG-311
- writing to PV-WAVE PG-194

C shell PG-299

- C_EDIT procedure R1-81, UG-331
- CALL_UNIX function R1-77, PG-359—PG-381
- CALL_WAVE PG-371—PG-382
- callback
 - adding to a Widget Toolbox widget PG-485
 - definition of PG-413, PG-485
 - for menu button PG-428
 - list PG-485
 - parameters, Motif PG-C-3—PG-C-6
 - parameters, OLIT PG-C-6—PG-C-12
 - procedures R1-311
 - registering R1-309
 - reason PG-485
 - scrolling list PG-447
- calling PV-WAVE
 - cwavec routine PG-333, PG-340
 - cwavefor routine PG-333, PG-340
 - from a C program R1-429, PG-316
 - from a FORTRAN program (wavecmd) PG-318
 - from a statically linked program PG-333
 - from an external program PG-313
 - in a UNIX unidirectional environments PG-313
- callwave object module PG-313
- carriage control
 - VMS FORTRAN, table of PG-227
 - VMS output PG-226
- case folding PG-127
- case selector expression PG-62
- CASE statement PG-10, PG-62
- CD procedure R1-79, PG-302
- CeditTool procedure R2-294
- CENTER_VIEW procedure R1-86, UG-194
- CGM output UG-A-9—UG-A-12
- changing the PV-WAVE prompt R2-552

- characters
 - changing case of R2-150, R2-166, PG-127
 - concatenation PG-123
 - converting
 - from byte data PG-125
 - parameters to R2-144, PG-124
 - extracting R2-154, PG-131
 - inserting R2-157, PG-131
 - locating substring R2-154, PG-131
 - non-printable, specifying PG-22
 - non-string arguments to string routines PG-133
 - obtaining length of strings R2-148, PG-130
 - removing white space R2-140, R2-162, PG-128
 - semicolon UG-34
 - setting to lowercase R2-150
 - setting to uppercase R2-166
 - size of R2-501, R2-518, R2-543
 - axis notation R2-501, R2-518, R2-524, R2-529, R2-533, R2-554
 - software UG-291
 - special UG-33
 - string operations supported PG-121
 - thickness of R2-543
 - unprintable PG-125
 - vector-drawn UG-291
 - working with PG-121
- CHEBYSHEV function R1-89
- CHECK_MATH function R1-90, PG-36, PG-263, PG-265
- checking for
 - keywords PG-269
 - number of elements PG-270
 - overflow in integer conversions PG-264
 - parameters in procedures and functions PG-269
 - positional parameters PG-269
 - presence of keywords PG-271
 - size and type of parameters PG-272
 - validity of operands PG-264
- child processes PG-309–PG-310
 - spawning R2-126
- CINDGEN function R1-93
- clear output buffer R1-294, R1-344
- clear screen of graphics device R1-298
- client
 - C program as PG-382
 - definition of PG-359
 - in X Window Systems UG-A-69
 - linking with PV-WAVE PG-361
 - PV-WAVE as (CALL_UNIX)
 - PG-362, PG-377–PG-381, PG-387
- clipping
 - graphics output R2-503
 - of graphics window R2-543
 - rectangle R2-503
 - strings R2-158
 - suppressing R2-511, R2-546
- CLOSE procedure R1-94
- closing
 - files R1-94
 - DC (Data Connection) functions PG-140
 - description PG-140
 - LUNs PG-140
 - graphics output file UG-A-5
 - widget hierarchy PG-468, PG-471, PG-485
- colon (:):
 - format code PG-A-10, PG-A-12
 - use in PV-WAVE UG-35
- color
 - See also color tables, colormaps
 - See also TEK_COLOR procedure as 6-digit hexadecimal value UG-A-91
 - can be viewed in bar R2-284, UG-317, UG-331
 - characteristics, determining UG-A-82
 - contours, filling UG-109
 - decomposed into red, green, and blue UG-A-79

- editing interactively R2-290, R2-292, R2-301, UG-316, UG-331
- histogram equalizing R1-383
- images PG-189
- in resource file PG-415
- in TIFF files PG-192
- indices
 - changing default UG-57
 - definition of UG-57, UG-305
 - displaying UG-332
 - for Tektronix 4510 rasterizer UG-A-62
 - interpretation of UG-58
- list of PG-463
- lists, converting R1-535
- making color table with HLS or HSV system UG-331
- map
 - See colormap, color tables
- model
 - See color systems
- number available on graphics device R2-538, UG-322
- obtaining tables from common block UG-321
- of
 - annotation R2-504
 - axes R2-504
 - contour levels R2-500
 - data R2-504
 - elements inside plots UG-311, UG-321
 - graphics output R2-504
 - surface bottom R2-499
- on 24-bit display UG-A-79
- on 8-bit display UG-A-79
- PostScript devices UG-A-39
- pseudo PG-189
- raster graphics UG-A-92
- represented by shades of gray PG-189
- reserving for other applications UG-A-84
- rotate R2-286, R2-296, R2-303
- running out of UG-A-81
- setting background R2-498, R2-543
- setting in WAVE Widgets PG-463
- specifying by name UG-A-92
- translation
 - See also color table
 - in X Windows UG-A-84
 - table UG-308, UG-A-71, UG-A-78, UG-A-85
- true-color UG-147
- unexpected UG-A-96
- vector graphics UG-311, UG-A-92
- with monochrome devices UG-310
- X Windows UG-A-77
- color bar
 - purpose of R2-284, UG-317, UG-331
 - shown in figure R2-298
- COLOR field of !P, changing of UG-A-66
- color systems
 - definition UG-305
 - HLS UG-309
 - HSV UG-309
 - RGB UG-306
- color tables
 - adding new UG-330
 - based on
 - HLS system UG-329–UG-330
 - HSV system UG-329
 - changing predefined UG-330
 - controlling contrast UG-331
 - copying R2-67, UG-328
 - creating R1-81, R1-95, R1-97, UG-149, UG-329
 - creating interactively R1-81, R1-97, R1-396, R1-398, R1-400, R1-494, R1-588
 - custom, creating R2-293
 - discussion of UG-145, UG-310
 - editing UG-149
 - from Standard Library procedures UG-329
 - interactively R2-290, R2-292, R2-301, UG-316, UG-331
 - expanding R2-142

- histogram equalizing R1-383, R1-386, UG-329
 - indices R2-294
 - listed in table UG-312
 - loading R2-219, UG-311, UG-329
 - from colors.tbl file UG-145
 - from variables UG-145, UG-311, UG-330
 - into device UG-145
 - predefined tables R1-435
 - lookup table UG-308, UG-A-79
 - modifying R1-36, R1-464, UG-330
 - to enhance contrast UG-155
 - obtaining UG-321
 - predefined UG-145, UG-311, UG-329
 - replacing R1-464
 - reversing R2-302, UG-320
 - rotating R2-302, UG-320
 - saving in TIFF file PG-192
 - stretching R2-142, R2-302, R2-303, UG-320, UG-330
 - supplied with PV-WAVE R1-435, R1-464, UG-145, UG-311, UG-329
 - switching between devices UG-327
 - Tektronix 4115 color table R2-195
 - utility widgets, shown in figure R2-298
 - color wheel, shown in figure R2-298
 - COLOR_CONVERT procedure R1-95
 - COLOR_EDIT procedure R1-97, UG-331
 - COLOR_PALETTE procedure R1-101, UG-332
- colormap
 - See also color tables
 - compared to color table UG-A-78
 - how PV-WAVE
 - chooses one UG-A-80
 - obtains UG-A-83
 - private, advantages, disadvantages UG-A-82
 - shared, advantages, disadvantages UG-A-81
- column-oriented
 - ASCII files PG-144
 - data
 - in arrays PG-47
 - reading R1-168, R1-184, R1-207, R1-216
 - transposing with rows PG-47
 - FORTRAN write PG-181
- combining
 - image and contour plots UG-100
 - images with three-dimensional graphics UG-129–UG-130
 - surface and contour plots UG-125
- command files
 - description of UG-19
 - difference from main programs UG-25
 - examples UG-20
 - executing UG-19
 - at startup UG-13, UG-48
 - guidelines for creating UG-21
 - terminating execution of UG-20
 - three methods of running UG-19
 - using ampersand and dollar sign with UG-21
- command interpreter, VMS PG-299
- command line
 - editing PG-401
 - entering interactive commands UG-17
 - keys UG-33
- command recall
 - description of UG-32
 - using function keys UG-33
 - with INFO procedure PG-401
- command widget R2-410, PG-459
- comment field
 - in statements PG-52
 - semicolon UG-34
- common block
 - colors UG-321
 - COMMON block procedure
 - examples PG-249
 - COMMON statement PG-63
 - creating common variables for procedures PG-249

- definition PG-12
- not needed with user data PG-467
- use of variables PG-63
- used to pass client data PG-486–PG-489
- communicating with
 - a child process PG-309
 - operating systems from PV-WAVE PG-291
- compilation
 - automatic PG-69
 - of one or more statements R1-313
 - run-time PG-273
- COMPILE procedure R1-104
- compiler messages, suppressing R2-553
- compiling
 - .RUN with filename PG-239
 - automatic PG-239, UG-22
 - compiler messages, suppressing PG-29
 - external programs PG-313
 - procedures and functions PG-239
 - saving compiled procedure R1-104
 - system limits PG-241
 - under UNIX for LINKNLOAD PG-322, PG-326, PG-328
 - with EXECUTE function PG-273
- complex
 - arrays, creating R1-109, R1-112
 - conjugate R1-118
 - constants PG-20
 - data type R1-109, PG-3, PG-18
 - converting to R1-109
 - shown in figure PG-146
 - numbers PG-21
 - type arrays, creating R1-109
 - variables, importing data into PG-160
- COMPLEX function R1-109, PG-37
- COMPLEXARR function R1-112
- compressing date/time variables R1-272
- compression, of TIFF files PG-192
- Computer Graphics Metafile
 - See CGM
- concatenating
 - arrays PG-47
 - characters PG-123
 - strings PG-122–PG-123
 - text PG-123
- concurrent processes R1-309
- CONE function R1-113, UG-196
- conformance levels, for TIFF PG-192
- CONGRID function
 - description of R1-115
 - using to magnify or minify an image UG-141
- CONJ function R1-118
- connecting data points with lines UG-70
- constant shading UG-131
- constants
 - complex PG-20
 - double-precision PG-20
 - examples PG-19
 - floating-point PG-20
 - hexadecimal PG-18
 - integer PG-18
 - numeric PG-4
 - octal PG-18
 - ranges of PG-19
 - string PG-4, PG-21
 - using correct data types PG-280
- .CONTINUE command UG-16
- continuing program execution UG-15
- contour plots
 - 2D arrays UG-94
 - adding a Z axis R2-533
 - algorithms used to draw R2-506, UG-95
 - annotation of R2-499
 - cell drawing method R2-506
 - closing open contours with arrays UG-110
 - color of contour levels R2-500
 - coloring of UG-109
 - combining with
 - PLOT procedure UG-124
 - SURFACE procedure UG-125
 - combining with surface plots and images R2-104
 - creating R1-119

- cubic spline interpolation of R2-519
- default number of levels R2-511
- enhancing UG-97
- examples UG-94, UG-96–UG-99, UG-102, UG-105
- filling with color R1-529, UG-109
- filling with pattern R2-513
- follow method algorithm UG-95
- gridding irregular data R1-362
- labeling R2-501, UG-105
- levels to contour R2-508, R2-511
- line thickness of R2-504
- line-following drawing method R2-506
- linestyle of R2-502
- maximum value to contour R2-510
- overlying
 - an image UG-100
 - with scalable pixel devices UG-102
- placement of Z axis R2-533
- shading of R1-529
- size of annotation R2-500
- smoothing UG-108
- superimposing on surface plots R2-517
- CONTOUR procedure R1-119
- contrast
 - control UG-331
 - enhancement UG-153, UG-330
- control points, using to correct linear distortion UG-170
- Control-\, aborting PV-WAVE UG-15
- Control-C
 - interrupting PV-WAVE UG-15
 - using to abort plots UG-62
- Control-D, exiting PV-WAVE on a VMS system UG-14
- controls box, see sliders
- Control-Y UG-16, UG-34
- Control-Z
 - exiting
 - PV-WAVE on a VMS system UG-14
 - suspending PV-WAVE on a UNIX system UG-14
- CONV_FROM_RECT function R1-123, UG-193
- CONV_TO_RECT function R1-129, UG-193
- convenience widgets (Motif) PG-B-3
- conversion characters
 - C code PG-A-23–PG-A-25
 - C export PG-A-25
 - C import PG-A-23
 - FORTRAN PG-A-8
- conversion functions PG-36
- converting
 - between graphics coordinate systems R1-21, R1-123, R1-129, R1-538, R1-559, UG-60, UG-193
 - color lists R1-524, R1-535
 - data to
 - See also extracting data
 - byte type R1-68, R1-73
 - complex type R1-109
 - date/time R1-421, R2-58, R2-159, R2-238, UG-223, UG-229
 - double-precision type R1-267
 - floating-point type R1-340
 - integer type R1-338
 - longword integer type R1-442
 - string type R2-144
 - strings PG-124
 - date/time variables to
 - double-precision variables R1-286
 - numerical data R1-292
 - string data R1-289
 - strings for tables UG-282
 - degrees to radians PG-27
 - DT_TO_SEC function UG-258
 - DT_TO_STR procedure UG-256
 - DT_TO_VAR procedure UG-257
 - JUL_TO_DT function UG-233
 - Julian day number to PV-WAVE date/time R1-421
 - mixed data types in expressions PG-281
 - radians to degrees PG-27

- scalars to date/time variables
 - R2-238
- SEC_TO_DT function UG-233
- STR_TO_DT function UG-229, UG-231
- string data to date/time variables
 - R2-159
- strings to
 - lower case PG-127
 - upper and lower case PG-122
 - upper case PG-127
- three-dimensional to
 - two-dimensional coordinates UG-119
- VAR_TO_DT function UG-232
- CONVOL function R1-125, UG-159, UG-162
- convolution R1-125, UG-159
- coordinate systems
 - constructing 3D UG-121
 - converting from one to another
 - R1-21, R1-123, R1-129, R1-538, R1-559, UG-60, UG-193
 - data R2-505
 - device R2-505
 - graphics UG-59, UG-61
 - homogeneous UG-115
 - in PV-WAVE UG-59
 - normalized R2-512
 - polar R2-515, UG-89
 - reading the cursor position R1-143, UG-90
 - right-handed UG-116
 - screen display UG-138
- copying
 - color tables R2-67, UG-328
 - pixels UG-A-72, UG-A-85
- CORRELATE function R1-133
- correlation and regression analysis, Hilbert transform R1-381
- correlation coefficient, computing for arrays R1-133
- COS function R1-135
- COSH function R1-136
- COSINES function R1-137
- count-intensity distributions R1-386
- CREATE_HOLIDAYS procedure
 - R1-138, UG-239–UG-240
- CREATE_WEEKENDS procedure
 - R1-140, UG-241
- cross product, calculating R1-142
- CROSSP function R1-142
- CSV files, generating R1-213
- cube, establishes frame of reference
 - R2-311, R2-332
- cubic splines
 - interpolation R2-132
 - to smooth contours R2-519, UG-108
- cursor
 - controlling position of with TVCRS UG-143
 - for X Windows system UG-A-72
 - hot spot of UG-A-73
 - manipulating with images R2-217
 - position, reading R1-143
 - positioning text with UG-91
 - selecting default UG-A-72
 - specifying pattern of UG-A-72
 - system variable, IC R2-537
- CURSOR procedure R1-143, UG-91
- curve fitting
 - Gaussian R1-353
 - least squares fitting R2-181
 - multiple linear regression R2-26
 - non-linear least squares R1-147
 - polynomial R1-552, R1-555
 - singular value decomposition
 - method R2-181
 - to a surface R2-172
- CURVEFIT function R1-147
- custom color tables, creating R2-293
- cut-away volumes, defining R2-253
- cwavec routine PG-333–PG-340
- cwavefor routine PG-333–PG-340
 - calling PV-WAVE PG-334
- CYLINDER function R1-150, UG-197

