HP VEE Advanced Programming Techniques

Notice

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

Hewlett-Packard Company (HP) shall not be liable for any errors contained in this document. *HP makes no warranties of any kind with regard to this document, whether express or implied. HP specifically disclaims the implied warranties of merchantability and fitness for a particular purpose.* HP shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory, in connection with the furnishing of this document or the use of the information in this document.

Warranty Information.

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Printing History.

Edition 1 - August 1993

© Copyright 1991, 1992, 1993 Hewlett-Packard Company. All rights reserved.

This document contains information which is protected by copyright. All rights are reserved. Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

UNIX® is a registered trademark of UNIX System Laboratories in the U.S.A. and other countries.

Microsoft® and MS-DOS® are registered trademarks of Microsoft Corporation.

Sun, SunOS and Openwindows are trademarks of Sun Microsystems, Inc.

SPARCstation is a registered trademark of SPARC International, Inc.

Conventions Used in this Manual

This manual uses the following typographical conventions:

Example	Represents
Installing HP VEE	Italicized words are used for book titles and for emphasis.
`File`	Computer font represents text you will see on the screen, including menu names, features, or text you have to enter.
`cat` *filename*	In this context, the word in computer font represents text you type exactly as shown, and the italicized word represents an argument that you must replace with an actual value.
`File` ⟹ `Open`	Features separated with the arrow indicate the order of selection from a menu.
`Zoom Out | In 2x | In 5x`	Choices in computer font, separated with a bar (|), indicates that you should choose one of the options.
Return	The keycap font graphically represents a key on the computer's keyboard.
Press CTRL-O	Dash-separated keys represent a combination of keys on the computer's keyboard that you should press at the same time.
Dialog Box	Bold font indicates the first instance of a word defined in the glossary.

Contents

1. Introduction

About This Manual . . . 1-2
What is HP VEE? . . . 1-3
HP VEE Example Programs . . . 1-5
Examples Directories . . . 1-5
Running Examples . . . 1-5
UNIX Systems . . . 1-5
Windows Systems . . . 1-6

2. Global Variables

Using Global Variables . . . 2-2
Using Global Variables in UserObjects . . . 2-6

3. Using Instruments

Instrument Control Fundamentals . . . 3-3
Driver-Based Objects . . . 3-4
Instrument Panels . . . 3-4
Component Drivers . . . 3-5
Direct I/O . . . 3-6
Summary of Instrument Control Objects . . . 3-8
Terminating I/O Operations (UNIX) . . . 3-8
Terminating I/O Operations (PC) . . . 3-9
Using Instrument Control Examples . . . 3-10
Understanding Instrument Panels and Component Drivers . . . 3-11
Inside HP Drivers . . . 3-11
Driver Files . . . 3-11
Components . . . 3-11
States . . . 3-13
How Driver-Based I/O Works . . . 3-14
Instrument Panel Operation . . . 3-15
Component Driver Operation . . . 3-15
Multiple Driver Objects . . . 3-16
The Importance of Names . . . 3-16
Reusing Driver Files . . . 3-17
Choosing the Correct Instrument Object . . . 3-18
Configuring Instruments . . . 3-19

Basic Instrument Configuration . . . 3-19
Driver Configuration . . . 3-22
Direct I/O Configuration . . . 3-23
A16 Space Configuration (VXI only) . . . 3-24
A24/A32 Space Configuration (VXI only) . . . 3-25
Details of Configure I/O Dialog Boxes . . . 3-26
Device Configuration Dialog Box . . . 3-26
Name . . . 3-26
Interface . . . 3-27
Address . . . 3-27
HP-IB Address Examples . . . 3-28
VXI Address Examples . . . 3-28
Serial Address Examples . . . 3-28
GPIO Address Example . . . 3-29
Device Type . . . 3-29
Timeout . . . 3-29
Byte Ordering . . . 3-29
Live Mode . . . 3-29
Config Buttons . . . 3-30
Instrument Driver Configuration Dialog Box . . . 3-30
ID Filename . . . 3-31
Sub Address . . . 3-31
Incremental Mode . . . 3-32
Error Checking . . . 3-32
Direct I/O Configuration Dialog Box . . . 3-33
Read Terminator . . . 3-33
EOL Sequence . . . 3-34
Multi-field As . . . 3-34
Array Separator . . . 3-35
Array Format . . . 3-35
Writing Arrays with Direct I/O . . . 3-36
END On EOL (HP-IB Only) . . . 3-37
Conformance . . . 3-37
Binblock . . . 3-37
State . . . 3-38
Upload String . . . 3-38
Download String . . . 3-38
Serial Interface Settings . . . 3-38
Data Width (GPIO only) . . . 3-39
A16 Space Configuration Dialog Box (VXI Only) . . . 3-40
Byte Access . . . 3-40

Word Access . . . 3-40
LongWord Access . . . 3-40
Add Register . . . 3-40
Delete Register . . . 3-41
A24/A32 Space Configuration Dialog Box (VXI only) . . . 3-42
Byte Access . . . 3-42
Word Access . . . 3-42
LongWord Access . . . 3-42
Byte Ordering . . . 3-43
Add Location . . . 3-43
Delete Location . . . 3-44
Example of Configuring a VXI Device . . . 3-44
Advanced Topic - I/O Configuration File . . . 3-46
Sharing Programs . . . 3-47
Running Example Programs . . . 3-47
Using Instrument Panels . . . 3-49
Using Instrument Panels Interactively . . . 3-49
Using Instrument Panels Programmatically . . . 3-50
Using Component Drivers . . . 3-51
Using Component Drivers in a Program . . . 3-51
Advanced I/O Control . . . 3-53
Polling . . . 3-53
Service Requests . . . 3-55
Monitoring Bus Activity . . . 3-57
Low-level Bus Control . . . 3-59
Instrument Downloading . . . 3-59
Example of Downloading . . . 3-60
Troubleshooting . . . 3-63
Related Reading . . . 3-65

4. Using Records and DataSets
Record Containers . . . 4-3
Accessing Records . . . 4-5
Building Records . . . 4-8
Editing Record Fields . . . 4-10
Building Records Containing Waveforms . . . 4-12
Using Global Records . . . 4-14
Using DataSets . . . 4-16

5. Creating User-Defined Functions

User Functions 5-3
Creating a User Function 5-3
Editing a User Function 5-6
Calling a User Function from an Expression 5-11
Creating a User Function Library 5-12
Compiled Functions 5-17
Design Considerations for Compiled Functions 5-17
Importing and Calling a Compiled Function 5-19
Creating a Compiled Function (UNIX) 5-21
The Definition File 5-21
Building a C Function 5-22
Creating a Shared Library 5-26
Binding the Shared Library 5-27
Creating a Dynamic Linked Library (MS-Windows) 5-28
Creating the DLL 5-28
Declaring DLL Functions 5-28
Creating the Definition File 5-29
Examples 5-29
Parameter Limitations 5-29
veedata type 5-29
The IMPORT LIBRARY Object 5-30
The CALL FUNCTION Object 5-30
The DELETE LIBRARY Object 5-31
Using DLL Functions in Formula Objects 5-31
Remote Functions (UNIX) 5-32
UNIX Security, UIDs, and Names 5-35
The .veeio and .veerc files 5-36
Timeouts 5-37
Errors 5-37

6. Using Transaction I/O

Using Transactions 6-3
Creating and Editing Transactions 6-5
Editing the Data Field 6-6
Adding Terminals 6-9
Reading Data 6-10
Transactions that Read a Specified Number of Data Elements 6-11
Read-To-End Transactions 6-13
Non-blocking Reads 6-15

Suggestions for Experimentation . . . 6-18
Details About Transaction-Based Objects . . . 6-19
Execution Rules . . . 6-19
Object Configuration . . . 6-19
EOL Sequence . . . 6-21
Multi-field As . . . 6-21
Array Separator . . . 6-21
Array Format . . . 6-22
READ and WRITE Compatibility . . . 6-23
Choosing the Correct Transaction . . . 6-24
Selecting the Correct Object and Transaction . . . 6-26
Example of Selecting an Object and Transaction . . . 6-27
Using To String and From String . . . 6-28
Communicating with Files . . . 6-29
Details About File Pointers . . . 6-29
Read Pointers . . . 6-30
Write Pointers . . . 6-30
Closing Files . . . 6-30
The EOF Data Output . . . 6-32
Common Tasks for Importing Data . . . 6-34
Importing X-Y Values . . . 6-34
Importing Waveforms . . . 6-36
Fixed-Format Header . . . 6-37
Variable-Format Header . . . 6-39
Communicating with Programs (UNIX) . . . 6-41
Execute Program (UNIX) . . . 6-41
Execute Program (UNIX) Fields . . . 6-42
Shell . . . 6-42
Wait for Prog Exit . . . 6-43
Pgm With Params . . . 6-43
Running a Shell Command . . . 6-44
Running a C Program . . . 6-46
To/From Named Pipe (UNIX) . . . 6-47
Hints for Using Named Pipes . . . 6-48
HP BASIC/UX Objects (HP VEE-Test, Series 300/400 Only) . . . 6-49
Init HP BASIC/UX . . . 6-49
To/From HP BASIC/UX . . . 6-50
Examples Using To/From HP BASIC/UX . . . 6-50
Sharing Scalar Data . . . 6-50
Sharing Array Data . . . 6-51
Sharing Binary Data . . . 6-52

Communicating with Programs (PC) . . . 6-53
Execute Program (PC) . . . 6-53
Execute Program (PC) Fields . . . 6-54
Pgm With Params . . . 6-54
Wait for Prog Exit . . . 6-54
Using Dynamic Data Exchange (DDE) . . . 6-54
`To/From DDE` Object . . . 6-56
DDE Examples . . . 6-58
Dynamic Linked Libraries (DLL) . . . 6-64
Communicating with Instruments . . . 6-65
Direct I/O . . . 6-66
Sending Commands . . . 6-66
WRITE TEXT Transactions . . . 6-66
WRITE BINBLOCK Transactions . . . 6-67
WRITE STATE Transactions . . . 6-67
Learn String Example . . . 6-68
Reading Data . . . 6-69
READ TEXT Transactions . . . 6-70
Interface Operations . . . 6-71
Related Reading . . . 6-74

7. Using the Sequencer Object
Sequence Transactions . . . 7-3
Logging Test Results . . . 7-9
Logging to a DataSet . . . 7-12
Some Restrictions in Logging Test Results . . . 7-13
A Practical Test Example . . . 7-15

8. Troubleshooting Problems

A. Configuring HP VEE
Changing X11 Attributes (UNIX) . . . A-3
Configuring HP VEE for Windows . . . A-4
General HP VEE Settings . . . A-4
Color Settings . . . A-5
Fonts . . . A-9
Using Multiple Color Sets/Fonts (UNIX) . . . A-11
Customizing Icon Bitmaps (UNIX & Windows) . . . A-13
Selecting a Bitmap for a Panel View . . . A-14
If You See Colors Changing On Your Screen (UNIX) . . . A-15
Too Many Colors . . . A-15

Applications that Use a Local Color Map (UNIX) A-16
Using Non-USASCII Keyboards (UNIX) A-19
Using HP-GL Plotters (UNIX Only) A-21
Using Two-Byte Character Sets (UNIX Only) A-22

B. Example Programs and Library Objects
Using the Examples B-3
Using Library Objects B-4

C. ASCII Table

D. HP VEE Utilities
The veedoc Utility for Documenting Programs D-3
The HP Driver Writer Tool D-6
The HP Instrument Driver Compiler D-7
The Instrument Finder (MS-Windows Only) D-8
Install Drivers (PC) D-9
Configure I/O Utility D-10

E. I/O Transaction Reference
WRITE Transactions E-4
Path-Specific Behaviors E-4
TEXT Encoding E-6
DEFAULT Format E-7
STRING Format E-9
Field Width and Justification E-9
Number of Characters E-11
Writing Arrays with Direct I/O E-12
QUOTED STRING Format E-13
Field Width and Justification E-13
Number of Characters E-15
Embedded Control and Escape Characters E-16
INTEGER Format E-18
Number of Digits E-18
Sign Prefixes E-20
OCTAL Format E-21
Number of Digits E-21
Octal Prefixes E-21
HEX Format E-23
Hexadecimal Prefixes E-23
REAL Format E-25

Notations and Digits E-25
COMPLEX, PCOMPLEX, and COORD Formats E-27
COMPLEX Format E-28
PCOMPLEX Format E-29
COORD Format E-30
TIME STAMP Format E-31
BYTE Encoding E-33
CASE Encoding E-34
BINARY Encoding E-35
BINBLOCK Encoding E-37
Non-HP-IB BINBLOCK E-37
HP-IB BINBLOCK E-38
CONTAINER Encoding E-39
STATE Encoding E-40
REGISTER Encoding E-41
MEMORY Encoding E-42
IOCONTROL Encoding E-43
READ Transactions E-45
TEXT Encoding E-47
General Notes for READ TEXT E-48
Read to End E-48
Number of Characters Per READ E-49
Effects of Quoted Strings E-50
CHAR Format E-52
TOKEN Format E-53
SPACE DELIM E-54
INCLUDE CHARS E-55
EXCLUDE CHARS E-57
STRING Format E-58
Effects of Control and Escape Characters E-59
QUOTED STRING Format E-60
Effects of Control and Escape Characters E-61
INTEGER Format E-63
OCTAL Format E-65
HEX Format E-66
REAL Format E-68
COMPLEX, PCOMPLEX, and COORD Formats E-71
COMPLEX Format E-71
PCOMPLEX Format E-71
COORD Format E-72
BINARY Encoding E-73

BINBLOCK Encoding E-75
CONTAINER Encoding E-76
REGISTER Encoding E-77
MEMORY Encoding E-78
IOSTATUS Encoding E-79
EXECUTE Transactions E-81
Details About HP-IB E-84
Details About VXI E-86
WAIT Transactions E-88
SEND Transactions E-90
WRITE(POKE) Transactions E-92
READ(REQUEST) Transactions E-93

F. HP VEE-Test and HP VEE for Windows Differences
I/O F-3
Execute Program F-4
DLL versus Shared Library F-5
Data Files F-6
Memory Usage F-7

G. HP VEE for Windows Instrument I/O Select Codes
I/O Select Code Map G-2

Glossary

Index

Figures

2-1. A Simple Global Variable Example 2-2
2-2. Accessing an Undefined Global Variable 2-3
2-3. Accessing a Global Variable Multiple Times 2-4
2-4. Waveform Data in a Global Variable 2-5
2-5. Retrieving a Global Variable in a UserObject 2-7
2-6. Retrieving Multiple Global Variables in a UserObject 2-8
3-1. Instrument Control Objects 3-3
3-2. Two `Instrument Panels` 3-5
3-3. Combining `Instrument Panels` and `Component Drivers` . 3-6
3-4. Combining `Instrument Panels` and `Direct I/O` 3-7
3-5. Default I/O Configuration 3-10
3-6. Accessing Driver Components 3-13
3-7. Two Voltmeter States 3-13
3-8. A16 Configuration for the HP E1411B Multimeter 3-45
3-9. A Typical Component Driver 3-51
3-10. Using Instrument Panels and Component Drivers 3-52
3-11. Device Event Configured for Serial Polling 3-54
3-12. Handling Service Requests 3-57
3-13. The Bus I/O Monitor 3-58
3-14. Two Methods of Low-Level HP-IB Control 3-59
3-15. Downloading To An Instrument 3-62
4-1. A Simple Record Container 4-3
4-2. Retrieving Record Fields with Get Field 4-5
4-3. Using Array Syntax in Get Field 4-6
4-4. Retrieving Record Fields with UnBuild Record 4-7
4-5. The Effect of Output Shape in Build Record 4-9
4-6. Mixing Scalar and Array Input Data 4-10
4-7. Using Set Field to Edit a Record 4-11
4-8. Building a Record from Waveform Data 4-12
4-9. Using a Global Record 4-14
4-10. Using To DataSet to Save a Record 4-16
4-11. Using From DataSet to Retrieve a Record 4-17
5-1. Program with UserObject 5-4
5-2. UserObject Replaced by Call Function 5-5
5-3. Editing a User Function 5-7
5-4. The Edited User Function 5-8
5-5. Program Using Edited User Function 5-9

5-6. Using Multiple Call Function Objects . . . 5-10
5-7. Calling a User Function from Expressions . . . 5-11
5-8. Creating UserObjects for a User Function Library . . . 5-13
5-9. Importing a User Function Library . . . 5-14
5-10. Importing and Deleting a User Function Library . . . 5-15
5-11. Using Import Library for Compiled Functions . . . 5-19
5-12. Using Call Function for Compiled Functions . . . 5-20
5-13. Program Calling a Compiled Function . . . 5-25
5-14. Import Library for Remote Functions . . . 5-33
6-1. Default Transaction in `To String` . . . 6-3
6-2. A Simple Program Using `To String` . . . 6-4
6-3. Editing the Default Transaction in `To String` . . . 6-6
6-4. The Data Field . . . 6-7
6-5. Terminals Correspond to Variables . . . 6-9
6-6. Select Read Dimension Dialog Box . . . 6-11
6-7. Transaction Dialog Box for Multi-Dimensional Read . . . 6-12
6-8. Transaction Dialog Box for Multi-Dimensional Read-To-End . 6-14
6-9. Using `READ IOSTATUS DATAREADY` for a Non-Blocking Read . 6-17
6-10. Experimenting with `To String` . . . 6-18
6-11. The `Format Configuration` Dialog Box . . . 6-20
6-12. Using the `EXECUTE CLOSE` Transaction . . . 6-31
6-13. Typical Use of EOF to Read a File . . . 6-33
6-14. Importing `XY` Values . . . 6-35
6-15. Importing a Waveform File . . . 6-38
6-16. Importing a Waveform File . . . 6-40
6-17. The `Execute Program (UNIX)` Object . . . 6-42
6-18. `Execute Program (UNIX)` Running a Shell Command . . . 6-44
6-19. `Execute Program (UNIX)` Running a Shell Command using Read-To-End . . . 6-45
6-20. `Execute Program` Running a C Program . . . 6-46
6-21. C Program Listing . . . 6-47
6-22. `To/From HP BASIC/UX` Settings . . . 6-51
6-23. The `Execute Program (PC)` Object . . . 6-53
6-24. The `To/From DDE` Object . . . 6-56
6-25. The `To/From DDE` Example . . . 6-56
6-26. Execute PC before To/From DDE . . . 6-57
6-27. I/O Terminals and To/From DDE . . . 6-58
6-28. Lotus 123 DDE Example . . . 6-59
6-29. HP ITG DDE Example . . . 6-60
6-30. Instrument BASIC for Windows DDE Example . . . 6-61
6-31. Excel DDE Example . . . 6-62

6-32. Reflections DDE Example 6-62
6-33. Word for Windows DDE Example 6-63
6-34. WordPerfect DDE Example 6-63
6-35. Configuring For Learn Strings 6-69
7-1. A Simple Sequencer Program 7-3
7-2. Running the Program 7-6
7-3. A Logged Record of Records 7-8
7-4. A Simple Logging Example 7-9
7-5. A Logged Array of Records of Records 7-10
7-6. Analyzing the Logged Test Results 7-11
7-7. Logging to a DataSet 7-12
7-8. Simple Bin Sort Example 7-16
7-9. Improved Bin Sort Example 7-19
A-1. Color Map File Using Words A-17
A-2. Color Map File Using Hex Numbers A-17
E-1. A `WRITE TEXT` Transaction E-8
E-2. Numeric Data E-8
E-3. Two `WRITE TEXT STRING` Transactions E-9
E-4. Two `WRITE TEXT STRING` Transactions E-10
E-5. A `WRITE TEXT STRING` Transaction E-11
E-6. Two `WRITE TEXT STRING` Transactions E-11
E-7. Two `WRITE TEXT QUOTED STRING` Transactions E-13
E-8. Two `WRITE TEXT QUOTED STRING` Transactions E-14
E-9. A `WRITE TEXT QUOTED STRING` Transaction E-15
E-10. Two `WRITE TEXT QUOTED STRING` Transactions E-15
E-11. A `WRITE TEXT QUOTED STRING` Transaction E-17
E-12. Two `WRITE TEXT INTEGER` Transactions E-19
E-13. A `WRITE TEXT INTEGER` Transaction E-19
E-14. Two `WRITE TEXT INTEGER` Transactions E-20
E-15. A `WRITE TEXT OCTAL` Transaction E-22
E-16. A `WRITE TEXT OCTAL` Transaction E-22
E-17. A `WRITE TEXT HEX` Transaction E-24
E-18. A `WRITE TEXT HEX` Transaction E-24
E-19. Three `WRITE TEXT REAL` Transactions E-26
E-20. Three `WRITE TEXT REAL` Transactions E-26
E-21. Three `WRITE TEXT REAL` Transactions E-27
E-22. A `WRITE TEXT COMPLEX` Transaction E-28
E-23. Two `WRITE TEXT PCOMPLEX` Transactions E-29
E-24. A `WRITE TEXT PCOMPLEX` Transaction E-30
E-25. Two `WRITE BYTE` Transactions E-33
E-26. Character Data E-33

E-27. Two WRITE CASE Transactions E-34
E-28. Quoted and Non-Quoted Data E-51
E-29. Data for READ TOKEN E-54
E-30. Data for READ TOKEN E-56
E-31. Data for READ TOKEN E-58
E-32. String Data E-59
E-33. String Data E-61
E-34. Octal Data E-65
E-35. Octal Data E-66
E-36. Hexadecimal Data E-67
E-37. Real Data E-68
E-38. Example of Real Notations E-70
E-39. Data Containing Parentheses E-72

Tables

3-1. Instrument I/O Support . . . 3-2
3-2. Instrument Control Objects . . . 3-8
3-3. Escape Characters . . . 3-34
6-1. Editing Transactions With A Mouse . . . 6-5
6-2. Editing Transactions With the Keyboard . . . 6-5
6-3. Typical Data Field Entries . . . 6-7
6-4. Escape Characters . . . 6-8
6-5. Summary of Transaction-Based Objects . . . 6-25
6-6. Summary of Transaction Types . . . 6-26
6-7. Summary of `EXECUTE` Commands (`Interface Operations`) . 6-72
6-8. SEND Bus Commands . . . 6-73
8-1. Problems, Causes, and Solutions . . . 8-2
E-1. Summary of Transaction Types . . . E-2
E-2. Summary of I/O Transaction Objects . . . E-3
E-3. `WRITE` Encodings and Formats . . . E-5
E-4. Formats for `WRITE TEXT` Transactions . . . E-7
E-5. Escape Characters . . . E-17
E-6. Sign Prefixes . . . E-20
E-7. Octal Prefixes . . . E-21
E-8. Hexadecimal Prefixes . . . E-23
E-9. `REAL` Notations . . . E-25
E-10. `PCOMPLEX` Phase Units . . . E-29
E-11. `READ` Encodings and Formats . . . E-45
E-12. Formats for `READ TEXT` Transactions . . . E-47
E-13. Suffixes for `REAL` Numbers . . . E-70
E-14. `IOSTATUS` Values . . . E-80
E-15. Summary of `EXECUTE` Commands . . . E-81
E-16. `EXECUTE ABORT` HP-IB Actions . . . E-84
E-17. `EXECUTE CLEAR` HP-IB Actions . . . E-84
E-18. `EXECUTE TRIGGER` HP-IB Actions . . . E-84
E-19. `EXECUTE LOCAL` HP-IB Actions . . . E-85
E-20. `EXECUTE REMOTE` HP-IB Actions . . . E-85
E-21. `EXECUTE LOCAL LOCKOUT` HP-IB Actions . . . E-85
E-22. `EXECUTE CLEAR` VXI Actions . . . E-87
E-23. `EXECUTE TRIGGER` VXI Actions . . . E-87
E-24. `EXECUTE LOCAL` VXI Actions . . . E-87
E-25. `EXECUTE REMOTE` VXI Actions . . . E-87

E-26. SEND Bus Commands E-90
G-1. HP VEE for Windows I/O Select Codes G-2

Contents

1

Introduction

Introduction

About This Manual

This manual gives detailed information on using advanced features of HP VEE for tasks that you may want to perform. If you are new to HP VEE, you may want to start by working through the examples in *Getting Started with HP VEE*. You will also find that *Using HP VEE* contains useful introductory materials. For reference information on specific HP VEE features, refer to the *HP VEE Reference* manual. If you haven't already installed HP VEE, refer to *Installing HP VEE* on HP-UX and Sun platforms. See *Getting Started with HP VEE for Windows* for installation instructions for MS-DOS platforms.

This manual is meant to be used as needed, rather than read from beginning to end.

What is HP VEE?

HP VEE is HP's *visual engineering environment*, an iconic programming language for engineering problem solving. HP's visual engineering environment includes HP VEE-Test, HP VEE-Engine, and HP VEE for Windows collectively referred to in this manual as HP VEE.

NOTE

Throughout this manual references to HP VEE apply to HP VEE-Engine, HP VEE-Test, and HP VEE for Windows except where noted otherwise.

HP VEE gives you the ability to gather, analyze, and display data without conventional (text-based) programming. HP VEE increases your productivity by shortening the time it takes you to solve engineering problems.

The HP VEE family includes several separate products:

- **HP VEE for Windows** is designed for the test and measurement professional. It allows you to analyze and display data stored in a file, input by the user, or generated mathematically, plus it allows you to communicate with instruments from the visual environment running on an MS-Windows platform.
- **HP VEE-Test** is designed for the test and measurement professional. It allows you to analyze and display data stored in a file, input by the user, or generated mathematically, plus it allows you to communicate with instruments from the visual environment. This product runs only on HP-UX platforms.
- **HP VEE-Engine** is designed for the engineer and scientist. It allows you to analyze and display data stored in a file, input by the user, or generated mathematically. This product runs only on HP-UX platforms.
- **HP VEE RunOnly** is a *run-only* environment that runs programs developed with HP VEE-Test or HP VEE for Windows. (Programs developed

with HP VEE-Engine also will run.) No program editing capabilities are provided by HP VEE RunOnly.

- **HP VEE RunOnly for Windows** is a *run-only* environment that runs programs developed with HP VEE-Test or HP VEE for Windows on MS-Windows platforms. (Programs developed with HP VEE-Engine also will run.) No program editing capabilities are provided by HP VEE RunOnly for Windows.
- **HP VEE for Sun SparcStation** is designed for the test and measurement professional. It includes everything in HP VEE-Engine, plus it allows you to communicate with instruments from the visual environment. This product runs only on Sun SPARCstation® platforms.

HP VEE Example Programs

HP VEE includes many example programs. You can load, run and modify these programs to help you use and understand HP VEE.

Examples Directories

You will find the example programs in the following directories:

- HP VEE-Test

  `/usr/lib/veetest/examples`

- HP VEE-Engine

  `/usr/lib/veeengine/examples`

- HP VEE for Windows

  `C:\VEE\EXAMPLES`

Running Examples

UNIX Systems

The example programs for HP VEE are included in the **`/usr/lib/`** directory as indicated above. To load and execute these programs use the **`Open`** menu pick under the **`File`** menu. Load the file from the appropriate **`examples`** directory and press the **`Run`** button on the control bar.

Windows Systems

The example programs are included as part of the help system on HP VEE for Windows. To load and run an example program click on `Contents` under the `Help` menu on the menu bar. Next, click on `Examples` topic. This will bring up a list of examples available to you. Scroll through the list until you find the desired example. Click on the example name and HP VEE will give you more information about that example. Then press the `Open` button on the `Help` control bar to load the example. You can then execute the example program by pressing the `Run` button on the control bar.

2

Global Variables

Global Variables

Using Global Variables

A global variable is a named variable that is set globally, and which can be used by name in any context of an HP VEE program.

You can set a global variable with the `Set Global` object. The simplest way to get, or retrieve, a global variable is with the `Get Global` object. In the following example, the Real array at left is output to the `Set Global` object, which sets the global variable named `globalA`. The `Get Global` object retrieves `globalA` and outputs the array to the `AlphaNumeric` object.

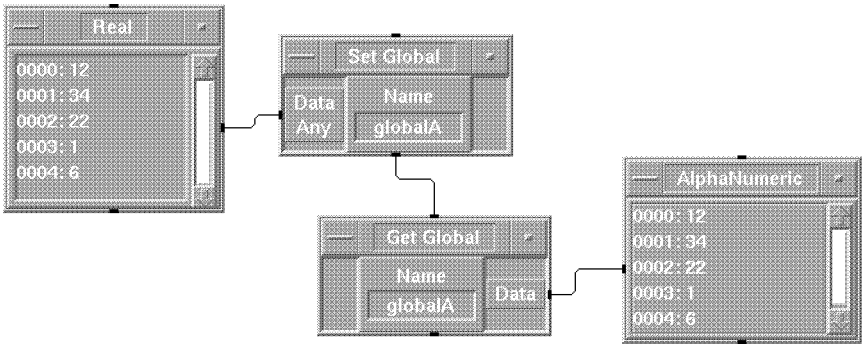


Figure 2-1. A Simple Global Variable Example

There is a very important point to note in the previous example. The `Set Global` must set the global variable *before* the `Get Global` attempts to retrieve it. To ensure this, the sequence output pin of the `Set Global` object is connected to the sequence input pin of the `Get Global` object. If this is not done, the `Get Global` may try to access a non-existent global variable, and an error will occur, as shown below:

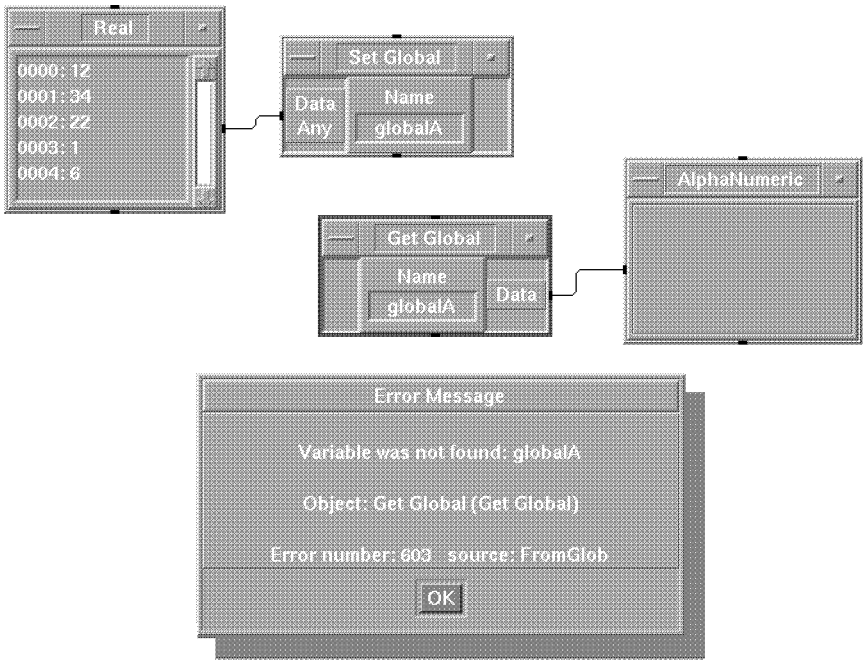


Figure 2-2. Accessing an Undefined Global Variable

Note that if `Delete Globals at PreRun` is off, you may not receive this error and receive old data instead. See the *HP VEE Reference* manual for more information regarding `Delete Globals at PreRun`.

Once you have defined a global variable, you can access it as many times as you want in your program. You don't have to use `Get Global` to retrieve the global variable. In the following example, the global variable `globalA` is retrieved once with a `Get Global` object, a second time by including the name `globalA` in an expression in a `Formula` object, and a third time by including the name `globalA` in a transaction in a `To File` object:

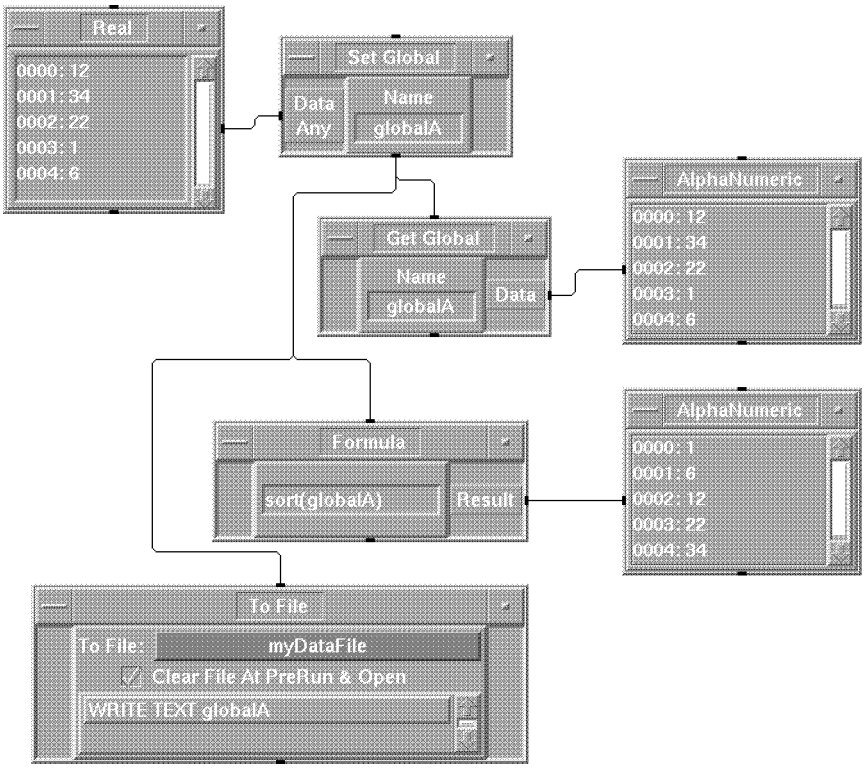


Figure 2-3. Accessing a Global Variable Multiple Times

In the previous example, the `Get Global` object just retrieves the global variable and outputs the data to the `AlphaNumeric` object. In the `Formula` object, the `sort` function reorders the array and outputs the data to the second `AlphaNumeric` object. The `To File` transaction retrieves the global variable and outputs the data to the file `myDataFile`.

NOTE

You can include the name of any global variable in any expression in a `Formula` object, or in any other expression that is evaluated at run time.