D

- !D system variables, fields of R2-537–R2-540
- damage repair of windows UG-A-7
- data
 - See also data files
 - 3D graphics UG-115
 - accessing with wavevars PG-350
 - adding to existing plots UG-66
 - annotating R2-490
 - building tables from R1-63
 - changing to another coordinate system UG-59, UG-119
 - color of R2-504
 - column-oriented R1-168, R1-184, R1-207, R1-216
 - converting
 - byte type to characters PG-125
 - coordinates UG-193
 - date/time variables to
 - double-precision variables R1-286
 - date/time variables to numerical data R1-292
 - date/time variables to strings R1-289
 - into date/time UG-229
 - scalars to date/time variables R2-238
 - coordinate systems R2-505, UG-59, UG-61
 - dense R1-320
 - drop-outs PG-58
 - export
 - with C format strings PG-A-22, PG-A-24
 - with FORTRAN format strings PG-A-22
 - extracting
 - See extracting data
 - fitting, cubic spline R2-132
 - fixed format I/O PG-152, PG-165, PG-175
 - floating-point, with format reversion PG-A-6
 - formatting
 - into strings PG-124, PG-A-1–PG-A-2
 - with STRING function PG-199
 - free format PG-139
 - ASCII I/O PG-159, PG-161, PG-164
 - output PG-164
 - gridding irregular R1-362
 - importing
 - from more than one file UG-187
 - with C format strings PG-A-22
 - with FORTRAN format strings PG-A-22
 - input/output
 - from a file PG-12
 - of unformatted string variables PG-197
 - irregular R1-362
 - linear regression fit to R2-26
 - logarithmic scaling UG-84
 - magnetic tape storage R2-38
 - manipulation UG-8, UG-191
 - maximum value to contour R2-510
 - range R2-526, R2-530, R2-534, R2-556
 - reading
 - 8-bit image PG-189
 - for date/time UG-228
 - from magnetic tapes R2-193
 - into arrays PG-180, PG-181
 - tables of formatted data PG-175
 - unformatted R2-17, PG-193
 - record-oriented PG-150
 - reduction before plotting R2-512
 - row-oriented PG-146
 - scaling to byte values R1-73
 - skipping over R2-115
 - smoothing of R2-119
 - sorting tables of formatted data PG-175
 - sparse R1-368

- three-dimensional scaling R2-57
- TIFF format PG-191
- transfer
 - using FORTRAN or C formats PG-165
 - with XDR files PG-206
- transformation
 - by T3D procedure R2-185, UG-116
 - rotating R2-48, R2-51, UG-117
 - scaling UG-117
- type conversion PG-125
- types of PG-17
 - XDR routines for PG-210
- unformatted PG-193
- user-defined coordinate system UG-121
- writing
 - date/time UG-255
 - to a file PG-12
 - to magnetic tape R2-194
 - unformatted PG-193
 - unformatted output R2-361
 - with output formats PG-164
- data files
 - CSV format R1-213, PG-154
 - logical records PG-147
 - reading
 - 24-bit image data R1-195
 - 8-bit image data R1-192
 - ASCII data from a specific file R2-17
 - ASCII data from the standard input stream R2-17
 - binary data from a specific file R2-17
 - fixed-format ASCII data R1-161
 - freely-formatted ASCII data R1-177
 - TIFF R1-198
 - row-oriented ASCII PG-146
 - writing
 - 24-bit image data R1-221
 - 8-bit image data R1-219
 - ASCII free format data R1-203
 - fixed format ASCII data R1-203
 - TIFF R1-221, R1-223
 - to a file in ASCII free format R1-212
- data types
 - byte PG-3
 - combining in arrays PG-39–PG-40
 - complex PG-3
 - constants PG-17
 - double precision PG-3
 - floating point PG-3
 - integer PG-3
 - longword PG-3
 - mixed, in
 - expressions PG-35
 - relational operators PG-42
 - seven basics PG-34
 - string PG-3
 - structure PG-3
 - variables PG-24
- date/time
 - creating empty variables UG-227
 - data
 - adding R1-270
 - compressing R1-270, R1-272
 - converting to UG-229
 - converting to double-precision variables R1-286
 - converting to numerical data R1-292
 - converting to string data R1-289
 - converting to strings for tables UG-282
 - decrementing values R1-283
 - determining elapsed time R1-277
 - duration R1-277
 - formats UG-230
 - generating day of the year for each date R1-155
 - holidays UG-239
 - in tables UG-281
 - incrementing values R1-270
 - plotting UG-243–UG-253, UG-267

- printing values R1-282
- reading into PV-WAVE PG-170
- removing holidays and week-ends R1-272
- subtracting R1-283
- templates for transferring PG-169
- transfer with
 - DC_READ_FIXED PG-174
- transferring with templates PG-168
- using Julian day for tables UG-283
- using to generate specific arrays R1-279
- weekends UG-241
- writing to a file UG-255
- DC_WRITE functions UG-255
- description of UG-221
- excluding days UG-239
- Julian day UG-225
- reading data UG-228
- recalc flag UG-226
- structures
 - elements of UG-225
 - importing data into PG-162
- templates PG-169
- variables
 - creating from scalars R2-238
 - with current system date and time R2-199
- DAY_NAME function R1-153, UG-259
- DAY_OF_WEEK function R1-154, UG-260
- DAY_OF_YEAR function R1-155, UG-261
- DBLARR function R1-156
- DC (Data Connection) functions
 - advantages of PG-153
 - description of PG-13
 - for simplified data connection PG-153
 - opening, closing files PG-140
 - table of PG-14
- DC_ERROR_MSG function R1-157
- DC_OPTIONS function R1-159
- DC_READ_24_BIT function R1-195, PG-154
- DC_READ_8_BIT function R1-192, PG-154
- DC_READ_FIXED function R1-161, PG-157, PG-174–PG-184, PG-187, PG-A-8, UG-252
- DC_READ_FREE function R1-177, PG-136, PG-156, UG-246–UG-248
- DC_READ_TIFF function R1-198, PG-154
- DC_WRITE_24_BIT function R1-221, PG-154
- DC_WRITE_8_BIT function R1-219, PG-154, PG-191
- DC_WRITE_FIXED function R1-203, PG-157
- DC_WRITE_FREE function R1-212, PG-137, PG-156
- DC_WRITE_TIFF function R1-223, PG-154, PG-191
- DCL
 - logical names R2-74
 - symbols
 - defining R2-74
 - deleting R1-238
- deallocating
 - file units R1-344
 - example R1-348, PG-222
 - LUNs PG-141, PG-143
- DEC
 - printers UG-A-58
 - terminals, generating output for UG-A-54
- dec, definition of UG-200–UG-201
- declaring variables PG-5
- decomposed color UG-A-79
- DECStation 3100, error handling PG-268
- DECW\$DISPLAY logical, for X Windows UG-A-69
- DECwindows Motif UG-A-69
- DEFINE_KEY procedure R1-226
- DEFROI function R1-232
- DEFSYSV procedure R1-236

- degrees
 - convert from radians PG-27
 - convert to radians R2-541, PG-27
- delaying program execution R2-274
- DELETE_SYMBOL procedure R1-238, PG-297
- deleting
 - compiled functions from memory R1-239
 - compiled procedures from memory R1-241
 - DCL symbols R1-238
 - ERASE procedure R1-298
 - variables R1-244
 - VMS logical names R1-240
 - VMS symbols R1-238
- DELFUNC procedure R1-239
- DELLOG procedure R1-240, PG-296
- DELPROC procedure R1-241
- DELSTRUCT procedure R1-242, PG-104
- DELVAR procedure R1-244, PG-104
- demonstration gallery UG-179
- demonstration programs UG-183
- dense data R1-320
- density function, calculating histogram R1-386, UG-329
- DERIV function R1-246
- Desc qualifier, for sorting columns UG-276
- DETERM function R1-248
- determinant, calculating R1-246, R1-248
- device
 - controlling output of R1-249
 - coordinate systems UG-59, UG-61
 - current parameters R2-537
 - name of R2-539
 - selecting output from R2-66
- device drivers
 - general information about UG-A-1
 - getting information about PG-398
 - selecting output from UG-A-2
 - table of supported UG-A-2
- DEVICE procedure R1-249, UG-A-4
- dialog box R2-419, PG-453–PG-454
- differentiation, numerical R1-246
- diffuse component of color, for RENDER function UG-198
- digital gradient function PG-237
- DIGITAL_FILTER function R1-250
- DILATE function R1-254
- dimensions of expressions, determining PG-272
- DINDGEN function R1-259
- !Dir system variable R2-540
- directory path
 - !Path UG-20
 - searching for procedures and functions R2-550
- directory stack
 - popping directories off of R1-578
 - printing out R1-580
 - pushing R1-590
- directory, changing to R1-79
- disappearing variables PG-251, PG-258
- display
 - color, control of UG-145, UG-310
 - dithering UG-148
 - of images R2-225
 - of multiple plots on a page UG-83
 - reading from R2-223, UG-142
- \$DISPLAY environment variable, for X Windows UG-A-69
- !Display_Size system variable R2-540
- DIST function R1-260, UG-163
- Distinct qualifier, for removing duplicate rows in tables UG-272
- distortion, linear UG-170
- dithering, different methods compared UG-148
- division operator (/) PG-32
- DOC_LIBRARY procedure R1-264
- documentation of user routines R1-264
- dollar sign
 - as continuation character UG-34
 - format code PG-A-10, PG-A-12
 - in command files UG-21
- DOUBLE function R1-267
- double-buffering UG-A-79

- double-precision
 - arrays, creating R1-156, R1-259
 - constants PG-20
 - data type PG-17
 - data, converting to R1-267
 - variables, converting date/time variables to R1-286
- !Dpi system variable R2-540
- drawing area R2-422, PG-439
- driver
 - default settings
 - SIXEL UG-A-58
 - See device drivers
- DROP_EXEC_ON_SELECT procedure R1-269
- DT_ADD function R1-270, UG-237
- DT_COMPRESS function R1-272, UG-242
- DT_DURATION function R1-277, UG-239
- DT_PRINT procedure R1-282, UG-261
- DT_SUBTRACT function R1-283, UG-238
- DT_TO_SEC function R1-286, UG-258
- DT_TO_STR procedure R1-289, UG-256
- DT_TO_VAR procedure R1-292, UG-257
- DTGEN function R1-279
- !Dtor system variable R2-541
- dynamic
 - data types PG-35
 - structures PG-35

E

- eavesdrop mode, HPGL plotter output UG-A-18
- edge enhancement
 - Roberts method R2-45
 - Sobel method R2-122
- !Edit_Input system variable R2-541
- editing, command line UG-33

- editor
 - See also text widget
 - creating text PG-445
 - eigenvalues and eigenvectors, determining R2-201
 - 8-bit image data
 - how stored PG-189
 - writing data to a file R1-219
 - 8-bit color UG-A-79
 - elements
 - definition of PG-4
 - finding the number of R1-469
 - ELSE
 - in CASE statement PG-62
 - in IF statement PG-71
 - empty output buffer R1-294, R1-344
 - EMPTY procedure R1-294
 - Encapsulated PostScript
 - Interchange Format UG-A-34
 - keyword UG-A-33
 - with Microsoft Word UG-A-46
 - END identifier PG-60–PG-61
 - end of file
 - testing for R1-296
 - when applied to a pipe PG-310
 - writing to tape R2-276
 - ENDCASE, end statement PG-62
 - ENDIF, end statement PG-72
 - ending PV-WAVE sessions PG-315, PG-335, PG-341
 - ENDREPEAT, end statement PG-61
 - ENDWHILE, end statement PG-61
 - enhancing contrast, of images R1-383
 - environment
 - after an error PG-250
 - manipulating R1-77
 - modifying UG-44
 - ENVIRONMENT function R1-295, PG-294
 - environment variables
 - \$DISPLAY for X Windows UG-A-69
 - adding R2-64, PG-293
 - changing R2-64, PG-292
 - DELETE_SYMBOL procedure (VMS) PG-297
 - DELLOG procedure (VMS) PG-296

ENVIRONMENT function (UNIX)
 PG-294
 GETENV function (UNIX) PG-294
 obtaining
 all R1-295
 values of R1-356, PG-294
 SET_SYMBOL procedure (VMS)
 PG-296
 SETENV procedure (UNIX) PG-293
 SETLOG procedure (VMS) PG-295
 translating PG-294
 TRNLOG function (VMS) PG-295
 UNIX PG-292
 VMS PG-295–PG-296
 WAVE_DEVICE UG-44, UG-45
 WAVE_DIR UG-45
 WAVE_STARTUP UG-48, UG-51
 EOF
 function R1-296, PG-171, PG-219,
 PG-310
 statement, example of PG-78
 testing for PG-219
 WEOF procedure, writing to end of
 mark on tape PG-229
 writing to tape R2-276
 EQ operator
 description of PG-48
 operator precedence PG-33
 equality
 See EQ operator
 ERASE procedure R1-298
 erasing
 screen of graphics device R1-298
 variables UG-27
 ERODE function R1-300
 !Err system variable R2-541
 !Err_String system variable R2-541
 error bars, plotting R1-305
 error handling
 "Plot truncated" message R2-515
 See also math errors
 arithmetic PG-263
 categories of PG-257, PG-258
 CHECK_MATH function R1-90,
 PG-263
 DECStation 3100 PG-268
 during program execution PG-258
 enabling/disabling math traps
 PG-266
 example of checking for PG-267
 floating-point PG-263
 for I/O PG-140
 function, evaluating R1-304
 input/output R1-479, PG-259
 list of procedures PG-257
 machine dependent handling
 PG-268
 math
 See math errors
 mechanism in procedures and func-
 tions PG-250
 MESSAGE procedure PG-251
 messages PG-261–PG-262
 prefix for R2-542
 setting report level for "DC" func-
 tions R1-159
 obtaining text of message R2-541
 ON_ERROR procedure PG-258
 ON_IOERROR procedure PG-259
 options for recovery PG-258
 overflow PG-264
 recovery from PG-250, PG-259
 running SunOS Version 4 PG-268
 Sun-3 and Sun-4 PG-268
 VMS PG-268
 ERRORF function R1-304
 ERRPLOT procedure R1-305
 escape sequences
 creating PG-22
 SIXEL output UG-A-61
 event handler
 definition R2-367, PG-487
 registering R2-367, PG-487
 event loop
 WAVE Widgets PG-471
 Widget Toolbox PG-494
 exclamation point UG-34
 excluding days from date/time variables
 UG-239
 exclusive OR operator PG-50
 EXEC_ON_SELECT procedure R1-309

EXECUTE function R1-313, PG-243–
PG-244, PG-273

executing

- command files UG-19
- existing functions and procedures
UG-23
- interactive programs UG-25
- main programs UG-24
- operating system commands
R2-126
- statements one at a time R1-313

executive commands

- ..LOCALS compiler directive
PG-244
- .CON UG-15, UG-29
- .GO UG-29
- .LOCALS PG-243, UG-31
- .RNEW UG-28, UG-29
- .RUN
 - compiling functions and proce-
dures PG-239
 - description UG-27
 - examples UG-29
 - l argument UG-28
 - syntax PG-239
 - t argument UG-28
- .SIZE PG-242–PG-243, UG-31
- .SKIP UG-30
- .STEP UG-30
- definition of UG-26
- table of UG-27

EXIT procedure R1-316, PG-335,
PG-341, UG-14

exiting

- an application
 - See closing widget hierarchy
- PV-WAVE
 - EXIT procedure, description
R1-316
 - on a UNIX system UG-14
 - on a VMS system UG-14

EXP function R1-317

expanding

- color tables R2-142
- images R2-21, R2-493

explicitly formatted

- See fixed format

exponential function, natural R1-317

exponentiation operator (^)

- examples PG-46
- operator precedence PG-32

exporting data PG-191

exposing windows R2-363

expressions

- data type and structure evaluation
PG-34
- description of PG-6
- efficiency of evaluation PG-279
- evaluation of PG-35
- finding attributes of PG-272
- forcing to specific data types PG-36
- general description PG-31
- invariant PG-281
- mixed data types in expressions
PG-35
- overview of PG-6
- structure of PG-34, PG-39
- type of PG-34
- with arrays PG-40
- with mixed data types PG-281