Global variables can contain complex data types such as waveforms. In the following example, the output of a `Function Generator` is output to a `Set Global` object, which sets the global variable `globW`. The `Get Global` object retrieves `globW` and outputs the data to the `XY Trace` object.

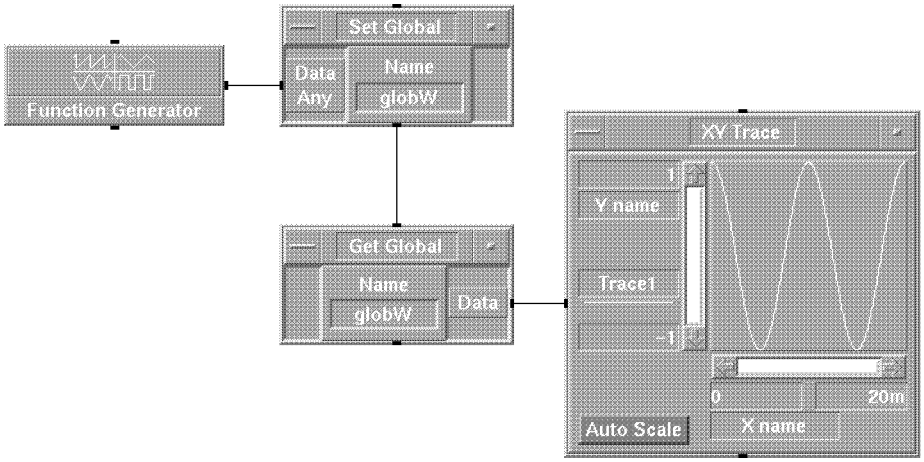


Figure 2-4. Waveform Data in a Global Variable

NOTE

You can use any valid variable name for a global variable. The first character must be a letter, and letters, numbers and the underscore may be used. Variable names are not case sensitive (uppercase and lowercase letters are equivalent). Special characters, including spaces, are not allowed.

To retrieve the global variable, you must use the name that you specified in the `Set Global` object. If there is a local variable with the same name, the local variable takes precedence.

Now let's look at using global variables within different contexts in an HP VEE program; that is, within UserObjects. (For information on UserObjects, see Help ⟹ How to Use HP VEE.)

Using Global Variables in UserObjects

A global variable is a named variable that is set globally, and which can be used by name in any context of an HP VEE program. For example, a global variable can be set with `Set Global` in the root context of the program, and can be accessed by name within the context of any UserObject or User Function (even though nested recursively) *within the same HP VEE process*. Thus, global variables can be used throughout your HP VEE program, but not in a remote process such as a Remote Function or Compiled Function.

Let's look at some examples of using global variables in UserObjects. In the program of the following figure, a Real array is output to the `Set Global` object, which sets the global variable `globalA`. The global variable is accessed by name three times within the UserObject, once by the `Get Global` object, once by the expression `sort(globalA)` in the `Formula` object, and once by the expression in the transaction of the `To File` object.

Note that if you did not use global variables, the Real array would have to be an input to the UserObject. When you have many nested UserObjects this could require many input and output connections, making your program harder to understand and maintain. By using global variables you can eliminate many connecting lines and improve the readability of your program.

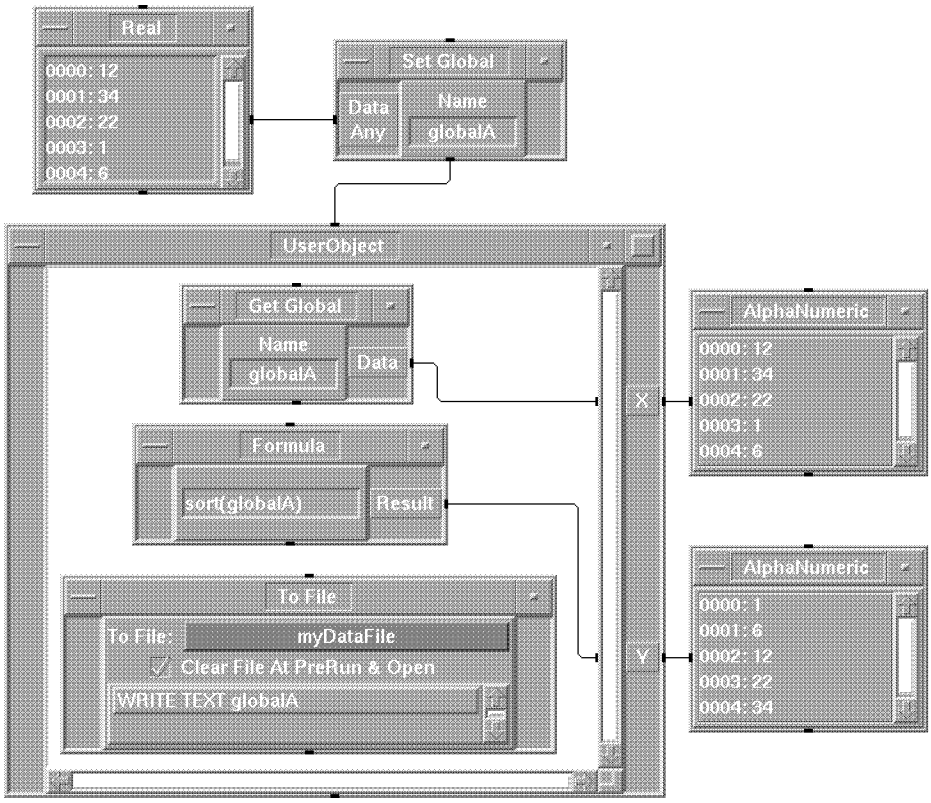


Figure 2-5. Retrieving a Global Variable in a UserObject

Note that, to ensure that `Set Global` sets the global variable before it is retrieved, you must connect the sequence output pin of the `Set Global` object to the sequence input pin of the UserObject, but you don't have to connect the sequence input pins of the individual `Get Global`, `Formula`, and `To File` objects. This is because none of the objects in the UserObject will execute until the UserObject sequence input pin is activated. This is a major advantage of accessing global variables within UserObjects — you don't have to make sequence connections to every object that accesses a global variable.

Let's look at another example. In the figure that follows, two waveforms (a sine wave and a noise waveform) are set as global variables with `Set Global` objects.

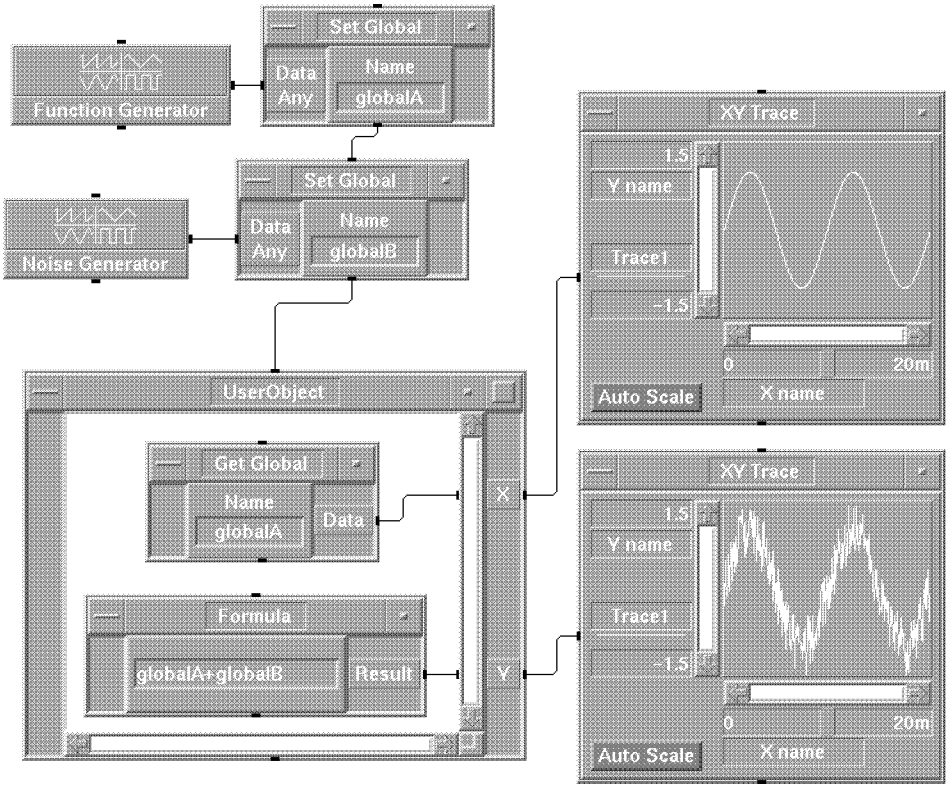


Figure 2-6. Retrieving Multiple Global Variables in a UserObject

The sine wave becomes `globalA` and the noise waveform becomes `globalB`, respectively. These waveforms are retrieved within the UserObject by the `Get Global` object (which retrieves and outputs the sine wave) and the `Formula` object (which retrieves both `globalA` and `globalB`, adds them, and outputs the combined waveform). Again the sequence input pin of the UserObject is used to hold off retrieval of the global variables until they have been set. However, in this case, you must ensure that *both* `Set Global` objects execute before the UserObject executes. To do this, you can "chain" the sequence pins of the two `Set Global` objects. The order in which the two `Set Global` objects execute does not matter, but the both must execute before the UserObject executes.

For further information about global variables, refer to "Using Global Records" in Chapter 4, and to the `Set Global` and `Get Global` reference sections in the *HP VEE Reference* manual.

3

Using Instruments

Using Instruments

HP VEE-Test, HP VEE for Sun SparcStation and HP VEE for Windows provide a variety of methods for controlling test and measurement instruments. These capabilities are supported by HP VEE-Test, HP VEE for Sun SparcStation and HP VEE for Windows only; they are *not* supported by HP VEE-Engine. This chapter describes how to configure and use instrument control objects.

NOTE

Before you can communicate with any instrument, the computer running HP VEE must be properly configured. The necessary procedures are described in the "Configuring HP VEE-Test" section of the manual *Installing HP VEE*.

Table 3-1. Instrument I/O Support

Platform	Supported I/O Interfaces
HP VEE for Windows (PC)	• HP-IB or GPIB • Serial (COM1, COM2, COM3, COM4) • VXI devices may be accessed via HP-IB or GPIB. Direct access to the VXI backplane is *not* supported.
HP VEE-Test (HP 9000)	• HP-IB • Serial • GPIO (S300/400 only) • VXI (MXI) backplane (V/382, S700 only)
HP VEE (Sun SPARCstation)	• GPIB • Serial
HP VEE-Engine	• No Instrument I/O Support

For the details of specific interface hardware supported, refer to the installation instructions for HP VEE on your platform.

Instrument Control Fundamentals

HP VEE supports three types of objects for controlling instruments:

- `Instrument Panels` (previously called `State Drivers`.)
- `Component Drivers`
- `Direct I/O`

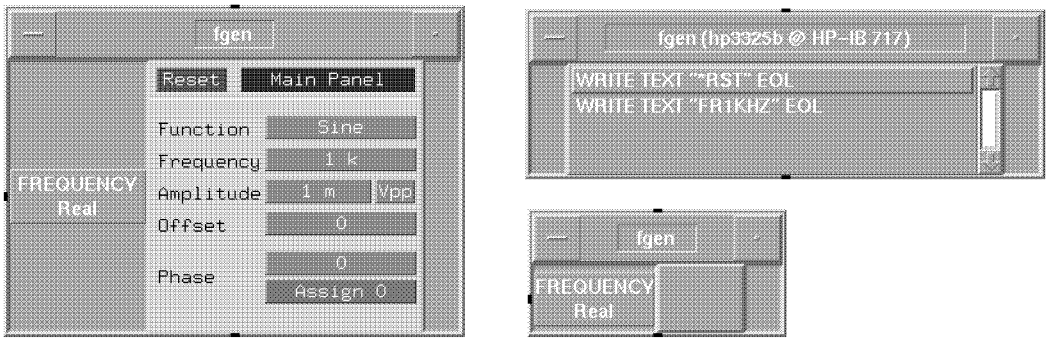


Figure 3-1. Instrument Control Objects

Each of the open-view objects shown in Figure 3-1 controls an HP 3325B function generator. Notice that each type of instrument control object has a different appearance. This appearance directly relates to the differences in how the objects operate and how you use them.

`Instrument Panels` and `Component Drivers` allow you to control instruments without learning the details of the instrument's programming mnemonics and syntax.

If you prefer to communicate with your instruments by sending low-level mnemonics or if a driver is not available for your instrument, you can use `Direct I/O`.

Driver-Based Objects

`Instrument Panels` and `Component Drivers` are available for a particular instrument only if there is a **driver file** to support that instrument. The installation procedure for HP VEE-Test automatically copies driver files onto your system disk. HP VEE for Windows allows you to select which drivers you wish to install. Subsequent sections in this chapter will explain how to locate and configure the proper driver files for your instruments.

Instrument Panels

`Instrument Panels` serve two purposes in HP VEE:

- They allow you to define a measurement state that specifies all the instrument function settings. When an `Instrument Panel` operates, the corresponding physical instrument is automatically programmed to match the settings defined in the `Instrument Panel`.
- They act as remote control panels for interactively controlling instruments. This is useful during development and debugging of your programs. It is also useful when your instruments do not have a physical front panel.

As shown in Figure 3-1, the open-view of an `Instrument Panel` contains a graphical control panel for the associated physical instrument. If the physical instrument is properly connected to your computer, you can control the instrument by clicking on the fields in the graphical control panel. You can also make measurements and display the results by clicking on numeric and XY displays.

Even if the instrument is not connected to your computer, you can still use the graphical panel to define a measurement state. In fact, this can be a great benefit if you wish to develop programs before instruments are purchased or while they are being used elsewhere.

For example, suppose you want to program the HP 3325B function generator to provide two different output signals:

1. A square wave with a frequency of 20kHz and an amplitude of 20mV rms
2. A sine wave with a frequency of 50kHz and an amplitude of 50mV rms

Figure 3-2 shows the two `Instrument Panels` that provide the desired signals.

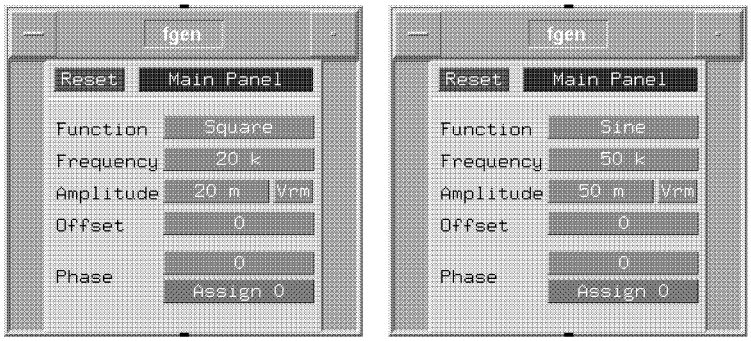


Figure 3-2. Two `Instrument Panels`

Component Drivers

In an HP instrument driver, each instrument function and measured value is called a **component**. A component is like a variable inside the driver that records the function setting or measured value. Thus, a **`Component Driver`** is an object that reads or writes only the components you specify as input and output terminals. This is in contrast to an **`Instrument Panel`**, which automatically writes values for many or all components.

`Component Drivers` are provided to help you improve the execution speed of your program; speed is the only advantage they provide over **`Instrument Panels`**. The execution speed of a program is generally impacted most when an instrument control object is attached to an iterator object where it must operate many times. In these cases, it is common for only one or two components to be changing; this is exactly the situation **`Component Drivers`** are designed to handle.

The increase in execution speed provided by a **`Component Driver`** will vary considerably from one situation to another. The increase depends primarily on the particular driver file used. There is no easy way to predict the exact increase in execution speed.

For example, suppose you want to program the HP 3325B Function Generator to do the following:

1. Output a sine wave with an initial frequency of 10kHz and an amplitude determined by operator input.
2. Sweep the frequency output from 10kHz to 1MHz using 5 steps per decade.

In this case, it makes sense to use an `Instrument Panel` to perform the initial setup and a `Component Driver` to repeatedly set the output frequency. Figure 3-3 shows a program that does this.

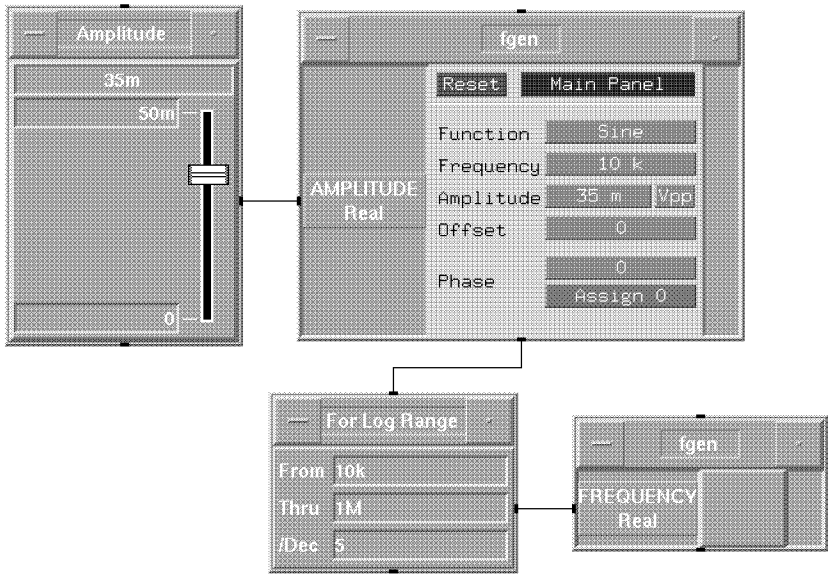


Figure 3-3. Combining `Instrument Panels` and `Component Drivers`

Direct I/O

`Direct I/O` objects allow you to read and write arbitrary data to instruments in much the same way you read from and write to files. This allows you full access to any programmable feature of any instrument including non-HP instruments; no instrument driver file is required. However, you must have a detailed understanding of your instrument's programming commands to use `Direct I/O`.

`Direct I/O` objects also provide convenient support for learn strings. A learn string is a special feature supported by some instruments that allows you to

set up measurement states from the front panel of the physical instrument. Once the instrument is configured, you simply select `Upload` from the `Direct I/O` object menu to upload the entire measurement state of the instrument to HP VEE. You can recall the measurement state from within your program by using the `Direct I/O` object.

This chapter describes how to *configure* HP VEE to use `Direct I/O`. For details about how to *use* `Direct I/O` objects, please refer to "Communicating with Instruments" in Chapter 6.

To complete the comparisons of different types of I/O objects, consider the previous program in Figure 3-3. You could replace the `Component Driver` in Figure 3-3 with `Direct I/O` as shown in Figure 3-4.

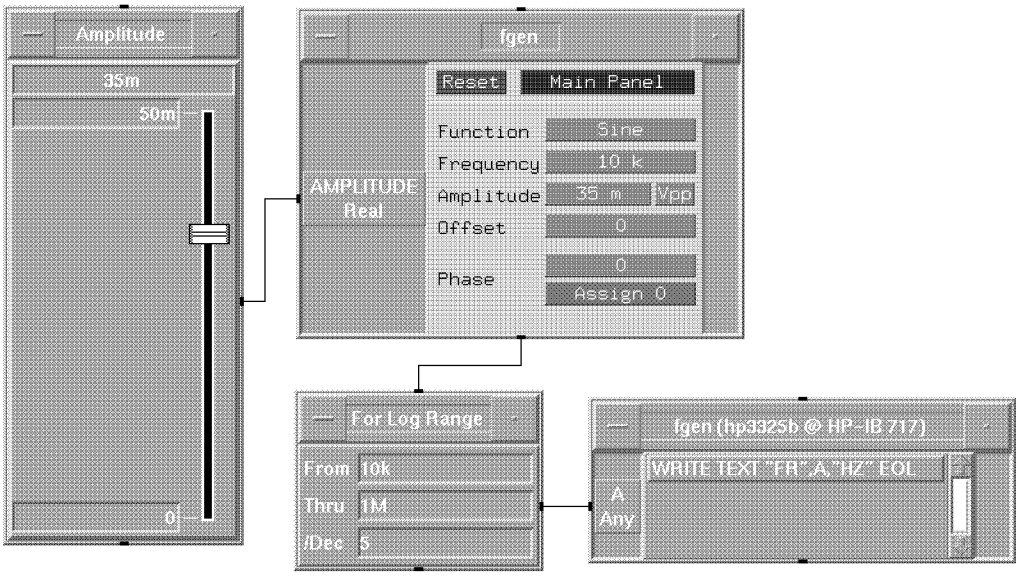


Figure 3-4. Combining `Instrument Panels` and `Direct I/O`

Summary of Instrument Control Objects

Table 3-2. Instrument Control Objects

Object	Advantages	Disadvantages
`Instrument Panel`	• Very easy to use • Good for interactive control • Acceptable execution speed in most cases	• Drivers not available for all instruments • Limited support for non-HP-IB interfaces
`Component Driver`	• Almost as easy to use as `Instrument Panels` • Execution speed approaching `Direct I/O`	• Drivers not available for all instruments • Limited support for non-HP-IB interfaces
`Direct I/O`	• Best execution speed • No driver required • Access to all supported interfaces, not just HP-IB	• Not as easy as driver-based I/O; you must know the instrument programming mnemonics

Terminating I/O Operations (UNIX)

In some cases you may wish to terminate an HP VEE I/O operation from the keyboard.

If you started HP VEE from a window as a foreground process by typing `veetest` (without using `&`), use this procedure to terminate I/O operations:

1. Use the mouse to position the pointer in the window in which you typed `veetest`.
2. Press CTRL-C (or the key indicated by the `intr` setting when you run the UNIX `stty` command). If you have problems with this, ask your system administrator for help.

If you started HP VEE from a window as a background process by typing `veetest &` , use this procedure to terminate I/O operations:

1. Use the mouse to position the pointer in a terminal window.
2. Determine the UNIX process identification number (PID) for HP VEE using the UNIX `ps` command.
3. Type `kill -2` *vee_pid*, where *vee_pid* is the PID number determined in the previous step.

Terminating I/O Operations (PC)

If you are using HP VEE for Windows and wish to terminate an I/O operation from the keyboard use the following procedure.

Place the cursor in the HP VEE for Windows window and press Ctrl-C. If that does not stop the I/O operation and halt the program continue with the following.

Press CTRL-ALT-DEL. Windows will display a screen giving you the option of:

1. Aborting the application that is no longer responding to Windows,
2. Returning to Windows and continuing your application, or
3. Rebooting your system.

NOTE

Aborting the application or rebooting you system may cause you to lose data. If you have not saved your program to a file, you will also lose your program.

Using Instrument Control Examples

HP VEE includes a number of on-line examples that are copied to your system disk automatically when you install HP VEE. In addition, the first time you execute HP VEE, it copies a default instrument configuration file to your home directory. *You must have this I/O configuration to open the on-line examples involving instruments.*

You can always configure additional instruments, but do not delete the entries in Figure 3-5 from the I/O configuration if you want to open the on-line instrument examples:

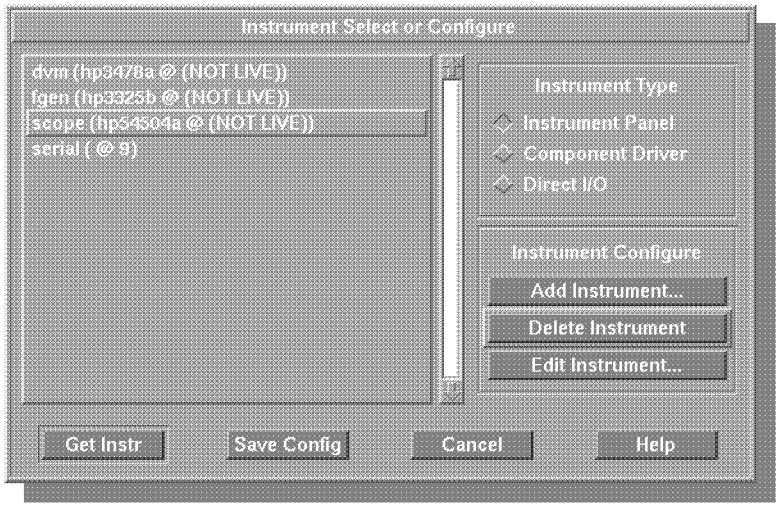


Figure 3-5. Default I/O Configuration

If HP VEE reports errors when you attempt to load the example programs referenced in this chapter, please refer to the section " Advanced Topic - I/O Configuration File" later in this chapter.

Understanding Instrument Panels and Component Drivers

This section explains some background and details that will help you use `Instrument Panels` and `Component Drivers` more effectively.

Inside HP Drivers

The term **driver** is used so frequently in computer terminology that it can be confusing. In HP VEE, the term driver has specific meaning, but you should be aware that people may use the term very casually.

Driver Files

> **Key Idea**
>
> Each HP instrument **driver file** describes the unique personality of a particular test and measurement instrument. A driver file is required to control any instrument using a `Instrument Panel` or `Component Driver` object.

Driver files (`.cid` files) are copied onto your system disk when HP VEE is installed or with the `Install Drivers` program. Each driver file contains two basic types of information:

1. A description of the instrument's functions and the commands used to set and query them.
2. A description of the appearance and behavior of the graphical control panel visible in the open view of an `Instrument Panel`.

Components

Key Idea

Internally, `Instrument Panels` and `Component Drivers` represent each instrument function as a **component**. Component names are analogous to variable names in programming languages; components are used to hold the value of an instrument function setting or measured values.

For example, the HP 3478A voltmeter contains these and other components:

Typical Voltmeter Driver Components

Component Name	Instrument Function
`ARANGE`	Autoranging is on or off.
`FUNCTION`	The measurement function is voltage, current, or resistance.
`TRIGGER`	The trigger source is internal, external, fast, or single.
`READING`	The most recent measured value.

Components can be accessed interactively or through a program. To access a component interactively, click on a labeled button or display in the open view of an `Instrument Panel`. To access components using a graphical program, add them as input or output terminals. For detailed procedures on using components, refer to the sections "Using Instrument Panels" and "Using Component Drivers" later in this chapter.

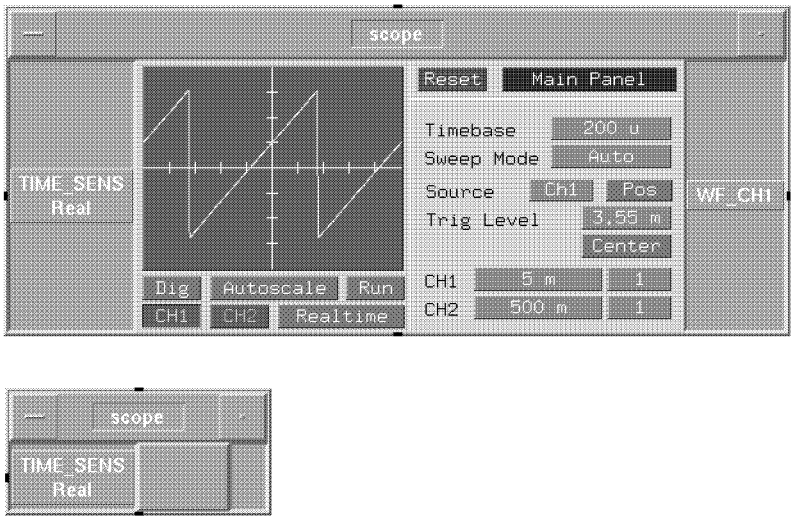


Figure 3-6. Accessing Driver Components

States

An instrument **state** is a specific set of values for all components in a particular driver. For example, you must set all the components in a voltmeter driver to particular values for AC voltage measurements. You must use a different set of component values to measure DC current. In other words, these two different measurements require two different states.

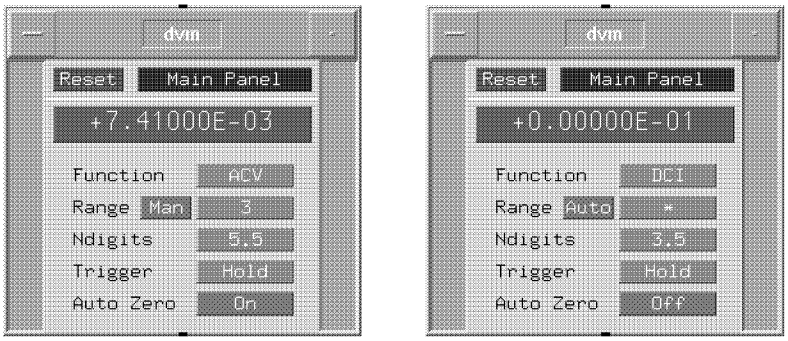


Figure 3-7. Two Voltmeter States

Key Idea

In HP VEE, each instance of an `Instrument Panel` represents a separate measurement state. It is common to have more than one `Instrument Panel` in a program, where each `Instrument Panel` programs the *same* physical instrument to a unique measurement state.

Each `Instrument Panel` object you create using the same instrument `Name` will communicate with the same physical instrument.

How Driver-Based I/O Works

When you place an `Instrument Panel` or `Component Driver` in a program, HP VEE establishes a state record in memory. This state record is specific to a particular instrument `Name`. `Name`s are very important and are discussed in greater detail in the section "The Importance of Names" later in this chapter.

All the driver-based objects that reference a particular `Name` share a single state record. The state record reflects the *current* values of each the instrument's components. When you write to components using `Instrument Panels` or `Component Drivers`, HP VEE updates both the physical instrument and the state record. If you write to the instrument using `Direct I/O`, HP VEE marks the state record as invalid because the state record no longer matches the true state of the physical instrument.

Important differences occur when the `Instrument Panel` and `Component Driver` objects operate.

Instrument Panel Operation

> **Key Idea**
>
> When an `Instrument Panel` operates, it sends whatever commands are necessary to make the state of the physical instrument match the state defined in the graphical control panel.

If necessary, an `Instrument Panel` will send commands to reset and update all settings in the corresponding physical instrument. This behavior is affected by the `Incremental Mode` setting described in the section, "Instrument Driver Configuration Dialog Box" later in this chapter.

If you set `Incremental Mode` to `ON`, HP VEE compares the current state record to the desired state defined in the `Instrument Panel` and determines which components must be changed. HP VEE sends *only* those commands required to update the affected components.

If you set `Incremental Mode` to `OFF` or if the current state record is marked as invalid, HP VEE will explicitly send commands to update each and every component in order to guarantee synchronization between the desired state and the state of the physical instrument.

Note that an `Instrument Panel` operates when its sequence input pin is activated *or* when you click on one of the control panel buttons visible in the open view.

Component Driver Operation

> **Key Idea**
>
> When a `Component Driver` operates, it writes *only* to those components that appear as input terminals and reads *only* from those components that appear as output terminals.

This is why `Component Drivers` generally operate faster than `Instrument Panels`. An `Instrument Panel` potentially writes to *many* components to achieve a particular state; a `Component Driver` writes to only the components you specify.

Note that components are read and written in the order that they appear as terminals, from top to bottom. This order of operation is important in some cases where you want the instrument to change the value of one component, based on the value of another. This interaction is called **coupling**. With component drivers you must do this manually.

Multiple Driver Objects

This section discusses some situations that may be confusing when you are using multiple objects that:

- Use the same instrument `Name`
- Use the same instrument address
- Use the same driver file

The Importance of Names. This section discusses some concepts related to configuring instruments. If you find this discussion difficult to understand, you may wish to wait and read it after you have read the following section, "Configuring Instruments".

Consider how HP VEE maps an instrument object to a specific instrument configuration created via `I/O` ⟹ `Instrument` . . . .

> **Key Idea**
>
> It is the `Name` field in the `Device Configuration` dialog box that logically maps each instrument object to the address of a physical instrument and the other configuration information. To determine the `Name` of an instrument object, click on `Show Config` in the object menu; the text in the object title is *not* necessarily the same as the `Name`.

For example, the `Name`s of the instruments in the default I/O configuration are `scope`, `dvm`, and `fgen`. `Name`s must be unique; there cannot be more than one configured instrument with the `Name` of `scope`.

In general, you should have only one configured `Name` referencing a particular physical instrument. While it is possible to have more than one `Name` referencing the same instrument address, it will cause unpredictable results in a program using `Instrument Panels`. HP VEE's internal records of instrument states are organized by `Names`. Two `Instrument Panels` with different names will blindly write to the same address, thus invalidating each other's state records.

In some cases involving `Direct I/O`, you may need to have more than one `Name` for the same physical instrument. This may be necessary if certain settings in the `Direct I/O Configuration` dialog box need to be varied depending on the direct I/O operation. For example, you may wish to send some commands to an oscilloscope with EOI asserted on the last character of data and some commands without EOI. In such a case, you can configure one instrument with the `Name Scope (EOI)` and another instrument with the `Name Scope`. Both `Scope` and `Scope (EOI)` have the same `Address` setting, but different settings for `END on EOL`.

Note that the configured `Name` appears as the default title in instrument objects at the time you select them from the menu. However, editing the title *in no way* affects the relationship to the `Name`.

`Names` are also important for saving and opening programs containing instruments. When you save a program, the `Name` of each instrument object in the program is saved. When you open a program, HP VEE looks in the current I/O configuration for the `Name` of each instrument being loaded. For example, if you saved a program containing an `Direct I/O` object with a name of `My Scope`, there must be an instrument named `My Scope` in the current I/O configuration. `Name`s must match *exactly*, including any spaces. However, `Name` is not case-sensitive. Furthermore, if the object under consideration is a `Instrument Panel` or `Component Driver`, the `ID Filename` (driver file) in the current I/O configuration must match the one used in the saved program.

Reusing Driver Files. It is valid (and not uncommon) to have several objects with different names that use the same driver file. For example, you might have a test system that uses three programmable power supplies named `Supply1`, `Supply2`, and `Supply3` at three separate addresses that all use the `hp665x.cid` driver file. Since the `Names` are different, HP VEE maintains a separate state record for each name; an `Instrument Panel` for `Supply1` will have no effect on anything related to `Supply2` or `Supply3`.

Choosing the Correct Instrument Object

This section presents a simplified set of rules for choosing the appropriate object for an instrument control application. In this procedure you will take some steps that appear to be configuring an instrument. You are not actually configuring an instrument, this is just the easiest way to locate certain information.

1. Determine whether a driver file is available for your instrument:
   a. Click on `I/O` ⟹ `Instruments` . . . .
   b. Click on the `Add Instrument` button in the `Instrument Select or Configure` box.
   c. Click on `Instrument Driver Config` in the `Device Configuration` dialog box.
   d. Click on the `ID Filename` field.
   e. Scroll through the list of available driver files.
2. If a driver file *is available* for your instrument:
   a. Use it as an `Instrument Panel`. For details, refer to the following sections, "Configuring Instruments" and "Using Component Drivers". Go to step 4.
   b. If the execution speed of an `Instrument Panel` is not acceptable, use a `Component Driver`. For details, refer to the following sections, "Configuring Instruments" and "Using Instrument Panels". Go to step 4.
3. If a driver file *is not available* for your instrument, or if you prefer to program it directly, use a `Direct I/O` object. For details, refer to the following sections, "Configuring Instruments" and "Direct I/O Configuration Dialog Box". Go to step 4.
4. Exit all the pending dialog boxes on your screen by clicking on the `Cancel` buttons.

Configuring Instruments

This section contains the step-by-step procedures for configuring HP VEE to communicate with your instruments using **Instrument Panels**, **Component Drivers**, and **Direct I/O**.

As you follow these procedures, note that some steps apply only to driver configuration, some apply only to direct I/O, and some apply to both. It is possible to configure a single instrument for both driver-based communication and direct I/O; if you follow the sections in order, this situation is addressed.

Begin the configuration procedure with the following section, "Basic Instrument Configuration". If you need help interpreting any of the fields in the dialog boxes used in this procedure, refer to the section "Details of Configure I/O Dialog Boxes" later in this chapter.

Basic Instrument Configuration

Follow this procedure for *any* instrument you plan to use with HP VEE.

1. Click on **I/O** ⟹ **Instrument** . . . .
2. Note the device (instrument) entries listed in the **Instrument Select or Configure** dialog box.
   a. If you need to use instruments that do not appear in the list, go to step 3 in this procedure.
   b. If the instruments you need are listed but the addresses or other settings are incorrect, click on the instrument you wish to modify, then click on the **Edit Instrument** button. The **Device Configuration** dialog box appears. Go to step 4 in this procedure.
   c. If the list properly specifies all your instruments, click on the **Cancel** button and exit this procedure.
3. Click on the **Add Instrument** button. The **Device Configuration** dialog box appears.

4. HP VEE enters default values in the fields inside the `Device Configuration` dialog box. If you have difficulty interpreting any of the fields in this dialog box, refer to the section, "Device Configuration Dialog Box" later in this chapter. Complete or modify each entry field as follows:

   - `Name:` Enter the name for this instrument. Note that any spaces are significant and `Name` does not distinguish between upper- and lower-case letters. Typical entries are `Scope`, `Voltmeter`, and `Switch`.

     Note that the entry you specify for `Name` is a symbolic link between each instrument object created using this `Name` and the configuration information you specify in this procedure. This concept is very important and it is explained in detail in the section, "The Importance of Names".

   - `Interface:` Identify the type of interface used by the instrument. The default is `HP-IB`. Other interface selections are `VXI`, `GPIO`, and `Serial`.

   - `Address:` Enter the digits of the address that identifies the instrument. For HP-IB instruments, the address is of the form *xxyyzz*, where *xx* is the one- or two-digit **interface select code** and *yy* is the two-digit bus address. *zz* is the two-digit secondary bus address; it is rarely used and, if not used, it is omitted entirely. The factory default select code for most HP-IB interfaces is `7`.

     For example, if an oscilloscope is at HP-IB address 6, set `Address` to `706`.

     If you need help determining the proper address for HP-IB or other interfaces, refer to the "Device Configuration Dialog Box" and "Troubleshooting" sections later in this chapter.

   - `Device Type:` Enter the manufacturer's program number for the instrument. This field is for your convenience only; HP VEE does not use it. If you are going to configure this instrument with a driver, HP VEE will fill in this field for you when you specify the driver file.

   - `Timeout:` Enter the timeout in seconds. The default value of five seconds works for most applications. It is unadvisable to specify `0` in this field; if you do, HP VEE will *never* detect a timeout. Certain `Direct I/O` transactions for register or memory access of VXI devices do not support a timeout. Note that some interfaces and some platforms limit the value of `Timeout`. For example, RS-232 on UNIX is limited to a 25 second timeout.

- `Byte Ordering:` Specifies the order the device uses for reading and writing binary data. HP VEE will determine if byte swapping is necessary. Click on this field to choose between `MSB` (send Most-Significant Byte first) and `LSB` (send Least-Significant Byte first). All IEEE 488.2-compliant devices must default to MSB order. Please refer to your device manual for more specific information.
- `Live Mode:` Set this field to `ON` if the corresponding instrument is connected to your computer. Set this field to `OFF` if the corresponding instrument is not connected to your computer so that objects will not attempt to read or write to a non-existent instrument.

5. At this point, you have completed all the configuration steps that are common to drivers and direct I/O. You must still complete the steps for driver configuration, direct I/O configuration, or both.
   - To configure your instrument for use with a driver, go to the following section,"Driver Configuration".
   - To configure your instrument for use with direct I/O, go to the following section, "Direct I/O Configuration".
   - To configure a VXI device for register access with direct I/O, go to the "A16 Space Configuration (VXI only)" section, later in this chapter.
   - To configure a VXI device for extended memory access with direct I/O, go to the "A24/A32 Space Configuration (VXI only)" section, later in this chapter. *We will use the term "extended memory" to indicate either A24 or A32 memory in a VXI device.* (A VXI device can implement either A24 or A32 memory, but not both.)

   If you are following this procedure out of sequence and wish to exit now, you must click on the `OK` button in the `Device Configuration` dialog box and the `Save Config` button in the `Instrument Select or Configure` dialog box to save your newly configured instruments.

Driver Configuration

As you begin this procedure, you should have already completed the steps in the previous section, "Basic Instrument Configuration". The `Device Configuration` dialog box should be on the screen.

1. Click on the `Instrument Driver Config` button. The `Instrument Driver Configuration` dialog box appears.
2. Click on the `ID Filename` field. A list of driver files appears.
3. Click on the driver file corresponding to your instrument, then click on the `OK` button.
4. HP VEE enters default values in the fields inside the `Instrument Driver Configuration`. In general, you do not need to change these values. Consult the section, "Instrument Driver Configuration Dialog Box" later in this chapter for details about this dialog box.
5. Click on the `OK` button in the `Instrument Driver Configuration` dialog box.
6. At this point you have completed all the steps necessary to configure this instrument for use with a driver.
   - To configure this instrument for use with direct I/O, go to the following section, "Direct I/O Configuration".
   - To configure additional instruments, click on the `OK` button in the `Device Configuration` dialog box. Go to step 3 of the section, "Basic Instrument Configuration" earlier in this chapter.
   - To stop configuring instruments and save the current configuration, click on the `OK` button in the `Device Configuration` dialog box, then click on the `Save Config` button in the `Instrument Select or Configure` dialog box.

Direct I/O Configuration

As you begin this procedure, you should have already completed the steps in the previous section, "Basic Instrument Configuration". The `Device Configuration` dialog box should be on the screen.

1. Click on the `Direct I/O Config` button. The `Direct I/O Configuration` dialog box appears.
2. HP VEE selects default values for all of the fields in the `Direct I/O Configuration` dialog box based on your previous selection of `Interface`. Modify these fields as required. For a detailed description of each field, please refer to the section, "Direct I/O Configuration Dialog Box" later in this chapter.
3. Click on the `OK` button in the `Direct I/O Configuration` dialog box.
4. At this point you have completed all the steps required to configure this instrument for use with direct I/O.
   a. To configure additional instruments, click on the `OK` button in the `Device Configuration` dialog box. Go to step 3 of the section, "Basic Instrument Configuration" earlier in this chapter.
   b. To stop configuring instruments and save the current configuration, click on the `OK` button in the `Device Configuration` dialog box, then click on the `Save Config` button in the `Insturment Select or Configure` dialog box.

To learn how to use the `Direct I/O` instruments you have just configured, refer to "Communicating with Instruments" in Chapter 6.

A16 Space Configuration (VXI only)

Before you begin, you should complete the steps in the "Basic Instrument Configuration"section, earlier in this chapter. The `Device Configuration` dialog box should be on the screen. From this dialog box you can configure a VXI device's registers for access with `WRITE REGISTER` or `READ REGISTER` transactions in a `Direct I/O` object.

1. Click on the `A16 Space Config` button. The `A16 Space Configuration` dialog box appears.
2. The dialog box displays fields which are used to configure the memory access data width, and to configure individual registers within a VXI device's A16 memory. For a detailed description of each field, refer to the "A16 Space Configuration Dialog Box (VXI Only)" section, later in this chapter.
3. Click on the `OK` button in the `A16 Space Configuration` dialog box.
4. At this point you have completed all the steps required to configure this VXI instrument for register access with direct I/O.
   a. To configure additional instruments, click on the `OK` button in the `Device Configuration` dialog box. Go to step 3 of the section, "Basic Instrument Configuration" earlier in this chapter.
   b. To stop configuring instruments and save the current configuration, click on the `OK` button in the `Device Configuration` dialog box, then click on the `Save Config` button in the `Instrument Select or Configure` dialog box.

A24/A32 Space Configuration (VXI only)

Before you begin, you should complete the steps in the "Basic Instrument Configuration" section, earlier in this chapter. The `Device Configuration` dialog box should be on the screen. From this dialog box you can configure a VXI device's extended memory (A24 or A32) for access with `WRITE MEMORY` or `READ MEMORY` transactions in a `Direct I/O` object.

1. Click on the `A24/A32 Space Config` button. The `A24/A32 Space Configuration` dialog box appears.
2. The dialog box displays fields which are used to configure the memory access data width, and to configure individual registers within a VXI device's extended memory. For a detailed description of each field, refer to the "A24/A32 Space Configuration Dialog Box (VXI only)" section, later in this chapter.
3. Click on the `OK` button in the `A24/A32 Space Configuration` dialog box.
4. At this point you have completed all the steps required to configure this VXI instrument for memory access with direct I/O.
   a. To configure additional instruments, click on the `OK` button in the `Device Configuration` dialog box. Go to step 3 of the section, "Basic Instrument Configuration" earlier in this chapter.
   b. To stop configuring instruments and save the current configuration, click on the `OK` button in the `Device Configuration` dialog box, then click on the `Save Config` button in the `Instrument Select or Configure` dialog box.

Details of Configure I/O Dialog Boxes

This section explains in detail the meaning of each field in the dialog boxes you encounter while configuring instruments using `I/O` ⟹ `Instrument` . . . .

Device Configuration Dialog Box

Name

The **Name** field uniquely identifies a particular instrument configuration. The instrument **Name** is a symbolic link between each instance of an instrument control object and all the configuration information corresponding to that **Name**. Usually, this field is used to give a descriptive name to the instrument, such as `Oscilloscope` or `Power Supply`.

Names must be unique; you cannot configure two instruments with a **Name** of `Scope`. While it is possible to create two different **Name**s that refer to the same physical instrument, it can cause problems if you use both **Name**s with `Instrument Panel` in the same program.

Do not confuse the **Name** of an instrument with the text that appears as the title in an instrument control object. The default title of an instrument control object is the name, but you can change the title and it has no effect on the **Name**. If you need to determine the **Name** of a particular instance of an instrument control object, select `Show Config` in the object menu.

NOTE

It is very important that you use **Name**s correctly. This section discusses only the more common situations. For more details about how HP VEE uses **Name**s please refer to the section "The Importance of Names" earlier in this chapter.

Interface

The `Interface` field specifies the type of hardware interface used to communicate with the instrument: `HP-IB`, `VXI`, `GPIO`, or `Serial`.

Address

The `Address` field specifies the address of the instrument. For instruments using GPIO or serial interfaces, the address is the same as the interface select code. An interface select code is a number used by the computer to identify a particular interface.

For instruments using HP-IB interfaces, the address is of the form *xxyyzz*, where:

- *xx* is the one- or two-digit interface select code. The factory default select code for most HP-IB interfaces is 7.
- *yy* is the two-digit bus address of the instrument. Use a leading zero for bus addresses less than `10`; for example, use `09` not `9`.
- *zz* is the secondary address of the instrument. *In most cases, secondary addressing is not used* and this portion of the `Address` field entry is omitted. Secondary addresses are typically used by cardcage-type instruments that incorporate multiple plug-in modules.

NOTE

The secondary address is the secondary address as defined in IEEE 488.1; it is part of the interface specification of the instrument hardware. The instrument *hardware* design determines whether or not a secondary address is required; secondary addresses are *not* related to *driver* configuration.

Do not confuse secondary addresses with the `Sub Address` field used in the `Instrument Driver Configuration` dialog box. Subaddresses are a *driver-related* feature and are used *very rarely*.

For instruments using VXI interfaces (connected to either the HP 75000 Series C V/382 VXI controller or the E1489I MXI Interface for HP 9000 Series 700 systems), the address is of the form *xxyyy*, where:

- *xx* is the one- or two-digit select code of the VXI backplane interface of an embedded or external controller. The factory default select code is 16 for

either the HP 75000 Series C Program V/382 VXI Controller or the *first* E1489I MXI Interface in an HP 9000 Series 700 system.

(If more than one E1489I MXI Interface is installed in an HP 9000 Series 700 system, the first defaults to select code 16, and subsequent interfaces default to select codes 18, 20, and 22, in that order.)

- *yyy* is the logical address of the VXI device. Use leading zeros for logical addresses less than `100`. (For example, use `008` not `8`.)

NOTE

Setting the **`Address`** field to `0` has special meaning. Setting the **`Address`** field to `0` (for any interface) means that there is no physical instrument matching this device description connected to the computer. An address of `0` automatically sets **`Live Mode`** to **`OFF`**.

HP-IB Address Examples. For example, if you wish to control an HP-IB instrument at bus address 9 using the built-in HP-IB interface on an HP computer. The factory default select code for built-in HP-IB interfaces is 7. Assuming that the select code has not been changed, the proper **`Address`** field setting for this instrument is `709`.

If you wish to address a plug-in module in this same instrument with a secondary bus address of 2, the proper **`Address`** field setting is `70902`.

VXI Address Examples. Assume you wish to control a VXI instrument, logical address 28, using either the HP 75000 Series C V/382 VXI Controller or the E1489I MXI Interface (Series 700). The factory default select code is 16, so the proper **`Address`** field setting is `16028`, assuming the select code has not been changed. (Logical addresses for VXI instruments are in the range 1–255, inclusive.)

Serial Address Examples. Assume you wish to control an instrument using an HP 98644 Serial Interface. The factory default select code for this interface is `9`. Assuming the select code has not been changed, the proper **`Address`** field setting for this instrument is `9`.

Assume you wish to control an instrument using an HP 98642 Four-Channel Multiplexer. The instrument is connected to port 3, the highest-numbered

port available on the interface. The default interface select code is 17. Assuming that the select code has not been changed, the proper `Address` field setting for this instrument is `1703`. Note that the HP 98642 interface supports separate addresses for each port: `1700`, `1701`, `1702`, and `1703`.

GPIO Address Example. Assume you wish to control a custom-built instrument using an HP 98622 GPIO Interface. The factory default select code for this interface is `12`. Assuming the select code has not been changed, the proper `Address` field setting for this instrument is `12`.

Device Type

The `Device Type` field is used to record the manufacturer's program number. For example, the `Device Type` for the HP 54504A oscilloscope could be `hp54504a`. This field is provided for your convenience; HP VEE does not use it.

You may notice that if you configure the instrument for use with a driver, HP VEE will automatically fill in the `Device Type` field using the driver file name as a default. You can change this default to anything you want; it will not affect HP VEE.

Timeout

The `Timeout` field specifies how many seconds HP VEE will wait for an instrument to respond to a request for communication before generating an error. The default value of five seconds works well for most applications. In general, you should not set this field to `0`; if you do, HP VEE will *never* detect a timeout. Certain `Direct I/O` transactions for register or memory access of VXI devices do not support a timeout.

Byte Ordering

Use this field to specify the order the device uses for reading and writing binary data. HP VEE will determine if byte swapping is necessary. Click on this field to choose between `MSB` (send Most-Significant Byte first) and `LSB` (send Least-Significant Byte first). All IEEE 488.2-compliant devices must default to MSB order. Please refer to your device manual for more specific information.

Live Mode

The `Live Mode` field determines whether or not HP VEE will attempt to communicate with an instrument at the specified address. To actually communicate with a physical instrument connected to your computer, you *must* set `Live Mode` to `ON`.

Note that if `Live Mode` is `OFF` for instrument X, you can run programs containing `Instrument Panels`, `Component Drivers`, or `Direct I/O`

objects that would otherwise read and write to instrument X. However, no instrument communication actually takes place. The latter behavior can be useful if you wish to develop or debug portions of a program while instruments are not available.

Config Buttons

If you plan to control the configured instrument using `Instrument Panels` or `Component Drivers`, you must click on the `Instrument Driver Config` button and complete the resulting dialog box. Please refer to the section "Driver Configuration" earlier in this chapter for details.

If you plan to control the configured instrument using `Direct I/O`, you must click on the `Direct I/O Config` button and complete the resulting dialog box. Please refer to the section "Direct I/O Configuration" earlier in this chapter for details.

However, if you want to control a VXI instrument by using `Direct I/O` to access the instrument's registers or extended memory, there is an additional step:

- If you want to use `Direct I/O` to access the device's *registers*, click on the `A16 Space Config` button and complete the resulting dialog box. (Refer to the "A16 Space Configuration (VXI only)" section, earlier in this chapter, for details.)
- If you want to use `Direct I/O` to access the device's *extended memory*, click on the `A24/32 Space Config` button and complete the resulting dialog box. (Refer to the "A24/A32 Space Configuration (VXI only)" section, earlier in this chapter, for details.)

Instrument Driver Configuration Dialog Box

This section describes the meaning of all the fields in the `Instrument Driver Configuration` dialog box. Interfaces currently supported by Instrument Drivers include HP-IB and VXI (message-based devices only).

ID Filename

The `ID Filename` field specifies the file that contains the desired driver. Note that files are named according to instrument model number. Be certain to choose the name corresponding to the exact model number you are using; there are similar file names, such as `hp3325a.cid` and `hp3325b.cid`.

If you are unsure which driver to use, please refer to the on-line information in `Help` ⟹ `Instruments`.

Sub Address

The `Sub Address` field specifies the subaddress used by certain drivers to identify plug-in modules in cardcage-type instruments, such as data acquisition systems and switches. If you are *not* configuring a driver for one of these plug-ins, set this field to `""` (the `NULL` string).

NOTE

Since *very* few drivers use subaddresses, the default setting of `""` (the `NULL` string) is the proper setting in 99% of all situations.

If you *are* configuring a driver for one of these plug-ins, refer to the on-line help for the instrument driver to determine if and how subaddresses are used. To get help on instrument drivers, click on `Help` ⟹ `Instruments`.

NOTE

Do not confuse the `Sub Address` field with a secondary address for HP-IB instruments. Subaddresses are part of the *driver* configuration; they are *not* part of the hardware address.

Incremental Mode

The `Incremental Mode` field specifies whether or not incremental state recall is used with `Instrument Panel` objects.

> **NOTE**
>
> The proper setting for `Incremental Mode` is `ON` in 99% of all situations.

When `Incremental Mode` is set to `ON`, HP VEE automatically minimizes the commands sent to the instrument to change its state. To do this, HP VEE compares its record of the current state the physical instrument to the new state specified in the `Instrument Panel`. HP VEE determines which component settings are different, then sends only those commands needed to change components that do not match the desired state. In most cases, you should set `Incremental Mode` to `ON`; it provides the best execution speed.

When `Incremental Mode` is set to `OFF`, HP VEE explicitly sets the values of *every* component when a corresponding `Instrument Panel` operates. This is generally used only when there is a chance that HP VEE's record of the instrument state does not match the true state of the physical instrument.

Note that the `Incremental Mode` setting affects only `Instrument Panels`.

These things *do* suggest setting `Incremental Mode` to `OFF`:

- Allowing front panel operation of an instrument at any time
- Changing instrument settings outside of the HP VEE environment through C programs, HP BASIC programs, or shell commands

Using combinations of `Component Drivers`, `Instrument Panels`, and `Direct I/O` objects in a program does *not* imply that you need to set `Incremental Mode` to `OFF`.

Error Checking

The `Error Checking` field determines whether or not HP VEE queries the instrument for errors after setting component values. Set this field to `ON` unless execution speed is not acceptable.

Direct I/O Configuration Dialog Box

This section explains each field in the `Direct I/O Configuration` dialog box. Interfaces which support transactions configured with

this dialog box include HP-IB, VXI (message-based devices only), GPIO, and serial.

> **NOTE**
>
> Register-based VXI devices can only be controlled by means of read/write access to device registers or device memory. Refer to "A16 Space Configuration Dialog Box (VXI Only)" or "A24/A32 Space Configuration Dialog Box (VXI only)" for details.

Read Terminator

The `Read Terminator` field specifies the character that terminates `READ` transactions. The entry in this field must be a a single character surrounded by double quotes. "Double quote" means ASCII 34 decimal. HP VEE recognizes any ASCII character as a `Read Terminator` as well as the escape characters shown in Table 3-3.

The character you should specify is determined by the design of your instrument. Most HP-IB instruments send newline after sending data to the computer. Consult your instrument programming manual for details.

Table 3-3. Escape Characters

Escape Character	ASCII Code (decimal)	Meaning
`\n`	10	Newline
`\t`	9	Horizontal Tab
`\v`	11	Vertical Tab
`\b`	8	Backspace
`\r`	13	Carriage Return
`\f`	12	Form Feed
`\"`	34	Double Quote
`\'`	39	Single Quote
`\\`	92	Backslash
`\`*ddd*		The ASCII character corresponding to the three-digit octal value *ddd*.

EOL Sequence

The `EOL Sequence` field specifies the characters that are sent at the end of `WRITE` transactions that use `EOL ON`. The entry in this field must be zero or more characters surrounded by double quotes. "Double quote" means ASCII 34 decimal. HP VEE recognizes any ASCII characters within `EOL Sequence` including the escape characters shown previously in Table 3-3.

Multi-field As

The `Multi-field As` field specifies the formatting style for multi-field data types for `WRITE TEXT` transactions. The multi-field data types in HP VEE are Coord, Complex, PComplex, and Spectrum. Other data types and other formats are unaffected by this setting.

Specifying a multi-field format of `( . . . ) Syntax` surrounds each multi-field item with parentheses. Specifying `Data Only` omits the parentheses, but retains the separating comma. For example, the complex number `2+2`*j* could be written as `(2,2)` using `( . . . ) Syntax` or as `2,2` using `Data Only` syntax.

Array Separator

The **Array Separator** field specifies the character string used to separate elements of an array written by **WRITE TEXT** transactions. The entry in this field must be a a single character surrounded by double quotes. "Double quote" means ASCII 34 decimal. HP VEE recognizes any ASCII character as an **Array Separator** as well as the escape characters shown previously in Table 3-3.

WRITE TEXT STR transactions in **Direct I/O** objects that write arrays are a special case. In this case, the value in the **Array Separator** field is ignored and the linefeed character (ASCII 10 decimal) is used to separate the elements of an array. This behavior is consistent with the needs of most instruments.

Note that HP VEE allows arrays of multi-field data types; for example you can create an array of Complex data. In such a case, if **Multi-Field Format** is set to **(. . .) Syntax**, the array will be written as:

`(1,1)`*array_sep*`(2,2)`*array_sep* `...`

where *array_sep* is the character specified in the **Array Separator** field.

Array Format

The **Array Format** determines the manner in which multidimensional arrays are written. For example, mathematicians write a matrix like this:

```
1  2  3
4  5  6
7  8  9
```

HP VEE writes the same matrix in one of two ways, depending on the setting of **Array Format**. In the two examples that follow, **EOL Sequence** is set to "**\n**" (newline) and **Array Separator** is set to

" " (space).

```
1 2 3     Block Array Format
4 5 6
7 8 9

1 2 3 4 5 6 7 8 9   Linear Array Format
```

Either array format separates each element of the array with the **Array Separator** character. **Block Array Format** takes the additional step of separating each row in the array using the **EOL Sequence** character.

In the more general case (arrays greater than two dimensions), `Block Array Format` outputs an `EOL Sequence` character each time a subscript other than the right-most subscript changes. For example, if you write the three-dimensional array `A[x,y,z]` using `Block` array format with this transaction:

```
WRITE TEXT A
```

an `EOL Sequence` will be output each time `x` or `y` changes value. If the size of each dimension in `A` is two, the elements will be written in this order:

```
A[0,0,0]  A[0,0,1]<EOL Sequence>
A[0,1,0]  A[0,1,1]<EOL Sequence>
<EOL Sequence>
A[1,0,0]  A[1,0,1]<EOL Sequence>
A[1,1,0]  A[1,1,1]<EOL Sequence>
```

Notice that after `A[0,1,1]]` is written, `x` and `y` change simultaneously and consequently two `<EOL Sequence>`s are written.

Writing Arrays with Direct I/O. `WRITE TEXT STR` transactions that write arrays to direct I/O paths ignore the `Array Separator` setting for the `Direct I/O` object. These transactions always use linefeed (ASCII decimal 10) to separate each element of an array as it is written. This behavior is consistent with the needs of most instruments.

NOTE

This special behavior for arrays does not apply to any other types of transactions.

END On EOL (HP-IB Only)

`END on EOL` controls the behavior of EOI (End Or Identify). If `END on EOL` is `YES`, the EOI line is asserted on the bus at the time the last data byte is written under one of the following circumstances:

1. A `WRITE` transaction with `EOL ON` executes.
2. A `WRITE` transaction executes as the last transaction listed in the `Direct I/O` object.
3. One or more `WRITE` transactions execute without asserting EOI and are followed by a non-`WRITE` transaction, such as `READ`.

Many instruments accept *either* EOI or a newline as valid message terminators. Some block transfers may require EOI. Consult your instrument's programming manual for details.

Conformance

`Conformance` specifies whether an instrument conforms to the IEEE 488.1 or IEEE 488.2 standard. Refer to your instrument programming manual to determine the standard to which your instrument conforms, and then set the `Conformance` field accordingly.

Each of these standards defines communication protocols for the HP-IB interface. However, IEEE 488.2 specifies rules for block headers and learn strings that are left undefined in IEEE 488.1. All message-based VXI instruments are IEEE 488.2 compliant.

If you set `Conformance` to `IEEE 488` (which denotes IEEE 488.1), you may optionally specify additional settings to handle block headers and learn strings. Do this using the fields that appear below `Conformance`:

- `Binblock`
- `State`
- `Upload String`
- `Download String`

Binblock. The `Binblock` field specifies the block data format used for `WRITE BINBLOCK` transactions. `Binblock` may specify IEEE 728 `#A`, `#T`, or `#I` block headers. If `Binblock` is `None`, `WRITE BINBLOCK` writes an IEEE 488.2 Definite Length Arbitrary Block Response Data block.

IEEE 728 block headers are of the following forms:

```
#A<Byte_Count><Data>
#T<Byte_Count><Data>
#I<Data><END>
```

where:

`<Byte_Count>` is a 16-bit unsigned integer that specifies the number of bytes that follow in `<Data>`.

`<Data>` is a stream of arbitrary bytes.

`<END>` indicates that EOI is asserted with the last data byte transmitted.

State. The `State` field indicates whether or not the instrument has been configured for uploading and downloading learn strings. If the `State` entry is `Not Config'd` and you wish to configure the instrument for use with learn strings, click on the `State` field and the `Upload String` and `Download` fields will appear. If the `State` entry is `Not Config'd`, the `Upload String` and `Download String` fields are set to the null string.

Upload String. The `Upload String` field specifies the command that is sent to the instrument when you select `Upload State` from the `Direct I/O` object menu. Specify the command that causes the instrument to output its learn string; consult your instrument programming manual for details. Note that you must surround the command with double quotes.

Download String. The `Download String` field specifies the string that is sent to the instrument immediately before the learn string as the result of a `WRITE STATE` transaction in a `Direct I/O` object. This field is provided to support instruments that require a command prefix when downloading a learn string; consult your instrument programming manual for details.

Serial Interface Settings

Several fields are provided to allow flexible configuration of serial interfaces.

- `Baud Rate`
- `Character Size`
- `Stop Bits`
- `Parity`
- `Handshake`

For UNIX the transmission and reception queue size is set at 512 characters. For the PC the reception queue size can be set by the user. Refer to Appendix A for more information.

Data Width (GPIO only)

The **Data Width** field specifies the number of bits of parallel data transmitted as a unit across the GPIO interface. This field configures the interface to read and write data eight or sixteen bits wide. No hardware switches need to be set in conjunction with this field.

A16 Space Configuration Dialog Box (VXI Only)

This section explains each field in the `A16 Space Configuration` dialog box. Register-based VXI devices can only be controlled by read/write access to device registers or device memory.

Byte Access

The `Byte Access` field specifies whether the VXI device supports 8-bit A16 memory accesses. The possible choices for this field are:

- `NONE` - Device does not support byte access.
- `ODD ACCESS` - Device supports byte access, but only on odd byte boundaries (D08(O)).
- `ODD/EVEN ACCESS` - Device supports byte access on all boundaries (D08(EO)).

Word Access

The `Word Access` field is not editable. All VXI devices must support 16-bit access (D16).

LongWord Access

The `LongWord Access` field specifies whether the VXI device supports 32-bit A16 memory accesses. The possible choices are:

- `NONE` - Device does not support 32-bit access.
- `D32 ACCESS` - Device supports 32-bit A16 memory access.

Add Register

When you click on the `Add Register` field, it adds a row of fields to the bottom of the dialog box. These fields allow you to configure access to a device's A16 memory. The four fields are:

- `Name` - The symbolic name of the register, which is used to refer to the particular register in a `Direct I/O` object using

  `READ REGISTER` or `WRITE REGISTER` transactions.
- `Offset` - The offset in *bytes* from the *relative* base of a device's A16 memory for the register being configured.

- `Format` - The data format that will be read from, or written to, the register being configured. The read or write access will take place at the byte specified in the `Offset` field. The possible formats are:
  - `BYTE` - Read or write a byte. The device must support and be configured correctly for 8-bit access by using the `BYTE` field discussed above. If the `BYTE` field is `ODD`, the byte location specified in the `Offset` field must be an odd number.
  - `WORD16` - Read or write a 16-bit word. The 16-bits are represented as a two's complement integer. All VXI devices explicitly support this format.
  - `WORD32` - Read or write a 32-bit word. The 32-bits are represented as a two's complement integer. HP VEE supports this format even if the `LongWord Access` field is specified as `NONE` (by using two D16 accesses to read or write all 32 bits). If the `LongWord Access` field is specified as `D32 ACCESS`, all 32 bits are accessed.
  - `REAL32` - Read or write a 32-bit word. The 32-bits are represented as a IEEE 754 32-bit floating-point number. HP VEE supports this format even if the `LongWord Access` field is specified as `NONE` (by using two D16 accesses to read or write all 32 bits). If the `LongWord Access` field is specified as `D32 ACCESS`, all 32 bits are accessed.
- `Mode` - Specify what I/O mode the register will support. The choices are:
  - `READ` - This register will appear as a choice in a `READ REGISTER` transaction only.
  - `WRITE` - This register will appear as a choice in a `WRITE REGISTER` transaction only.
  - `READ/WRITE` - This register will appear as a choice in both a `READ REGISTER` and `WRITE REGISTER` transaction.

Delete Register

When you click on the `Delete Register` field, it will display a list of the symbolic names of the currently configured registers. The selected register will be removed from the dialog box.

A24/A32 Space Configuration Dialog Box (VXI only)

This section explains each field in the `A24/A32 Space Configuration` dialog box. Register-based VXI devices can only be controlled by read/write access to device registers or device memory.

> **NOTE**
>
> We use the term "extended memory" to indicate either A24 or A32 memory in a VXI device. (A VXI device can implement either A24 or A32 memory, but not both.)

Byte Access

The `Byte Access` field specifies whether the VXI device supports 8-bit extended memory accesses. The possible choices for this field are:

- `NONE` - Device does not support byte access.
- `ODD ACCESS` - Device supports byte access, but only on odd byte boundaries (D08(O)).
- `ODD/EVEN ACCESS` - Device supports byte access on all boundaries (D08(EO)).

Word Access

The `Word Access` field is not editable. All VXI devices must support 16-bit access (D16) for all memory spaces.

LongWord Access

The `LongWord Access` field is specifies whether the VXI device supports 32-bit extended memory accesses. The possible choices are:

- `NONE` - Device does not support 32-bit access.
- `D32 ACCESS` - Device supports 32-bit extended memory access.

Byte Ordering

The `Byte Ordering` field specifies the internal order of bytes within a VXI device's extended memory. The choices are:

- `MSB First` - Most significant bit first.
- `LSB First` - Least significant bit first.

The VXIbus Specification recommends that extended memory devices implement MSB First architecture. All Hewlett-Packard VXI devices conform to this recommendation.

Add Location

When you click on the `Add Location` field, it adds a row of fields to the bottom of the dialog box. These fields allow you to configure access to a device's extended memory. The four fields are:

- `Name` - The symbolic name of the location, which is used to refer to the particular memory location in a `Direct I/O` object using `READ MEMORY` or `WRITE MEMORY` transactions.
- `Offset` - The offset in *bytes* from the *relative* base of a device's extended memory for the location being configured.
- `Format` - The data format that will be read from, or written to, the location being configured. The read or write access will take place at the byte specified in the `Offset` field. The possible formats are:
  - `BYTE` - Read or write a byte. The device must support and be configured correctly for 8-bit access by using the `BYTE` field discussed above. If the `BYTE` field is `ODD`, the byte location specified in the `Offset` field must be an odd number.
  - `WORD16` - Read or write a 16-bit word. The 16-bits are represented as a two's complement integer. All VXI devices explicitly support this format.
  - `WORD32` - Read or write a 32-bit word. The 32-bits are represented as a two's complement integer. HP VEE supports this format even if the `LongWord Access` field is specified as `NONE` (by using two D16 accesses to read or write all 32 bits). If the `LongWord Access` field is specified as `D32 ACCESS`, all 32 bits are accessed.
  - `REAL32` - Read or write a 32-bit word. The 32-bits are represented as a IEEE 754 32-bit floating-point number. HP VEE supports this format even if the `LongWord Access` field is specified as `NONE` (by using two D16 accesses to read or write all 32 bits). If the `LongWord Access` field is specified as `D32 ACCESS`, all 32 bits are accessed.