Extended Metafile System UG-A-13

extracting data

- See also converting data to
double-precision floating-point data
type R1-267
- from variables PG-37, PG-38
- Julian day numbers R1-421
- longword integer type R1-442
- string type R2-144

extracting image plane UG-A-95

F

false

- definitions for different data types
PG-71
- representation of PG-42

Fast Fourier Transform R1-328

FAST_GRID2 function R1-319, UG-191

FAST_GRID3 function R1-322, UG-191

FAST_GRID4 function R1-325, UG-191
 FFT function
 applied to images UG-162
 description R1-328
 fields
 definition of PG-101
 extraction and conversion of PG-37
 reference to PG-108
 syntax PG-108
 file selection box (widget) R2-426,
 PG-456–PG-458
 FILEPATH function R1-331
 files
 access mode, VMS PG-225
 allocating units R1-359, PG-139,
 PG-143
 closing R1-94, PG-140
 column-oriented ASCII PG-144
 compression of TIFF files PG-192
 CSV format PG-154
 deallocating LUNs R1-345,
 PG-141, PG-143
 formatting, codes for PG-A-9–
 PG-A-25
 free format ASCII I/O PG-156
 FREE_LUN procedure, deallocating
 files PG-143
 getting information about PG-220,
 PG-399
 header, example PG-A-3
 I/O with structures PG-116
 in UNIX PG-223
 indexed files for VMS PG-228
 inputting data from PG-12
 locating on disk PG-218
 logical records, changing size of
 PG-147
 logical units PG-138
 on magnetic tape R2-38
 opening R1-480, PG-138–PG-139
 organization of VMS PG-224
 organization options PG-144
 pointer, positioning R1-519,
 PG-219
 portable binary PG-205
 printing R1-580
 procedures for fixed format I/O
 PG-165
 reading R2-17
 C programs PG-196
 DC_READ_FIXED PG-177–
 PG-184, PG-187, PG-A-8
 FORTRAN binary data with
 PV-WAVE PG-224
 one input character R1-358
 READF PG-175–PG-185
 READU PG-195, PG-203
 unformatted R2-17
 record-oriented binary files PG-149
 row-oriented PG-146
 skipping over R2-115
 standard input, output and error
 PG-140
 startup command UG-48
 stopping batch R2-138
 testing for end of file R1-296,
 PG-219
 units, closing R1-94
 VMS
 attributes PG-227
 fixed-length record format
 PG-217, PG-226
 record oriented PG-226
 when to open PG-140
 writing
 8-bit image data to R1-219,
 R1-221
 ASCII free format to R1-203,
 R1-212
 data to PG-12
 unformatted output R2-361
 with UNIX FORTRAN programs
 PG-200
 with WRITEU PG-195
 XDR PG-205
 filling
 contour plots UG-110
 pattern R2-506
 filters
 2D UG-163
 bandpass UG-164
 Butterworth UG-164

- creating digital filters R1-250
- exponential high or low pass UG-164
- for
 - images UG-162–UG-163
 - mean smoothing R2-119
 - median smoothing R1-454
 - signals or images R1-260
- high pass UG-160, UG-164
- low pass UG-164
- Roberts UG-160
- Sobel UG-160
- FINDFILE function R1-333, PG-218
- FINDGEN function R1-335
- FINITE function R1-336, PG-36, PG-264
- finite values, checking for R1-336
- fitting with Gaussian curve R1-353
- FIX function
 - description R1-338
 - example PG-36–PG-37
- fixed format PG-156
 - ASCII I/O PG-155, PG-157
 - comparison with free format PG-152
 - data, examples of reading PG-176
 - I/O PG-165, PG-175
- fixed-length record format, VMS PG-217, PG-226
- FLOAT function R1-340, PG-37
 - example PG-21
- floating-point
 - arrays, creating R1-340, R1-343
 - constants PG-20
 - data type PG-3, PG-17
 - errors PG-263
 - format code defaults PG-A-14
 - type, converting to R1-340, R1-343
- Floyd-Steinberg dithering UG-148
- FLTARR function R1-343
- flush output buffer R1-294, R1-344
- FLUSH procedure R1-344, PG-158, PG-219
- flushing file units PG-218
- Font field of !P UG-293
- fonts
 - 3D transformations UG-292
 - appearance (hardware vs. software) UG-291
 - changing UG-295
 - character size R2-501, R2-518
 - computer time for drawing UG-293
 - flexibility (hardware vs. software) UG-293
 - formatting commands UG-293
 - hardware R2-507
 - hardware vs. software UG-291
 - Hershey character sets UG-291
 - index of
 - graphics text font R2-544
 - hardware font R2-544
 - list of PG-464
 - portability (hardware vs. software) UG-292
 - positioning commands UG-301
 - PostScript
 - See PostScript fonts
 - selecting R2-507
 - to annotate plots UG-68
 - selection commands UG-295
 - setting in WAVE Widgets PG-464
 - software R2-507, R2-563
 - specifying R2-507
 - vector-drawn R2-507, UG-291
 - with graphic routines R2-507
 - with plotting routines R2-507
- FOR statement
 - definition PG-9
 - examples PG-65, PG-66
 - (variable increment) PG-67
 - explicit increment PG-67
 - format with a block statement PG-60
 - two forms of PG-65
- force fields
 - examples PG-65
 - increment parameter PG-65
 - plotting R1-510, R2-244, R2-248
 - simple PG-65

- form layout (widgets)
 - See also layout (WAVE widgets) PG-422
- formal parameters
 - copying actual parameters into PG-236
 - definition of PG-235
- format
 - C and FORTRAN for data transfer PG-165
 - for
 - DC (Data Connection) functions PG-169
 - writing data PG-164
 - interpreting format strings PG-166
 - tick labels UG-81–UG-82
- format codes
 - C PG-A-22
 - examples of integer output PG-A-17
 - floating-point
 - defaults PG-A-14
 - output PG-A-15
 - FORTRAN PG-A-9–PG-A-21
 - group repeat specifications PG-A-7
 - integer defaults PG-A-16
- format reversion PG-167, PG-A-5
- format strings
 - C, for data import and export PG-A-22
 - description of PG-A-1
 - FORTRAN PG-185, PG-A-22
 - when to use PG-A-2
- formatted data
 - commands for software fonts UG-293
 - rules for PG-159, PG-165
 - strings PG-124
 - structures for I/O PG-116
 - using STRING function PG-187
- FORTRAN
 - binary data, reading with PV-WAVE PG-224
 - ending PV-WAVE with waveterm PG-315
 - format codes PG-A-9–PG-A-21
 - format strings PG-A-8
 - reading multiple array elements PG-185
 - programs
 - calling PV-WAVE (wavcmd) PG-318
 - linking to PV-WAVE under UNIX PG-348
 - reading
 - data in relation to UNIX PG-199
 - multiple array elements PG-185
 - sending commands to PV-WAVE wavcmd PG-315
 - writing data to PV-WAVE PG-181–PG-185
 - 4D gridding R1-325, R1-369, UG-191
 - Fourier
 - spectrum, 2D UG-165
 - transform
 - See FFT function
 - free format
 - ASCII I/O PG-155–PG-156
 - input PG-159–PG-160
 - output PG-164
 - FREE_LUN procedure R1-345, PG-143–PG-144
 - frequency
 - domain techniques UG-162
 - image UG-163
 - FSTAT function R1-346, PG-220–PG-221
 - FUNCT procedure R1-350
 - FUNCTION definition statement PG-235
 - function keys
 - defining R1-226
 - equating to character strings UG-52
 - for line editing UG-33
 - getting information about R1-410, PG-400
 - functions
 - actual parameters PG-234
 - calling PG-11
 - calling mechanism PG-248

- checking for
 - keyword parameters R1-422
 - number of elements R1-469
 - parameters PG-269
 - positional parameters R1-471
- compiling PG-239
 - automatically PG-69, PG-239
 - with .RUN and filename PG-239
 - with interactive mode PG-240
- conditions for automatic compilation PG-234
- copying actual parameters into formal parameters PG-236
- creating PG-15
 - interactively UG-26
 - with a text editor UG-22
- definition of PG-233
- definition statement of FUNCTION PG-235
- deleting from memory R1-239
- determining number of parameters PG-269
- directory path for R2-550, PG-28
- documentation of user R1-264
- error handling PG-250
- formal parameters PG-235
- I/O errors in R1-479
- I/O for simplified data connection PG-13
- information, obtaining with INFO procedure PG-401
- keyword parameters PG-68, PG-74
- libraries of VMS PG-251
- library UG-23
- maximum size of code PG-397
- number of parameters required PG-236
- overview of PG-2, PG-14
- parameters PG-74, PG-234
 - passing mechanism PG-246
 - positional PG-68, PG-74, PG-235
- program control routines PG-273
- recursion PG-248
- required components of PG-237

- search path UG-46—UG-50
- syntax for
 - calling PG-15
 - defining PG-15
- type conversion PG-36
- user-defined PG-241

G

- GAMMA function R1-352
- GAUSSFIT function R1-353
- Gaussian
 - curve fitting R1-353
 - function, evaluating R1-355
 - integral R1-355
- GAUSSINT function R1-355
- GE operator
 - description of PG-49
 - for masking arrays PG-49
 - operator precedence PG-33
- geometric transformations UG-167, UG-168
- GET_KBRD function R1-358, PG-154, PG-222, PG-223
- GET_LUN procedure R1-359, PG-143
- GET_SYMBOL function R1-361
- GETENV function R1-356, PG-294
- GOTO statement PG-10, PG-52, PG-70
- Gouraud shading R1-564, R2-70—R2-71, UG-131
- graphical user interface
 - See GUI
- graphics
 - 2D array as 3D plot R2-197
 - 3D to 2D transformations R2-549
 - bar charts UG-76
 - changing graphics device UG-45
 - clipping R2-503
 - output R2-503
 - window R2-543
 - combining
 - CONTOUR and PLOT procedures UG-127
 - CONTOUR and SURFACE procedures UG-126

- connecting symbols with lines
 - R2-548
- converting from 3D to 2D coordinates
 - UG-119
- coordinate systems for
 - UG-59,
 - UG-116
- current device parameters
 - R2-537,
 - PG-398
- cursor
 - R1-143, UG-90
- device driver
 - changing
 - UG-45
 - controlling
 - R1-249
 - size of display
 - R2-539
 - supported by PV-WAVE
 - UG-A-2
- establishing a 3D coordinate system
 - UG-121
- exposing and hiding windows
 - R2-363
- extracting image profiles
 - R1-583
- keywords
 - R2-497
- line thickness
 - R2-522, R2-549
- multiple graphs on a page
 - R2-545
- output device
 - configuring with DEVICE
 - UG-A-4
 - selecting with SET_PLOT
 - UG-A-3
- overlying an image on a contour
 - R1-402, UG-100
- polygon
 - filling
 - R1-542, UG-74
 - shading
 - R1-564
- positioning in window
 - R2-547
- procedures for plotting data
 - UG-55
- reading
 - cursor position
 - R1-143
 - values from cursor
 - R2-15
- shaded surfaces
 - R2-83, R2-91,
 - UG-131
- specifying range of data to plot
 - R2-556
- speeding up plotting
 - R2-547
- subtitle
 - R2-549
- symbols
 - R2-547
- text
 - See annotation
- thickness of lines
 - R2-522
- 3D
 - UG-115
- three-dimensional scaling
 - R2-57
- tick marks
 - See tick
- title
 - R2-550
- transformation matrices
 - R2-185,
 - UG-116
- window display
 - UG-A-6
- graphics window (drawing area)
 - PG-439
- gray
 - levels
 - dithering
 - UG-148
 - transformations
 - UG-152
 - scales
 - PG-189, PG-192
- greater than
 - See GT operator
- greater than or equal
 - See GE operator
- GRID function
 - R1-362
- GRID_2D function
 - R1-364, UG-191
- GRID_3D function
 - R1-367, UG-191
- GRID_4D function
 - R1-369, UG-191
- GRID_SPHERE function
 - R1-373,
 - UG-192
- gridding
 - 2D
 - R1-319, R1-364
 - 3D
 - R1-322, R1-367
 - 4D
 - R1-325, R1-369
 - definition of
 - UG-191
 - spheres
 - R1-373
 - summary of routines
 - R1-22
 - table of
 - UG-184
 - with dense data points
 - R1-319,
 - R1-322, R1-325
 - with sparse data points
 - R1-364,
 - R1-367, R1-369
- grids
 - making plots with
 - R1-362, UG-81
 - plotting keyword for
 - UG-78
- Group By clause
 - UG-273–UG-275
- group repeat specifications
 - PG-A-6–
 - PG-A-7
- groups of statements
 - PG-60

- GT operator
 - description of PG-49
 - operator precedence PG-33
- GUI
 - methods of creating PG-408
 - selecting look-and-feel PG-411, UG-51

H

- HAK procedure R1-377
- handler, event R2-367, PG-487
- HANNING function R1-378
- hardcopy devices
 - See output devices
- hardware fonts
 - See fonts
- hardware pixels UG-A-96
- hardware polygon fill UG-A-18
- help
 - See also information
 - documentation of user-written routines R1-264
 - getting R1-410
- HELP procedure R1-410
- Hershey fonts R2-507, UG-291
- Hewlett-Packard
 - Graphics Language plotters UG-A-13
 - ink jet printers UG-A-23
 - laser jet printers UG-A-23
 - Printer Control Language printers
 - See PCL output
- hexadecimal characters, for representing non-printable characters PG-23
- hexadecimal value, specifies color UG-A-91
- hide a widget PG-469
- hiding windows R2-363
- hierarchy of operators PG-32
- high pass filters R1-250, UG-160, UG-164
- HILBERT function R1-381
- HIST function, example of PG-66
- HIST_EQUAL function R1-383, UG-158

- HIST_EQUAL_CT procedure R1-386, UG-156
- histogram
 - calculating density function R1-386, UG-329
 - equalization R1-383, UG-155
 - HISTOGRAM function R1-388, UG-155
 - mode UG-70
 - of volumetric surface data R2-312
- HLS
 - color model UG-308, UG-309
 - compared to HSV UG-308
 - procedure R1-396, UG-329
- !Holiday_List system variable R1-437
- holidays, removing from date/time variables with DT_COMPRESS R1-272
- homogeneous coordinate systems UG-115
- HPGL output UG-A-13—UG-A-19
- HSV
 - color model UG-308, UG-309
 - compared to HLS UG-308
 - procedure R1-398, UG-329
- HSV_TO_RGB procedure R1-400
- hyperbolic
 - cosine R1-136
 - sine R2-112
 - tangent R2-191

I

- icons
 - on menu PG-431
 - on tool box PG-433
 - turning windows into R2-363
- IF statement PG-70—PG-72
 - avoiding PG-276, PG-277
 - definition PG-9
- image processing
 - See also images
 - calculating histograms R1-386
 - convolution R1-125
 - creating digital filters R1-250

- edge enhancement R2-45, R2-122
- Fast Fourier Transform R1-328
- Hanning filter R1-378
- histogram equalization R1-383
- magnifying images R1-125, R2-21
- minimizing images R2-21
- morphological dilation R1-254
- morphological erosion R1-300
- polynomial warping R1-526, R1-574
- Roberts edge enhancement R2-45
- rotating images R2-48, R2-51
- selecting a region of interest R1-232
- smoothing R1-454, R2-119
- Sobel edge enhancement R2-122
- write mask UG-328
- IMAGE_CONT procedure R1-402
- images
 - See also color, image processing
 - 24-bit, reading R1-195
 - 8-bit, reading R1-192
 - animating R2-277, R2-315
 - annotation of R2-490
 - as pixels PG-192
 - combining
 - CONTOUR and SURFACE procedures UG-126
 - with surface and contour plots R2-104
 - with three-dimensional graphics UG-129–UG-130
 - contrast
 - control UG-331
 - enhancement UG-330
 - convolution of R1-125, UG-159
 - data
 - how stored PG-189
 - interleaving PG-190, PG-193
 - output PG-191
 - pixels PG-193
 - reading block of from tape PG-231
 - reading with associated variable method PG-213
 - definition of UG-135
 - dilation operator R1-254
 - direction of display (!Order) R2-542
 - display
 - order UG-138
 - routines UG-4, UG-136
 - TVRD function R2-223
 - without scaling intensity R2-212
 - dithering UG-148
 - 8-bit format PG-189
 - erosion operator R1-300
 - expanding R1-115, R2-24, R2-36
 - extracting plane UG-A-95
 - extracting profiles from R1-583
 - Fast Fourier Transform R1-328
 - filtering UG-162
 - frequency domain techniques UG-162
 - geometric transformations UG-167
 - interleaving R1-197, R1-201, R1-222
 - interpolation of UG-170
 - magnifying R2-24, R2-36, R2-54, R2-225, R2-493, UG-140
 - modifying intensities of UG-152
 - morphologic dilation of R1-254
 - morphologic erosion of R1-300
 - optimizing transfer of UG-A-82
 - orientation of UG-138
 - overlying with contour plots R1-402, UG-100
 - placing the cursor in UG-143
 - polynomial warping of R1-521, UG-168
 - position of on screen UG-138
 - PostScript display of UG-A-39
 - profiles R1-583
 - reading
 - from display device UG-142
 - pixel values from R1-550
 - reformatting of R1-574
 - resizing of R2-225
 - Roberts edge enhancement R2-45
 - rotating R2-54, UG-168
 - routines used to display UG-135
 - scaling to bytes UG-154