- `Mode` - Specify what I/O mode the location will support. The choices are:
  - `READ` - This location will appear as a choice in a `READ REGISTER` transaction only.
  - `WRITE` - This location will appear as a choice in a `WRITE REGISTER` transaction only.
  - `READ/WRITE` - This location will appear as a choice in both a `READ REGISTER` and `WRITE REGISTER` transaction.

Delete Location

When you click on the `Delete Location` field, it will display a list of the symbolic names of the currently configured location. The selected register will be removed from the dialog box.

Example of Configuring a VXI Device

The following examples of VXI device configuration apply to both A16 and extended (A24/A32) memory.

Figure 3-8 shows the `A16 Space Configuration` dialog box with the register configuration of an HP E1411B Multimeter. The `Offset` field is configured with the offset in bytes of each register from the *relative* base of the device's A16 space. Notice that there are two registers with an offset of four bytes, `status` and `control`. Register `status` is configured for `READ` mode only, while register `control` is configured for `WRITE` mode only. While two separate register locations could have the same mode, the `Name` field must be unique.

Note that it would be possible for the register at byte location 4 to be named `statuscontrol` with a mode of READ/WRITE.

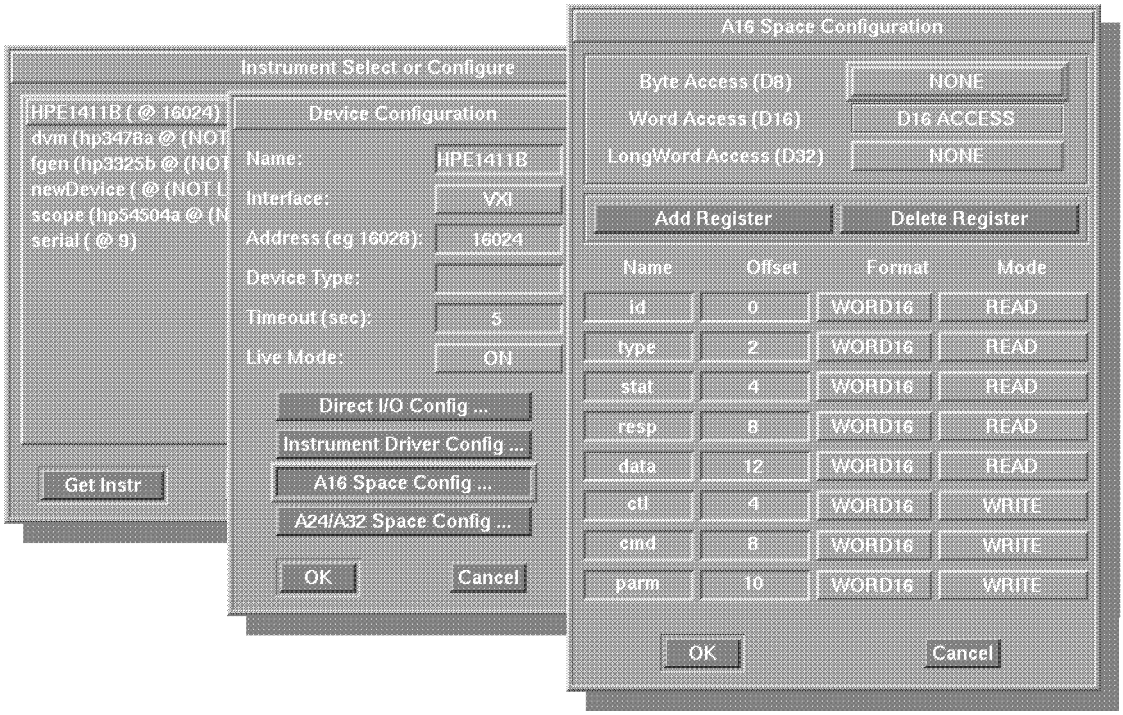


Figure 3-8. A16 Configuration for the HP E1411B Multimeter

Advanced Topic - I/O Configuration File

This section discusses a special file that you may occasionally need to modify. This file is `.veeio` on UNIX and `VEE.IO` on PC's, the I/O configuration file. It is stored in your `$HOME` directory (typically your `/users` directory), or `C:\VEE` on PC's. If you have difficulty understanding this section, ask your system administrator for help.

When you configure instruments using `I/O` $\Longrightarrow$ `Instrument`, you click on the `Save Config` button in the `Instrument Select or Configure` dialog box to save the settings. These settings are saved not only in memory for the remainder of your work session, but also in `.veeio` or `VEE.IO`. This way, the next time you start HP VEE, you can continue working with your existing I/O configuration.

If you do not have a `.veeio` or `VEE.IO` file in your `$HOME` directory when you run HP VEE, HP VEE creates a default `.veeio` or `VEE.IO` file for you. This default file is also created when you run HP VEE for the first time. The default configuration contains the instruments used in the on-line examples included with HP VEE.

You cannot open any program containing an instrument control object unless your I/O configuration contains a device with a matching `Name`. In this discussion, `Name` means the entry in the `Name` field in the `Instrument Select or Configure` dialog box, not the text in the object's title bar. Furthermore, if the object is an `Instrument Panel` or `Component Driver`, the `ID Filename` must also match your configuration. Settings other than `Name` and `ID Filename` do not affect your ability to *open* these programs, although other setting affect how the programs *execute*.

Most of the time, HP VEE takes care of `.veeio` or `VEE.IO` for you. But there may be times when you want to erase, update, or copy this file outside of the HP VEE environment. The rest of this section explains two situations when you might want to do this.

Sharing Programs

Assume Susan develops an instrument control program that she wants to share with you. How can you get the same I/O configuration as Susan so you can run her program? You can either manually add all of Susan's instruments to your configuration via `I/O ⟹ Instrument` or you can copy Susan's `.veeio` or `VEE.IO` file to your `$HOME` directory, or `C:\VEE` directory on PC's. If you use the file copying method, save a copy of your original `.veeio` file to another name (such as `.oldveeio` or `veeio.old` on PC's) in case you need it later. Make sure that any `.veeio` file you place in your `$HOME` directory has write permissions set to allow HP VEE to write to it.

Running Example Programs

Assume that you want to open one of the on-line example programs. Unfortunately, you have accidentally deleted the default instrument configuration. There are two ways to solve this problem:

1. Manually add the default instrument configuration to your current configuration using `I/O ⟹ Instrument` . . . .
2. Rename your `.veeio` or `VEE.IO` file and restart HP VEE.
3. Simply load the program. If HP VEE finds any conflicts it will ask if you wish to add the device now. If you answer `yes`, a `Device Configuration` dialog box will appear with the correct name in the name field. Fill in any addition information needed and press `OK`. HP VEE will then continue loading the file.

If you choose method 1, configure the following instruments using the procedures outlined in the section, "Configuring Instruments" earlier in this chapter:

Default I/O Configuration

Name Field Entry	ID Filename Field Entry
`dvm`	`hp3478a.cid`
`scope`	`hp54504a.cid`
`fgen`	`hp3325b.cid`

If you choose method 2, follow these procedures:

On UNIX systems:

1. Exit HP VEE.
2. Go to your `$HOME` directory (`/users/`*YourName*).
3. Type `mv .veeio .oldveeio` Return. This renames your `.veeio` file.
4. Execute `veetest`. HP VEE will look for `.veeio` and when it finds that it does not exist, it will create one for you using the default I/O configuration.

On PC systems:

1. Exit HP VEE.
2. Go to your `C:\VEE` directory.
3. Type `rename vee.io veeio.old` Return. This renames your `VEE.IO` file.
4. Run HP VEE. It will look for `VEE.IO` and when it finds that it does not exist, it will create one for you using the default I/O configuration.

Using Instrument Panels

Using Instrument Panels Interactively

The open view of an **Instrument Panel** provides a graphical control panel that you can use to interactively construct a measurement state. If you connect the corresponding physical instrument to your computer and turn **Live Mode** on, you can control the physical instrument interactively as you build the measurement state. To change an individual setting, click on the corresponding field in the graphical control panel and complete the resulting dialog box. To make a measurement and view the result, click on the display region of a numeric or XY display. Note that XY displays may take a few seconds to update.

Using Instrument Panels Programmatically

To add an `Instrument Panel` to your program:

1. Click on `I/O` ⟹ `Instrument` . . . . The `Instrument Select or Configure` dialog box appears.
2. Click on the desired instrument to highlight it, then click on the `Instrument Panel` button.

   If the instrument has not been configured with an instrument driver, HP VEE will give you a choice of instrument drivers to use. Refer to the section "Configuring Instruments" earlier in this chapter for configuration procedures.
3. Click on the desired instrument again, or on the `Get Instr` button.
4. When the object outline appears, position the cursor and click once to place the object in the work area.

To use `Instrument Panels` in a program, you will often use input or output terminals to set the values of components. Each input or output terminal actually corresponds to a component in the driver. There are two ways to add a terminal:

- Select `Terminals` ⟹ `Add Data Input` or `Terminals` ⟹ `Add Data Output` from the `Instrument Panel` object menu. A list box appears that lists all the valid driver components not yet used as terminals. Double-click on the component in the list that you wish to add as a terminal.
- Select `Add Terminals` ⟹ `Select Input Terminal` or `Add Terminals` ⟹ `Select Output Terminal` from the `Instrument Panel` object menu. After making this selection, click on one of the fields or display areas in the graphical control panel to add the corresponding component as a terminal.

In general, it is more convenient to use the first method listed above because you do not need to guess the name of the component you wish to use. However, some components are not visible on any part of the graphical control panel; you must access these using the second method.

Using Component Drivers

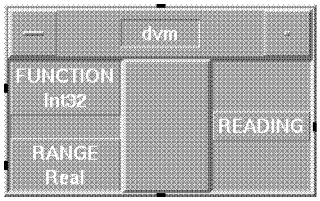


Figure 3-9. A Typical Component Driver

Using Component Drivers in a Program

To add a `Component Driver` to a program:

1. Click on `I/O` ⟹ `Instrument` . . . . A list of configured instruments appears.
2. Click on the desired instrument to highlight it, then click on the `Component Driver` radio button.

   If the instrument has not been configured with an instrument driver, HP VEE will give you a choice of instrument drivers to use. Refer to the section "Configuring Instruments" earlier in this chapter for configuration procedures.
3. Click on the desired instrument again, or on the `Get Instr` button.
4. When the object outline appears, position the pointer and click once to place the object in the work area.

`Component Drivers` are generally used when you need to repeatedly execute an instrument control object while changing only a few components. `Component Drivers` are preferred over `Instrument Panels` in these situations because `Component Drivers` write and read only the components you specify and, as result, they execute faster.

Figure 3-10 illustrates this type of situation. This program measures the frequency response of a filter by sweeping the input frequency sourced by `fgen` and measuring the response using `dvm`. Since the subthread attached to `For Log Range` executes repeatedly, component drivers are used to improve execution speed. Note that state drivers are still appropriate for the initial set up of `fgen` and `dvm`.

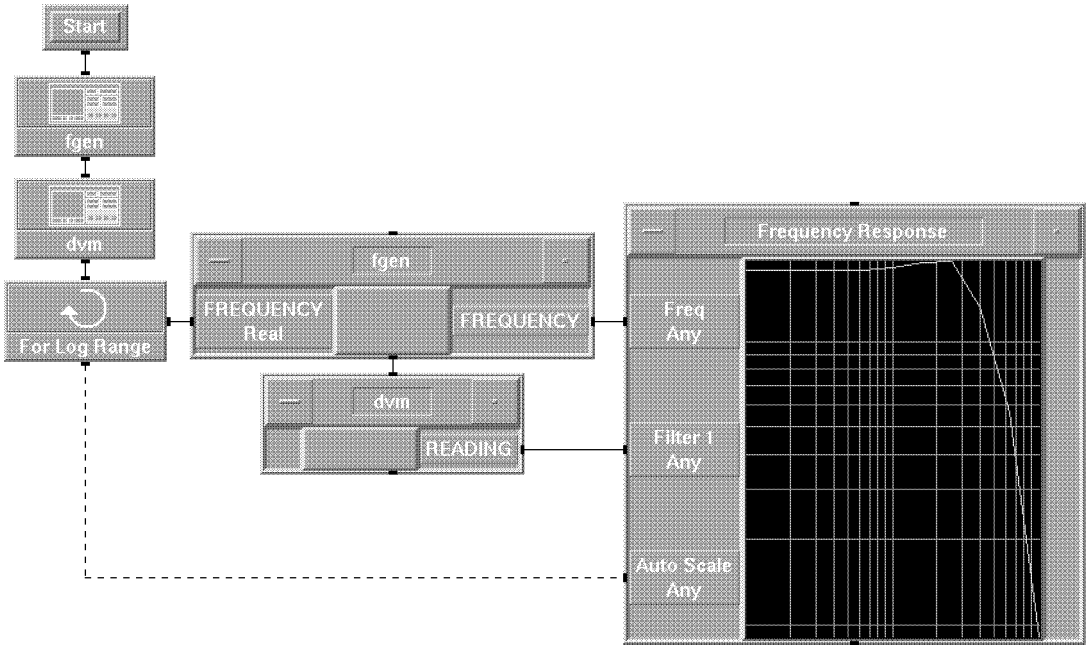


Figure 3-10. Using Instrument Panels and Component Drivers

The program shown in Figure 3-10 is stored in the file `manual15.vee` in your `examples` directory.

Advanced I/O Control

This section explains the objects accessed via the `I/O ⟹ Advanced I/O` menu and how to download measurement subprograms to instruments.

Polling

HP VEE supports all the serial poll operations defined by IEEE 488.1. All HP-IB instruments, and all VXI message-based instruments, support serial poll operations. VXI message-based devices are, by definition, IEEE 488.2 compliant. HP VEE does not support parallel poll operations.

You can obtain an instrument's serial poll response in two ways:

Object	Serial Poll Behavior
`Device Event`	The `Device Event` object can poll the specified instrument once and output a scalar integer, which is the serial poll response using the `NO WAIT` option. The `Device Event` object can also wait for a specific bit pattern within the serial poll response byte by using a user supplied bit mask and the `ALL CLEAR` and `ANY SET` options.
`Direct I/O`	`Direct I/O` objects for HP-IB instruments support a `WAIT SPOLL` transaction. This transaction repeatedly polls an instrument until the serial poll response byte matches a specific bit pattern, using a user-supplied bit mask and the `ALL CLEAR` or `ANY SET` options. Refer to Chapter 6 for details about `Direct I/O`.

A serial poll of a message-based VXI instrument is achieved by sending the Word Serial Protocol command *Read STB* to the specified instrument.

This is the exact bus sequence used by `Device Event` for an HP-IB instrument:

ATN
UNL
MLA
TAD
SPE
$\overline{\text{ATN}}$
serial poll response
ATN
SPD
UNT

The `Device Event` object has special execution properties when configured for `Spoll` that are discussed in the next section, "Service Requests." This behavior allows for other concurrent threads to continue execution while waiting for a specific bit pattern using the mask value and the `ALL CLEAR` or `ANY SET` options. `NO WAIT` will simply execute immediately and return the status byte of the HP-IB or message-based VXI instrument.

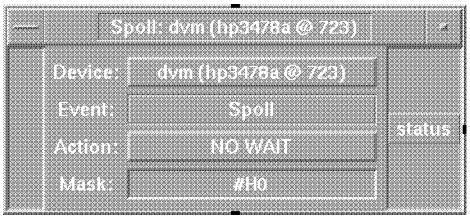


Figure 3-11. Device Event Configured for Serial Polling

Service Requests

To detect a service request (SRQ message) for a VXI instrument, use the `Device Event` object (`I/O` ⟹ `Advanced I/O` ⟹ `Device Event`). To detect a service request for an HP-IB instrument, use the `Interface Event` object (`I/O` ⟹ `Advanced I/O` ⟹ `Interface Event`).

The `Device Event` and `Interface Event` objects provide special behavior for interrupt-like execution. To view this behavior, you may wish to execute your program with `Edit` ⟹ `Show Exec Flow` enabled.

For example, `Interface Event` behaves in a program as follows:

1. Before a `Interface Event` object (configured for HP-IB and with the `WAIT` option specified) operates, execution proceeds normally with each thread sharing execution with equal priority.
2. When a `Interface Event` object operates, execution of the thread attached to the `Interface Event` data output pauses at the `Interface Event` object. Other threads not attached to `Interface Event` *will continue to execute.*
3. When an SRQ is detected on the specified interface, the data output of `Interface Event` is activated.

   At this point, *execution of all other threads is blocked* until the thread attached to the data output of `Interface Event` completes execution.

The program shown in Figure 3-12 shows how to handle service requests. In the case shown, it is possible that either `dvm` or `scope` is responsible for a service request. The program determines the originator of the service request by using `Device Event` to obtain the status byte of each instrument. Each status byte is tested using

`If/Then/Else` and the `bit(x,n)` function to determine if bit 6 is true. If bit 6 is set, then the corresponding instrument is responsible for the service request. The `Until Break` object automatically re-enables the entire thread to handle any subsequent service requests. The `Device Event` object is configured for `NO WAIT`, meaning the status byte is returned without using the mask value. If a mask value of `64` is used and the `Device Event` object is configured for `ANY SET`, the `If/Then/Else` and `bit(x,n)` function will not have to be used.

Note that different instruments have different requirements for clearing and re-enabling service requests. In Figure 3-12, `dvm` requires only a serial poll to clear and re-enable its SRQ capability. However, `scope` requires the additional step of clearing the originating event register.

The `Device Event` object can be used to detect a service request from a message-based VXI instrument. The instrument that writes a request true event (RT), which is evaluated as a request for service, into the VXI controller's signal register will receive a *Read STB* word serial protocol command. The message-based instrument will send its status byte back to the controller, and will write a request false event (RF) into the VXI controller's signal register. The status byte will be used with the supplied mask value and the `ANY SET` or `ALL CLEAR` options to determine which bit (besides bit 6) is set. Thus one object, the `Device Event` can be used to detect a service request from a message-based VXI device and determine why the request occurred.

For further information see the `Device Event` and `Interface Event` entries in the Reference Manual.

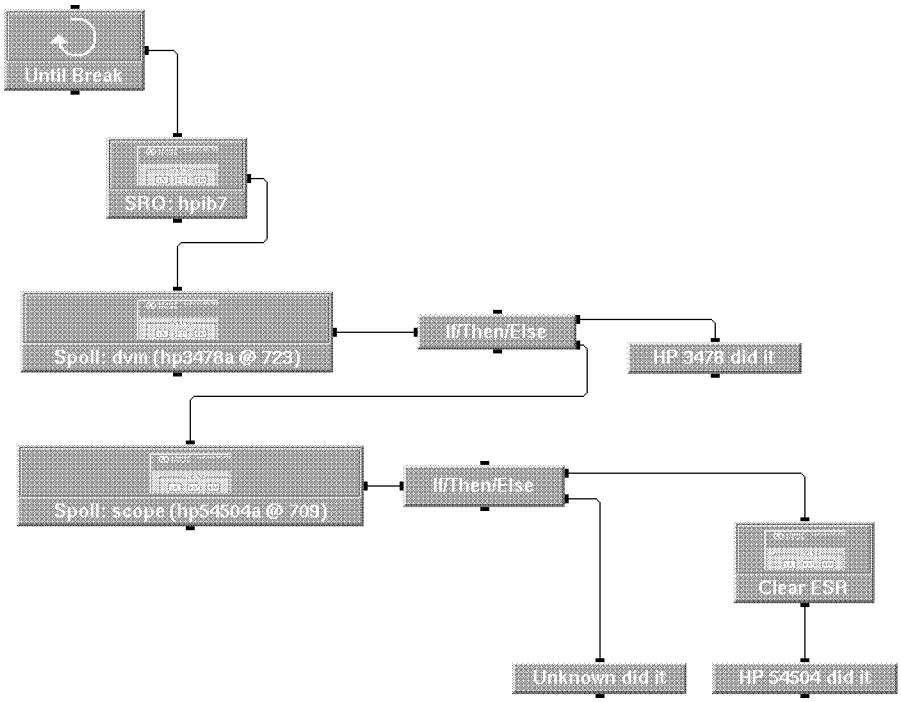


Figure 3-12. Handling Service Requests

The program shown in Figure 3-12 is saved in the file `manual16.vee` in your `examples` directory.

Monitoring Bus Activity

You can use `Bus I/O Monitor` to record all bus messages transmitted between HP VEE and any talkers and listeners. Note that `Bus I/O Monitor` records *only* those bus messages inbound or outbound from HP VEE.

You can monitor any supported interface (HP-IB, VXI, serial, or GPIO) using a `Bus I/O Monitor`. Each instance of a `Bus I/O Monitor` monitors exactly one hardware interface.

Figure 3-13 shows the bus messages sent to write `*RST` to an instrument at HP-IB address 717.

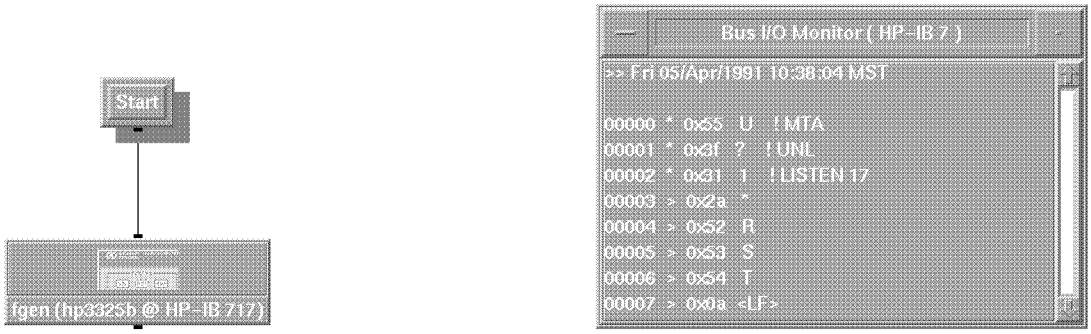


Figure 3-13. The Bus I/O Monitor

The display area of `Bus I/O Monitor` contains five columns:

- Column 1 - Line number
- Column 2 - Bus command (`*`), or outbound data (`>`), or inbound data (`<`)
- Column 3 - Hexadecimal value of the byte transmitted
- Column 4 - 7-bit ASCII character corresponding to the byte transmitted
- Column 5 - Bus command mnemonic (bus commands only, blank for data)

Note that the `Bus I/O Monitor` executes much faster as an icon than as an open view.

Low-level Bus Control

You can send low-level bus messages in two ways:

Object	Bus Message Capability
`Interface Operations`	This object allows you to send arbitrary bus messages to any HP-IB device, or reset the VXI interface and fire various VXI backplane trigger lines.
`Direct I/O`	`Direct I/O` objects for HP-IB and message-based VXI instruments allows you to send `CLEAR`, `LOCAL`, `REMOTE`, and `TRIGGER` commands using `EXECUTE` transactions.

For details about `Interface Operations` and `Direct I/O`, please refer to "Communicating with Instruments" in Chapter 6.

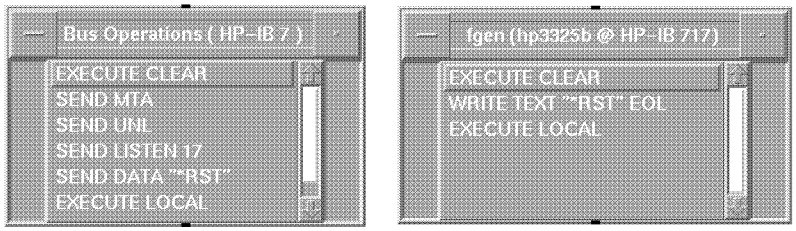


Figure 3-14. Two Methods of Low-Level HP-IB Control

Instrument Downloading

Some instruments allow you to download macros, measurement routines, or complete measurement programs. For example, some HP instruments support HP Instrument BASIC; you can write complete HP Instrument BASIC programs that execute inside the instrument. Here is one approach for using HP VEE to download a measurement routine to an instrument:

1. Create and maintain your measurement routine using a text editor, such as `vi`. Save the measurement routine in an ordinary text file.
2. Use `From File` to read the file.
3. Use `Direct I/O` to write the contents of the file to the instrument.

The following section presents a complete example of downloading using this approach. Please refer to Chapter 6 for details about `From File` and `Direct I/O`.

Example of Downloading

Figure 3-15 shows a program that downloads a measurement subprogram to the HP 3852A. This example downloads a simple subprogram, `BEEP2`, that beeps twice and displays a message.

Since the HP 3852A is not included in the default I/O configuration, you must follow these steps to open the on-line example:

1. Use `I/O` ⟹ `Instrument` . . . to add a device with the settings listed here. Enter these settings in the `Device Configuration` dialog box *exactly* including spaces:

   `Name: HP 3852A`

   `Interface: HP-IB`

   `Address:` Enter `0` if you do not have an HP 3852A connected to your computer. If you do have an HP 3852A, enter its address instead; the factory default is `709`.

   `Device Type: HP 3852A`

   `Timeout: 5`

   `Byte Ordering: MSB`

   `Live Mode:` Enter `OFF` if an HP 3852A is *not* connected to your computer or `ON` if an HP 3852A is connected.

2. Click on the `Save` button.

Here are the contents of the downloaded file `io_dload.dat`:

```
DISP MSG "LOADING BEEP2"
WAIT 1

SUB BEEP2
DISP "SUB BEEP2 CALLED"
BEEP
BEEP
SUBEND

DISP MSG "BEEP2 LOADED"
```

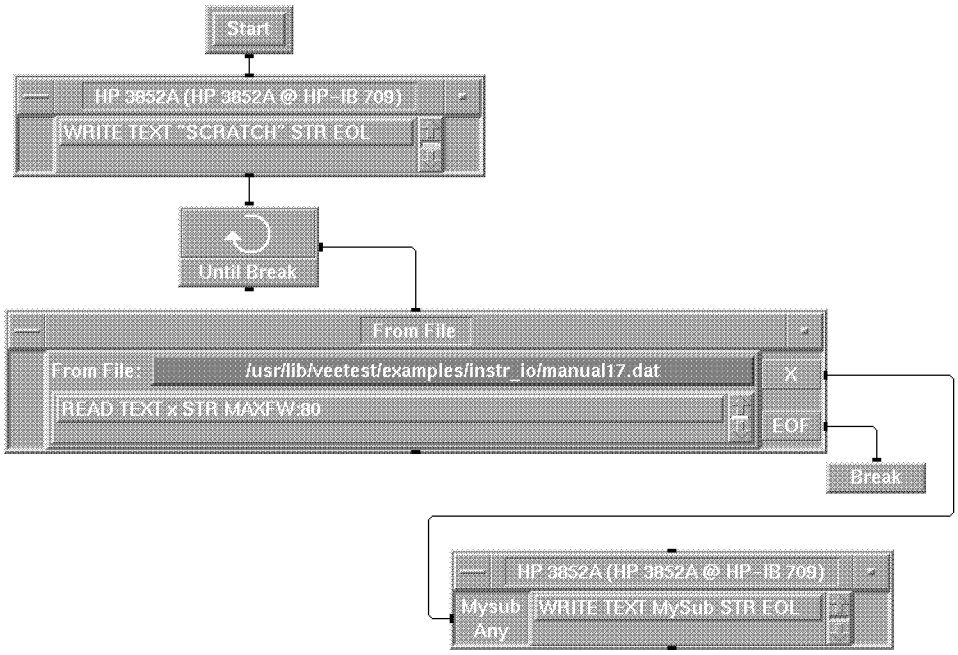


Figure 3-15. Downloading To An Instrument

The program shown in Figure 3-15 is saved in the file `manual17.vee` in the Instrument I/O section of your `examples` directory.

Instrument Control Troubleshooting

Problem	Remedy/Cause
Instruments do not respond at all.	All these conditions must be met: • Instruments must be powered on. • Instruments must be connected to the interface by a functioning cable. • The interface select code and instrument addresses must match settings in the **Address** field of the **Device Configuration** dialog box. The address for each physical instrument must be unique. • The **Live Mode** field in the Device Configuration dialog box must be set to **ON**. • You or your system administrator must properly configure HP VEE to work with instruments. Normally this is done during HP VEE installation; refer to the "Configuring HP VEE-Test" section of *Installing HP VEE*. • The system on which you are working may not be configured properly. You or your system administrator must properly configure the UNIX kernel with the proper drivers and/or interface cards.
HP VEE locks up while trying to communicate with an instrument.	(UNIX Only) If you are running HP VEE as a foreground process, position the cursor in the window in which you typed **veetest** and press CTRL-C (or the key indicated by the **intr** setting when you run the UNIX **stty** command). If you have problems with this, ask your system administrator for help. If your are running HP VEE as a background process, use the UNIX command **kill -2** *vee_pid*, where *vee_pid* is the process identification number for HP VEE. (MS-Windows Only) Position the cursor in the HP VEE for Windows window and press CTRL-C. If the program does not halt and return control to you press CTRL-ALT-DEL. Windows will display a menu allowing you to continue, abort the program or reboot your computer.

Instrument Control Troubleshooting (continued)

Problem	Remedy/Cause
Cannot determine the instrument address.	For GPIO and serial interfaces, the instrument address is the same as the interface select code. HP-IB instrument addresses are set by hardware switches or front panel commands. Older instruments use small switches located on the rear panel near the HP-IB connector. Newer instruments set and query the address via front panel buttons. Consult your instrument's programming manual for details. Refer to the section "Device Configuration Dialog Box" earlier in this chapter for examples of specifying addresses. VXI devices have logical addresses set by switches on the outside of the cards (usually the cards will have to be removed from the card cage to access the switches).
Cannot determine the interface select code.	In some cases, the select code is printed on the rear panel of the interface itself. Contact your system administrator for details about your hardware configuration. Refer to the section "Device Configuration Dialog Box" earlier in this chapter for examples of specifying addresses. These are the factory default select codes for commonly used interfaces: • HP-IB : 7 • Serial: 9, 17 • GPIO : 12 • VXI : 16

Related Reading

This section lists publications that contain more detailed information about instrument control topics discussed in this chapter.

- *Tutorial Description of the Hewlett-Packard Interface Bus* (Hewlett-Packard Company, 1987), part number 5021-1927.

  This document provides a condensed description of the important concepts contained in IEEE 488.1 and IEEE 488.2. If you are unfamiliar with the IEEE 488.1 interface, this is the best place to start.

- *IEEE Standard 488.1-1987, IEEE Standard Digital Interface for Programmable Instrumentation* (The Institute of Electrical and Electronics Engineers, 1987).

  This standard defines the technical details required to design and build an HP-IB (IEEE 488.1) interface. This standard contains electrical specifications and information on protocol that is beyond the needs of most programmers. However, it can be useful to clarify formal definitions of certain terms used in related documents.

- *IEEE Standard 488.2-1987, IEEE Standard Codes, Formats, Protocols, and Common Commands For Use with ANSI/IEEE Std 488.1-1987* (The Institute of Electrical and Electronics Engineers, 1987).

  This document describes the underlying message formats and data types used by instruments that implement the Standard Commands for Programmable Instruments (SCPI). It is intended more for instrument firmware engineers than for instrument users and programmers. However, you may find it useful if you need to know the precise definition of certain message formats, data types, or common commands.

- *IEEE Standard 728-1982, IEEE Recommended Practice For Code and Format Conventions For Use with ANSI/IEEE Std 488-1978, etc.* (The Institute of Electrical and Electronics Engineers, 1983).

- *VMEbus Extensions for Instrumentation*, including: "VXI-0, Rev. 1.0: Overview of VXIbus Specifications" and "VXI-1, Rev. 1.4: System Specification," VXIbus Consortium, Inc., 1992.

Related Reading

4

Using Records and DataSets

Using Records and DataSets

This chapter introduces two concepts: the **Record** data type and the **DataSet**, which is a collection of Record containers saved into a file for later retrieval. There are several HP VEE objects that allow you to create and manipulate records, including: `Record Constant`, `Build Record`, `UnBuild Record`, `Merge Record`, `SubRecord`, `Set Field`, and `Get Field`. The `To DataSet` and `From DataSet` objects allow you to store and retrieve records to and from DataSets. This chapter gives an overview of how to use the Record data type, and how to use DataSets to store Record containers. However, for specific information on an individual object, refer to the corresponding reference section in the *HP VEE Reference* manual.

Let's begin by looking at the Record data type.

Record Containers

A container of the Record data type has named data fields which can contain multiple values. Each field can contain another record, a scalar, or an array. Let's look at a simple record, created with the **Record Constant** object.

The **Record Constant** object allows you to create records "by hand." Just configure the **Record Constant** as a scalar (array elements = 0) or as an array (array elements = non-zero) with **Config** in the object menu. The **Record Constant** in the following example is configured as a record array with four array elements. The record consists of five fields: the Text fields **Name**, **Address**, and **City**, and the Int32 fields **EmplNo** and **Zip**. The **Record Constant** allows you to step through the record, from one array element to the next, with the **First**, **Prev**, **Next**, and **Last** buttons. You can edit each field as you go.