- sharpening UG-160
- shrinking R1-115
 - or expanding R2-21, UG-140
- size of display UG-139
- smoothing R2-119
- Sobel edge enhancement of R2-122
- spatial warping R1-574
- special effects R1-543, R1-561, UG-328, UG-A-78
- subscripts of pixels inside polygon region R1-550
- transformation matrices UG-116
- transposing of R2-204
- true-color UG-146–UG-147
- 24-bit format PG-189
- value of individual pixels R2-327
- warping UG-168
 - with contour plots UG-100
 - zooming of R2-493
- IMAGINARY function R1-405
 - example PG-21
- IMG_TRUE8 procedure R1-406
- In operator UG-280
- include files for widgets PG-495
- including program files UG-33
- inclusive OR operator PG-50
- increment parameter
 - error if equal to zero PG-67
 - in FOR statement PG-67
- index number, color index UG-306, UG-322
- indexed files, VMS PG-228
- INDGEN function
 - description R1-409
 - using to make color tables UG-313
- indices of colors in color table R2-294
- indirect compilation UG-21
- Infinity PG-263
- INFO procedure
 - description of R1-410, PG-395, UG-16
 - getting information about files PG-220
 - viewing table structure UG-268
- information, obtaining with INFO procedure R1-410
- informational procedures, list of PG-257
- initializing WAVE Widgets PG-413
- input/output R1-479
 - advantages and disadvantages of each type PG-152
 - ASCII PG-151
 - ASSOC with unformatted data PG-212
 - binary data PG-151, PG-188
 - C binary data PG-194
 - C-generated XDR data PG-207
 - date/time data PG-162, PG-170
 - DC (Data Connection) functions PG-140, PG-153
 - efficiency of associated variables PG-212
 - error handling R1-479, PG-140, PG-259
 - fixed formats PG-152, PG-165
 - flushing file units PG-218
 - format reversion PG-167
 - free format R1-580, R2-17, PG-159, PG-164
 - from
 - keyboard R1-358, PG-222
 - word-processing application PG-175
 - image data PG-189
 - into
 - complex variables PG-160
 - structures PG-161
 - overview of PG-12
 - portable binary PG-206
 - procedures
 - for ASCII data PG-155
 - for binary data PG-154
 - READF procedure PG-175
 - reading records with multiple array elements PG-180–PG-182, PG-185
 - rules for
 - binary transfer of string variables PG-197
 - fixed format PG-156

- formatted data PG-159, PG-165
 - record-oriented binary files PG-149
 - selecting type of PG-138
 - strings with structures PG-117
 - structures PG-116
 - types of PG-151
 - unformatted R2-17
 - using UNIX pipes PG-310
 - VMS binary files PG-149
 - waiting for input R2-61
 - when to open PG-140
 - XDR files PG-205
 - INTARR function R1-413
 - integer
 - bit shifting R1-418
 - constants
 - examples PG-19
 - range of PG-19
 - syntax of PG-18
 - conversions, errors in PG-264
 - data
 - shown in figure PG-146
 - type PG-3, PG-17
 - writing with format reversion PG-A-5–PG-A-6
 - format defaults PG-A-16
 - output, for format codes PG-A-17
 - syntax of constants PG-18
 - type array, creating R1-413
 - type, converting to R1-338
 - Integral of Gaussian R1-355
 - intensities, modifying images UG-152
 - intensity, in 24-bit color UG-A-91
 - interapplication communication
 - bidirectional and unidirectional PG-307
 - calling PV-WAVE from a C program PG-337
 - methods of PG-305
 - using SPAWN PG-309
 - with a child process PG-309
 - with RPCs (remote procedure calls) PG-359
 - interleaving
 - description of PG-190
 - image data PG-190, PG-193
 - in 24-bit images R1-197, R1-201, R1-222
 - pixels PG-192
 - INTERPOL function R1-414
 - interpolated shading UG-131
 - interpolation
 - bilinear R1-57
 - cubic spline R2-132
 - linear, between colortable values R2-294
 - interprocess communication
 - See interapplication communication
 - invariant expressions, removing from loops PG-281
 - INVERT function R1-416
 - inverting pixels UG-A-95
 - irregular data, gridding R1-362
 - ISHFT function R1-418
 - iso-surfaces
 - example of UG-212, UG-214, UG-216
 - viewing interactively R2-307
- J**
- JOURNAL procedure R1-419
 - !Journal system variable R2-542
 - journaling R1-419
 - description of UG-36
 - examples UG-38
 - in relation to PRINTF UG-37
 - obtaining unit number of output R2-542, PG-28
 - JUL_TO_DT function R1-421
 - description of UG-233
 - example of UG-234, UG-253
 - Julian day
 - converting to a date/time variable R1-421
 - description of UG-225
 - in !DT_Base R2-60
 - using with tables UG-283

K

- keyboard
 - accelerators UG-52
 - defining keys R1-226, UG-52
 - getting input from R1-358, PG-222
 - interrupt UG-15, UG-29
 - key definitions PG-400
 - line editing, enabling R2-541, PG-28
 - using for command recall UG-32
- KEYWORD_SET function R1-422, PG-238, PG-239, PG-269
- keywords
 - checking for presence of PG-271
 - description of UG-23
 - examples PG-74
 - with functions PG-238
 - parameters
 - abbreviating PG-74
 - advantages of PG-75
 - and functions PG-68
 - checking for presence of R1-422, PG-239, PG-269
 - definition of PG-74, PG-235
 - graphics and plotting routines R2-497
 - passing of PG-234, PG-235
 - using the Keyword construct PG-74, PG-235
 - relationship to system variables UG-24, UG-57
- Korn shell PG-299

L

- labels, destinations of GOTO statements PG-52
- Lambertian
 - ambient component UG-199
 - diffuse component UG-198
 - transmission component UG-199
- landscape orientation UG-A-34
- LaTeX documents
 - inserting plots UG-A-40
 - using PostScript with UG-A-40, UG-A-43
- layout (WAVE Widgets)
 - arranging a PG-422
 - example PG-417, PG-420–PG-421
 - form PG-422
 - row/column PG-420
- LE operator
 - description of PG-49
 - operator precedence PG-33
- least square
 - curve fitting R1-552, R1-574, UG-72
 - non-linear curve fitting R1-147
 - problems, solving R2-179
- Lee filter algorithm R1-424
- LEEFILT function R1-424
- LEGEND procedure R1-426
- legend, adding to a plot R1-426
- less than
 - See LT operator
- less than or equal
 - See LE operator
- libraries
 - creating and revising for VMS PG-253
 - PV-WAVE Users' PG-255
 - searching (VMS) PG-252
 - Standard PG-255
- light source
 - lighting model, for RENDER function UG-197
 - modifying R2-70
 - shading R1-564, R2-70, UG-131
 - setting parameters for R2-70
- LINDGEN function R1-427
- line
 - color of R2-504
 - connecting symbols with UG-72
 - drawing R1-510, UG-121
 - fitting, example using POLY_FIT UG-72
 - linestyle index R2-545

- style of R2-502, R2-545
- thickness of R2-522, R2-549
- linear
 - algebra
 - eigenvalues and eigenvectors R2-201
 - LU decomposition R1-445, R1-447
 - reducing matrices R2-207
 - rules PG-46
 - solving matrices R2-207, R2-208
 - axes, specifying R2-560
 - distortion UG-170
 - equations
 - solution vector R1-468
 - solving R2-177
 - least squares problems, solving R2-179
 - regression, fit to data R2-26
- linking
 - applications to PV-WAVE PG-346
 - C code to PV-WAVE R1-429
 - C program under
 - UNIX PG-347
 - VMS PG-349
 - client with PV-WAVE PG-361
 - FORTRAN applications under
 - UNIX PG-348
 - LINKNLOAD function
 - accessing external functions R1-433, PG-323, PG-327
 - calling external programs PG-319
 - compiling under UNIX PG-322, PG-326, PG-328
 - description R1-429, PG-319
 - example R1-431, PG-321, PG-324
 - operating systems supported under PG-320
- list, scrolling
 - See scrolling list
- LJ-250 output UG-A-20—UG-A-23
- LN03 procedure R1-434
- LOAD_HOLIDAYS procedure R1-437, UG-240
- LOAD_WEEKENDS procedure R1-439, UG-241, UG-242
- LOADCT procedure R1-435, UG-145, UG-311, UG-329
 - description UG-136
- local variables, definition of PG-237
- log scaling, plotting R1-497
- logarithm
 - base 10 R1-40
 - natural R1-38
- logarithmic
 - axes R1-497
 - specifying R2-560
 - plotting R1-497
 - scaling R1-497
 - keywords UG-84
 - PLOT_IO procedure UG-85
- logical names
 - defining R2-65
 - VMS, DCL R2-209
 - VMS, deleting R1-240
- logical operators
 - evaluating PG-42
 - representation of PG-42
 - used with arrays PG-42
- logical records PG-147
- logical unit number
 - allocating R1-359
 - See LUNs
- LONARR function R1-440
- LONG function R1-442
- longword
 - data type PG-17
 - integer arrays, creating R1-427, R1-440
 - integer, converting to R1-442
- look-and-feel, of GUI PG-411, UG-51
- lookup table, color UG-308, UG-A-79
 - See also color tables
- loop, event
 - See event loop
- low pass filters UG-159, UG-164
 - creating R1-250

- LT operator
 - description of PG-49
 - operator precedence PG-33
- LUBKSB procedure R1-445
- LUDCMP procedure R1-447
- LUNs
 - closing PG-140
 - deallocating R1-345, PG-141
 - description PG-138
 - for
 - opening files PG-139
 - standard I/O PG-140
 - getting information using FSTAT PG-220
 - of current output file R2-539
 - of journal output R2-542, PG-28
 - operating system dependencies PG-142
 - range of PG-141
 - general use PG-143
 - reserved numbers PG-140, PG-141
 - standard
 - error output PG-142
 - input PG-141
 - output PG-141
 - used by
 - FREE_LUN PG-143
 - GET_LUN PG-143
 - with
 - UNIX PG-142
 - VMS PG-142
- skipping
 - backward on a tape PG-231
 - forward on the tape (VMS) PG-230
- TAPRD procedure PG-229
- TAPWRT procedure PG-229
- WEOF procedure PG-229
- writing to R2-194, PG-229
- magnifying images R2-21, R2-36, R2-54, R2-493, UG-140
- main programs
 - definition of UG-24
 - difference from command files UG-25
 - executing UG-24
- main PV-WAVE directory R2-540, PG-27
- main window
 - See shell window
- MAKE_ARRAY function R1-449
- makefile, using PG-346
- management, of widgets PG-484
- mapping, of widget PG-484
- margin, around plot R2-555
- marker symbols
 - displaying in a volume R2-263
 - for data points UG-72
 - user-defined UG-73
- masking
 - arrays PG-49
 - unsharp UG-161
- math errors
 - See also error handling
 - accumulated R1-90
 - accumulated math error status PG-263
 - CHECK_MATH function PG-265
 - detection of PG-263
 - hardware-dependent PG-268
 - procedures for controlling PG-258
- math messages, issuing R1-460
- math traps, enabling/disabling PG-266
- mathematical function
 - abbreviated list of UG-9
 - absolute value R1-30
 - arc-cosine R1-31

M

- Macintosh computers
 - See PICT output
- magnetic tape
 - accessing under VMS PG-229
 - mounting a tape drive (VMS) PG-230
 - reading from R2-193, PG-231
 - REWIND procedure PG-229
 - SKIPF procedure PG-229

arcsine R1-41
 arctangent R1-45
 area of polygon R1-522
 base 10 logarithm R1-40
 Bessel I function R1-51
 Bessel J function R1-53
 Bessel Y function R1-55
 bilinear interpolation R1-57
 bit shifting R1-418
 checking for finite values R1-336
 complex conjugate R1-118
 convolution R1-125
 correlation coefficient R1-133
 cosine R1-135
 cross product R1-142
 cubic splines, interpolation R2-132
 derivative R1-246
 determinant of matrix R1-248
 error function R1-304
 Fast Fourier Transform R1-328
 GAMMA function R1-352
 Gaussian integral R1-355
 Hilbert transform R1-381
 hyperbolic
 cosine R1-136
 sine R2-112
 tangent R2-191
 imaginary numbers R1-405
 improve solution vector R1-468
 LU decomposition R1-445, R1-447
 matrix
 reduction R2-207
 solutions R2-208
 maximum value R1-452
 mean R2-136
 minimum value R1-462
 natural exponent R1-317
 natural logarithm R1-38
 polynomial functions R1-521
 polynomial roots R2-495
 random numbers R2-13
 sine R2-109
 singular value decomposition
 R2-179
 solving R2-177
 square root R2-134
 standard deviation R2-136
 tangent R2-190
 mathematical morphology UG-173
 matrices
 reading and printing PG-93
 reading interactively R2-40
 transformation, See transformation
 matrices
 using subscripts with PG-82
 matrix
 expressions PG-99
 multiplication PG-94
 print to specified file unit R1-517
 print to standard output stream
 R1-515
 reading from a file R2-43, PG-96
 subarrays PG-98
 matrix inversion
 REVERSE function R2-36
 SVD procedure R2-179
 matrix multiplication operator (#)
 examples PG-46
 operator precedence PG-32
 MAX function R1-452
 !Max_Levels system variable R2-551
 maximum operator (>)
 examples PG-46
 operator precedence PG-32
 mean of an array R2-136
 mean smoothing UG-159
 median
 filter R1-454
 value of array R1-454
 MEDIAN function R1-454, UG-159
 median smoothing UG-159
 memory
 allocation PG-244
 deleting
 compiled functions from
 R1-239
 compiled procedures from
 R1-241
 structure definitions R1-242
 getting information about R1-410

- order and arrays PG-281, PG-282
- virtual
 - See virtual memory
- menus
 - bar PG-426
 - callback PG-428
 - creating R2-221, R2-359
 - creating and handling PG-425
 - defining text PG-429
 - example PG-431
 - option PG-428
 - popup PG-427
 - separators PG-430
 - using unnamed structure to define text PG-429
- MESH function
 - description of R1-457
 - example of UG-204, UG-206
- mesh surfaces, drawing R2-168, UG-112
- message
 - See also error handling
 - error, setting level to report in "DC" functions R1-159
 - for incomplete "DC" function R1-157
 - issuing error R1-460
 - popup message PG-449
 - suppressing compiler messages PG-29
- MESSAGE procedure R1-460, PG-251, PG-261–PG-262
- Microsoft Word, inserting PV-WAVE plots into UG-A-45–UG-A-47
- MIN function R1-462
- minimizing images UG-140
- minimum operator (<)
 - examples PG-46
 - operator precedence PG-32
- minimum value of array R1-462
- mixed data types PG-35, PG-42, PG-281
- MOD operator
 - description of PG-49
 - operator precedence PG-32
- MODIFYCT procedure R1-464, UG-330
- modulo operator PG-49
- monochrome
 - devices UG-148
 - displays UG-A-94
 - dithering UG-148
- MONTH_NAME function R1-465, UG-260
- morphologic
 - dilation operator R1-254
 - erosion operator R1-300
- morphology, mathematical UG-173
- Motif GUI PG-479
- mouse
 - accesses text editing functions R2-351
 - positions slice through data R2-332
 - rotates color table R2-286
 - use to rotate and flip surface R2-344
- !Mouse system variable R2-542
- MOVIE procedure R1-466
- MPROVE procedure R1-468
- !Msg_Prefix system variable R2-542
- multidimensional arrays, using subscripts PG-82
- multiple
 - array elements, reading records with PG-181–PG-182, PG-185
 - plots UG-83
 - statements, separating with ampersand UG-35
- multiplication operator (*) PG-32