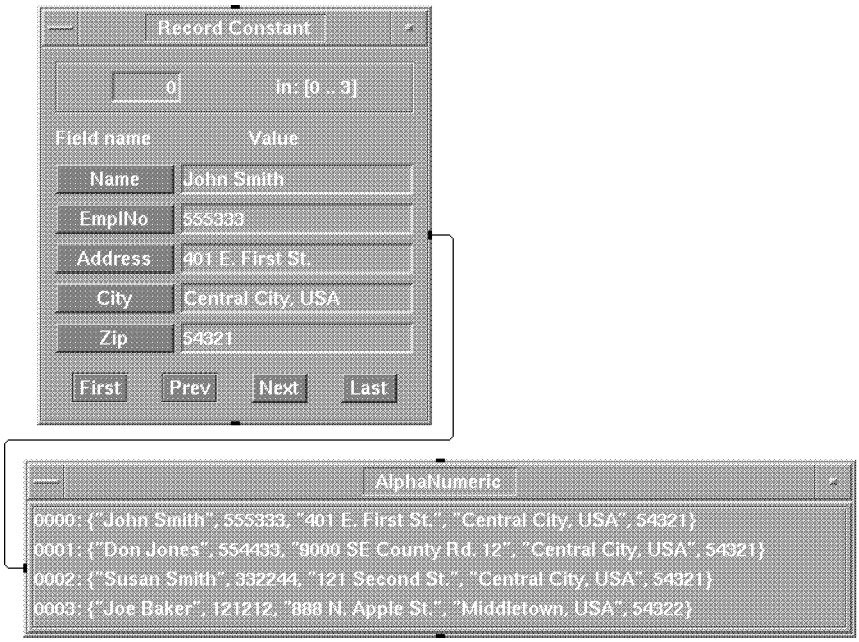


Figure 4-1. A Simple Record Container

When the program is run, the entire record is output on the `Record` output pin. The `AlphaNumeric` display shows the entire record, with four array elements (0000 through 0003), each consisting of five record fields enclosed in braces ("`{}`").

In the previous example we have seen how to output a record from a **Record Constant** and display the entire record in an **AlphaNumeric** display. This isn't very useful unless you can access the record and extract individual fields. Let's look at some ways to do this, using the same **Record Constant** example.

First, you can use the **Get Field** object to extract an individual field from the record. In the following example **Get Field** objects are used to extract the **Name** and **EmplNo** fields:

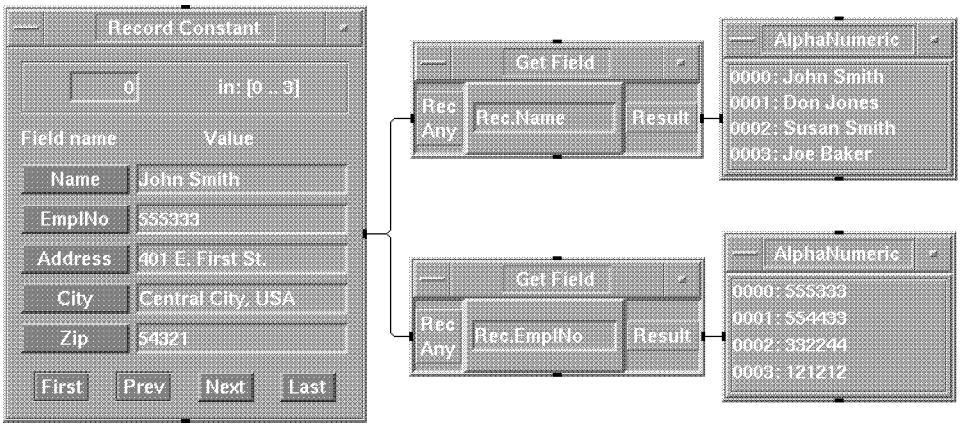


Figure 4-2. Retrieving Record Fields with Get Field

The "dot" syntax, for example: **Rec.Name** and **Rec.EmplNo**, is described in detail in "Using Records in Expressions" in Chapter 3 of the *HP VEE Reference* manual. Basically, **Rec.Name** means "get the **Name** field from the record on the **Rec** input pin." This syntax can be used in an expression in a **Formula** object, or in any other expression that is evaluated at run time. For example, you could use this syntax in a transaction in the **To String** object, but more about that later.

In the previous example, the entire **Name** and **EmplNo** fields were obtained, that is, the entire array for each field. But suppose you want the **Name** and **EmplNo** fields from a single array element. You can use the array syntax

`Rec[1].Name` and `Rec[1].EmplNo` to obtain just the second element ("element `1`") of each field:

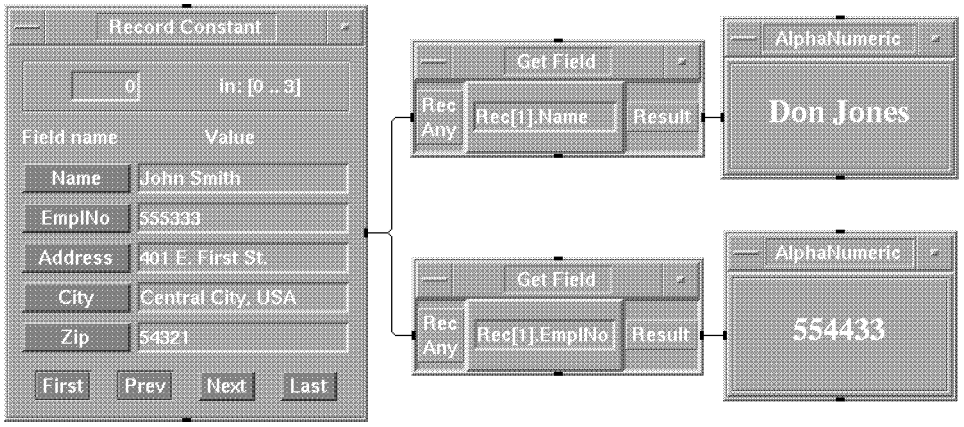


Figure 4-3. Using Array Syntax in Get Field

Another, often more efficient way to retrieve several or all fields from a record is to use the `UnBuild Record` object, as shown below:

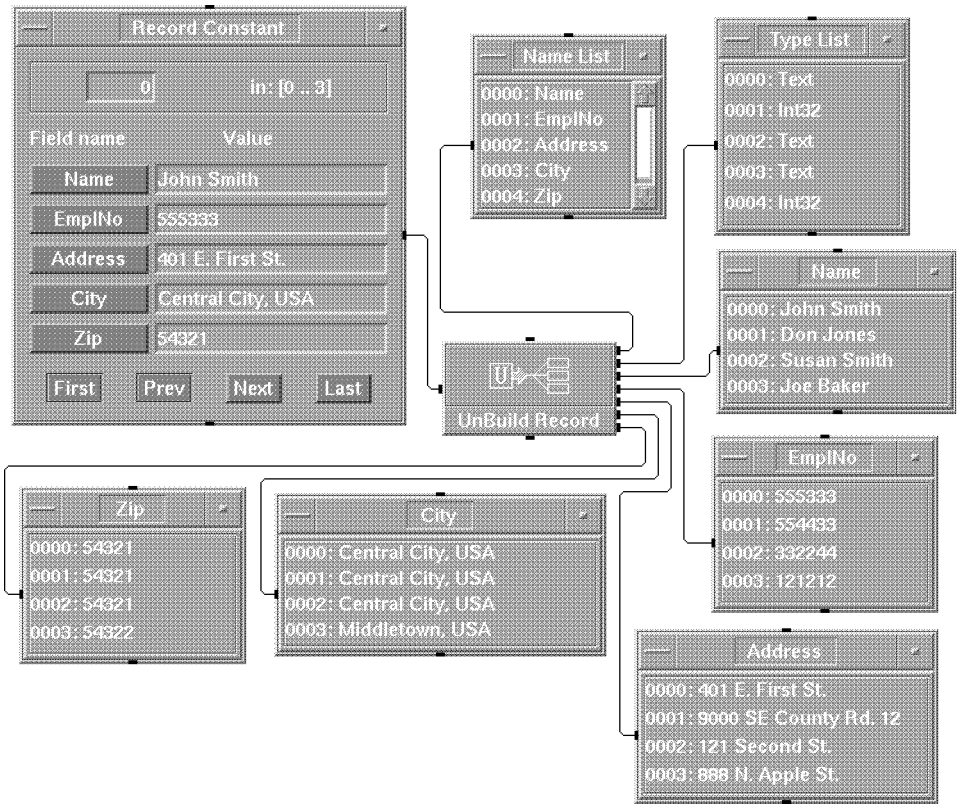


Figure 4-4. Retrieving Record Fields with UnBuild Record

The `UnBuild Record` object not only allows you to add outputs for every field in the record, but provides `Name List` and `Type List` outputs. These outputs list the name and type of each field in the record. To save space and make the program easier to read, `UnBuild Record` is shown in its icon view. However, just load the program and switch to the open view if you desire. The program is saved in the file `manual38.vee` in the Concepts section of your `examples` directory.

Building Records

Although the **Record Constant** object is useful to create and edit simple records, it would be cumbersome to create a large record that way. You may already have the data that you need in a file or in an array, and you may want to "build" a record from that data. In such cases, you can use **Build Record** to build a record just as you can use **UnBuild Record** to retrieve fields from a record.

NOTE

The Record data type has the highest precedence of all HP VEE data types. However, data cannot be converted to and from the Record data type through the automatic promotion/demotion of data types described in *Using HP VEE*. For example, you cannot send Record data into an input terminal constrained to be Real. Instead, you must perform these conversions by using **Build Record** and **Unbuild Record**, or with the **Rec.A** "dot" syntax described earlier.

When you build a record from individual data components with **Build Record**, you will have to define the data shape of the output Record container. The **Build Record** object gives you two **Output Shape** choices: **Scalar** and **Array 1D**. In most cases you will find that **Scalar**, the default, is the appropriate choice for **Output Shape**.

The following example shows the difference between **Scalar** and **Array 1D** in the output record built from two input arrays:

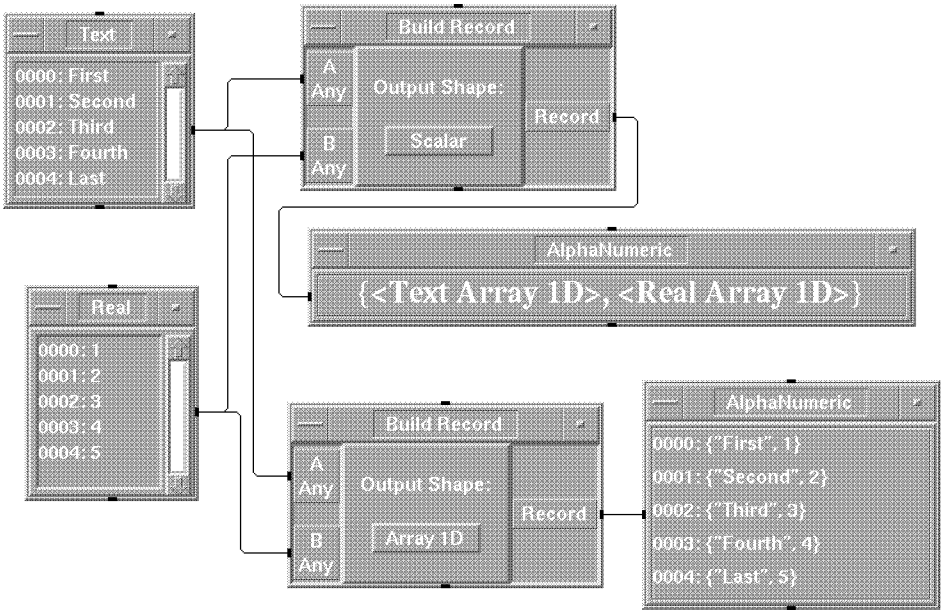


Figure 4-5. The Effect of Output Shape in Build Record

As you can see in the figure, when `Scalar` is selected, the output record is a scalar record consisting of two fields, each being one of the input arrays. On the other hand, when `Array 1D` is selected for the same input data, the output record is a record array with the same number of elements as the two input arrays. The data is matched, element for element, in the output record.

If two input arrays have different numbers of elements, only `Scalar` is allowed as the `Output Shape`. To create an `Array 1D` output record, all input arrays must have the same number of elements or an error will occur. However, you can mix scalar and array input data, as shown below:

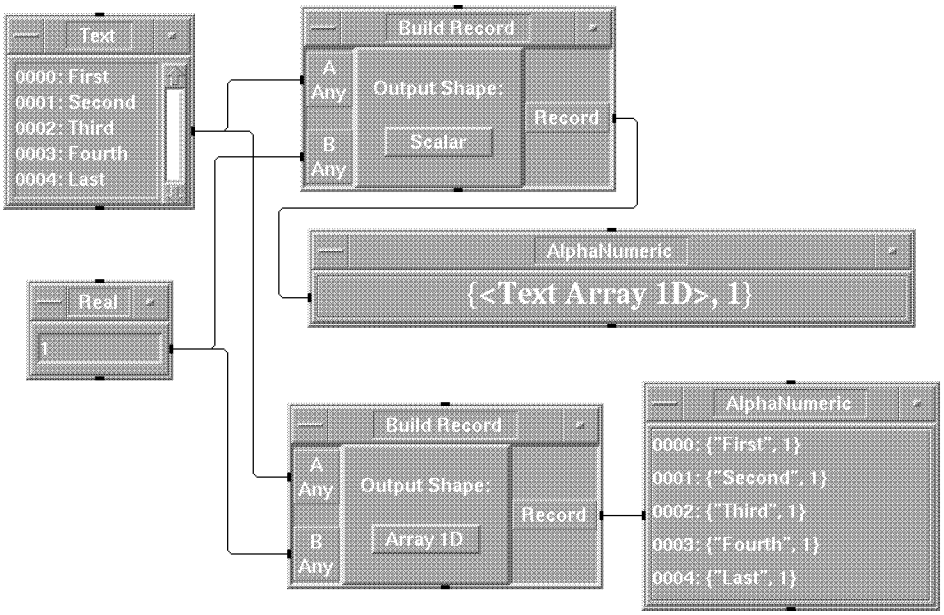


Figure 4-6. Mixing Scalar and Array Input Data

In this case, the scalar Real value `1` is repeated five times in the output record array if `Array 1D` is selected.

For further details, refer to the `Build Record` section in the *HP VEE Reference* manual.

Editing Record Fields

You can use the `Set Field` object to modify a field in a record. The `Set Field` object is an assignment statement consisting of a *left-hand expression* set equal to a *right-hand expression*. The left-hand expression specifies the field that you want to modify, so it is restricted to the "dot" syntax (for example, `Rec.A` or `Rec[1].A`). The right-hand expression can be any HP VEE expression. The right-hand expression is evaluated and the record

field specified by the left-hand expression is assigned that value. Let's look at an example:

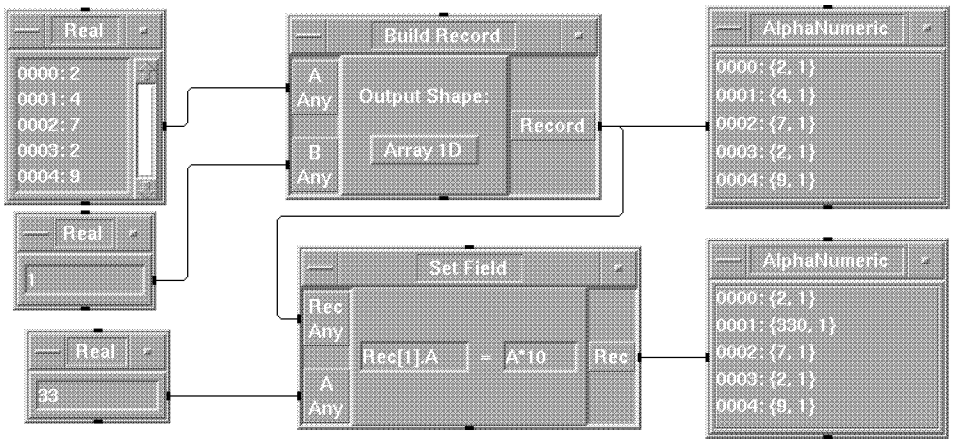


Figure 4-7. Using Set Field to Edit a Record

In the above example, a five element record array is built with `Build Record`. The `Set Field` object specifies that the field `Rec[1].A` (the `A` field of record element `1`) is to be assigned the value `A*10`. There is potentially a confusing point here. In the left-hand expression, the `A` in `Rec[1].A` refers to the `A` field of the record. However, in the right-hand expression, the `A` in `A*10` refers to the value at the `A` input of the `Set Field` object. Otherwise, the statement :

```
Rec[1].A = A*10
```

is very similar to a BASIC assignment statement such as:

```
A=B*10
```

At any rate, the variable `A` has the value 33, so `A*10` is evaluated as 330, which is assigned to `Rec[1].A`, as shown in the figure. Note that none of the other values of the record have changed.

For more information about the `Set Field` object, refer to the corresponding reference section in the *HP VEE Reference* manual.

Building Records Containing Waveforms

So far, we have only considered simple records containing scalar and array data. However, you'll find that the Record data type is very useful to contain waveform data. Let's look at an example:

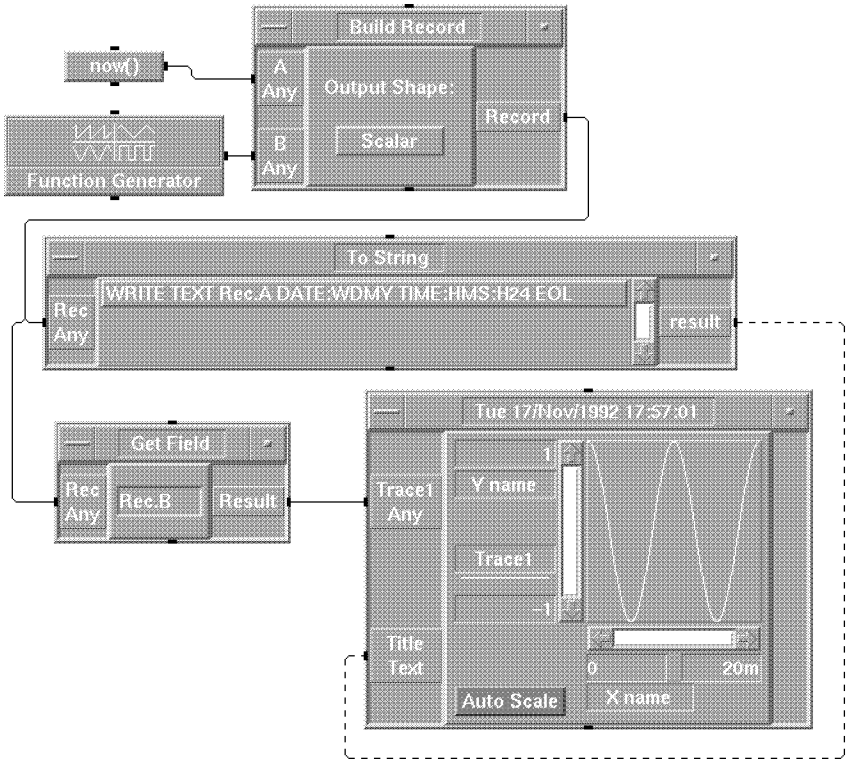


Figure 4-8. Building a Record from Waveform Data

In the above example the sine wave output by a `Function Generator` object and the current time from the `now()` function are built into a record. The sine wave is the `B` field of the record, so `Rec.B` in the `Get Field` object retrieves the waveform, which is output to an `XY Trace` object. But here's a useful "trick" — the time when the waveform was generated is displayed in

the title field of the `XY Trace` object. Here is how this works. First, the "dot" syntax `Rec.A` in the transaction in the `To String` object retrieves the time from field `A` in the record. Further, the transaction is configured to output the resulting time in the format `DATE:WDMY TIME:HMS:H24`. Thus, the date and time in that format is output to the `Title` control input on the `XY Trace` object. The title becomes: `Tue 29/Sep/1992 15:19:17`.

The above program is saved in the file `manual39.vee` in the Concepts section of your `examples` directory.

Using Global Records

In chapter Chapter 2 we briefly looked at global variables, and the `Set Global` and `Get Global` objects. We also looked at using global variables in UserObjects. In many cases you may have several related global variables of different data types and shapes. It is often useful to group these global variables together into one global variable, which is a record. Let's look at how to create and use global variables of the Record data type. The process is really quite simple. Just build the record with `Build Record` and output it to the `Set Global` object, as shown in the following example:

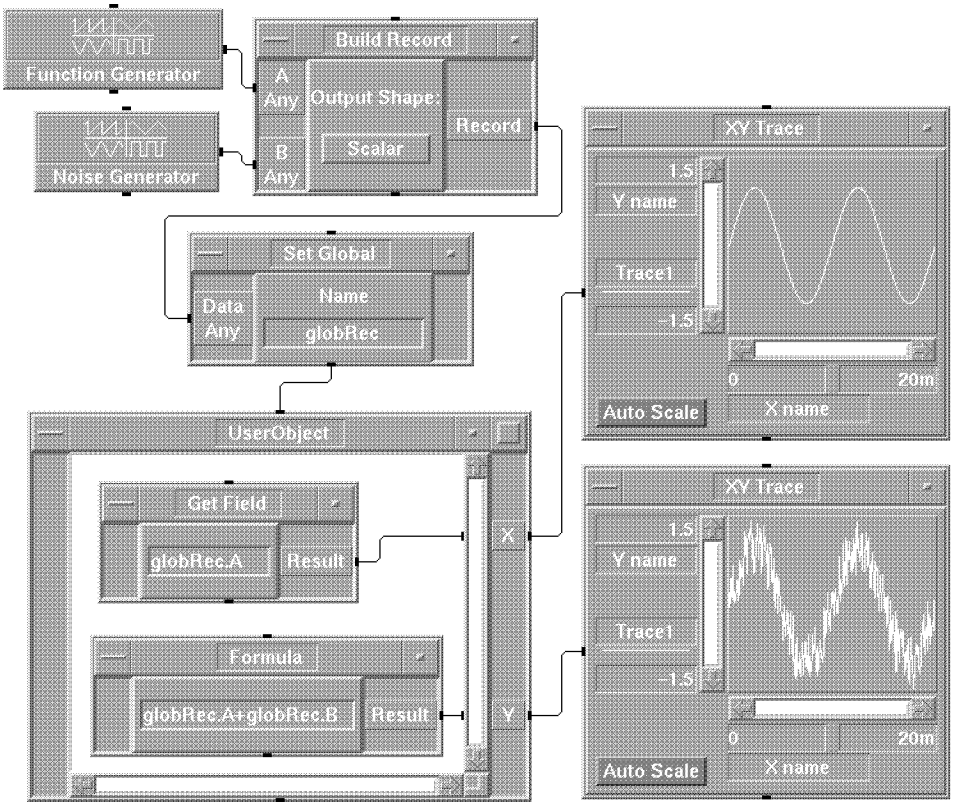


Figure 4-9. Using a Global Record

In the example, the output of the `Function Generator` and `Noise Generator` are built into a record, which is output to the `Set Global` object. `Set Global` creates the global variable named `globRec` (a record), which can be called with a `Get Global` or from an expression.

In the example, expressions in the `Get Field` and `Formula` objects, inside the UserObject, retrieve the waveform data from the global record. Note that a global record, like any global, is valid in any context, including a UserObject or User Function.

The expression `globRec.A` in the `Get Field` object retrieves field `A` of the record (the sine wave) and outputs it to the top `XY Trace` object. The expression `globRec.A+globRec.B` retrieves and adds both fields of the record, outputting the combined waveform (a noisy sine wave) to the bottom `XY Trace` object.

Using DataSets

As we have seen, HP VEE data (including waveforms) can be built into records and later retrieved. But what really makes this useful is the ability to store records using DataSets.

A **DataSet** is a collection of Record containers saved into a file for later retrieval. The `To DataSet` object collects Record data on its input and writes that data to a named file (the DataSet). Let's look at an example of how this is done.

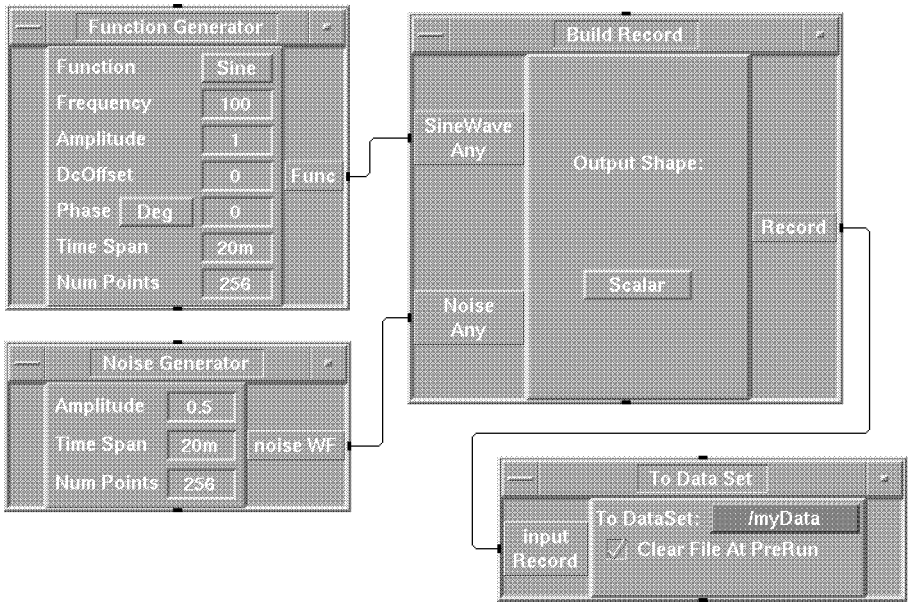


Figure 4-10. Using To DataSet to Save a Record

Two waveforms, a sine wave and a noise waveform, are output to the `Build Record` object, which builds a record. The record is then output to the `To DataSet` object, which writes the data to the file `myData`. Note that `Clear File at PreRun` is checked so that any "old" data already stored in `myData` will be cleared.

Once the data has been saved as a DataSet, you can use `From DataSet` to retrieve the record, which can then be unbuilt. The following program does this.

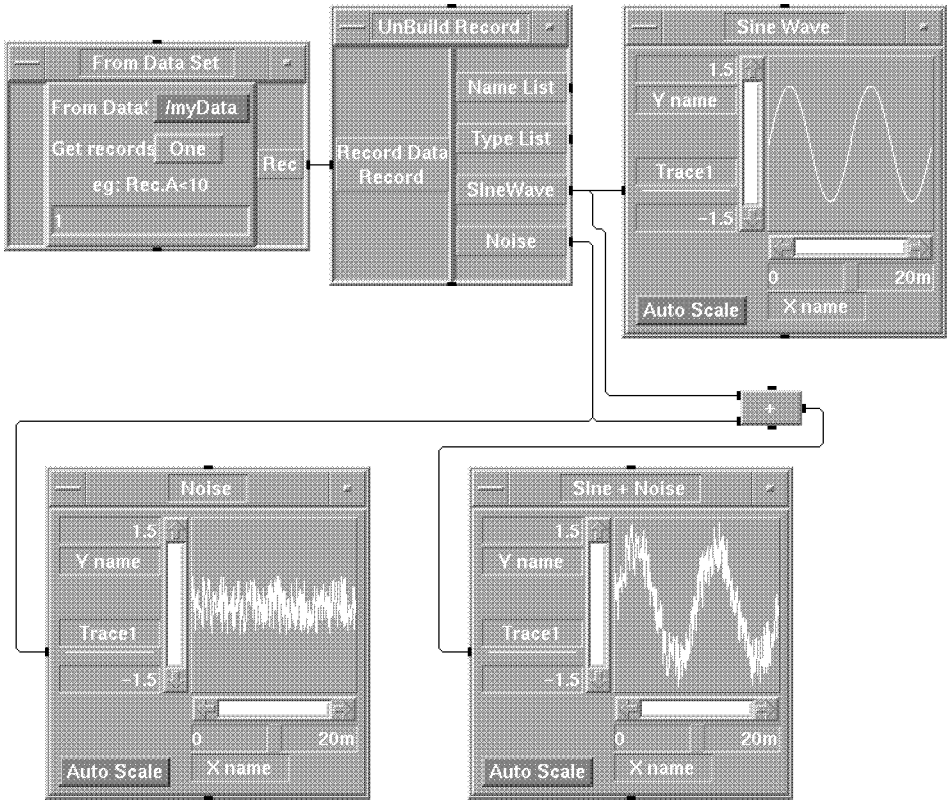


Figure 4-11. Using From DataSet to Retrieve a Record

The `From DataSet` object retrieves the record data from `myData`, and outputs the data to `Unbuild Record`, which separates out the sine wave and noise data fields. In this example, the sine wave, the noise waveform, and the sum of the two waveforms are each displayed in a separate `XY Trace` object.

The pair of programs of this last example are saved in the files `manual40.vee` and `manual41.vee` in the Concepts section of your `examples` directory.

For further information, refer to the `To DataSet` and `From DataSet` reference sections in the *HP VEE Reference* manual.

5

Creating User-Defined Functions

Creating User-Defined Functions

HP VEE supports three kinds of user-defined functions, the **User Function**, **Compiled Function**, and **Remote Function**. The method for creating each type of user-defined function, and for incorporating it into the HP VEE process, is different. However, all of these functions can be called using the `Call Function` object, or from certain expressions. Let's begin by looking at the easiest to use, the User Function.

User Functions

A User Function is a user-defined function created from a UserObject by executing `Make UserFunction` from the object menu. The User Function exists in the background within the HP VEE process, but provides the same functionality as the original UserObject. You can call a User Function with the `Call Function` object, or from certain expressions. The major advantage of creating a User Function is that you can call the same User Function several times in your program. Thus, there is only one User Function to edit and maintain, rather than several instances of a UserObject. A User Function can be created and called locally within an HP VEE program, or it can be saved in a library and imported into a program with `Import Library`.

Let's begin with an example of creating, editing, and calling a User Function locally within a program.

Creating a User Function

The first step in creating any User Function is to create a UserObject (refer to `UserObjects` in *Using HP VEE*). The following program contains a UserObject that adds a noise component to the waveform on its `Y` input, and then outputs the modified waveform on its `Y+Noise` output terminal. Note that the amplitude of the noise component is controlled by the `Real Slider` connected to the `Amplitude` input of the UserObject.

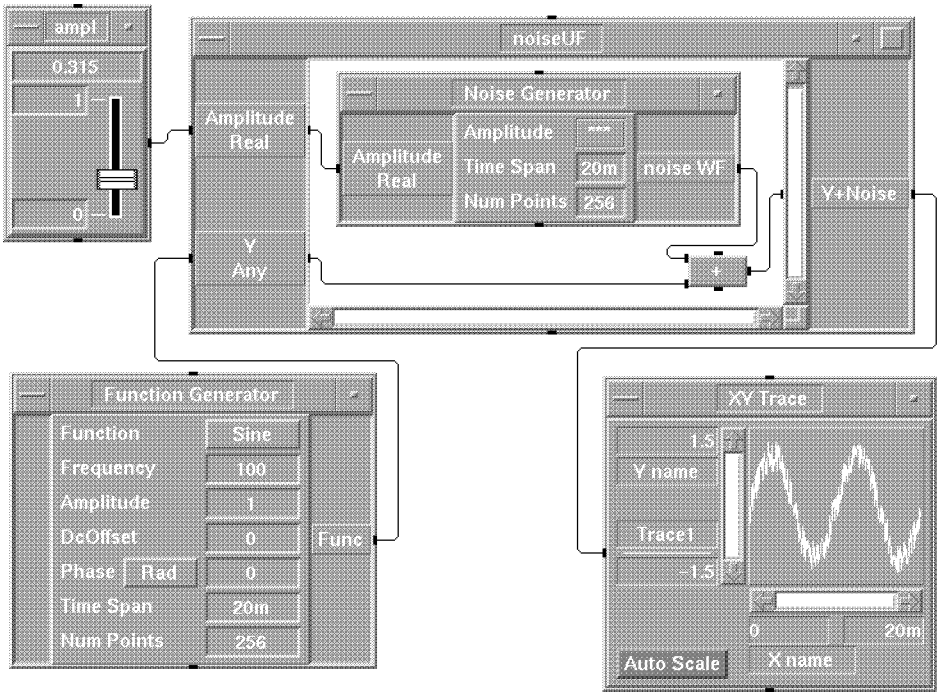


Figure 5-1. Program with UserObject

In our example, the UserObject adds a noise component to the 100 Hz sine wave output by the `Function Generator`. You may want to load this program and follow along with our example. The example is saved in the file `manual42.vee` in the Concepts section of your `examples` directory.

Note that the UserObject is named `noiseUF` in the example program. When you convert the UserObject into a User Function, the name in the title field will become the name of the User Function. You can then call the User Function by including this function name in a `Call Function` object, or in certain expressions.

NOTE

It is important that you enter the desired name for your User Function as the title of the original UserObject. If you leave the title field as `UserObject`, the User Function will end up with that name. If the name in the title field conflicts with any existing User Function, you will be prompted for a different name when you select `Make UserFunction`.

To convert the UserObject into a User Function, select `Make UserFunction` from the UserObject's object menu. The UserObject will disappear, being replaced by a `Call Function` object with the same input and output terminals, as shown on the next page.

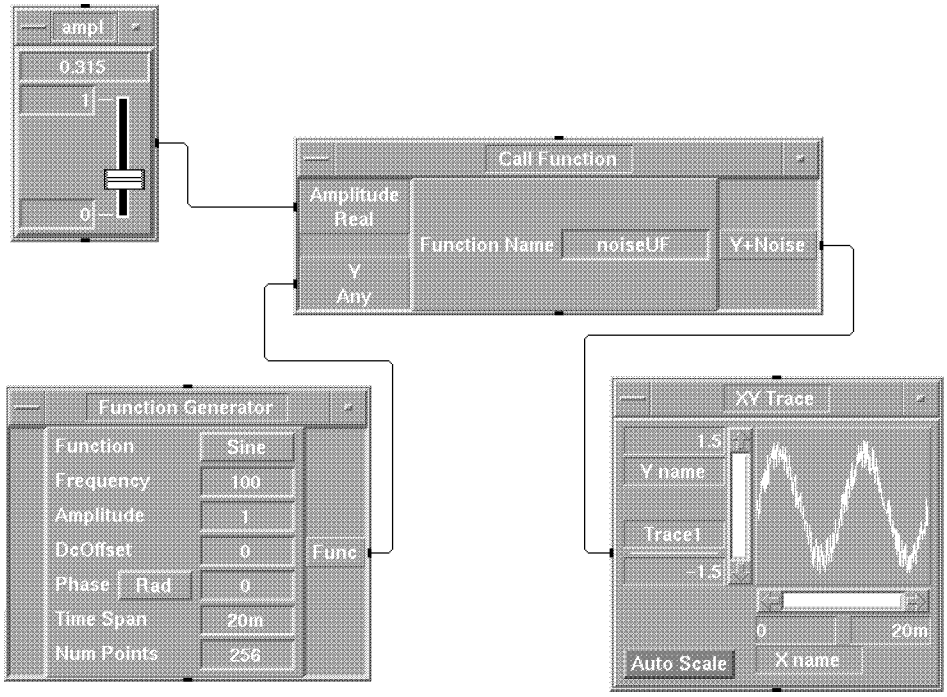


Figure 5-2. UserObject Replaced by Call Function

Actually, what happens is that the UserObject is converted into a User Function, named `noiseUF`, which exists in the "background" of the HP VEE process, but which contains the same functionality as the original UserObject. The `Call Function` object is automatically configured to call this User Function, and its pinout is automatically configured for that function. Thus, the connections in the original program are preserved.