N

- N_ELEMENTS function R1-469, PG-270
- N_PARAMS function R1-471, PG-269
- N_TAGS function R1-472, PG-119
- NaN, not a number PG-263
- natural
 - exponential function R1-317
 - logarithm R1-38

- NE operator
 - description of PG-50
 - operator precedence PG-33
- nearest neighbor method UG-170
- nested procedures, getting information on PG-403
- nonblocking window
 - dialog box PG-453
 - popup message PG-449
- non-printable characters
 - examples PG-23
 - specifying PG-22
- normal coordinate systems UG-60, UG-61
- normalized coordinates R2-512
- not equal
 - See NE operator
- NOT operator
 - description of PG-50
 - example PG-43
 - operator precedence PG-32
- number of elements in expression PG-270
- numeric
 - constants PG-4
 - errors
 - machine specifics PG-268
 - trapping PG-263
 - operators, table of PG-7

O

- object modules, callwave, compiling PG-313
- octal characters, for representing
 - non-printable characters PG-23
- offset parameter PG-216, PG-217
- ON_ERROR procedure
 - description R1-474, PG-258
 - examples PG-250
 - table of values PG-259
- ON_IOERROR procedure
 - description R1-479, PG-258
 - uses of PG-259
- OPEN procedures PG-139
- opening files
 - DC (Data Connection) functions PG-140
 - LUNs PG-138
 - OPEN procedures R1-480, PG-139
 - when to open PG-140
 - XDR PG-205
- OPENR procedure R1-480
- OPENU procedure R1-480
- OPENW procedure R1-480
- OpenWindows GUI PG-479
- operands, checking validity of PG-264
- operating system
 - accessing using SPAWN PG-297
 - communicating with from
 - PV-WAVE PG-291
 - UNIX, calling from PV-WAVE R1-77
- operator precedence
 - examples PG-33
 - table of PG-32
- operators PG-47
 - addition (+) PG-45, PG-123
 - AND PG-48
 - array concatenation, table of PG-8
 - assignment (=) PG-44
 - Boolean, table of PG-7
 - division PG-45
 - EQ PG-48
 - evaluating PG-42
 - exponentiation (^) PG-46
 - GE PG-49
 - general description PG-31
 - grouping PG-44
 - GT PG-49
 - hierarchy of PG-32
 - infix notation PG-2
 - LE PG-49
 - LT PG-49
 - matrix multiplication (#) PG-46
 - maximum (>) PG-46
 - minimum (<) PG-46
 - MOD PG-49
 - multiplication PG-45
 - NE PG-50
 - NOT PG-50
 - numeric, table of PG-7

- OR PG-50
- overview of PG-6
- parentheses () PG-44
- precedence PG-32
- relational, table of PG-8
- string, table of PG-8
- subtraction (-) PG-45
- table of
 - Boolean PG-43
 - relational PG-41
- using with arrays PG-42
- XOR PG-50
- OPLOT procedure R1-488, UG-61, UG-66, UG-68, UG-252
- OPLOTERR procedure R1-491
- option menu PG-428
- OR operator
 - description of PG-50
 - operator precedence PG-33
 - truth table PG-43
- Order By clause UG-276
- !Order system variable R2-542
- ordered dithering UG-148
- output
 - 24-bit image data R1-221
 - 8-bit image data R1-219
 - See also device drivers
 - ASCII free format data R1-203, R1-212
 - binary string variables PG-197
 - capturing with SPAWN procedure PG-301
 - clipping graphics R2-503
 - color PostScript
 - See color PostScript devices
 - controlling graphics device R1-249
 - file, sending to plotter or printer UG-A-5
 - flushing buffers R1-344
 - formats, for writing data PG-164
 - generated by SPAWN command PG-301
 - hardcopy
 - See device drivers
 - image data PG-191
 - procedures, table of PG-13

- selecting device R2-66
- steps to produce UG-A-3
- TIFF image data R1-221, R1-223
- to a file, basic steps PG-12
- using UNIX pipes PG-310
- writing unformatted data R2-361
- overflow
 - checking for in integer conversion PG-264
 - in type conversions PG-36
 - with integer arithmetic PG-19
- overlying
 - image and contour plots R1-402
 - images and graphics UG-100
 - using devices with
 - fixed pixels UG-103—UG-104
 - scalable pixels UG-102
- overplotting data R1-488, UG-56, UG-66

P

- !P system variable, fields of R2-543—R2-550
- padding
 - bytes PG-117
 - volumes R2-266
- page faults PG-283
- page file quota PG-287
- palette
 - color PG-192
 - of color cells R2-293
- PALETTE procedure R1-494, UG-332
- parameters
 - See also keywords
 - actual PG-234
 - changing for graphics device R2-537
 - checking for PG-269
 - copying PG-236
 - correspondence between formal and actual PG-235—PG-236
 - determining number of PG-269
 - for procedures and functions PG-74
 - formal PG-235

keyword PG-74
 number of non-keyword in routines
 R1-471
 number required in routines PG-236
 passing PG-75, PG-246
 by reference PG-246
 by value PG-246
 mechanism PG-111, PG-234,
 PG-246
 positional, definition of PG-235
 parent widget, manages child PG-484
 parentheses operator() PG-44
 !Path system variable R2-550
 patterns
 See polygon fill, pattern
 PCL output UG-A-23–UG-A-27
 percent sign (%), meaning of PG-A-6,
 PG-A-22
 period symbol UG-35
 Pqflquo quota PG-287
 physical memory, need for PG-283
 physical record, definition of PG-147
 !Pi system variable R2-552
 PICT output UG-A-27–UG-A-30
 pipes, UNIX, description of PG-310
 pixel interleaving PG-190, PG-192
 pixels
 copying UG-A-72, UG-A-85
 data PG-192
 easy way to ascertain exact value
 R2-327
 hardware UG-A-96
 inverting region of UG-A-95
 number per centimeter on graphics
 device R2-539
 palette color PG-192
 reading back UG-142
 scalable UG-139
 values, reading from an image
 R2-15
 pixmaps
 animating R2-278, R2-316
 creating with WINDOW procedure
 UG-A-86
 examples UG-A-86
 wider than the physical width of the
 screen UG-A-87
 with X Windows UG-A-8, UG-A-85
 plot
 area, defining R2-67, R2-79
 axis format R2-526
 combining images and contours
 UG-100
 data window UG-85
 inhibiting clipping of R2-546
 line styles R2-502
 margin for annotation R2-525,
 R2-530, R2-534
 multiple plots on a page R2-545
 position of in window R2-547
 positioning in display window R2-79
 region UG-85
 specifying
 margin around R2-515,
 R2-555
 range of data to plot R2-526,
 R2-530, R2-534, R2-556
 symbol index R2-516, R2-547
 PLOT procedure
 2D graphs R1-497
 basic description of UG-61
 date/time examples UG-244,
 UG-246, UG-249
 for tables UG-286–UG-287
 PLOT_FIELD procedure R1-507
 PLOT_IO procedure UG-84–UG-85
 2D graphs R1-497
 PLOT_OI procedure UG-61
 2D graphs R1-497
 PLOT_OO procedure
 2D graphs R1-497
 example of UG-82
 PLOTERR procedure R1-505
 PLOTS procedure R1-510
 plotters, HPGL UG-A-13
 plotting
 See also annotation, axes, color,
 ticks, image processing
 !P system variable R2-543, PG-27
 2D array as 3D plot R2-197
 3D data UG-93

3D orientation of text R2-521
 3D to 2D transformations R2-549
 adding
 axes UG-87
 bar charts UG-76
 changing coordinate systems
 UG-59
 clipping
 output R2-503
 window R2-543
 combination plots R2-104
 connecting symbols with lines
 R2-548
 contour plots R1-119, R2-506
 controlling
 placement of cursor R2-217
 converting from 3D to 2D coordinates
 UG-119
 coordinate systems UG-59,
 UG-115
 creating menus R2-221
 date/time data UG-243–UG-253
 examples UG-243–UG-253
 in tables UG-267
 defining a plotting symbol R2-235,
 UG-72
 display of images R2-212, R2-225
 enhanced contour plots UG-99
 error bars R1-305
 exchange of axes UG-126
 exposing and hiding windows
 R2-363
 filling contour plots UG-109
 flushing output buffer R1-294
 font selection R2-507
 ganging UG-83
 highlighting with POLYFILL UG-74
 histogram UG-70
 input from the cursor UG-90
 irregular polygon fill R1-542
 keywords R2-497
 line thickness R2-522
 linear log scaling R1-497
 location of plot in window R2-548,
 UG-85
 logarithmic
 axes R2-533
 scaling UG-84
 main title R2-523
 marker symbols UG-72
 multiple UG-83
 number of points to sum R2-547
 overlaying contour plots and images
 UG-100
 overplotting R1-488, UG-56,
 UG-66
 polar
 coordinates R2-515
 plots UG-89
 polygon filling R1-542, UG-74
 polygons R1-560
 position of graphics cursor R1-143,
 UG-90
 region filling UG-74
 routines, summary of UG-56
 scaling XY UG-63–UG-64
 smoothing contour plots UG-108
 speeding up R2-547
 surface shading parameters
 UG-132
 surfaces UG-112
 symbols R2-547, UG-72
 system variables UG-55
 tables UG-286–UG-287
 with a date/time axis UG-284
 thickness of lines R2-549
 three-dimensional graphics UG-115
 transformation matrices UG-116
 user-defined symbols UG-73
 vector fields from 3D arrays R2-240
 window display UG-A-6
 windows
 creating R2-356
 deleting R2-275
 POINT_LUN procedure
 description R1-519, PG-219
 example PG-197
 used to access VMS files PG-225

- pointer
 - function
 - example PG-488
 - WtPointer R2-390
 - into files R1-519
- polar
 - coordinates R2-515, UG-89
 - plots R2-515, UG-89
- POLY function R1-521
- POLY_2D function R1-526, UG-103, UG-168, UG-172
- POLY_AREA function R1-522
- POLY_C_CONV function R1-524, R1-535, UG-192
- POLY_COUNT function R1-537, UG-192
- POLY_DEV function R1-538, UG-194
- POLY_FIT function R1-552, UG-72
- POLY_MERGE procedure R1-557, UG-192
- POLY_NORM function R1-559, UG-194
- POLY_PLOT procedure R1-560, UG-195
- POLY_SPHERE procedure R1-568, UG-190
- POLY_SURF procedure R1-571, UG-189
- POLY_TRANS function R1-573, UG-192, UG-194
- POLYCONTOUR procedure
 - description R1-529
 - example of UG-110
 - filling closed contours R2-514, UG-109
 - syntax UG-109
- POLYFILL procedure
 - description R1-542
 - example UG-74, UG-76
 - fill pattern R2-506, R2-514
- POLYFILLV function R1-550
- POLYFITW function R1-555
- polygon fill
 - 2D or 3D polygon R1-542
 - color R2-504
 - example UG-74
 - hardware UG-A-18
 - pattern R2-506, R2-514
- polygon lists
 - description of UG-187
 - merging R1-557
- polygonal meshes
 - defining with MESH function R1-457
 - example of UG-204–UG-205
- polygons
 - calculating area of R1-522
 - filling
 - See polygon fill
 - generating R1-22, R1-571, UG-187
 - manipulating R1-23, UG-192
 - merging lists for rendering R1-557
 - plotting R1-560
 - querying subscripts of pixels inside R1-550
 - rendering R1-23, R1-560, R2-28, UG-176, UG-195
 - table of UG-183
 - returning
 - list of colors for R1-524, R1-535
 - number contained in list R1-537
 - shading R1-564
- polylines UG-121
- polynomial
 - curve fitting R1-552, R1-555
 - functions, evaluating R1-521
 - warping of images R1-526, R1-574
- POLYSHADE function R1-564, UG-195
- POLYWARP procedure R1-574, UG-173
- POPD procedure R1-578, PG-303
- popup menu
 - avoid using with text widget R2-462
 - creating PG-427
 - defining menu items PG-429
 - description R2-461
 - example PG-427
 - popup message PG-449, PG-451
 - popup widget PG-419, PG-449, PG-453, PG-457, PG-459

- portable data, XDR PG-205–PG-206
- portrait orientation UG-A-36
- position, of plot R2-515, UG-85
- positional parameters
 - checking for PG-269
 - examples PG-235
 - with functions PG-238
- positioning
 - file pointer PG-219
 - images on the display UG-138
 - text in PostScript UG-A-39
- PostScript output UG-A-31–UG-A-48
- PRINT procedure
 - description R1-580, PG-156
 - for tables with columns UG-285
 - for writing structures PG-116
- PRINTD procedure R1-582, PG-303
- Printer Control Language output
 - See PCL output
- printers supported by PV-WAVE UG-A-2
- PRINTF procedure
 - description R1-580, PG-156
 - for writing structures PG-116
 - in relation to journaling UG-37
- printing
 - files R1-580
 - graphics output UG-A-1
 - LN03 printer R1-434
 - tables with column titles UG-285
 - to LJ-250 printer UG-A-20
 - to PostScript printer UG-A-32
 - values of date/time variables
 - R1-282
 - variables R1-580
- private colormaps UG-A-81
- PRO statement
 - format of PG-76
 - syntax PG-235
- Procedure Call statement PG-73
- procedures
 - action to take on error R1-474
 - actual parameters PG-234
 - automatic compiling, conditions for
 - PG-69
 - call statement PG-76
 - calling PG-11
 - call stack PG-262
 - mechanism PG-248
 - checking for
 - number of elements R1-469, PG-270
 - parameters R1-422, R1-472, PG-269
 - compiling
 - automatically PG-69, PG-234, PG-239
 - three methods of PG-239
 - with .RUN and filename PG-239
 - with interactive mode PG-240
 - creating PG-15
 - with a text editor UG-22
 - definition of PG-11, PG-233
 - definition statement of PRO PG-76, PG-235
 - deleting from memory R1-241
 - determining number of parameters
 - PG-269
 - directory path for R2-550, PG-28
 - documentation of user-written routines R1-264
 - error handling PG-250, PG-257–PG-258
 - formal parameters PG-235
 - getting a list of those compiled
 - PG-401
 - help with R1-410
 - I/O errors in R1-479
 - information on nested PG-403
 - keyword parameters PG-74
 - libraries of VMS PG-251
 - library UG-23
 - maximum size of code PG-397
 - number of
 - non-keyword parameters R1-471, PG-269
 - parameters required PG-236
 - parameters PG-74, PG-234
 - passing mechanism PG-75, PG-246
 - positional PG-235

- program control PG-257, PG-273
- recursion PG-248
- required components of PG-237
- search path UG-20, UG-46
- stopping execution R2-138
- syntax for
 - calling PG-15
 - creating PG-15
 - user-written PG-69
 - various sources of PG-73
- process, spawning R2-126, PG-297, PG-298
- processes, concurrent R1-309
- processing images
 - See image processing
- product-moment correlation coefficient R1-133
- PROFILE function R1-583
- PROFILES procedure R1-585
- profiles, extracting from images R1-583
- program
 - calling from within PV-WAVE PG-310
 - code area full PG-242
 - control routines PG-257, PG-273
 - creating and running UG-22
 - data area full PG-243
 - file search method UG-48
 - files containing PV-WAVE procedures UG-28
 - format of UG-21, UG-24
 - help with nested PG-403
 - increasing speed of PG-276
 - linking to PV-WAVE PG-310
 - listings UG-28
 - main PV-WAVE UG-24
 - maximum size of code PG-397
 - pausing execution R2-274
 - preparing and running UG-12
 - running as batch UG-19
 - stopping R2-138
 - submitting to Standard Library PG-255
- programming
 - accessing large arrays PG-281
 - avoid IF statements PG-276
 - code size PG-397
 - commenting programs PG-52
 - delaying program execution R2-274
 - format strings PG-A-1—PG-A-2
 - menus, creating R2-221, R2-359
 - minimize virtual memory allocation PG-288
 - running out of virtual memory PG-284
 - tips PG-498
 - and cautions PG-472
 - for writing efficient code PG-275
 - on loops PG-281
 - use correct type of constants PG-280
 - use optimized system routines PG-280
 - use vector and array data operations PG-278
 - use virtual memory efficiently PG-285
 - using arrays efficiently PG-279
 - writing efficient programs PG-275
- prompt
 - changing R2-552, UG-52
 - PV-WAVE prompt string UG-12
- PROMPT procedure R1-587
- !Prompt system variable R2-552
- PSEUDO procedure R1-588, UG-330
- Pseudo_Color, 12-bit UG-A-94
- pseudo-color PG-189
 - compared to true-color UG-146
 - images, PostScript UG-146
- PUSHD procedure R1-590, PG-303
- PV-WAVE session
 - exiting R1-316
 - getting information about PG-395
 - recording R1-419
 - restoring R2-33
 - saving R2-56