When you run the program containing the User Function (called with `Call Function`), the result is the same as for the original program with the UserObject.

Editing a User Function

Now let's continue with the program of the previous section and edit the User Function. Just select `Edit UserFunction` (either from the `Call Function` object menu or from the `Edit` menu), and the `Edit UserFunction` dialog box appears, listing the available User Functions that you can edit. If you select `noiseUF` (the only choice in this case), the `Edit UserFunction` dialog box displays the following work area:

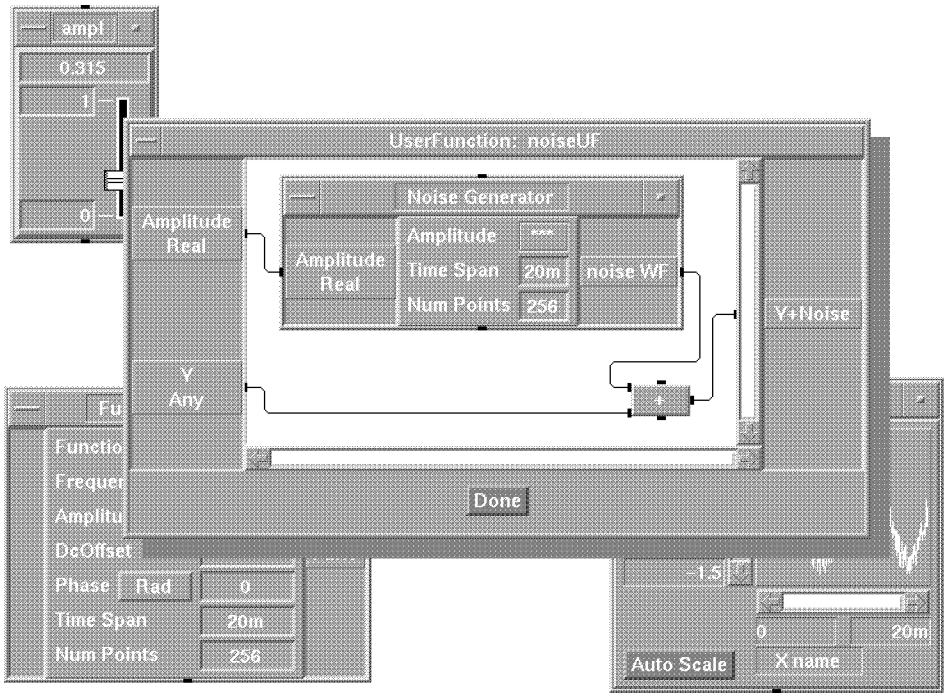


Figure 5-3. Editing a User Function

NOTE

At this point, if you want to reconvert the User Function back into a UserObject, just select `Make UserObject` from the object menu of the dialog box. If you do, the `Call Function` object will remain, but the User Function will be converted back into a UserObject. Note that you will have to provide another User Function of the same name before the `Call Function` can execute. This technique is useful when you want to test your program with a substitute User Function before importing the "real" User Function from a library.

You can edit the User Function in the work area just as you would the original UserObject. You can move and resize the work area, you can remove and add input and output terminals, and you can add and delete objects in the work area. In fact, you can do about anything that you can do in a UserObject work area, except you can't connect to any objects *outside* the User Function.

For example, suppose you want to use a global variable for the amplitude of the noise waveform. Just delete the `Amplitude` input pin from the User Function, add a `Get Global` object (with `ampl` in the `Name` field), and connect it to the `Amplitude` input pin of the `Noise Generator`, as shown in the following figure.

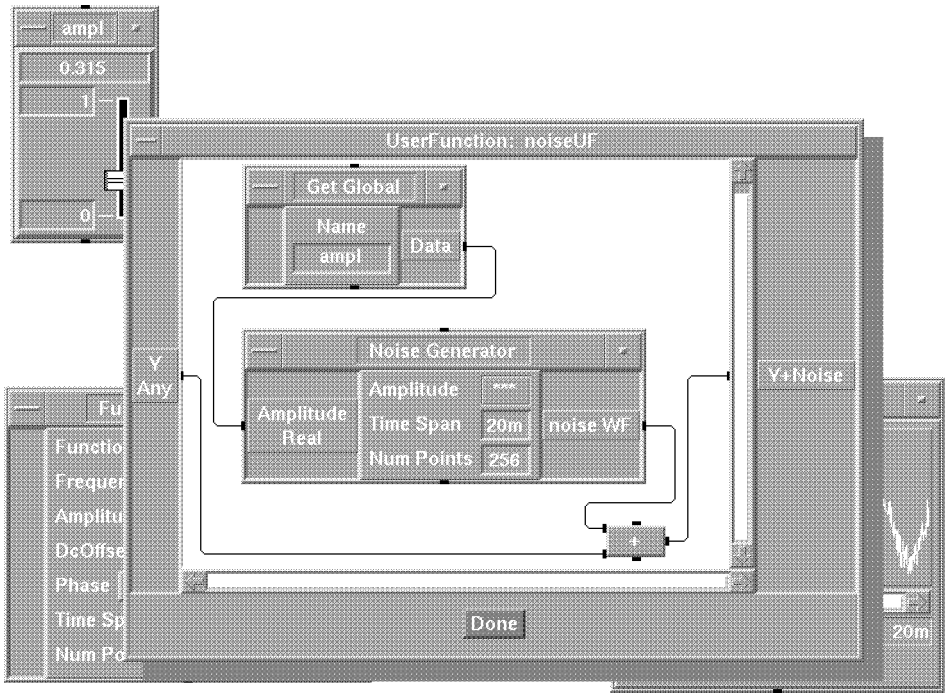


Figure 5-4. The Edited User Function

Click on `Done` when you have finished editing. To complete the program, add a `Set Global` object, and connect it as shown in the following figure. (To save space, we've iconized the `Function Generator` and we've selected `Graph Only` on the `XY Trace` so that we can reduce its size.)

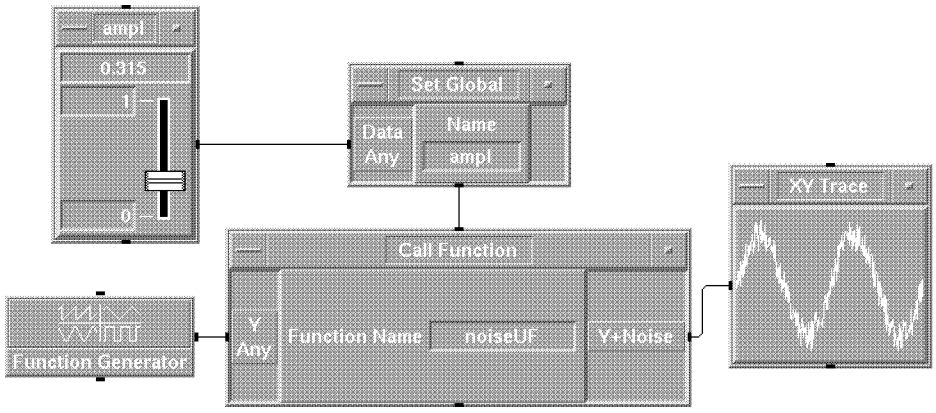


Figure 5-5. Program Using Edited User Function

This program performs the same task as the program of Figure 5-2, but uses a global variable to set the amplitude of the noise component. Note that the sequence output pin of the `Set Global` is connected to the sequence input pin of the `Call Function` object. This ensures that the global variable `ampl` will be set before it is called by the `Get Global` within the User Function.

You can call the same User Function several times by including multiple `Call Function` objects in your program. Let's add another `Call Function` object to our example program.

If you add a `Call Function` object by selecting `Device` ⟹`Function` ⟹ `Call`, you'll get the default `Call Function` object with no input or output terminals, and with `myFunction` as the called function. Just type in the User Function name `noiseUF`, or execute `Select Function` from the object menu and select `noiseUF` from the dialog box. In either case, the input and output terminals of the `Call Function` object is automatically configured for the selected function, provided that function is recognized by HP VEE. (Once your function is recognized, you can reconfigure the `Call Function` pinout at any time by selecting `Configure pinout` from the object menu.)

> **NOTE**
>
> As a "short cut," you could just clone the existing `Call Function` object in this case, since it is already configured for `noiseUF`.

In the following example we've used the second `Call Function` object to apply the User Function `noiseUF` to a different sine wave. (The first `Function Generator` outputs a 100 Hz sine wave, as before. The second `Function Generator` outputs a 50 Hz sine wave.)

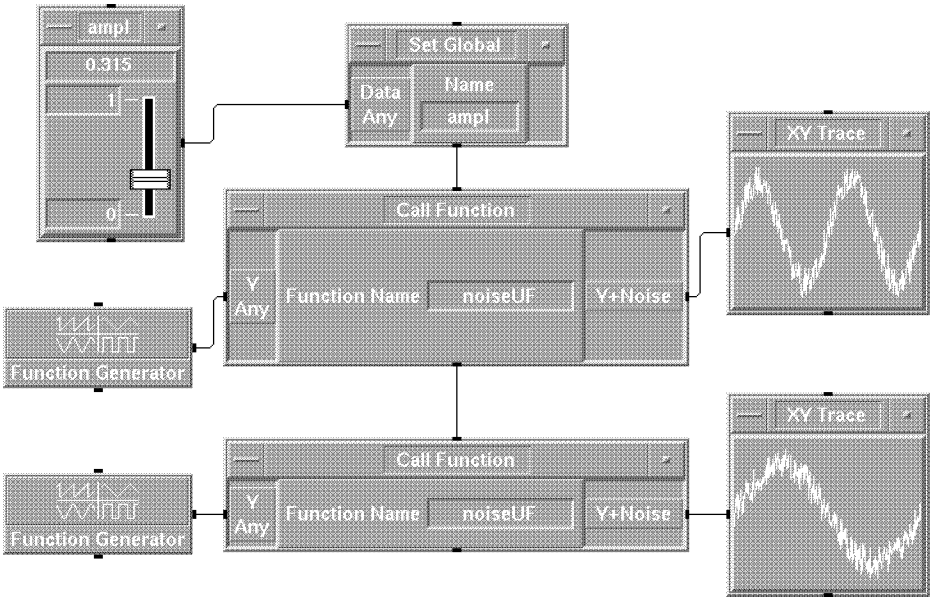


Figure 5-6. Using Multiple Call Function Objects

Calling a User Function from an Expression.

You don't have to use the `Call Function` object to call a User Function. In fact you can call a User Function from an expression in a `Formula` object, or from any expression evaluated at run time. The following program extends the example of Figure 5-5 by adding two `Formula` objects.

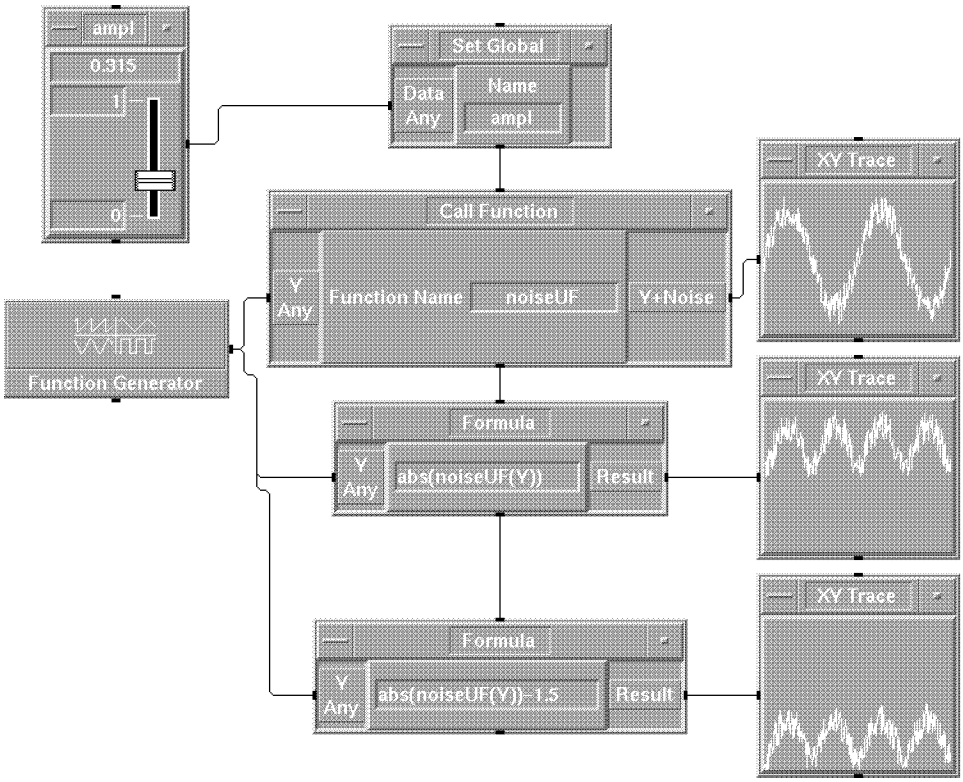


Figure 5-7. Calling a User Function from Expressions

In the program, the `Call Function` object calls the User Function `noiseUF` and returns a sine wave with an added noise component, as before. The expression `abs(noiseUF(Y))` in the first `Formula` object returns the absolute value of the waveform returned by the

User Function `noiseUF`. Thus, the displayed noisy sine wave is "rectified" in the positive direction. The expression `abs(noiseUF(Y))-1.5` in the second `Formula` object does the same, but also adds a negative "dc offset" to the waveform. Note that, as in the previous example, the sequence pins are used to ensure correct propagation with the global variable.

This program is saved in the file `manual43.vee` in the Concepts section of your `examples` directory.

The ability to call a User Function from an expression is very useful — especially when you include such an expression in a transaction in the `Sequencer` object. Refer to Chapter 7 for more information about this topic.

Creating a User Function Library

So far we have only looked at *local* User Functions, which are created and used within the same program. However, you can create a *library* of User Functions, save it in a file, and later import the library into a program.

To create a library of User Functions, you just create the individual User Functions in the HP VEE work area, and then save to a file. For example, suppose you want to create two User Functions, `myRand1` (which adds a random number, range 0 to 1, to an input value) and `myRand2` (which adds a random number, range 0 to 100, to an input value). You could start by creating the following UserObjects in a blank work area.

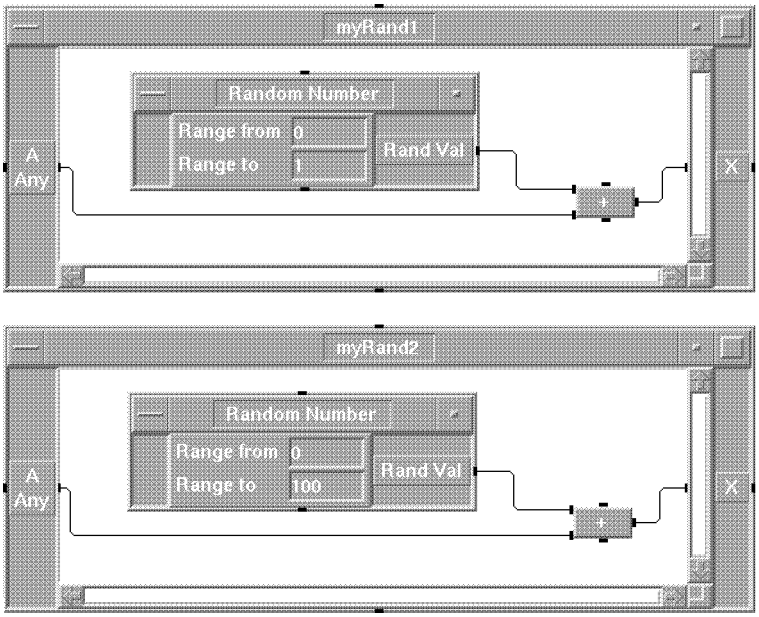


Figure 5-8. Creating UserObjects for a User Function Library

To create a User Function library, just execute `Make UserFunction` from the object menu for each UserObject, and then save to a file (for example, `user_func_libr`).

To import the User Function library into your program, use the `Import Library` object. For example, the following program imports the library from the file `user_func_libr` and calls the User Functions `myRand1` and `myRand2`.

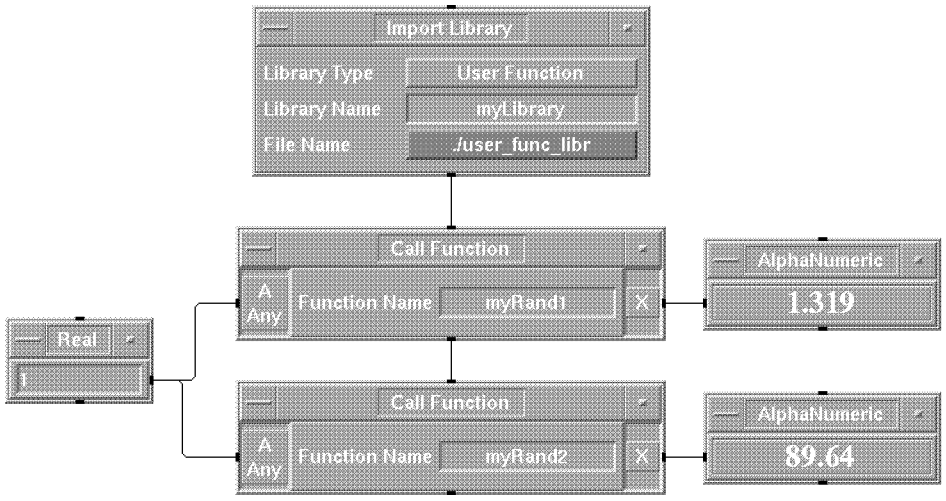


Figure 5-9. Importing a User Function Library

The `Import Library` object allows you to specify the type of library: `User Function`, `Compiled Function`, or `Remote Function`. For a `User Function` library, you can also specify a `Library Name` and `File Name`. The `File Name` field specifies the file from which to import the library, `user_func_libr` in this case. But what about the `Library Name` field? The `Library Name` just specifies a local name by which the library can be identified within the program. In this case, `Import Library` attaches the name `myLibrary` to the library imported from the file `user_func_libr`. This makes it possible for the `Delete Library` object to delete the library from the program.

Let's look at another example. In the following program, `Import Library` imports the User Function library from the file `user_func_libr` and attaches the name `myLibrary` to it. The User Functions `myRand1` and `myRand2` are called and their output values are set as global variables. At this point, the `Delete Library` object deletes `myLibrary` (both `myRand1` and `myRand2`) from the program, and then the `Formula` object evaluates an expression involving the global variables.

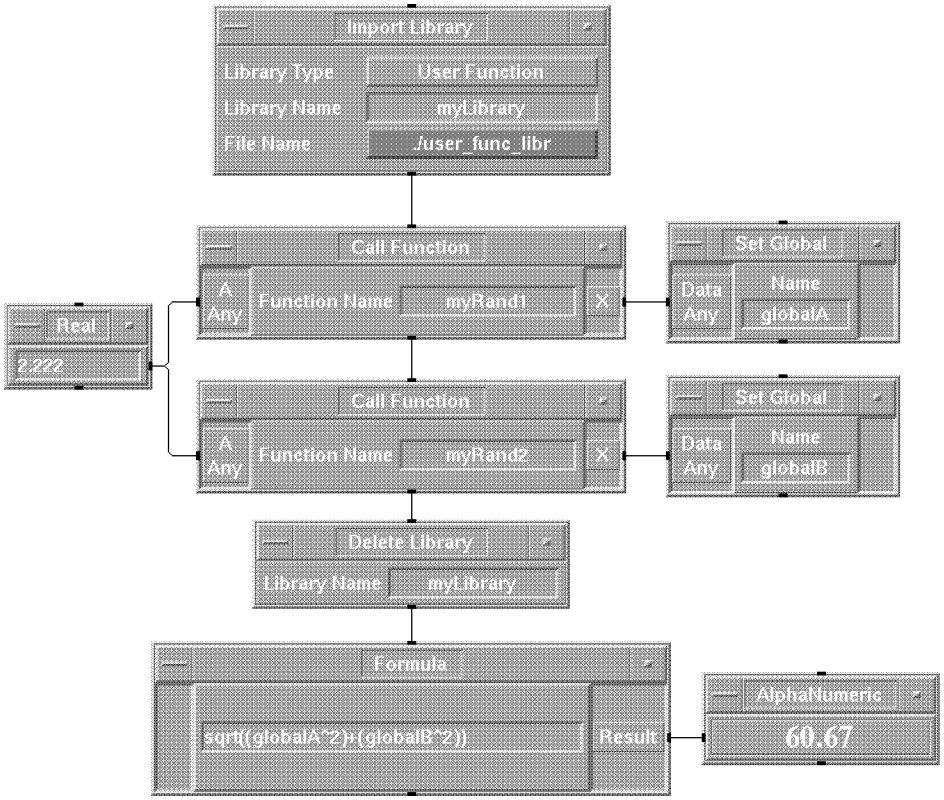


Figure 5-10. Importing and Deleting a User Function Library

Note that the sequence input and output pins are used to ensure correct propagation.

This program is simple, so it isn't really necessary to delete the User Function library. However, in a large program with multiple calls to large libraries, the ability to import a library, and then delete it when you no longer need it, significantly reduces the memory requirement.

NOTE

You cannot edit the User Functions imported with `Import Library` — you can only call them. However, you can *merge* a library of User Functions using `Merge Library` from the `File` menu.

Once the library is merged into your program, you can edit the individual User Functions with `Edit UserFunction`.

Compiled Functions

The second type of user-defined function is the Compiled Function, which is created by dynamically linking a program, written in C, C++, FORTRAN, or Pascal, into the HP VEE process. In order to use a Compiled Function, you will have to write the external program, create a shared library and definition file, and then import the library and call the function from HP VEE.

NOTE

Pascal shared libraries are supported only for HP 9000 Series 700 computers.

Basically, the methods for importing a Compiled Function library and for calling the function are very similar to what we've already discussed for User Functions. The `Import Library` object attaches the shared library to the HP VEE process and parses the definition file declarations. The Compiled Function can then be called with the `Call Function` object, or from certain expressions. On the other hand, you'll find that creating a Compiled Function is considerably more difficult than creating a User Function. Obviously, you cannot create a Compiled Function locally within an HP VEE program. Once you have written a program in C or another language, you'll have to create the shared library and definition file for the program to be linked.

Before we look at the process of creating and using Compiled Functions, let's look at some design considerations.

Design Considerations for Compiled Functions

There are several reasons for using Compiled Functions in your HP VEE program. You can develop your own data filters in another language and

integrate them directly into your HP VEE program by using Compiled Functions. Also, you can use Compiled Functions as a means of providing security for proprietary routines. Although you can extend the capabilities of your HP VEE program by using Compiled Functions, it is at the expense of adding complexity to the HP VEE process. *The key design goal should be to keep the purpose of the external routine highly focused on a specific task, and to use Compiled Functions only when the capability or performance that you need is not available using an HP VEE User Function, or an* `Execute Program` *escape to the operating system.*

You can use any facilities available to the operating system in the program to be linked. These include math routines, instrument I/O, and so forth. *However, you cannot access any of the HP VEE internals from within the external program to be linked.*

Although the use of Compiled Functions provides enhanced HP VEE capabilities, there are some pitfalls. Here are a few key ones:

- HP VEE can't trap errors originating in the external routine. Because your external routine becomes part of the HP VEE process, any errors in that routine will propagate back to HP VEE, and a failure in the external routine may cause HP VEE to "hang" or otherwise fail. Thus, you need to be sure of what you want the external routine to do, and provide for error checking in the routine. Also, if your external routine exits, so will HP VEE.
- Your routine must manage all memory that it needs. Be sure to deallocate any memory that you may have allocated when the routine was running.
- Your external routine cannot convert data types the way HP VEE does. Thus, you should configure the data input terminals of the `Call Function` object to accept *only* the type and shape of data that is compatible with the external routine.
- If your external routine accepts arrays, it must have a valid pointer for the type of data it will examine. Also, the routine must check the size of the array on which it is working. The best way to do this is to pass the size of the array from HP VEE as an input to the routine, separate from the array itself. If your routine overwrites values of an array passed to it, use the return value of the function to indicate how many of the array elements are valid.
- System I/O resources may become locked. Your external routine is responsible for timeout provisions, and so forth.

Importing and Calling a Compiled Function

Once you have created a dynamically linked library, you can import the library into your HP VEE program with the `Import Library` object and then call the Compiled Function with the `Call Function` object. The process is very much like that of importing a library of User Functions and then calling the functions, as described at the beginning of this chapter.

We've already discussed the `Import Library` object in the "User Functions" section at the beginning of this chapter. To import a Compiled Function library, just select `Compiled Function` in the `Library Type` field. Just as for a User Function, the `Library Name` field attaches a name to identify the library within the program, and the `File Name` field specifies the file from which to import the library. In addition, there is a fourth field, which specifies the name of the `Definition File`:

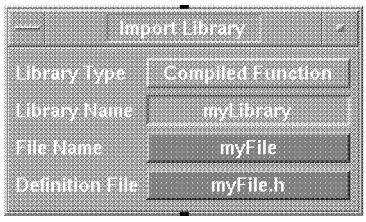


Figure 5-11. Using Import Library for Compiled Functions

The definition file defines the type of data that is passed between the external routine and HP VEE. We'll discuss this file later.

Once you have imported the library with `Import Library`, you can call the Compiled Function by specifying the function name in the `Call Function` object. For example, the `Call Function` object below calls the Compiled Function named `myFunction`.

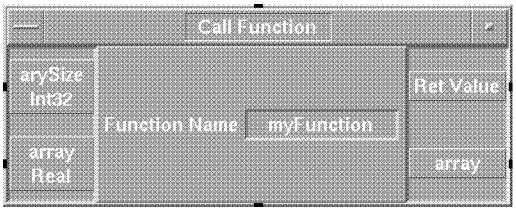


Figure 5-12. Using Call Function for Compiled Functions

You can select a Compiled Function just as you would select a User Function, as described earlier in this chapter. You can either select the desired function using `Select Function` from the `Call Function` object menu, or you can type in the name. In either case, provided HP VEE recognizes the function, the input and output terminals of the `Call Function` object will be configured automatically for the function. (The necessary information is supplied by the definition file.) Or, you can reconfigure the `Call Function` input and output terminals by selecting `Configure pinout` in the object menu. Whichever method you use, the HP VEE will configure the `Call Function` object with the input terminals required by the function, and with a `Ret Value` output terminal for the return value of the function. In addition, there will be an output terminal corresponding to each input that is passed by reference.

You can also call the Compiled Function by name from an expression in a `Formula` object, or from other expressions evaluated at run time. For example, you could call a Compiled Function by including its name in an expression in a `Sequencer` transaction. Note, however, that only the Compiled Function's return value (`Ret Value` in the `Call Function` object) can be obtained from within an expression. If you want to obtain other parameters from the function, you will have to use the `Call Function` object.

Creating a Compiled Function (UNIX)

There are several steps to the process of creating a Compiled Function. First you have to write a program in C, C++, FORTRAN, or Pascal (HP 9000 Series 700 only), and write a definition file for the function. Then you have to create a shared library containing the Compiled Function, and bind the shared library into the HP VEE process. We'll look at each step in turn. But first, let's look at the structure of the definition file.

The Definition File

The `Call Function` object determines the type of data it should pass to your function based on the contents of the definition file you provide. The definition file defines the type of data the function returns, the function name, and the arguments the function accepts. The function definition is of the following general form:

```
<return type> <function name> (<type> <paramname>, <type>
<paramname>, ...)
```

Where:

- `<return type>` can be: `long`, `double`, or `char*`.
- `<function name>` can be a string consisting of an alpha character followed by alphanumeric characters, up to a total of 512 characters.
- `<type>` can be: `long`, `double`, `char*`, `long*`, `double*`, or `char**`.
- `<paramname>` can be a string consisting of an alpha character followed by alphanumeric characters, up to a total of 512 characters. The parameter names are optional, but it is recommended to include them. If a parameter is to be passed by reference, the parameter name must be preceded by the indirection symbol (`*`).

The valid return types are character strings (`char*`, corresponding to the HP VEE Text data type), long integers (`long`, corresponding to the HP VEE Int32 data type), and double precision floating point real numbers (`double`, corresponding to the HP VEE Real data type).

If you specify "pass by reference" for a parameter by preceding the parameter name with `*`, HP VEE will pass the address of the information to your function. If you specify "pass by value" for a parameter by leaving out the `*`, HP VEE will copy the value (rather than the address of the value) to your function. You'll want to pass the data by reference if your external

routine changes that data for propagation back to HP VEE. *Also, all arrays must be passed by reference.*

Any parameter passed to a Compiled Function by reference will be available as an output terminal on the `Call Function` object. That is, the output terminals will be `Ret Value` for the function's return value, plus an output for each input parameter that was passed by reference.

HP VEE can only push 80 bytes on the stack. This means that it can allow up to 20 parameters to be passed by reference to a Compiled Function. This would also imply that up to 20 long integer parameters, or up to 10 double precision floating point parameters, may be passed by value.

NOTE

You must have the ANSI C compiler in order to generate the position independent code needed to build a shared library for a Compiled Function.

You may include comments in your definition file. The ANSI C conventions are used, allowing both "enclosed" comments and "to-end-of-line" comments. "Enclosed" comments use the delimiter sequence:

```
/*comments*/
```

Where `/*` and `*/` mark the beginning and end of the comment, respectively.

"To-end-of-line" comments use the delimiting characters `//` to indicate the beginning of a comment that runs to the end of the current line.

Building a C Function

Now let's look at an example of building an external routine. We'll use the C language in this example.

The following C function accepts a real array and adds `1` to each element in the array. The modified array is returned to HP VEE on the `Array` terminal, while the size of the array is returned on the `Ret Value` terminal. This function, once linked into HP VEE, becomes the Compiled Function called in the HP VEE program shown in Figure 5-13.

```
/*
  C code from manual49.c file
*/
#include <stdlib.h>

long myFunc(long arraySize, double *array)
{
        long i;

        for(i=0; i<arraySize; i++, array++){
                *array += 1.0;
        } /* for */

        return(arraySize);
} /* end myFunc() */
```

The definition file for this function is as follows:

```
/*
definition file for manual49.c
*/

long myFunc(long arraySize, double *array);
```

(This definition is exactly the same as the ANSI C prototype definition in the C file.)

Although this example is simple, it illustrates some important points.

First, you must include any header files on which the routine depends. In this case, the `stdlib.h` file isn't really necessary — it is there just to illustrate the point.

The example program uses the ANSI C function prototype. This isn't necessary, but it makes things a little easier to understand. The function prototype declares the data types that HP VEE should pass into the function. The array has been declared as a pointer variable. HP VEE will put the addresses of the information appearing on the `Call Function` data in

terminals into this variable. The array size has been declared as a long integer. HP VEE will put the value (not the address) of the size of the array into this variable. The positions of both the data input terminals and the variable declarations are important. The addresses of the data items (or their values) supplied to the data input pins (from top to bottom) are placed in the variables in the function prototype from left to right.

Note that one variable in the C function (and correspondingly, one data input terminal in the `Call Function` object) is used to indicate the size of the array. The `arraySize` variable is used to prevent data from being written beyond the end of the array. If you overwrite the bounds of an array, the result depends on the language you are using. In Pascal, which performs bounds checking, a run-time error will result, stopping HP VEE. In languages like C, where there is no bounds checking, the result will be unpredictable, but intermittent data corruption is probable.

Our example has passed a pointer to the array, so it is necessary to de-reference the data before the information can be used.

The `arraySize` variable has been passed by value, so it won't show up as a data output terminal. However, here we've used the function's return value to return the size of the output array to HP VEE. This technique is useful when you need to return an array that has fewer elements than the input array.

Note that the C routine is a function, not a procedure. The Compiled Function requires a return value, so if you use a language that distinguishes between procedures and functions, (FORTRAN, for example) make sure you write your routine as a function.