Q

QMS QUIC output UG-A-48–UG-A-51
quadric animation, example of UG-207
QUERY_TABLE function
 combining multiple clauses UG-280
 description R2-1, UG-7, UG-263
 Distinct qualifier UG-272
 examples PG-179, UG-265
 features UG-270
 Group By clause UG-273
 In operator UG-280
 passing variable parameters
 UG-279
 rearranging a table UG-271
 renaming columns UG-272
 sorting with Order By clause
 UG-276
 syntax UG-271
 Where clause UG-277
quick.mk makefile PG-346
!Quiet system variable R2-553
QUIT procedure R2-11, UG-14
quitting PV-WAVE R1-316, R2-11,
 UG-13
quotas
 Pqflquo PG-287
 Wsquo PG-287
quotation marks UG-35
quoted string format code PG-A-10,
 PG-A-19

R

!Radeg system variable R2-553
radians
 converting from degrees PG-27
 converting to degrees R2-553,
 PG-27
radio button box widget PG-436
random file access PG-225
random number, uniformly distributed
 R2-13
RANDOMN function R2-12
RANDOMU function R2-13

range
 setting default for axes R2-81
 subscript, for selecting a subarray
 PG-84
raster graphics, colors UG-A-92
raster images UG-135
rasterizer, Tektronix 4510 UG-A-61
ray tracing
 cone primitives R1-113
 cylinder primitives R1-150
 definition of UG-197
 description of UG-175, UG-196
 mesh primitives R1-457
 RENDER function R2-28
 sphere primitives R2-129
 summary of routines R1-23
 viewing demonstration programs
 UG-177
 volume data R2-272
RDPIX procedure R2-15
READ procedure R2-17, PG-156,
 PG-160–PG-161
READF procedure
 description PG-156
 example PG-176
 with STR_TO_DT function
 PG-170
 for fixed format PG-175
 for row-oriented FORTRAN write
 PG-183
 reading
 ASCII files R2-17
reading
 24-bit image file R1-195
 8-bit image data R1-192, PG-189
 ASCII files R1-161, R1-177, R2-17
 binary files between different sys-
 tems PG-205
 binary input R2-17
 byte data from an XDR file
 PG-206–PG-207
 C-generated XDR data PG-207–
 PG-208
 cursor position R1-143, UG-90

- data
 - date/time UG-228
 - files R2-17
 - from a file PG-12
 - from a word-processing application PG-175
 - from multiple array elements PG-180
- DC_READ routines PG-A-8
- files, using C programs PG-196
- fixed-formatted ASCII files R1-161
- freely-formatted ASCII files R1-177
- from magnetic tapes R2-193
 - TAPRD PG-229
- from the display device UG-142
- images from the display R2-223
- multiple array elements with FORTRAN format string PG-185
- READF PG-175–PG-176, PG-181, PG-183, PG-185
 - for arrays PG-181
- READU PG-195, PG-203
- records with multiple array elements PG-180, PG-182, PG-185
- TIFF files R1-198
- unformatted input R2-17
- XDR files with READU procedure PG-209

READU procedure

- binary files PG-195
- description PG-154
- reading binary input R2-17
- syntax PG-193
- with Segmented keyword PG-203
- XDR files PG-209

realize, widget hierarchy PG-484

REBIN function

- decrease sampling with UG-98
- description R2-21
- resampling original image with UG-103
- using to magnify or minify an image UG-141

recalc flag, in !DT structure UG-226

record attributes of VMS files PG-226

recording a PV-WAVE session R1-419

record-oriented

- binary files PG-149
- data, transferring in VMS files PG-150

records

- definition of PG-147
- extracting fields from PG-37
- fixed length format (VMS) PG-217, PG-226
- length of PG-147
- multiple array elements PG-180, PG-181, PG-182, PG-185

recovering from errors PG-258

- PV-WAVE session R2-33

rectangular surfaces UG-189

recursion PG-248

reference, parameter passing by PG-246

REFORM function R2-24

region of interest, selecting R1-232

Regis output UG-A-54–UG-A-56

register, event handler PG-487

REGRESS function R2-26

regression, fit to data R2-26

relational operators

- descriptions PG-48–PG-50
- evaluating with operands of mixed data types PG-42
- table of PG-8, PG-41
- using with arrays PG-42

Remote Procedure Call

- See RPC

RENDER function

- color, defining for UG-198
- cone objects R1-113, UG-196
- cylinder objects R1-150, UG-197
- default view UG-202
- defining material properties of objects UG-200
- description R2-28, UG-196, UG-203
- example of UG-204–UG-218
- invoking UG-203
- lighting model UG-197
- mesh objects R1-457, UG-197
- sphere objects R2-129, UG-197

- syntax UG-203
 - volume objects R2-272, UG-197
- rendering
 - images, displaying UG-219
 - iso-surfaces UG-216
 - polygons R1-23, R1-560, UG-183, UG-195
 - process of UG-185
 - ray-traced objects R2-28, UG-196
 - shaded surfaces R1-564, R2-83, R2-91
 - volumes R1-24, R2-96, R2-267, UG-182–UG-183, UG-195
 - with standard techniques UG-195
- REPEAT statement PG-10, PG-77
- REPLICATE function
 - description R2-31
 - description of PG-114
 - examples PG-195
- reserved LUNs
 - description PG-140–PG-141
 - operating system dependencies PG-142
- reserved words PG-25
- reserving colors for other applications use UG-A-84
- resizing arrays or images R2-21
- resource
 - data type of value PG-483
 - definition of PG-483
 - deriving name of PG-483
 - file PG-415, PG-482
 - setting for Widget Toolbox PG-483
- resource file
 - how to use PG-464
- RESTORE procedure R2-33, UG-40–UG-41
- RECALL procedure R2-34, PG-251
- RETURN procedure R2-35, PG-68, PG-76, PG-236, PG-251
- reversal of rows and columns PG-47
- REVERSE function R2-36
- reversing
 - current color table R2-302, UG-320
 - direction of vector R2-36
- reversion, format PG-167, PG-A-5
- REWIND procedure R2-38, PG-229
- RGB
 - color system UG-306
 - triplets PG-193
- RGB_TO_HSV procedure R2-39
- right-handed coordinate system UG-116
- RMS files, reading images PG-217
- Roberts edge enhancement R2-45
- ROBERTS function R2-45, UG-160
- root window
 - relationship to visual class UG-A-90
 - See shell window
- ROT function R2-48
- ROT_INT function R2-54
- ROTATE function
 - description R2-51
 - examples UG-138
 - syntax UG-168
- rotating
 - arrays or images R2-51
 - corresponding to surface R2-168
 - current color table R2-302, UG-320
 - data UG-117
 - images R2-48, R2-51, R2-54
- row-oriented
 - ASCII data PG-146
 - data, shown in figure PG-146
 - FORTRAN write PG-183
- rows
 - in arrays PG-47
 - removing duplicate from a table UG-270
 - transposing with columns PG-47
- RPC
 - description of PG-359
 - example, CALL_UNIX PG-377–PG-387
 - synchronization of processes PG-360
- running
 - See executing
- run-time compilation of statements PG-273

S

- sampled images UG-135
- SAVE procedure R2-56, UG-40
- saving
 - a PV-WAVE session R1-419, R2-56
 - compiled procedures R1-104
 - TIFF data PG-191
- scalable pixels UG-102, UG-139
- scalars
 - combining with subscript array, ranges PG-91
 - converting to date/time variables R2-238
 - definition of PG-4, PG-24
 - in relation to arrays PG-290
 - subscripting PG-84
- scale unit cube into viewing area R2-57
- SCALE3D procedure R2-57, UG-123
- scaling
 - corresponding to surface R2-168
 - data UG-117
 - images R2-225
 - input images with BYTSCL function UG-154
 - logarithmic UG-84
 - plots UG-63–UG-64
 - three-dimensional R2-57
 - Y axis with YNozero UG-64
- screen pixels, assigning color UG-A-78
- scroll bars
 - on drawing area PG-439
 - ScrollBar callback parameters (Motif) PG-C-5
 - scrollbar callback parameters (OLIT) PG-C-8
 - scrolling list PG-446
 - text widget PG-443
 - used to view
 - image R2-326
 - text R2-351
- scrolling list
 - callback PG-447
 - creating PG-446
 - description R2-439
 - example PG-448
 - multiple selection mode PG-447
 - single selection mode PG-447
- searching VMS libraries PG-252
- SEC_TO_DT function R2-58, UG-233
- seconds, converting to date/time variables R2-58
- semicolon after @ symbol UG-20
- sensitivity, of widgets PG-470
- servers
 - C program as PG-380
 - closing connections UG-A-72
 - definition of PG-359
 - example program(test_server.c) PG-369
 - in X Window systems UG-A-69
 - linking with PV-WAVE PG-361
 - UNIX_REPLY function PG-375
 - using PV-WAVE as PG-371–PG-373, PG-392–PG-394
- session
 - See PV-WAVE session
- SET_PLOT procedure
 - description R2-66
- SET_SCREEN procedure R2-67
- SET_SHADING procedure R2-70, UG-132
- SET_SYMBOL procedure R2-74, PG-296
- SET_VIEW3D procedure R2-77, UG-194
- SET_VIEWPORT procedure R2-79
- SET_XY procedure R2-81
- SETBUF function
 - example PG-311
 - with child program PG-310
- SETDEMO procedure R2-62
- setenv command
 - for WAVE_PATH UG-47
 - for WAVE_STARTUP UG-49
 - with WAVE_DEVICE UG-45
- SETENV procedure R2-64, PG-293
- SETLOG procedure R2-65, PG-295
- SETUP_KEYS procedure R2-75

- SHADE_SURF procedure
 - description R2-83, UG-131
 - examples UG-133
- SHADE_SURF_IRR procedure R2-91
- SHADE_VOLUME procedure
 - description R2-96, UG-190
 - example of UG-206
- shading
 - constant intensity R2-72
 - contour plots R1-529
 - examples UG-133
 - Gouraud R2-71
 - light source UG-131
 - methods UG-131
 - setting parameters UG-132
 - setting parameters for shading R2-70
 - surfaces R1-564, R2-83, R2-91, R2-345, UG-131
 - setting parameters R2-79
 - volumes R2-96
- shared colormaps UG-A-80–UG-A-81
- shared library, io.so R2-62
- sharpening, of images UG-160
- shell processes PG-299–PG-300
- shell window
 - creating PG-416, PG-482
 - with initial layout PG-414
 - destroying PG-468
 - managing PG-484
 - realizing PG-484
 - unmanaging PG-484
- SHELL, UNIX environment variable PG-293
- SHIFT function R2-99
- shifting elements of arrays R2-99
- show a widget PG-469
- SHOW3 procedure R2-104, UG-129
- shrinking images R1-115, R2-21, R2-36
- SIGMA function
 - computing standard deviation of array R2-107
 - description R2-107
- signal processing
 - convolution R1-125
 - creating filters R1-250
 - Fast Fourier Transform R1-328
 - Hanning filter R1-378
 - histogram equalization R1-383
 - Lee filter R1-424
- simultaneous equations, solution of R2-177
- SIN function R2-109
- SINDGEN function R2-111
- sine function R2-109
- sine, hyperbolic R2-112
- single step command UG-27, UG-30
- single-precision floating-point data
 - shown in figure PG-146
- singular value decomposition
 - curve R2-177
 - SVD function R2-179
- SINH function R2-112
- SIXEL output UG-A-58–UG-A-61
- SIZE function
 - description R2-114, PG-272
 - examples PG-272
 - table of type codes PG-272
- size of arrays, determining R2-114, PG-270
- SKIPF procedure R2-115, PG-229
- skirt
 - adding to surface R2-345
 - shown in figure R2-346
- SLICE_VOL function R2-117, UG-179, UG-193
- slicing
 - plane, defining R2-253
 - volumes R2-117, R2-329
- sliders
 - and text input field PG-437
 - creating PG-437
 - description R2-414
 - example PG-438
 - orientation R2-415
 - using PG-437
- SMOOTH function R2-119, UG-158
- smoothing
 - boxcar R2-119
 - contour plots R2-519
 - contours, using Spline keyword UG-108

- mean UG-159
- median UG-159
- of images UG-158–UG-159
- Sobel edge enhancement R2-122
- SOBEL function R2-122, UG-160
- software fonts
 - index of R2-544
 - See fonts
- solids, shading R1-564
- SORT function R2-125, UG-290
- sorting
 - array contents R2-125
 - SORT function UG-290
 - tables, using QUERY_TABLE
 - R2-1, PG-179–PG-180, UG-276
- spaces in statements PG-52
- sparse data R1-368
- spatial transformation of images
 - R1-526, R1-574
- SPAWN procedure
 - accessing C programs from
 - PV-WAVE PG-312
 - avoiding shell processes PG-300
 - calling without arguments PG-298
 - capturing output PG-301
 - communicating with a child process
 - PG-310
 - description R2-126
 - example of PG-298, PG-312
 - interapplication communication
 - PG-309
 - non-interactive use PG-299
 - syntax PG-297
 - when to use PG-307
- spawning a process R2-126
- special characters, list of UG-33
- special effects
 - created with color UG-328
 - using Z-buffer UG-A-99
 - with color UG-A-94
- SPHERE function
 - description of R2-129
 - example of UG-207
- spheres
 - defining with SPHERE function
 - R2-129, UG-197
 - generating R1-568
 - gridding R1-373
- spherical
 - gridding UG-192
 - surfaces UG-190
- SPLINE function R2-132
- splines
 - cubic R2-132
 - interpolation of contour plots
 - R2-519
- SQL, description of UG-264
- SQRT function R2-134
- square root, calculating R2-134
- stacking plots UG-83
- standard deviation, calculating R2-136
- standard error output
 - LUNs PG-142
 - reserved LUNs PG-140
- standard input
 - LUNs PG-141
 - reserved LUNs PG-140
- Standard Library
 - location of PG-16, PG-255, UG-23
 - submitting programs PG-255
 - suggestions for writing routines
 - PG-256
- standard output
 - LUNs PG-141
 - reserved LUNs PG-140
- starting PV-WAVE
 - description UG-12
 - executing a command file UG-13
 - from an external program PG-314
 - interactive UG-12
- startup file
 - for UNIX UG-49
 - for VMS UG-50
- statements
 - assignment PG-9, PG-53
 - blocks of PG-10, PG-60
 - can change data types PG-55
 - CASE PG-10, PG-62
 - COMMON block PG-12, PG-63

- compiling and executing with
 - EXECUTE function PG-273
- components of PG-52
- examples of assignment PG-54
- executing one at a time R1-313
- FOR PG-9
- function call PG-11
- function definition PG-11
- GOTO PG-10, PG-70
- groups of PG-60
- IF PG-9, PG-70, PG-71
- labels PG-52
- list of 12 types PG-51
- procedure
 - calls PG-11, PG-73
 - definition PG-11, PG-76
- REPEAT PG-10, PG-77
- spaces in PG-52
- tabs in PG-52
- types of PG-9, PG-51
- WHILE PG-10, PG-78
- statically linked programs PG-333
- STDEV function
 - computing standard deviation R2-136
 - description R2-136
- stdio facility PG-310
- STOP procedure R2-138
- stopping PV-WAVE UG-13
- storing image data PG-189
- STR_TO_DT function
 - description R2-159, UG-229, UG-231
 - example PG-171, UG-232
 - templates UG-231
- STRARR function R2-139
- STRCOMPRESS function R2-140
- stream mode files, VMS PG-226
- STRETCH procedure R2-142, UG-155, UG-330
- stretching the current color table R2-302, UG-320
- string data
 - a basic type PG-3
 - shown in figure PG-146
- STRING function
 - description R2-144
 - example PG-37, PG-124
 - formatting data PG-187
 - syntax PG-124
 - with
 - BYTE function PG-199
 - single byte argument PG-125
 - structures PG-118
- string processing PG-121
 - compressing white space R2-140
 - converting
 - lower to upper case R2-167
 - upper to lower case R2-150
 - extracting substrings R2-154
 - inserting substrings R2-157
 - locating substrings R2-156
 - removing
 - blanks R2-140
 - excess white space R2-141
 - leading and trailing blanks R2-162
- string variables
 - binary transfer of PG-197
 - determining number of characters to transfer PG-197
 - passing into QUERY_TABLE UG-279
- strings
 - array arguments PG-133
 - arrays, creating R2-111, R2-139
 - changing case PG-127
 - concatenating PG-123
 - constants PG-4, PG-21
 - definition of PG-121
 - determining length of PG-122
 - examples of string constants PG-21
 - extracting substrings PG-131
 - formatting PG-122
 - FORTRAN and C formats PG-166
 - importing with free format PG-160
 - initializing to a known length PG-199
 - input/output with structures PG-117
 - inserting substrings PG-131