The following HP VEE program calls the Compiled Function created from our example C program:

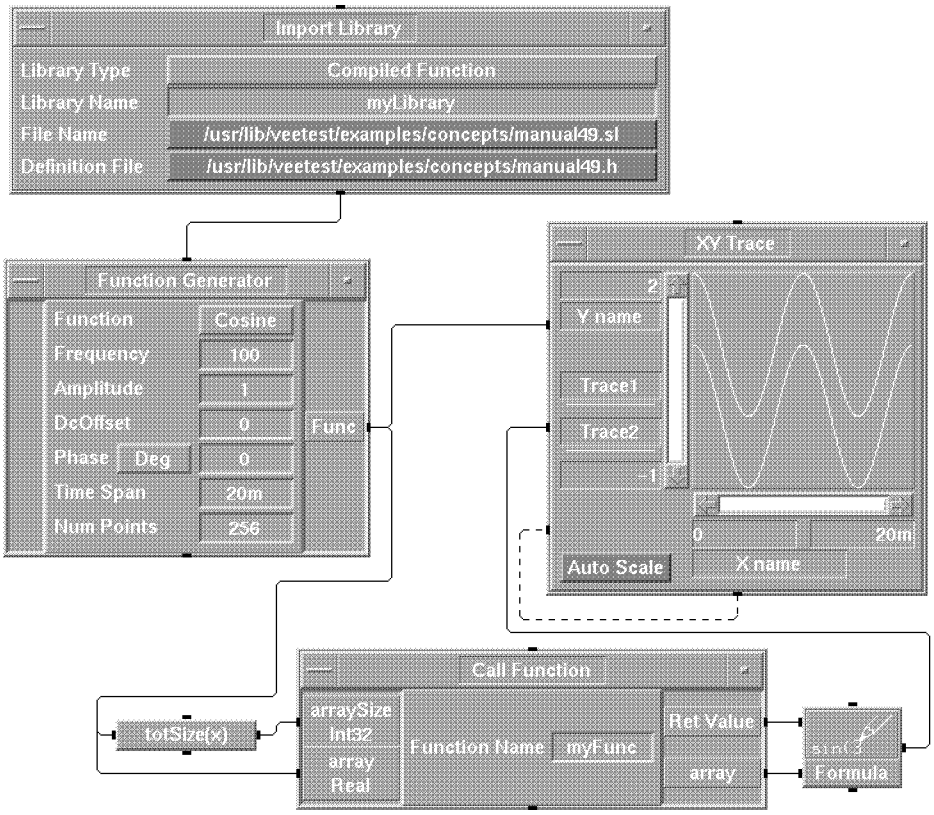


Figure 5-13. Program Calling a Compiled Function

However, before you can run the program, you'll have to create a shared library to be linked to the HP VEE process.

The example in Figure 5-13 is saved in the file `manual49.vee` in your `examples` directory. The C file is saved as `manual49.c`, the definition file as `manual49.h`, and the shared library as `manual49.sl`. (On the SPARCstation, the shared library is saved as `manual49.so`.)

Creating a Shared Library

To create a shared library, your function must be compiled as position-independent code. This means that, instead of having entry points to your routines exist as absolute addresses, your routine's symbol table will hold a symbolic reference to your function's name. The symbol table is updated to reflect the absolute address of your named function when the function is bound into the HP VEE environment. It must then be linked with a special option to create a shared library.

Let's suppose that our example C routine is in the file named `dLink.c`. To compile the file to be position independent, you can use the `+z` compiler option. You also need to prevent the compiler from performing the link phase by using the `-c` option. Thus, the compile command would look like this:

```
cc -Aa -c +z dLink.c
```

This produces an output file named `dLink.o`, which you can then link as a shared library with the following command:

```
ld  -b dLink.o
```

The `-b` option tells the linker to generate a shared library from position-independent code. This produces a shared library named `a.out`. Alternatively, you could use the command:

```
ld -b -o dLink.sl dLink.o
```

to obtain an output file (through the use of the `-o` option) called `dLink.sl`.

NOTE

For the SunOS 4.1.2 and 4.1.3 the above commands would be

```
cc -pic -c dLink.c

ld -o dLink.so -assert pure-text dLink.o
```

Binding the Shared Library

HP VEE binds the shared library into the HP VEE process. All you need to do is include an `Import Library` object in your program, specifying the library to import, and then call the function by name (i.e., with a `Call Function` object). When `Import Library` executes, HP VEE binds the shared library and makes the appropriate input and output terminals available to the `Call Function` object (`Configure Pinout` will now work). The shared library remains bound to the HP VEE process until HP VEE terminates, or until the library is expressly deleted.

You can delete the shared library from HP VEE either by selecting `Delete Lib` from the `Import Library` object menu, or by including the `Delete Library` object in your program. Note, however, that you may have more than one library name pointing to the same shared library file. In this case, you can use the `Delete Library` object to delete each library, but the shared library will remain bound until the last library pointing to it is deleted. However, the `Delete Lib` selection in the `Import Library` object menu will unbind the shared library with no regard to how may other `Import Library` objects have been executed.

When HP VEE binds a shared library, it defines the input and output terminals needed for each Compiled Function. When you select a Compiled Function for a `Call Function` object, or when you execute a `Configure Pinout`, HP VEE automatically configures `Call Function` with the appropriate terminals. The algorithm is as follows:

- The appropriate input terminals are created for each input parameter to be passed to the function (by reference or by value).
- An output terminal labeled `Ret Value` is configured to output the return value of the Compiled Function. This is always the top-most output pin.
- An output terminal is created for every input that is *passed by reference*.

The names of the input and output terminals (except for `Ret Value`) are determined by the parameter names in the definition file. However, the values output on the output terminals are a function of position, not name. Thus, the first (top-most) output pin is always the return value. The second output pin returns the value of the first parameter passed by reference, and so forth. This is normally not a problem unless you add terminals after the automatic pin configuration.

Creating a Dynamic Linked Library (MS-Windows)

HP VEE for Windows provides access to Dynamic Linked Libraries (DLL) through the `Call` object and through formula objects. Only DLL's specifically written for HP VEE will work with these objects because HP VEE does not support 8-bit characters, 16-bit integers or 32-bit reals.

> **NOTE**
>
> Writing DLL's requires considerable experience with Microsoft Windows C programming. This is not a task for the novice programmer. Contact your system administrator or Microsoft for further help with writing DLL's.
>
> This section tells you how to *call* a DLL, *not* how to write a DLL.

Creating the DLL

Create your DLL before writing your HP VEE program. Create your DLL as you would any other DLL with the following exceptions:

- Only a subset of C types are allowed. (See "Creating the Definition File" below.)
- Declare DLL functions as `_far _cdecl` (not `_far _pascal`) in the source code and with an underscore in the Windows definitions (`.DEF`) file.

Declaring DLL Functions. To work with HP VEE, DLL functions must be declared as `_far _cdecl` (not `_far _pascal`) in the source code and with an underscore in the Windows definitions (`.DEF`) file. For example, a generic function could be created as follows:

```
long _far _cdecl genericFunc(long a)
{
    return(a * 2);
}
```

The `.DEF` file would then contain:

```
EXPORTS _genericFunc
```

Creating the Definition File. The definition file contains a list of prototypes of the imported functions. HP VEE uses this file to configure the `Call` objects and to determine how to pass parameters to the DLL function. The format for these prototypes is:

```
<type>  <name>([ [<type> <pointer> [<name>]][,<type> <pointer> [<name>]]*])
```

where:

`<type>`	defines the data type `'long'|'double'|'char'|'*veedata'`
`<name>`	is a blank delimited string.
`<pointer>`	is `'*'|' '` .

and:

`long`	a 4-byte signed long integer.
`double`	an 8-byte double precision floating point.
`char*`	a far pointer to a null terminated string.
`veedata`	a pointer to a structure that contains information about the HP VEE container.

Examples.

```
long aFunc(double *,long param2,        Pass in four parameters, return a long
           long *param3, char *)
double aFunc()                          No input parameters, return a double
long aFunc(char *aString)               Pass in a string, return a long
aFunc(char *aString)                    Pass in an array of strings, return a string
```

Parameter Limitations

A DLL function call can only push 80 bytes on the stack. This limits the number of parameters used by the function. Any combination of parameters may be used as long as the 80 bytes limit is not exceeded. A long uses four bytes, a double uses eight bytes and a pointer uses four bytes. For example, a function could have 20 longs, 10 doubles, or 10 pointers and 5 doubles.

veedata type

The `veedata` type defines the state of an HP VEE container including type, size, number and size of dimensions, mappings, etc. The user DLL function can also send a `veedata` struct which HP VEE then uses to construct a container for the return value of the function call.

The IMPORT LIBRARY Object

Before you can use a `Call` object or `Formula` box to execute a DLL function you must import the function into the HP VEE environment via the `Import Library` object. On the `Import Library` object select `Compiled Function` under `Library Type`. Enter the correct definition file name using the `Definition File` button. Finally, select the correct file using the `File Name` button. The `Library Name` button assigns a logical name to a set of functions and does not need to be changed.

The CALL FUNCTION Object

Before using a DLL function with the `Call Function` object you must configure the `Call Function` object. The easiest way to do this is to select `Load Lib` on the `Import Library` object menu to load the DLL file into the HP VEE environment. Then select `Select Function` on the `Call Function` object menu. HP VEE will bring up a dialog box with a list of all the functions listed in the definitions file. When you select a function, HP VEE automatically configures the `Call Function` object with the correct input and output terminals and function name.

You can also configure the `Call Function` object manually by modifying the function name and adding the appropriate input and output terminals. First, configure the input terminals, with the same number of input terminals as there are parameters passed to the function. The top input terminal is the first parameter passed to the function. The next terminal down from the top is the second parameter, as so on. Next, configure the output terminals so that the parameters passed by reference appear as output terminals on the `Call Function` object. Note that parameters passed by value cannot be assigned as output terminals. The top output terminal is the value returned by the function, if it has a return value. The next terminal down is the first parameter passed by reference, etc. Finally, enter the correct DLL function name in the `Function Name` field. For example, for a DLL function defined as

```
long foo(double *x, double y, long *z);
```

you need three input terminals for `x`, `y`, and `z` and three output terminals, one for the return value and two for `x` and `z`. The `Function Name` field would contain `foo`. If the number of input and output terminals do not exactly match the number of parameters in the function HP VEE generates an error.

If the DLL library has already been loaded and you enter the function name in the `Function Name` field you can also use the `Configure Pinout` selection on the `Call Function` object menu to configure the terminals.

The DELETE LIBRARY Object

If you have very large programs you may want to delete libraries after you use them. The `Delete Library` object deletes libraries from memory just as the `Delete Lib` selection on the `Import Library` object menu does.

Using DLL Functions in Formula Objects

You can also use DLL functions in formula objects. With formula objects only the return value is used in the formula, the parameters passed by reference cannot be accessed. For example, the DLL function defined above in a formula:

```
4.5 + foo(a, b, c) * 10
```

where `a` is the top input terminal on the formula object, `b` is next and `c` is last. The call to `foo` must have the correct number of parameters or HP VEE generates an error.

Remote Functions (UNIX)

NOTE

Remote Functions are *not* supported by HP VEE for Windows.

The third type of user-defined function is the Remote Function. A Remote Function is actually a User Function that runs in another HP VEE process on a remote host computer. The remote function is called from the local HP VEE process over the LAN (Local Area Network). Just as for User Functions and Compiled Functions, you can import a library of Remote Functions with the `Import Library` object.

Once one or more Remote Functions have been imported, they can be called by either using the `Call Function` object, or by including function names in expressions. A library of Remote Functions can be deleted with the `Delete Library` object, again just as for User Functions or Compiled Functions. Thus, you can include Remote Function calls in your program just as you would User Functions. However, there are some differences, and some networking technicalities, which are described in this section.

You can create a library of Remote Functions just as you would a library of User Functions (as described earlier in this chapter). However, instead of saving the library file on your local computer, you'll need to save it on the intended remote host computer. When you import the library of Remote Functions, it is actually imported not in the local HP VEE process, but rather in a special invocation of HP VEE, called a "service", which runs on the remote host. The local HP VEE process is then called the "client."

The client HP VEE process imports the Remote Function library using the `Import Library` object. When you select `Remote Function` for the `Library Type` field, some new fields appear as shown below:

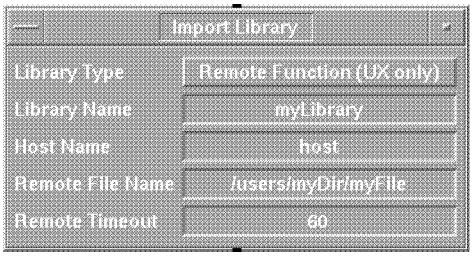


Figure 5-14. Import Library for Remote Functions

The `Library Type` and `Library Name` fields function exactly as for User Functions and Compiled Functions. However, we need to look at the other three fields:

- `Host Name` - This is the ;name of the host on which the "service" HP VEE process is to run (the "remot host"). This name can be the common or symbolic name of the host (for example `mike`). On the other hand, you can enter the IP address of the host in this field (for example `14.13.29.99`).
- `Remote File Name` - This is just the name of the Remote Function library file. The `File Name` is analogous to the `File Name` field for a User Function library. However, you must specify the *absolute* path to the file. Hence the path and file name can be rather long. You may want to have all users place remote function library files in a common place, for example: `/users/remfunc/`.
- `Remote Timeout` - This field specifies a timeout period in seconds for communication with the HP VEE service. If the HP VEE service has not returned the expected results of a Remote Function within this time period, an error occurs.

When the `Import Library` object is executed (either by selecting `Load Lib` from the object menu, or during normal program execution), a service HP VEE process is started on the remote host specified in the `Host Name` field. The client process and the service process are connected over the network, and are able to communicate. When a `Call Function` object in the client HP VEE calls a Remote Function, the arguments (the data input pins on the `Call Function` object) are sent over the network to the remote service, the remote function is executed, and the results are sent back to the `Call Function` object and output on its data output pins. If your program deletes the library

of Remote Functions with the **Delete Library** object, the service HP VEE process is terminated and the connection is broken.

The service HP VEE process can exist on the same computer or "host" as the client, or on another host as long as there is a network connection between them. The most common connection is between two hosts on a LAN. However, if a network path exists, the two hosts could be a continent apart.

NOTE

The remote HP VEE service invoked by the client is dependent on the **Host Name** and **File Name** specified in the **Import Library** object. Thus, if you have two **Import Library** objects importing the same **Host Name** and **Remote File Name**, only one service process will be invoked. If two different **Library Names** are used, each will communicate with the same service. On the other hand, if each **Import Library** specifies a different **File Name**, two separate services will be invoked.

The HP VEE service process has some attributes that are different than a normal HP VEE process:

1. The HP VEE service process will only execute User Functions that are contained in the Remote Function library named by **Import Library**. Any other objects, threads, and so forth in that file will be ignored.
2. The HP VEE service process has *no* views. This means that there is no HP VEE icon or work area appearing on the screen of the host where the HP VEE service is running. In fact, X Windows does not even need to be present on that host. What runs is just the pure functionality of the User Functions — there is no user interaction. This means that User Functions will run faster remotely than they will locally since there are no user events (keystrokes or mouse clicks) to detect.

UNIX Security, UIDs, and Names.

When you log onto an UNIX system you must enter your user name and password, and the system must have the user name and password in its `/etc/passwd` file. Also, you must have an assigned directory on the system. These requirements provide system security. There are also security requirements that must be met when one system attempts to run a process on another system. Thus, when your client HP VEE process attempts to run a service HP VEE process on a remote host, some security requirements must be satisfied.

The basic requirement is that, in order to invoke the service HP VEE process, you must have a user name on the remote host which is the same as your user name on the computer running the client HP VEE process. (However, the passwords need not be the same.) Also, you must have a directory in the `/users` directory. In addition, in order to establish network communication between the two hosts, either the remote host must have an `/etc/hosts.equiv` file with an entry for the client host, *or* the user must have an `.rhosts` file in the `$HOME` directory on the remote host, which contains an entry for the client host.

Let's look at an example. Suppose the client host can be identified as follows:

Client host: `myhost`

User: `mike`

Password: `twoheads`

and the service host can be identified as follows:

Service host: `remhost`

User: `mike`

Passwd: `arebetter`

Directory: `/users/mike`

In this case, you must have one of the following on the service host:

- An `/etc/hosts.equiv` file with the entry: `myhost`

  or

- A `/users/mike/.rhosts` file with the entry: `myhost mike`

The `/etc/hosts.equiv` file can be modified only by a super-user (usually the system administrator), while the `.rhosts` file can be modified by the user. It is a common practice to use the same `/etc/hosts.equiv` file on all computers in a particular subnet, listing all of those computers as entries. The `/etc/hosts.equiv` file is checked first for the proper entry for the client host. If no entry for the client host is found there, the `.rhosts` file is checked.

NOTE

In calling a service HP VEE process, the password is not required or called for. You must have the correct entry for the client in either the `hosts.equiv` file or the `.rhosts` file on the remote host.

Another factor in UNIX security is the user id and group id, called the UID and GID, respectively. The UID is a unique integer supplied to each user on a host by the `/etc/passwd` file. The GID is a unique integer supplied to groups of users. All UNIX processes have a UID and GID associated with them. The UID and GID determines which files or directories a user can read, write, and execute.

The HP VEE service on the service host will have the GID and UID of the user who invoked the process from the client host. This means that the file permissions are the same as if the user was running a normal interactive HP VEE session.

The .veeio and .veerc files

The .veeio and .veerc files used by the HP VEE service process are the .veeio and .veerc files of the user who invokes the process on host `remhost`. Thus, for the user `mike` in our previous example, the HP VEE service process will read the following files on host `remhost`:

```
/users/mike/.veeio
/users/mike/.veerc
```

(Only HP VEE-Test will read the .veeio file. The .veerc file is used for trig preferences only.)

Timeouts

The `Remote Timeout` field in the `Import Library` object specifies a maximum time (in seconds) to wait for the return of results from a Remote Function call. This time is also used by the `Import Library` object for the protocol used to obtain information about what functions are in the remote file loaded into the HP VEE service. If a timeout occurs, it is a *fatal error* as described in the next section. The HP VEE client will do everything possible to terminate the service. You will need to re-import the Remote Function library with a longer timeout period. (The default is 60 seconds.)

Errors

There are two classes of errors that can occur in a remote HP VEE service:

- *Fatal Errors* - These are errors, like the timeout violation

  discussed previously, that mean that the service is most likely in a unusable state. When a fatal error occurs in an HP VEE service, an error message is displayed, advising the user that the error was fatal. If this occurs, you'll need to re-import the Remote Function library. The HP VEE client will attempt to terminate the remote service.

  In most cases, a fatal error will only occur if something has gone wrong with the network, or in calling the remote service. Normally, a fatal error won't be caused by a problem in the Remote Function

  itself.

- *Non-Fatal Errors* - These are almost exclusively errors that occur within the Remote Function itself (for example a divide-by-zero error). Such errors would normally occur regardless of whether the function were local or remote. The normal error message display occurs, and gives the name of the Remote Function in which the error occurred.

N O T E

It is possible to write a Remote Function that will hang, such as an infinite loop. In this case, the Remote Function will time out with a fatal error message. The HP VEE client will attempt to remove the service, but will fail since the service will never respond. In this case, the user will have to log onto the remote host and terminate the process with `ps` and `kill`.

6

Using Transaction I/O

Using Transaction I/O

HP VEE-Engine includes objects for communicating with files, printers, named pipes, and other processes. HP VEE-Test includes all the capabilities of HP VEE-Engine, plus the ability to communicate with HP BASIC, and various hardware interfaces and the instruments connected to them.

HP VEE for Windows includes the capabilities of communicating with files, printers, other programs, and various hardware interfaces and the instruments connected to them.

All of these types of communication are controlled by I/O objects using **transactions**. This chapter explains the general concepts common to all objects using transactions and the details of how to use each type of object.

Using Transactions

All I/O objects discussed in this chapter contain **transactions**. A transaction is simply a specification for a low-level input or output operation, such as how to read or write data. Each transaction appears as a line of text listed in the open view of an I/O object. To view a typical transaction, click on `I/O` ⟹ `To` ⟹ `String` to create a `To String` object.

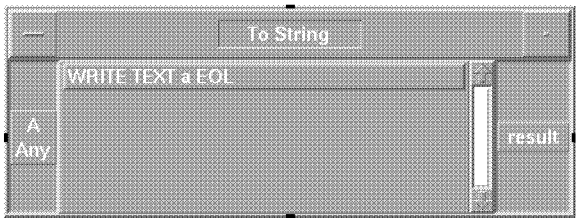


Figure 6-1. Default Transaction in `To String`

The default transaction in `To String` is:

```
WRITE TEXT a EOL
```

Before exploring too many details, consider a simple program using the `To String` object to illustrate how transactions operate. The program in Figure 6-2 uses two transactions, one to write a string literal and one to write a number in fixed decimal format.

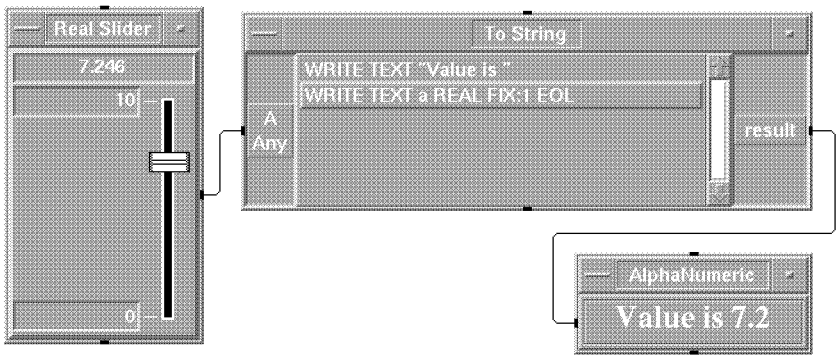


Figure 6-2. A Simple Program Using `To String`

To accomplish something useful with a transaction-based I/O object, you generally need to do at least two things:

1. Modify the default transaction or add additional transactions as required.
2. Add input terminals, output terminals, or both.

The following sections explain how to edit transactions and add terminals.

Creating and Editing Transactions

Table 6-1. Editing Transactions With A Mouse

To Do This . . .	Click On This . . .
Add another transaction to the end of the list.	`Add Trans` in the object menu. Or, double-click in the list area immediately below the last transaction.
Move the highlight bar to highlight a different transaction.	Any non-highlighted transaction.
Insert a transaction above the highlighted transaction.	`Insert Trans` in the object menu.
Cut (delete) the highlighted transaction, saving it in the transaction "cut-and-paste" buffer.	`Cut Trans` in the object menu.
Copy the highlighted transaction to the transaction "cut-and-paste" buffer.	`Copy Trans` in the object menu.
Paste the transaction currently in the buffer above the highlighted transaction.	`Paste Trans` in the object menu.
Edit the highlighted transaction.	The highlighted transaction.

Table 6-2. Editing Transactions With the Keyboard

To Do This . . .	Press This Key . . .
Move the highlight bar to highlight a different transaction.	(▲), (▼), (▶), or (Shift)-(▶)
Insert a transaction above the highlighted transaction.	(Insert line) or (CTRL)-(O).
Cut (delete) the highlighted transaction.	(Delete line) or (CTRL)-(K).
Edit the highlighted transaction.	(Select).

To edit the fields within a transaction, double-click on the transaction to expand it to an `I/O Transaction` dialog box.

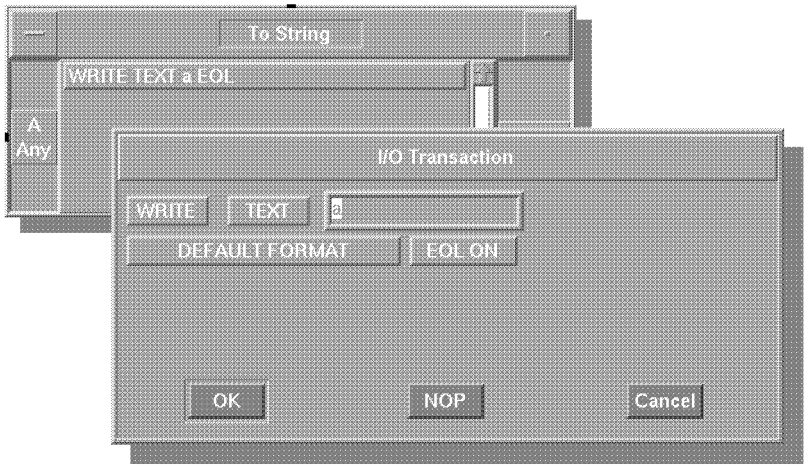


Figure 6-3. Editing the Default Transaction in `To String`

The fields shown in the `I/O Transaction` dialog box will be different for the different types of I/O operations. To edit any field, click on the field and type in information or complete the resulting dialog box. Detailed information about these fields is provided later in this chapter and in Appendix E.

Notice that the fields in the `I/O Transaction` dialog box map directly to the mnemonics that appear in the transaction listed in the open view.

The `NOP` button is unique to the `I/O Transaction` dialog box. Clicking on `NOP` saves the latest settings shown in the dialog box, but it also makes that transaction a "no operation" or a "no op." Its effect is the same as commenting out a line of code in a computer program.

Editing the Data Field

Most of the I/O specifications in a transaction are easy to edit because a dialog box helps you select the proper choice. However, the **data field** does not use a dialog box; you can type in many different combinations of variables and expressions.

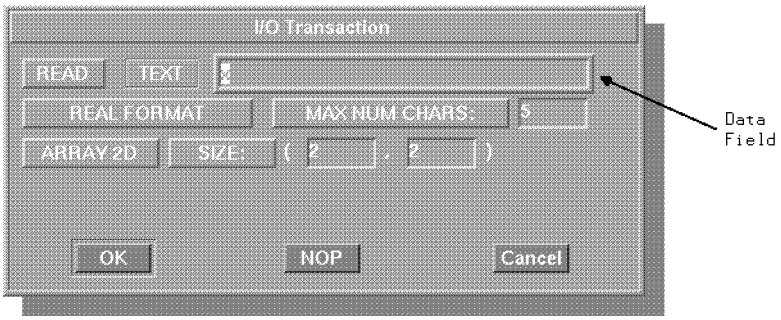


Figure 6-4. The Data Field

You must type in the proper list of what you wish to read or write. Table 6-3 lists typical entries for the data field. Note that `WRITE` transactions allow you to specify an expression list (variables, constants, and operators), but `READ` allows only a variable list.

Table 6-3. Typical Data Field Entries

Data Field Entry	Meaning
`X`	(READ) Read data into the variable `X`.
`A`	(WRITE) Write the value of the variable `A`.
`X,Y`	(READ) Read data into the variable `X` and then read data into the variable `Y`.
`A,B`	(WRITE) Write the value of the variable `A` and then write the value of the variable `B`.
`null`	(READ only) Read the specified value and throw it away. `null` is a special variable defined by HP VEE.
`A,A*1.1`	(WRITE only) Write the value of `A` and then write the value of `A` multiplied by 1.1.
`"hello\n"`	(WRITE) Write the Text literal `hello` followed by a newline character.
`"FR ",Fr," MHZ"`	(WRITE) Write a combination of Text literals and a numeric value. If the transaction is `WRITE TEXT REAL` and `Fr` has the Real value 1.234, then HP VEE writes `FR 1.234 MHZ`.

The expressions allowed in a `WRITE` data field are the same as those allowed in `Formula` objects. Note that you may include the escape characters shown in Table 6-4 in any field that accepts Text input in the form of a string delimited by double quotes.

NOTE

`READ` transactions allow a special variable named `null` in the data field. Reading data into the null variable simply throws the data away; this is useful when you need to strip away unneeded data in a controlled fashion.

Table 6-4. Escape Characters

Escape Character	ASCII Code (decimal)	Meaning
`\n`	10	Newline
`\t`	9	Horizontal Tab
`\v`	11	Vertical Tab
`\b`	8	Backspace
`\r`	13	Carriage Return
`\f`	12	Form Feed
`\"`	34	Double Quote
`\'`	39	Single Quote
`\\`	92	Backslash
`\`*ddd*		The ASCII character corresponding to the three-digit octal value *ddd*.

Adding Terminals

Most often, you will want to add input or output terminals to a transaction-based I/O object. To add terminals, click on the corresponding features in the object menu, or use the keyboard short cuts. (Use CTRL-A to add a terminal or CTRL-D to delete a terminal.)

For `WRITE` transactions, you will generally add a data input terminal. In a `WRITE` transaction, data is transferred from HP VEE to the destination associated with the object.

For `READ` transactions, you will generally add a data output terminal. In a `READ` transaction, data is transferred from the source associated with the object to HP VEE.

The variable names that appear on the terminal must match the variable names in the transaction specification to achieve useful results. This is easy to overlook, because HP VEE automatically assigns variable names such as `X`,`Y`, or `Z` when you add a terminal.

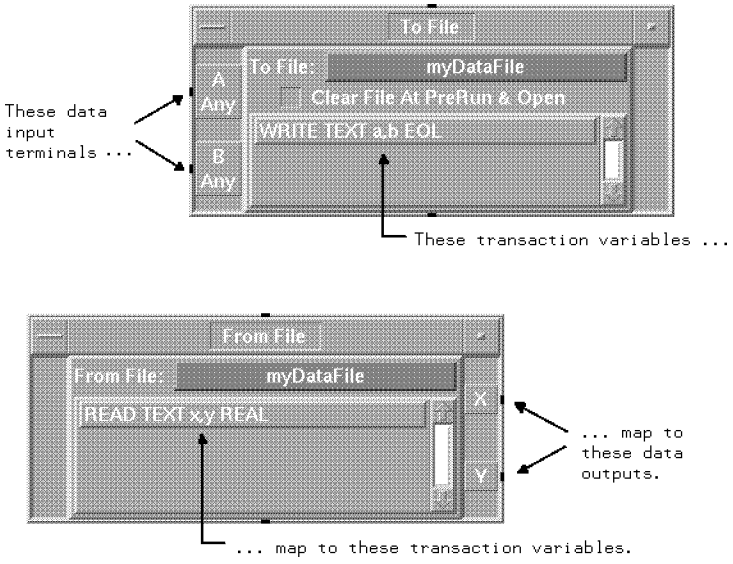


Figure 6-5. Terminals Correspond to Variables

To edit the variable name of a terminal:

1. Double click on the terminal to expand it into a `Terminal Information` dialog box.
2. Edit the `Name` field in the dialog box.

Recall that variable names in HP VEE are *not* case-sensitive. Thus, `s` is the same as `S` and `Signal` is the same as `signal`.

Reading Data

In order to read data into a variable, HP VEE must know either the number of data elements to read, or what specific terminating condition, such as EOF (end-of-file), is to be satisfied. Let's begin by looking at how to configure a transaction to read a specified number of data elements.

Transactions that Read a Specified Number of Data Elements

When you are editing a transaction, the last field in the dialog box has the default value `SCALAR`. This specifies that the `READ` transaction is to read only one element. To change this, just click on the `SCALAR` field and the following dialog box appears:

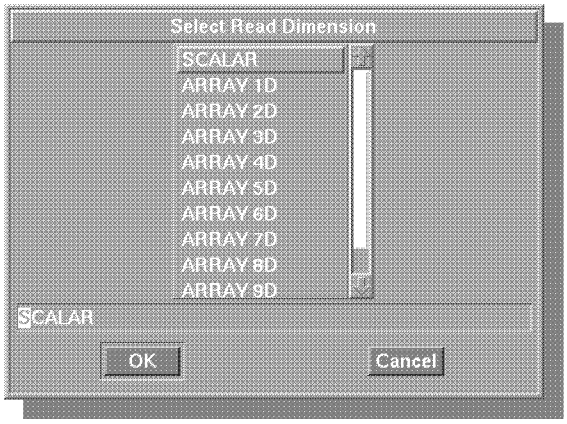


Figure 6-6. Select Read Dimension Dialog Box

The choices in the dialog box indicate the number of dimensions for the `READ` transaction. For example, `SCALAR` indicates a dimension of 0, `ARRAY 1D` indicates a one-dimensional array, `ARRAY 2D` indicates a two-dimensional array, and so forth.

When you click on the `OK` button in the `Select Read Dimension` dialog box, the transaction dialog box will reconfigure itself with a fill-in field for each of the dimensions specified. Figure 6-7 shows the transaction dialog box configured to read a three-dimensional array of binary integers into the variable named `matrix`. Each of the three fields after `SIZE:` contains the number of integers for the corresponding dimension. (In this case, each dimension has two elements.)

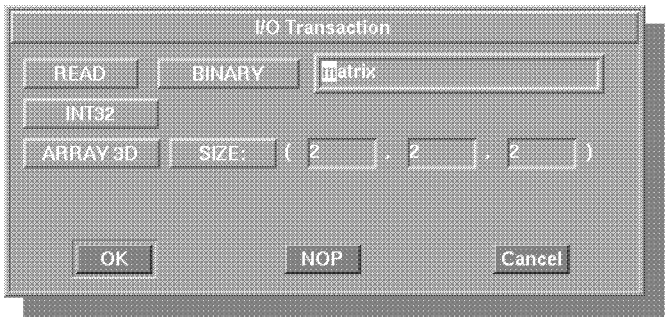


Figure 6-7. Transaction Dialog Box for Multi-Dimensional Read

Note that when more than one dimension is specified, the rightmost or "innermost" dimension is filled first. Thus, in this example, the elements are read in this order:

```
matrix[0,0,0]    read first
matrix[0,0,1]
matrix[0,1,0]
matrix[0,1,1]
matrix[1,0,0]
matrix[1,0,1]
matrix[1,1,0]
matrix[1,1,1]    read last
```

When you click on the `OK` button in the transaction dialog box, the resulting transaction appears with the `ARRAY:` keyword followed by the dimension sizes, for example:

```
READ BINARY matrix INT32 ARRAY:2,2,2
```

If the transaction is configured to read a scalar value, the transaction appears as follows:

```
READ BINARY x INT32
```

You can use variable names in the `SIZE:` fields to specify array dimensions programmatically. For example, the following transaction would read a three-dimensional matrix:

```
READ BINARY matrix INT32 ARRAY:xsize,ysize,zsize
```

In this case, `xsize`, `ysize`, and `zsize` could be either the names of input terminals, or the names of output terminals set by previous transactions in the same object.

Read-To-End Transactions

Certain HP VEE objects support `READ` transactions that will read to the end-of-file (`EOF`). Thus, it is possible to read the contents of a file with a single transaction. Such transactions are called **read-to-end** transactions. Note that `EOF`, besides indicating end-of-file for a standard disk file, can also indicate closure of a named-pipe or pipe.

The following HP VEE objects support read-to-end transactions:

- `From File`
- `From String`
- `From Stdin`
- `To/From Named Pipe` (UNIX only)
- `To/From HP BASIC/UX` (HP 300's only)
- `Execute Program (UNIX)`
- `To/From DDE (PC only)`

Figure 6-8 shows the transaction dialog box of a `From File` object, reading a three dimensional array of binary integers, but configured for read-to-end:

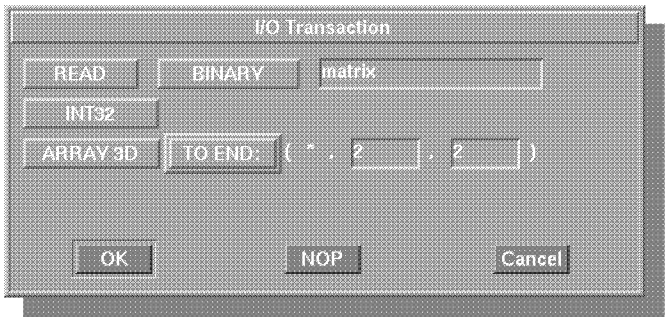


Figure 6-8. Transaction Dialog Box for Multi-Dimensional Read-To-End

Note that read-to-end transactions are not supported for scalars. The transaction must be configured for at least a one-dimensional array in order to be configured as read-to-end. If an HP VEE object supports read-to-end, the `SIZE:` field will appear as a button in the transaction dialog box. Clicking on the `SIZE:` field will enable read-to-end — the field will now appear as `TO END:`.