- length issues with structures
 - PG-118
- locating substrings PG-131
- non-string arguments PG-133
- obtaining length of PG-130
- operations supported PG-122
- operators, table of PG-8
- reading with Format keyword
 - PG-166
- removing white space PG-122,
 - PG-128
- STRCOMPRESS function PG-122
- STRING function PG-122
- string operations supported PG-121
- STRLEN function PG-122
- STRLOWCASE function PG-122
- STRTRIM function PG-122
- STRUPCASE function PG-127
- substrings PG-131
- transferring as binary data PG-197
- working with PG-121
- writing to a file PG-164
- strip chart
 - controlling the pace R2-339
 - displaying R2-336
- STRLEN function
 - description R2-148
 - determining string length PG-122
 - example PG-130
 - output PG-197
 - syntax PG-130
- STRLOWCASE function
 - converting to lower case PG-122
 - description R2-150
 - example PG-127
 - syntax PG-127
- STRMESSAGE function R2-152
- STRMID function R2-154, PG-132
- STRPOS function R2-156, PG-131
 - example PG-131
- STRPUT procedure R2-157, PG-132
- STRTRIM function R2-162
 - examples PG-128
 - removing white space PG-122
 - syntax PG-128
- STRUCTREF function PG-104
 - description R2-164
- structure references
 - examples PG-109
 - nesting of PG-108
 - syntax PG-109
- structures
 - See also unnamed structures
 - advanced usage PG-119
 - arrays PG-112, PG-114
 - creating unnamed PG-105
 - data type PG-3
 - date/time R1-204, R1-211, R1-290,
 - UG-225
 - defining PG-102
 - definition of PG-101
 - deleting PG-102
 - expressions PG-39
 - formatted input/output PG-116
 - getting information about R1-410,
 - PG-110, PG-402
 - importing data into PG-161
 - in relation to tables UG-287–
 - UG-288
 - input and output PG-116
 - introduction to PG-101
 - listing references to R2-164
 - number of tags in R1-472
 - N_TAGS function PG-119
 - parameter passing PG-111
 - PRINT and PRINTF PG-116
 - processing with tag numbers
 - PG-119
 - reference to a field PG-108
 - references to R2-164
 - removing definitions from memory
 - R1-242
 - REPLICATE function PG-114
 - scope of PG-105
 - storing into array fields PG-112–
 - PG-113
- string
 - input/output PG-117
 - length issues PG-118
 - structure references PG-109
 - subscripted references PG-108

- tag names R2-188
- TAG_NAMES function PG-119
- unformatted input/output PG-117
- using INFO PG-110
- STRUPCASE function R2-166, PG-127
- Style field of !X, !Y, !Z UG-64
- style, of axes R2-526, R2-531, R2-534, R2-557
- subarrays, structure of PG-86
- submatrices PG-98
- subscripts
 - * operator PG-85
 - combining arrays with PG-89–PG-90
 - notation for columns and rows PG-81
 - of pixels inside polygon region R1-550
 - ranges PG-56, PG-84
 - 4 types PG-84–PG-85
 - table of PG-86
 - using asterisk UG-36
 - reference syntax PG-82
 - storing elements in arrays PG-91
 - subscript arrays PG-53–PG-57
 - using arrays as PG-57
 - with
 - matrices PG-82
 - multidimensional arrays PG-82
 - scalars PG-84
 - structures PG-108
- subsetting
 - data
 - SORT function UG-290
 - Where clause UG-277
 - tables R2-1, UG-277
- substrings
 - extracting PG-131
 - from strings R2-154
 - inserting into strings R2-157–R2-158, PG-122, PG-131
 - locating in strings R2-156, PG-131
- subtraction operator (–)
 - examples PG-45
 - operator precedence PG-32
- SunOS Version 4, error handling PG-268
- superimposing
 - See overlaying
- surface data
 - in a file R2-87
 - volumetric R2-307, R2-329
- surface plot
 - adding skirt R2-345
 - angle of X rotation R2-497
 - angle of Z rotation R2-497
 - changing interactively R2-344
 - color of bottom R2-499
 - combining with image and contour plots R2-104
 - curve fitting to R2-172
 - display 3D array as R2-168
 - displaying data as R2-342
 - draw
 - a skirt on R2-518
 - horizontal lines only R2-508
 - lower surface only R2-509
 - upper surface only R2-523
 - easy way to change appearance R2-342, R2-344
 - example of combining with contour plot R2-517
 - placement of Z axis R2-533
 - rendering of shaded R1-564, R2-83, R2-91
 - rotating data UG-117
 - saving the 3D to 2D transformation R2-517
 - shaded R1-564, R2-345
 - shown with and without a skirt R2-346
 - superimposed with contours R2-517, UG-125
- SURFACE procedure
 - combining with CONTOUR procedure UG-125, UG-127
 - description R2-168, UG-93, UG-112
 - examples UG-112, UG-114, UG-119

- list of keywords UG-113
- transformation matrix R2-169
 - example UG-119
- SURFACE_FIT function R2-172
- SURFR procedure R2-174
- suspending PV-WAVE
 - on a UNIX system UG-14
 - on a VMS system UG-14
- SVBKSB procedure R2-177
- SVD procedure R2-179
- SVDFIT function
 - description R2-181
 - example of basis function for R1-137
- swap area, increasing PG-285
- symbol
 - displaying in a volume R2-263
 - plotting R2-547
 - VMS, defining R2-74
 - VMS, returnable R1-361
- symbols
 - connecting with lines R2-516, UG-72
 - marker UG-72, UG-73
 - plotting R2-516
 - size of plotting R2-520
 - user-defined R2-516, UG-73
 - VMS, deleting R1-238
- SYSGEN parameters
 - VirtualPageCount PG-286
 - Wsmax PG-286
- system
 - color tables UG-320
 - date, creating variable with R2-199
 - functions, using for efficient programming PG-280
 - time, creating variable with R2-184, R2-199
- system limits and compiler PG-241
- system variables
 - adding R1-236
 - creating R1-236
 - definition of PG-26
 - description of PG-5, UG-24
 - exclamation point UG-34
 - for tick label formats UG-82

- getting information about PG-403
- listing their values PG-403
- passing of PG-246
- relationship to keywords UG-24, UG-57
- summary of PG-27
- systems, window
 - See window systems
- SYSTIME function R2-184

T

- T3D procedure UG-116
 - data transformation UG-116, UG-118
 - description R2-185
 - examples UG-119
 - setting up data for viewing UG-194
- table functions
 - description of UG-263
 - overview UG-264
- table widget, creating PG-462
- tables
 - creating with BUILD_TABLE function R1-63, UG-266–UG-270
 - date/time data UG-281
 - determining unique values R2-230
 - GUI tool R2-472
 - in relation to structures UG-287–UG-288
 - plotting UG-286–UG-287
 - date/time axis UG-284
 - printing with column titles UG-285
 - rearranging with QUERY_TABLE UG-271
 - renaming columns UG-272
 - sorting R2-1
 - columns in descending order UG-276
 - with QUERY_TABLE function PG-179–PG-180
 - subsetting R2-1
 - with Where clause UG-277

- unique elements of R2-230
- viewing with INFO procedure
 - UG-268
- tabs in statements PG-52
- Tag Image File Format
 - See TIFF
- TAG_NAMES function R2-188, PG-119
- tags
 - definition of PG-101
 - names, of structures R2-188
 - number of R1-472
 - reference to structure PG-108
- TAN function R2-190
- tangent R2-190
 - hyperbolic R2-191
- TANH function R2-191
- tape, magnetic
 - accessing under VMS PG-229
 - mounting a tape drive (VMS)
 - PG-230
 - reading block of image data PG-231
 - REWIND procedure PG-229
 - rewinding R2-38
 - SKIPF procedure PG-229
 - skipping
 - backward on a tape PG-231
 - forward on the tape (VMS)
 - PG-230
 - TAPRD procedure PG-229
 - TAPWRT procedure PG-229
 - WEOF procedure PG-229
 - writing to PG-229
- TAPRD procedure R2-193, PG-229
- TAPWRT procedure R2-194, PG-229
- TEK_COLOR procedure R2-195,
 - UG-326, UG-330
- Tektronix 4115 device, mimicking colors
 - UG-330
- Tektronix 4510 rasterizer UG-A-61–
 - UG-A-64
- Tektronix terminal
 - color table R2-195
- Tektronix terminal output UG-A-65–
 - UG-A-68
- terminating, execution of a command file
 - UG-20

- text
 - 3D orientation of R2-521
 - See also annotation, fonts
 - alignment R2-497
 - angle of R2-513
 - centered R2-497
 - changing case R2-150, R2-166,
 - PG-127
 - character size R2-501, R2-518
 - color of R2-504
 - concatenation PG-123
 - converted from byte data PG-125
 - converting parameters to R2-168,
 - PG-124
 - editor programs UG-22
 - extracting PG-131
 - inserting PG-131
 - locating substrings PG-131
 - non-string arguments to string rou-
 - tines PG-133
 - obtaining length of strings R2-148,
 - PG-130
 - orientation of R2-513
 - positioning commands, for Post-
 - Script UG-A-39
 - removing white space PG-128
 - size of R2-501, R2-518
 - string operations supported PG-121
 - suppressing clipping R2-511
 - vector-drawn UG-291
 - viewed in scrolling window R2-349,
 - R2-351
 - width of R2-522
 - working with PG-121
 - writing to the display R2-490
- text widget
 - and popup menus R2-462
 - creating PG-443
 - description R2-481
 - multi-line PG-445
 - scroll bars PG-443
 - single line
 - editable PG-444
 - read-only PG-443

- thickness
 - of axis line R2-558
 - of characters R2-543
- THREED procedure R2-197
- three-dimensional
 - coordinate systems
 - converting to two-dimensional coordinate systems UG-119
 - user-defined UG-121
 - data, See three-dimensional data
 - graphics
 - combining with images UG-129–UG-130
 - drawing UG-115
 - rotating data UG-117
 - scaling data UG-117
 - shaded surfaces R1-564, R2-83, R2-91, R2-185
 - SHOW3 procedure R2-104
 - translating data UG-117
 - gridding R1-322, R1-367, UG-191
 - transformations
 - of text UG-292
 - with two-dimensional procedures UG-124
 - volumes UG-190
- three-dimensional data
 - contouring R1-119, UG-94
 - displayed as surface R2-168
 - MOVIE procedure R1-466
 - plotting UG-93
 - viewing as iso-surface R2-307
- threshold
 - dithering UG-148
 - value of an iso-surface R2-312
- thresholding of images UG-152
- tick
 - intervals, setting number of UG-66
 - label format UG-81–UG-82
 - marks
 - annotation of R2-527, R2-531, R2-535
 - controlling length of R2-522, R2-550, UG-78
 - extending away from the plot R2-522, UG-78
 - format of R2-527
 - intervals between R2-528, R2-532, R2-535
 - intervals, setting number of R2-560
 - linestyle of R2-507, R2-525, R2-529, UG-78
 - number of minor marks R2-525, R2-530, R2-534, R2-556, UG-78
 - producing non-linear marks UG-78
 - suppressing minor marks R2-556
 - values of R2-526, R2-530, R2-534
- TIFF
 - conformance levels PG-192
 - data
 - saving in TIFF format PG-191
 - writing image data to a file R1-221, R1-223
 - writing with DC_WRITE_TIFF PG-191
 - files
 - compression of PG-192
 - reading R1-198
 - reading a file in R1-198
 - writing image data to a file R1-221, R1-223
- time
 - current R2-184
 - elapsed between dates R1-277
 - formats, STR_TO_DT function UG-231
- timer, adding PG-489
- tips, programming PG-472, PG-498
- title
 - See also annotation
 - adding to two-dimensional plots UG-64
 - of axes R2-529, R2-532, R2-535, R2-560
 - of plot R2-523, R2-550
 - setting size of R2-501, R2-543
- TODAY function R2-199, UG-259

- toggle button, menu PG-430
- tool box R2-485, PG-433–PG-435
- top-level window
 - See shell window
- TOTAL function
 - description R2-200
 - example of PG-280
 - use of PG-47
- total, of array elements R2-200
- TQLI procedure R2-201
- traceback information PG-262
- transferring
 - binary data PG-154
 - binary string data PG-197
 - complex data, with XDR routines PG-210
 - data with
 - C or FORTRAN formats PG-165
 - READU, WRITEU PG-194
 - XDR files PG-206
 - date/time data PG-168
 - images in server UG-A-82
- transformation
 - 3D volumes R2-270
 - geometric UG-167–UG-168
 - gray level UG-152
 - matrix, See transformation matrices
 - saving R2-517
- transformation matrices R2-521
 - 3D points R1-573
 - 4-by-4 R2-270, R2-549
 - description UG-116
 - rotating data UG-117
 - scaling data UG-117
 - Transform keyword for RENDER function UG-202
 - translating data UG-117
 - with POLY_TRANS function R1-573
 - with SURFACE procedure UG-119
 - with T3D procedure UG-118
- translation table
 - bypassing UG-A-71, UG-A-85
 - color UG-308
- translucency, in images R1-561, R2-267
- transmission component of color, for RENDER function UG-199
- transparency, in images R1-543
- TRANPOSE function R2-204, PG-47
- transposing
 - arrays or images R2-51, R2-204
 - rows and columns PG-47
- TRED2 procedure R2-207
- TRIDAG procedure R2-208
- TRNLOG function R2-209, PG-295
- true
 - definition for IF statement PG-71
 - definitions for different data types PG-71
 - representation of PG-42
- true-color
 - compared to pseudo-color UG-146
 - images
 - definition UG-147
 - generating R1-406
- truth table
 - for AND operator PG-43
 - for OR operator PG-43
 - for XOR operator PG-43
- TV procedure
 - default coordinates for UG-59
 - description R2-212, UG-135
- TVCRS procedure R2-217
 - description UG-136, UG-143
 - example UG-143
- TVLCT procedure
 - description R2-219, UG-136, UG-311
 - examples UG-325
 - uses of UG-145
- TVMENU function PG-473
- TVRD function
 - description R2-223, UG-136, UG-142
 - examples UG-142
- TVSCL procedure
 - description R2-225, UG-136
 - examples UG-153
- 24-bit image data
 - writing data to a file R1-221
 - how stored PG-189

- 24-bit color, using color to differentiate data sets UG-A-93
 - two-dimensional
 - coordinate systems, converting from three-dimensional coordinate systems UG-119
 - data, summary of plotting routines UG-56
 - gridding R1-319, R1-364
 - 2D gridding UG-191
 - type conversion
 - byte R1-68
 - complex R1-109
 - double-precision R1-267
 - floating-point R1-340
 - functions
 - examples PG-37
 - overflow conditions PG-36
 - syntax PG-38
 - table of PG-36
 - integer R1-338
 - longword integer R1-442
 - scaled byte R1-73
 - string R2-144
 - type of variable, determining R2-114, PG-272
 - types
 - See data types
- U**
- undefined results
 - Infinity PG-263
 - NaN PG-263
 - undefined variables, checking for PG-270
 - unexpected colors UG-A-96
 - unformatted data
 - advantages and disadvantages of PG-151
 - FORTRAN generated, reading in UNIX environment PG-199
 - problems reading PG-205
 - reading with associated variable method PG-212
 - routines for transferring PG-156
 - VMS FORTRAN generated, example of reading PG-202
 - unformatted I/O
 - description PG-159
 - with associated variables PG-212
 - with structures PG-117
 - unformatted string variables, I/O PG-160
 - uniformly distributed random numbers R2-13
 - UNIQUE function R2-230, UG-7, UG-264
 - unique values, determining R2-230
 - UNIX
 - avoiding shells with SPAWN PG-300
 - calling from PV-WAVE R1-77
 - commands from within PV-WAVE R2-17, R2-33, R2-56, R2-64, R2-114, R2-126, PG-293–PG-297
 - CALL_UNIX PG-359
 - Control-\ UG-15
 - compiling for LINKNLOAD PG-322, PG-326, PG-328
 - description of files in PG-223
 - environment variable
 - adding R2-64
 - changing R2-64
 - inspecting R1-295
 - translation R1-356
 - WAVE_DEVICE UG-45
 - WAVE_DIR UG-45
 - WAVE_PATH UG-47
 - WAVE_STARTUP UG-49
 - environment, description of PG-292
 - FORTRAN programs
 - in relation to ASSOC function PG-218
 - writing to PV-WAVE PG-200
 - linking
 - C program to PV-WAVE PG-347
 - FORTRAN applications to PV-WAVE PG-348
 - offset parameter PG-216

- pipe, description of PG-310
- reading data in relation to FORTRAN
 - PG-199
- reserved LUNs PG-142
- sending output file to printer or plotter
 - UG-A-6
- shells PG-299
- virtual memory PG-285
- UNIX_LISTEN function R2-232
 - definition PG-371
 - example PG-392
 - PV-WAVE as a server PG-375
 - PV-WAVE as a server, example
 - PG-392
 - used with CALL_WAVE PG-371
- UNIX_REPLY function R2-234
 - example of PG-392, PG-394
 - PV-WAVE as a server PG-375,
 - PG-392
 - used with CALL_WAVE PG-371
- unmanage, shell widget PG-484
- unnamed structures
 - and variable-length fields PG-106
 - creating PG-105
 - internal names for PG-107
 - scope of PG-105
 - syntax of PG-106
 - to define menu items PG-429
 - uses for PG-105
 - specify resources PG-483
- unsharp masking UG-161
- updating a file, using OPENU PG-137
- user data, defining and passing PG-467
- user interface, graphical
 - See GUI
- Users' Library
 - documentation for PG-256
 - location of in PV-WAVE PG-255
 - submitting programs to PG-255
 - support for routines in PG-256
- USERSYM procedure R2-235, UG-73
- UT_VAR PG-387, PG-392

V

- VAR_TO_DT function
 - description of R2-238, UG-232
 - example of UG-232, UG-244–
 - UG-252
- variable length record format files, VMS
 - PG-226
- variables
 - associated R1-43, PG-24
 - in assignment statements
 - PG-59
 - attributes PG-23
 - checking for undefined PG-270
 - complex, importing data PG-160
 - creating
 - empty ones with date/time
 - type UG-227
 - new system variables R1-236
 - data types PG-24
 - declaring PG-5
 - definition of PG-4, PG-5, PG-23
 - deleting R1-244
 - determining number of elements in
 - PG-270
 - disappearing PG-251, PG-258
 - environment variables R1-410
 - erasing UG-27
 - forcing to specific data type PG-36
 - in common blocks PG-63
 - invalid names PG-26
 - local (definition of) PG-237
 - naming conventions PG-25
 - number of structure tags R1-472
 - obtaining environment variables
 - R1-295
 - passing PG-467
 - printing R1-580
 - reading R1-580
 - into from display device
 - UG-142
 - size and type information PG-24,
 - PG-272
 - storing in structures PG-112
 - structure of PG-24

- structure tag names R2-188
- system
 - See system variables
 - types of PG-24
 - valid names PG-26
 - virtual memory PG-284
- VAX/VMS Extended Metafile System UG-A-13
- vector
 - cross product R1-142
 - definition of PG-4, PG-24
 - fields, plotting R1-510, R2-244
 - from 3D arrays R2-240
 - of strings R2-149, R2-151, R2-167
 - reversing R2-36
 - solution, improving R1-468
 - subscripts PG-86
- vector graphics colors UG-311, UG-A-92
- VECTOR_FIELD3 procedure R2-240, UG-195
- vector-drawn text
 - See text
- vectors
 - BUILD_TABLE function UG-269
 - building tables from R1-63
 - definition of PG-24
 - deriving unique elements from R2-230
 - using as subscripts to other arrays PG-88
- VEL procedure R2-244
- VELOVECT procedure R2-248
- Ventura Publisher, inserting PV-WAVE plots into UG-A-47–UG-A-48
- version number of PV-WAVE R2-553, PG-29
- !Version system variable R2-553
- vertex lists
 - description of UG-187
 - merging R1-557
- video memory, of workstation UG-A-79
- View Control window R2-259
- View Orientation window R2-260
- view setup
 - 3D view R2-77
 - centering data R1-86
 - defining R2-253
 - description of UG-194
 - summary of routines R1-23
- VIEWER procedure R2-253, UG-179, UG-194
- viewport, defining R2-67, R2-79
- virtual memory
 - description PG-283
 - in relation to
 - PV-WAVE PG-283
 - swap area PG-285
 - minimizing use of PG-288
 - under UNIX PG-285
 - variable assignments PG-284
 - VMS PG-286
- VirtualPageCount parameter PG-286
- visual class(es)
 - for X windows UG-A-77
 - not inherited by PV-WAVE UG-A-90
- visual data analysis, importance of color UG-310
- VMS
 - access mode PG-225
 - accessing magnetic tape PG-229
 - binary CGM output UG-A-13
 - binary files PG-149
 - calling PV-WAVE R1-429
 - command interpreter PG-299
 - commands from within PV-WAVE PG-295–PG-297
 - environment variables PG-295–PG-296, UG-46
 - error handling PG-268
 - file
 - access PG-225
 - attributes PG-227
 - organization PG-224
 - formal parameters for procedures and functions PG-236
 - FORTRAN programs, writing to a file PG-202
 - libraries PG-251, PG-253

- linking a C program to PV-WAVE
 - PG-349
- logical names, deleting R1-240
- mounting a tape drive PG-230
- offset parameter PG-217
- record attributes PG-226
- reserved LUNs PG-142
- RMS files, reading images PG-217
- searching libraries PG-252
- sending output file to printer or plotter
 - UG-A-6
- specific information on data files
 - PG-224
- stream mode files PG-226
- symbols
 - defining R2-74
 - deleting R1-238
 - return value R1-361
- transferring record-oriented data
 - PG-150
- variable length format PG-226
- virtual memory PG-286
- working set size PG-286
- VOL_MARKER procedure R2-263,
 - UG-195
- VOL_PAD function R2-266, UG-192
- VOL_REND function R2-267, UG-195
- VOL_TRANS function R2-270, UG-193
- VOLUME function
 - description of R2-272
 - example of UG-211–UG-218
- volumes
 - defining R2-253
 - with VOLUME function
 - R2-272, UG-197
 - displaying markers in R2-263
 - generating UG-187
 - manipulating R1-24, UG-192
 - padding R2-266
 - rendering R1-24, R2-28, R2-240,
 - R2-263, R2-267, UG-183–
 - UG-195
 - shaded R2-96

- slicing
 - example of UG-210
 - with SLICE_VOL function
 - R2-117
 - transforming R2-270
 - volumetric surface data R2-307, R2-329
 - VT graphics terminals UG-A-54

W

- w_cmpnd_reply function PG-364,
 - PG-368
- w_get_par function PG-366
- w_listen function PG-365
- w_send_reply function PG-364, PG-367
- w_simpl_reply function PG-364
- w_smpl_reply function PG-367
- WAIT procedure R2-274
- waiting, in programs R2-274
- warping of images R1-526, R1-574,
 - UG-168
- WAVE Widgets
 - application example PG-473
 - arranging PG-420
 - basic steps in creating PG-413
 - combining with Widget Toolbox
 - PG-480
 - creating PG-416
 - event loop PG-471
 - getting values PG-466
 - hiding PG-469
 - initializing PG-414
 - introduction PG-410
 - list of PG-418
 - location of PG-411
 - passing user-defined data PG-467
 - sensitivity PG-470
 - setting
 - colors PG-463
 - fonts PG-464
 - values PG-466
 - showing PG-469
 - submitting to Users' Library PG-411
 - widget hierarchy PG-416
- WAVE_DEVICE UG-44–UG-45

WAVE_DIR UG-45–UG-46
 WAVE_PATH UG-46–UG-47
 WAVE_STARTUP UG-49–UG-50
 wavecmd routine
 description of PG-313
 examples PG-315, PG-316,
 PG-318
 when to use PG-307
 with FORTRAN programs PG-315
 waveinit routine PG-313–PG-314
 wavestartup file UG-49
 waveterm routine PG-313, PG-315
 wavevars function
 accessing data in PV-WAVE
 PG-350
 description of PG-321, PG-335
 examples PG-354–PG-358
 parameters PG-350
 syntax PG-321, PG-350
 WDELETE procedure R2-275, UG-A-98
 WEOF procedure R2-276, PG-229
 WgAnimateTool procedure
 description R2-277
 event handling R2-281
 WgCbarTool procedure
 description R2-284
 event handling R2-287
 stores colors in common block
 R2-288
 WgCeditTool procedure
 benefits of R2-291
 compared to WgCtTool procedure
 R2-291
 description R2-290
 event handling R2-299
 stores colors in common block
 R2-293
 utility widgets, shown in figure
 R2-298
 WgCtTool procedure
 compared to WgCeditTool R2-302
 description R2-301
 event handling R2-305
 stores colors in common block
 R2-303
 WgIsoSurfTool procedure
 description R2-307
 event handling R2-312
 WgMovieTool procedure
 benefits of R2-319
 description R2-315
 event handling R2-320
 WgSimageTool procedure
 description R2-323
 event handling R2-327
 WgSliceTool procedure
 description R2-329
 event handling R2-334
 shown in figure R2-332
 slices three-dimensional data
 R2-329
 WgStripTool procedure
 description R2-336
 event handling R2-340
 WgSurfaceTool procedure
 benefits of R2-344
 description R2-342
 event handling R2-347
 WgTextTool procedure
 description R2-349
 event handling R2-352
 Where clause UG-277
 WHERE function R2-354, PG-55,
 PG-58, PG-277, UG-A-23
 WHILE statement PG-10, PG-78
 white space
 compressing R2-141
 removing R2-140, R2-162
 removing from strings PG-122,
 PG-128
 widget
 class PG-482
 definition of PG-410, PG-482
 destroy PG-485
 hierarchy
 See widget hierarchy
 ID (handle) PG-418, PG-482
 input focus of PG-484
 managing PG-484
 mapped with an X window PG-484
 realizing PG-484

- resources PG-483
- resources, retrieving R2-376
- sensitivity PG-470
- setting resources PG-465
- unmanaging PG-484
- widget hierarchy
 - close PG-485
 - definition PG-416
 - displaying PG-471
- Widget Toolbox
 - adding
 - callbacks PG-485
 - event handlers R2-367, PG-487
 - timers PG-490
 - combining with WAVE Widgets PG-480
 - creating widgets PG-482
 - cursors PG-D1—PG-D5
 - destroying widgets PG-484
 - displaying widgets PG-484
 - example program PG-495
 - include files PG-495
 - initializing PG-481
 - managing widgets PG-484
 - resources PG-483
 - running an application PG-494
- window
 - creating R2-356
 - current R2-362
 - damage repair UG-A-7
 - deleting by ID number UG-A-98
 - exposing and hiding R2-363
 - looks dark and unreadable UG-A-82
 - margin around plot R2-525, R2-530, R2-534
 - menus, creating R2-221, R2-359
 - positioning plot in R2-67, R2-79
 - selecting R2-362
 - specifying plot coordinates R2-515
 - turning into icon R2-363
- window index, used with drawing area PG-439
- WINDOW procedure R2-356
- window systems
 - backing store UG-A-7
 - common features UG-A-6
 - defining with WAVE_DEVICE UG-44—UG-45
 - features of those supported UG-A-6
 - retained UG-A-7
 - table of UG-A-2
 - X Windows UG-53, UG-A-69
- WMENU function R2-359
- word-processing application, reading data from PG-175
- words, reserved PG-25
- working set
 - description PG-283
 - quota PG-287
 - VMS PG-286
- write mask
 - interacts with selected graphics function UG-A-95
 - using to create special effects UG-328, UG-A-94
- WRITEU procedure
 - description R2-361, PG-154
 - example PG-195
 - syntax PG-193
 - transferring strings with PG-197
- writing
 - 24-bit image data to a file R1-221
 - 8-bit image data to a file R1-219
 - ASCII free format data to a file R1-203, R1-212
 - data, output formats PG-164
 - date/time data UG-255
 - integer data, using format reversion PG-A-5—PG-A-6
 - strings to a file PG-164
 - TIFF image data to a file R1-223
 - to a data file, basic steps PG-12
 - to end of file on tape (VMS) PG-229
 - to tape (VMS) PG-229
 - using
 - UNIX FORTRAN programs PG-200
 - VMS FORTRAN PG-202
 - WRITEU PG-195

WSET procedure R2-362
 WSHOW procedure R2-363
 Wsquo quota PG-287
 WtAddCallback function R2-364,
 PG-485
 WtAddHandler function R2-367,
 PG-487
 WtClose function R2-369, PG-485
 WtCreate function R2-371, PG-480,
 PG-482
 WtCursor function R2-373
 WtGet function R2-376
 WtInit function R2-380, R2-432, PG-480
 WtInput function R2-382
 WtList function R2-385
 WtLoop function R2-387, PG-494
 WtMainLoop function R2-389
 WtPointer function R2-390
 WtSet function R2-392, PG-480
 WtTable function R2-395
 WtTimer function R2-402, PG-489
 WtWorkProc unction R2-404
 wsetup (WVSETUP.COM) file
 methods of executing UG-12
 WwButtonBox function R2-406, PG-434
 WwCommand function R2-410, PG-459
 WwControlsBox function R2-414,
 PG-437
 WwDialog function R2-419, PG-453
 WwDrawing function R2-422, PG-439
 WwFileSelection function R2-426,
 PG-456
 WwGetValue function PG-466
 description R2-430
 WwInit function PG-413—PG-414
 WwLayout function R2-434, PG-420
 WwList function R2-439, PG-446
 WwLoop function R2-443, PG-414
 WwMainWindow function R2-444,
 PG-416
 WwMenuBar function R2-447, PG-426
 WwMenuItem function R2-451
 WwMessage function R2-453, PG-449
 WwOptionMenu function R2-457,
 PG-428

WwPopupMenu function R2-461,
 PG-427
 WwRadioBox function R2-464, PG-436
 WwSetValue function R2-468, PG-466
 WwTable function R2-472
 WwText function R2-481, PG-443
 WwToolBox function R2-485, PG-435

X

X server
 closing connection UG-A-72
 connecting to PG-414
 !X system variable, fields of R2-554—
 R2-560
 X Window System
 \$DISPLAY environment variable
 UG-A-69
 clients UG-A-69
 color translation UG-A-84
 damage repair UG-A-7
 DECW\$DISPLAY logical UG-A-69
 DEVICE procedure UG-A-71
 graphics function codes UG-A-75
 IDs UG-A-97
 keywords UG-A-71
 overview UG-A-69
 pixmaps UG-A-85—UG-A-86
 private colormaps UG-A-82
 providing a GUI for application
 UG-A-70
 resources PG-464
 servers UG-A-69
 shared colormaps UG-A-81
 using with PV-WAVE UG-53
 visual classes UG-A-77
 X11 socket R2-61
 .Xdefaults file PG-465
 XDR
 data
 allowed I/O routines PG-205
 byte variables PG-206
 differences from normal I/O
 PG-205

- string variables PG-206
- transferring PG-206
- files
 - conventions for reading and writing PG-210
 - creating with C programs PG-208
 - description of PG-205
 - opening PG-205
 - reading, byte data PG-206, PG-207
 - reading, READU procedure PG-209
- routines
 - for transferring complex data PG-210
 - table of PG-210
- Xlib PG-406
- XOR operator
 - description of PG-50
 - example UG-A-95
 - operator precedence PG-33
 - truth table PG-43
- Xt Intrinsic PG-406, PG-414, PG-481
- Xt TimeOut function PG-489
- XY plots
 - examples UG-62
 - scaling axes UG-63—UG-64
- XYOUTS procedure R2-490, UG-69, UG-121

Y

- Y axis, scaling with YNozero UG-64
- !Y system variable, fields of R2-561

Z

- !Z system variable, fields of R2-561
- Z-buffer
 - output
 - description UG-A-99
 - DEVICE procedure UG-A-99
 - keywords UG-A-99
 - using to create special effects R1-543, UG-A-99