The trivial case of reading a one-dimensional array to end simply means that the number of elements in the array is equal to the number of elements read until `EOF` is found. The unknown size of the array is denoted by an asterisk (`*`) in the transaction.

On the other hand, reading a multi-dimensional array to end is somewhat more complicated. In this case the number of elements must be supplied for each dimension, except the left-most or "outer" dimension. Figure 6-8 shows that this dimension has an (`*`) in place of a size in the transaction. This dimension size is unknown until the read-to-end is transaction complete.

To better understand this concept, consider that a three-dimensional array is nothing more than a number of two-dimensional arrays grouped together. A two-dimensional array has the dimensions of "rows" and "columns". Stacking two-dimensional arrays, like cards, adds the third dimension, "depth". In a read-to-end transaction of a three-dimensional array, the number of "rows" and "columns" is specified, but the "depth" is unknown until EOF is encountered. The same is true for all multi-dimensional read-to-end transactions. If the array has `n` dimensions, the size of `n-1` of

those dimensions must be specified. Only one (the left-most) dimension can be of unknown size.

A further restriction on read-to-end transactions of dimensions greater than an `ARRAY 1D` is that the number of total elements read has to be evenly divisible by the product of the known dimensions. For example, let's assume that our read-to-end example of a three-dimensional array is from a file with 16 total elements. This means that the transaction will read four two-by-two arrays since the transaction specifies the number of "rows" and "columns" is equal to `2`. Hence, the unknown dimension size, "depth", is `4` when the read is complete.

If the file actually contained 18 elements, one of the two-by-two arrays would be incomplete — it would contain only two elements. A read-to-end of this file would result in an error, and no data would be read, if you specified a size of `2` for the "row" and "column" dimensions. On the other hand, you could read this file if the number of "rows" is equal to `1` and the number of "columns" is equal to `3`. A read-to-end of this file would then result in a "depth" of `6`.

NOTE

If you don't know the absolute number of data elements in a file, you can always use a read-to-end using `ARRAY 1D`.

The read-to-end transaction is useful with the `Execute Program` object for a program that is a shell command that will return an unknown number of elements.

Non-blocking Reads

A `READ` transaction finishes when the read is complete. Until the read is done, the transaction is said to **block**. When reading disk files the blocking action is not apparent since data is always available from the disk. However, for named-pipes, and for pipes where data is being made available from another process, a `READ` transaction could block, thereby effectively halting execution of an HP VEE program. In some cases, the `READ` transaction could block indefinitely.

The `READ IOSTATUS DATAREADY` transaction provides a means to *peek* at a named-pipe or pipe in order to see if there is data available for a `READ` transaction. The `READ IOSTATUS DATAREADY` transaction is available in the following HP VEE objects:

- `To/From Named Pipe` (UNIX only)
- `To/From HP BASIC/UX` (HP 300's only)
- `From StdIn`

NOTE

A `READ IOSTATUS DATAREADY` transaction, when executed, will block until the named pipe has been opened on the other end by the writing process. The transaction will then return the status of the pipe.

If the pipe has been closed by the writing process, effectively writing an EOF into the pipe, the `READ IOSTATUS DATAREADY` transaction will return a `1`, indicating that an EOF is in the pipe. A subsequent `READ` transaction will generate an EOF error. Use an error pin on the object reading the data to trap the EOF error.

Figure 6-9 shows a program where `READ IO STATUS DATA READY` is used to detect data on the StdIn pipe.

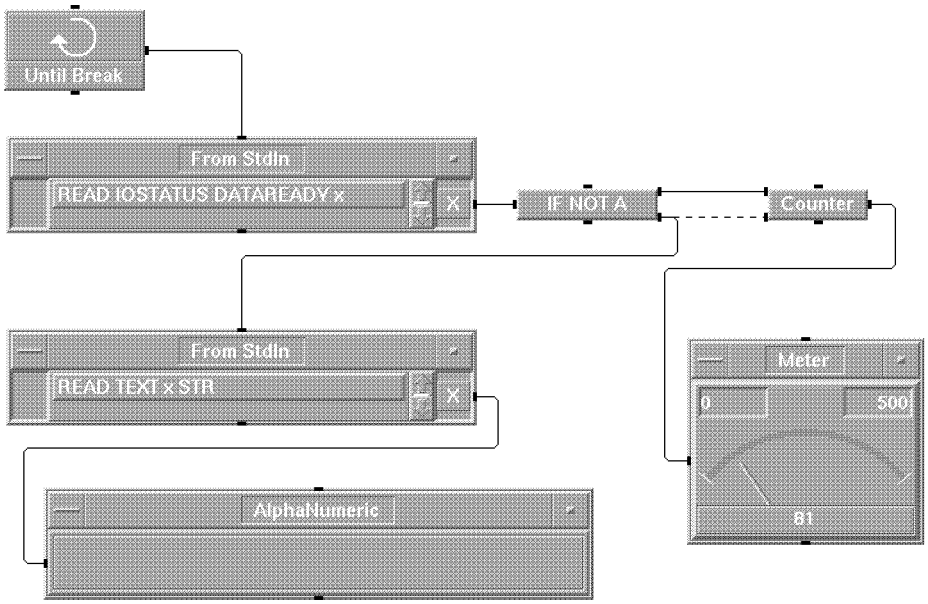


Figure 6-9. Using `READ IOSTATUS DATAREADY` for a Non-Blocking Read

This program is saved in the file `manual47.vee` in your `examples` directory.

The program in Figure 6-9 shows the use of a `READ IOSTATUS DATAREADY` transaction in `From StdIn`. The transaction returns a zero (`0`) if no data is present on the **stdin** pipe. If data is present, a one (`1`) is returned. The `If/Then/Else` is used to test the returned value of the `READ IOSTATUS DATAREADY` transaction. If the result is `1`, then the second `From StdIn` is allowed to execute, reading the data typed into the HP VEE start-up terminal window. If no data has been typed into the start-up terminal window (or a `Return` has not been typed), execution continues again at the start of the thread. Note the use of `Until Break` to iterate the thread so the `From StdIn` with the `READ IOSTATUS DATAREADY` transaction is continually tested.

To view complete programs that illustrate how to read arrays from files, open and run the programs `manual27.vee` and `manual28.vee` in the Concepts section of your `examples` directory.

Suggestions for Experimentation

Many times the best way to develop the transactions you need is by using trial and error. A large portion of the data handled by I/O transactions is text (as opposed to some type of binary data). Data written as `TEXT` is very useful for experimenting because it is human-readable. While using `TEXT` is not the most compact or fastest approach, you can use it to do just about anything.

You can use the `To String` object to accurately simulate the output behavior of other I/O objects writing text. The following program shows how you might do this.

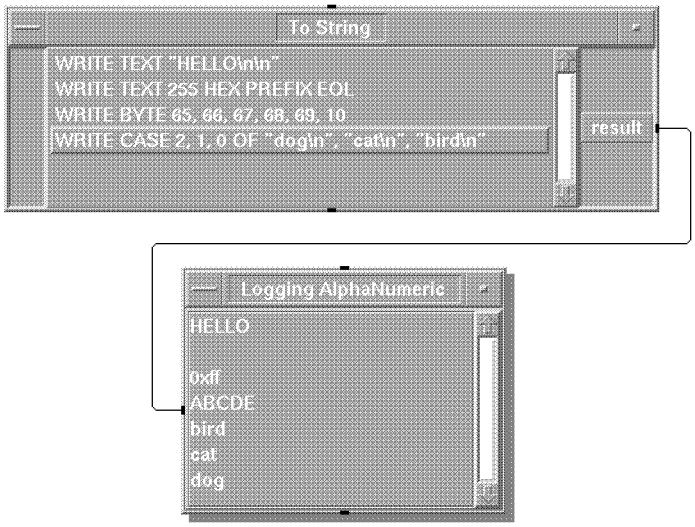


Figure 6-10. Experimenting with `To String`

Details About Transaction-Based Objects

Execution Rules

Transaction I/O objects obey all of the general propagation rules for HP VEE programs. In addition, there are a few simple rules for the transactions themselves:

1. Transactions execute beginning with the topmost transaction and proceed sequentially downward.
2. Each transaction in the list executes completely before the next one begins. Transactions within a given object do not execute in an overlapped fashion. Similarly, only one transaction object has access to a particular source or destination at a time.
3. Transaction-based I/O objects accessing the same source or destination may exist in separate threads within the same program.

   Note that for file-related objects, there is only one read pointer and one write pointer per file. The same pointers are shared by all objects accessing a particular file.

Object Configuration

In the most general case, the result of any transaction is actually determined by two things:

- The specifications in the transaction
- The settings accessed via `Config` in the object menu

In most cases you do not need to be concerned about the `Config` settings; the default values are generally suitable.

All transaction-based I/O objects that write data (except `Direct I/O`) include a `Config` feature in their object menu. The resulting dialog box allows you to view and edit various settings.

`Direct I/O` objects behave differently. `Direct I/O` objects include a `Show Config` feature in their object menu that allows you to view (but not edit) configuration settings. To edit the configuration of a `Direct I/O` object, you must use `I/O` ⟹ `Configure I/O`. Please refer to "Details of Configure I/O Dialog Boxes" in Chapter 3 for details.

Clicking on `Config` in the object menu of a transaction I/O object yields a `Format Configuration` dialog box like the one in Figure 6-11.

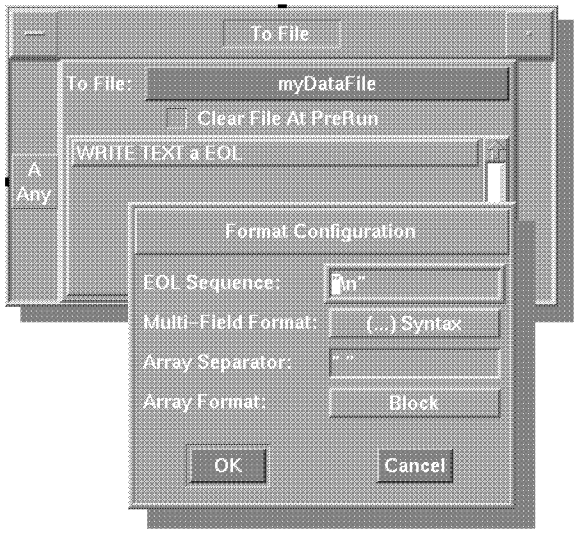


Figure 6-11. The `Format Configuration` Dialog Box

The `Format Configuration` dialog box contains settings that affect the way certain data is written by `WRITE` transactions. The `EOL Sequence` affects any `WRITE` in which `EOL ON` is set. The remaining `Format Configuration` fields affect only `WRITE TEXT` transactions.

The sections that follow explain the fields in the `Format Configuration` dialog box in detail.

EOL Sequence

The `EOL Sequence` field specifies the characters that are sent at the end of `WRITE` transactions that use `EOL ON`. The entry in this field must be zero or more characters surrounded by double quotes. "Double quote" means ASCII 34 decimal. HP VEE recognizes any ASCII characters within `EOL Sequence` including the escape characters shown previously in Table 6-4.

Multi-field As

The `Multi-field As` field specifies the formatting style for multi-field data types for `WRITE TEXT` transactions. The multi-field data types in HP VEE are Coord, Complex, PComplex, and Spectrum. Other data types and other formats are unaffected by this setting.

Specifying a multi-field format of `( ... ) Syntax` surrounds each multi-field item with parentheses. Specifying `Data Only` omits the parentheses, but retains the separating comma. For example, the complex number `2+2`*j* could be written as `(2,2)` using `( ... ) Syntax` or as `2,2` using `Data Only` syntax.

Array Separator

The `Array Separator` field specifies the character string used to separate elements of an array written by `WRITE TEXT` transactions. The entry in this field must be surrounded by double quotes. "Double quote" means ASCII 34 decimal. HP VEE recognizes any ASCII character as an `Array Separator` as well as the escape characters shown previously in Table 6-4.

`WRITE TEXT STR` transactions in `Direct I/O` objects that write arrays are a special case. In this case, the value in the `Array Separator` field is ignored and the linefeed character (ASCII 10 decimal) is used to separate the elements of an array. This behavior is consistent with the needs of most instruments.

Note that HP VEE allows arrays of multi-field data types; for example, you can create an array of Complex data. In such a case, if `Multi-Field Format` is set to `( ... ) Syntax`, the array will be written as:

`(1,1)`*array_sep*`(2,2)`*array_sep* `...`

where *array_sep* is the character specified in the `Array Separator` field.

Array Format

The `Array Format` determines the manner in which multidimensional arrays are written. For example, mathematicians write a matrix like this:

```
1  2  3
4  5  6
7  8  9
```

HP VEE writes the same matrix in one of two ways, depending on the setting of `Array Format`. In the two examples that follow, `EOL Sequence` is set to `"\n"` (newline) and `Array Separator` is set to `" "` (space).

```
1 2 3    Block Array Format
4 5 6
7 8 9

1 2 3 4 5 6 7 8 9   Linear Array Format
```

Either array format separates each element of the array with the `Array Separator` character. `Block Array Format` takes the additional step of separating each row in the array using the `EOL Sequence` character.

In the more general case (arrays greater than two dimensions), `Block Array Format` outputs an `EOL Sequence` character each time a subscript other than the right-most subscript changes.

For example, if you write the three-dimensional array `A[x,y,z]` using `Block` array format with this transaction:

```
WRITE TEXT A
```

an `EOL Sequence` will be output each time `x` or `y` changes value.

If the size of each dimension in `A` is two, the elements will be written in this order:

```
A[0,0,0]  A[0,0,1]<EOL Sequence>
A[0,1,0]  A[0,1,1]<EOL Sequence>
<EOL Sequence>
A[1,0,0]  A[1,0,1]<EOL Sequence>
A[1,1,0]  A[1,1,1]<EOL Sequence>
```

Notice that after `A[0,1,1]` is written, `x` and `y` change simultaneously and consequently two `<EOL Sequence>`s are written.

READ and WRITE Compatibility

In general, you must know how data was written in order to read it properly. This is particularly true when the data in question is in some type of binary format that cannot be examined directly to determine its format. You must read data in the same format it was written.

Choosing the Correct Transaction

This section summarizes the various I/O objects and the transactions they support. It also suggests a procedure for determining the correct object and transaction for a particular purpose. For details on transaction encodings and formats, please refer to Appendix E.

The two tables that follow summarize the transaction-based objects available in HP VEE and the actions they support. Use these tables together with the following section, "Selecting the Correct Object and Transaction", to determine the proper object and transaction for your needs.

Table 6-5. Summary of Transaction-Based Objects

Object	Description
`To File`	Writes data to a file.
`From File`	Reads data from a file.
`To String`	Writes text to an HP VEE container.
`From String`	Reads text from an HP VEE container.
`Execute Program (UNIX)`	Spawns an executable file; writes to standard input and reads from standard output of the spawned process. Note that `Execute Program (PC)` is not transaction based.
`To Printer`	Writes text to the HP VEE text printer.
`To StdOut`	Writes data to HP VEE standard output. (A file on the PC)
`To StdError`	Writes data to HP VEE standard error. (A file on the PC)
`From StdIn`	Reads data from HP VEE standard input. (A file on the PC)
`Direct I/O`	Communicates directly with HP-IB, VXI, serial, or GPIO instruments.
`Interface Operations`	Transmits low-level bus commands and data bytes on an HP-IB or VXI interface.
`To/From Named Pipe`	Transmits data to and from named pipes to support interprocess communications. (UNIX only)
`To/From HP BASIC/UX`	Transmits data to and from an HP BASIC/UX process via HP-UX named pipes. (HP 300's only)
`To/From DDE` (PC only)	Dynamically exchanges data between programs running under Microsoft Windows.

Table 6-6. Summary of Transaction Types

Action	Description
`EXECUTE`	Executes low-level commands to control the file, device, or interface associated with the transaction-based object. This action is used to adjust file pointers, clear buffers, close files and pipes, and provide low-level control of hardware interfaces.
`WAIT`	Waits for a specified period of time before executing the next transaction. In the case of `Direct I/O` to HP-IB or message-based VXI instruments, `WAIT` can also wait for a specific serial poll response.
`READ`	Reads data from the associated object.
`WRITE`	Writes data to the associated object.
`SEND`	Sends IEEE 488-defined bus messages (commands and data) to an HP-IB interface.

Selecting the Correct Object and Transaction

1. Determine the source or destination of your I/O operation and the form in which data is to be transmitted.
2. Determine the type of object that supports the source or destination using Table 6-5.
3. Determine the correct type of transaction using Table 6-6.
4. To determine the remaining specifications for the transaction, such as encodings and formats, consult Appendix E.

Example of Selecting an Object and Transaction

Assume you need to read a file containing two columns of text data. Each row contains a time stamp and a real number separated by a white space. Each line ends with a newline character. Here is a partial listing of the contents of the file.

```
14:18:00        1.001
14:18:30        -2.002
14:19:00        1.0E-03
.
.
.
```

Based on the previous procedure for selecting objects and transactions, here are the steps to solve this problem:

1. The source is a text file. The data consists of a time stamp in 24-hour hours-minutes-seconds notation and signed real numbers in scientific and decimal notation.
2. Consulting Table 6-5, note that the object used to read a file is `From File`.
3. Consulting Table 6-6, note that the type of transaction used to read data from a file is `READ`.
4. The desired transactions are:

```
READ TEXT x TIME
READ TEXT y REAL
```

Using To String and From String

Use `To String` to create formatted Text by using transactions. The Text is written to an HP VEE container.

Use `From String` to read formatted Text from an HP VEE container.

If only one string is generated by all the transactions in a `To String` object, the output container is a Text scalar. If more than one string is generated by the transactions in a `To String`, the output is a one-dimensional array of Text.

`WRITE` transactions using `EOL ON` always terminate the current output string. This causes the next transaction to begin writing to the next array element in the output container.

`WRITE` transactions ending with `EOL OFF` will not terminate the output string, causing the characters output by the next `WRITE` transaction to append to the end of the current string. The last transaction in a `To String` always terminates the current string, regardless of that transaction's `EOL` setting.

For most situations, the proper type of transaction for use with `To String` is `WRITE TEXT`. For details about encodings other than `TEXT`, please refer to Appendix E.

`From String` can read a Text scalar or an array depending on the configuration of the `READ TEXT` transaction. `READ TEXT` will either terminate a read upon encountering a `EOL` or will consume the `EOL` and continue with the read. This is dependent on the format. For details about formats, please refer to Appendix E.

Communicating with Files

Source or Destination	Object
Data Files	`To File`, `From File`
Standard Input	`From StdIn`
Standard Output	`To StdOut`
Standard Error	`To StdErr`

Details About File Pointers

HP VEE maintains one read pointer and one write pointer *per file* regardless of how many objects are accessing the file. A read pointer indicates the position of the next data item to be read. Similarly, a write pointer indicates the position where the next item should be written. The position of of these pointers can be affected by:

- A `READ`, `WRITE`, or `EXECUTE` action
- The `Clear File at PreRun & Open` setting in the open view of `To File`

All objects accessing the same file share the same read and write pointers, even if the objects are in different threads or different contexts.

A file is opened for reading and writing when either of these conditions is met:

- The first object to access a particular file operates for the first time after PreRun. This is the most common case.
- New data arrives at the optional control input terminal that specifies the filename. This case occurs less frequently.

Read Pointers

At the time `From File` opens a file, the read pointer is at the beginning of the file. Subsequent `READ` transactions advance the file pointer as required to satisfy the `READ`. You can force the read pointer to the beginning of the file at any time using an `EXECUTE REWIND` transaction in a `From File` object; data in the file is not affected by this action.

Write Pointers

The initial position of a write pointer depends on the `Clear File at PreRun & Open` setting in the open view of `To File`. If you enable `Clear File at PreRun & Open`, the file contents are erased and the write pointer is positioned at the beginning of the file when the file is opened. Otherwise, the write pointer is positioned at the end of the file and data is appended. You can force the write pointer to the beginning of the file at any time using an `EXECUTE REWIND` or `EXECUTE CLEAR` transaction. `REWIND` preserves any data already in the file. However, new data will overwrite old data starting at the new position. `CLEAR` erases data already in the file.

Closing Files

HP VEE guarantees that any data written by `To File` is written to the operating system when the last transaction completes execution and all output terminals have been activated.

The UNIX operating system physically writes data buffered by the operating system to disk periodically, typically every 15-30 seconds. This buffered operation is part of the operating system; it is not unique to HP VEE.

HP VEE automatically closes all files at PostRun. PostRun occurs when all active threads finish executing.

Files may be closed programmatically by using the `EXECUTE CLOSE` transaction in both `To File` and `From File`. This provides a means to continually read or write a file that may have been created by another process.

Files may also be deleted programmatically by using the `EXECUTE DELETE` transaction. This is useful for deleting temporary files.

Figure 6-12 shows an example of how to use `EXECUTE CLOSE`.

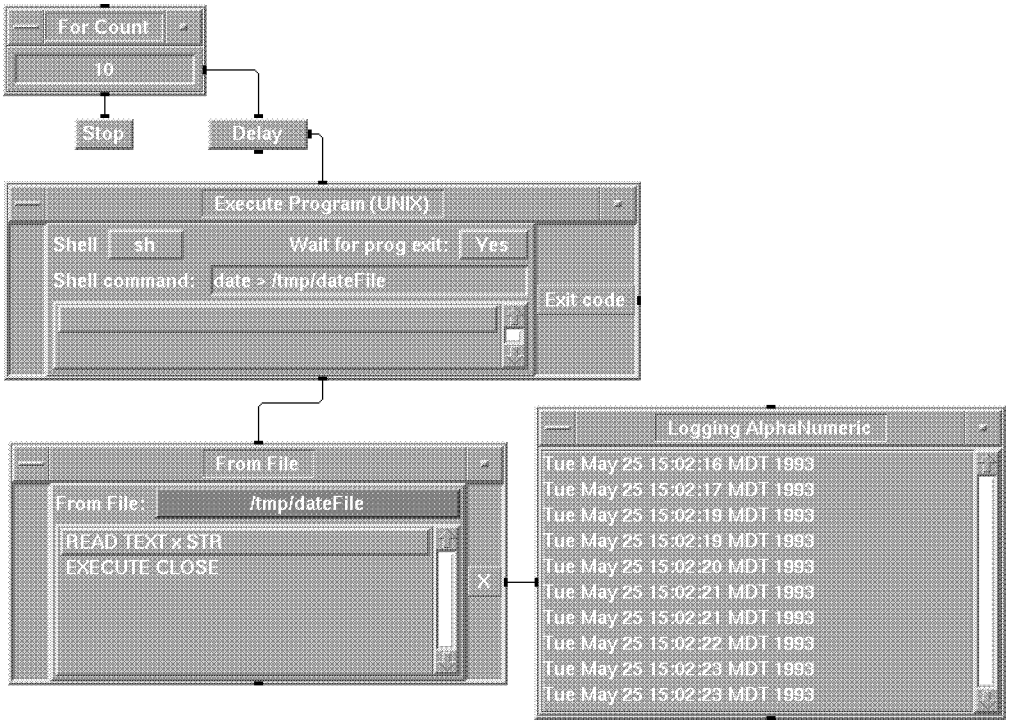


Figure 6-12. Using the `EXECUTE CLOSE` Transaction

This program is saved in the file **`manual48.vee`** in the Concepts section of your **`examples`** directory.

In Figure 6-12 **`Execute Program`** executes a shell command (**`date`**) that creates and writes the date and time to a file (**`/tmp/dateFile`**). Within the same thread, a **`From File`** reads the date from that file using a **`READ TEXT x STR`** transaction. The **`EXECUTE CLOSE`** transaction is necessary because the subthread is executed multiple times by **`For Count`**. Succeeding executions of **`Execute Program`** will overwrite the file. However, since **`From File`** only opens the file once, upon the second execution of **`From File`** the read pointer will be *stale* — it will no longer point to the file since **`Execute Program`** has *re-created* the file. An error will occur.

`From File` must close the file after reading the data by using an `EXECUTE CLOSE` transactions. The `EXECUTE CLOSE` transaction forces `From File` to re-open the file on every execution.

In the example of Figure 6-12, the error can be shown by using a `NOP` to "comment out" the `EXECUTE CLOSE` transaction. The error will state `End of file or no data found`. Removing the `NOP` will allow the program to run normally.

The EOF Data Output

`From File` supports a unique data output terminal named `EOF` (end-of-file). This terminal is activated whenever you attempt to read beyond the end of a file. The `EOF` terminal is useful when you wish to read a file of unknown length.

The read-to-end feature, discussed in "Reading Data", also provides a means of reading a file of unknown length. However, the contents of the file will be in a single HP VEE container. If the file is to be read an-element-at-a-time, with each element residing in its own container, use the `EOF` terminal.

Figure 6-13 illustrates a typical use of `EOF`. The file being read contains a list of `X-Y` data of unknown length. Here are typical contents of the file:

```
1.0
5.5
2.1
8
.
.
.
```

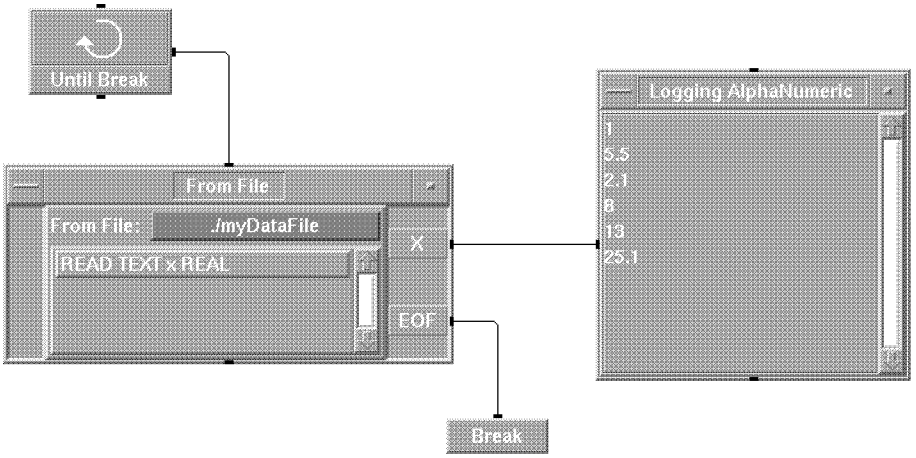


Figure 6-13. Typical Use of EOF to Read a File

Common Tasks for Importing Data

Because HP VEE provides a convenient environment for analyzing and displaying data, you may wish to import data into HP VEE from other programs. This is the general procedure to use for importing data from another software application:

1. Save the data in a text file (ASCII file).
2. Examine the data file with a text editor to determine the format of the data.
3. Use a `From File` object with a `READ TEXT` transaction to read the data file.

Importing X-Y Values

One very common problem is reading a text file containing an unknown number of `X` and `Y` values and plotting them. The program shown in Figure 6-14 solves this problem.

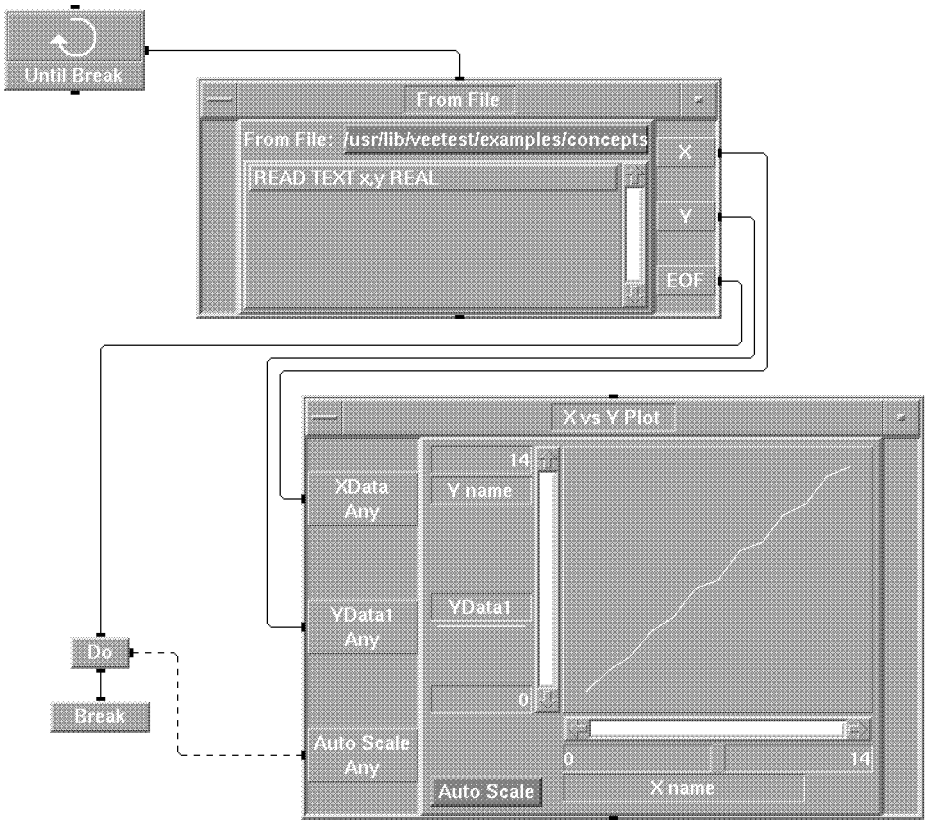


Figure 6-14. Importing XY Values

The program shown in Figure 6-14 is saved in the file `manual29.vee` in the Concepts section of your `examples` directory.

Note that the `READ TEXT REAL` transaction easily handles all the different notations used for `Y` values including signs, decimals, and exponents. Here is a portion of the data file:

```
.
.
.
8    8.555555
9    9e0
10   1.05e+01
11   +11.
12   12.5
13   1.3E1
```

Importing Waveforms

There are many different conventions used by other software applications for saving waveforms as text files. In general, the file consists of a number of individual values that describe attributes of the waveform and a one-dimensional array of `Y` values. This section illustrates how to import waveforms saved using one of these conventions:

- Fixed-format file header. Waveform attributes are listed in fixed positions at the beginning of the file followed by a one-dimensional array of `Y` data.
- Variable-format file header. A variable number of attributes are listed at the beginning of the file followed by a one-dimensional array of `Y` data. Their positions are marked by special text tokens.

Fixed-Format Header. Here is a portion of the data file read by the program in Figure 6-15:

```
NAME        Noise1
START_TIME  0.0
STOP_TIME   1.0E-03
SAMPLES     32
DATA

              .243545
             .2345776
.
.
.
```

Since this is a fixed-format header, labels such as `NAME` and `SAMPLES` are irrelevant. The waveform attributes *always appear and are in the same position*. Figure 6-15 shows a program that reads the waveform data file.

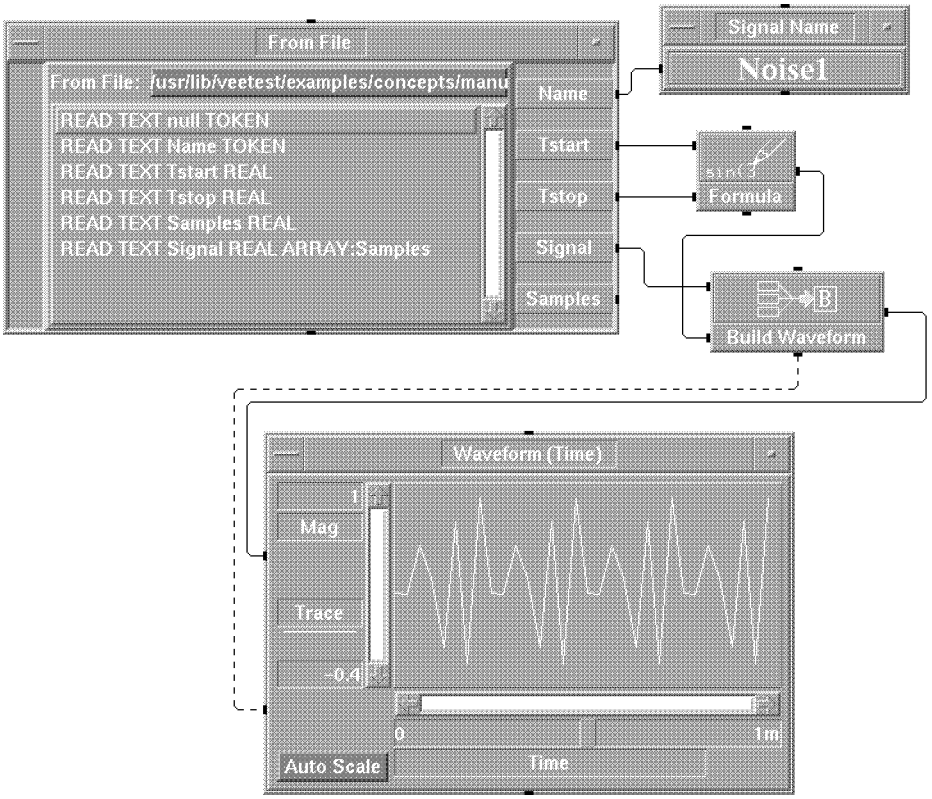


Figure 6-15. Importing a Waveform File

The program shown in Figure 6-15 is saved in the file `manual30.vee` in the Concepts section of your `examples` directory.

The transactions in **From File** do most of the work here. Here is how each transaction works:

1. The first transaction strips away the **NAME** label. This must be done before attempting to read the string that names the waveform, or else **NAME** and **Noise1** would be read together as a single string.
2. The second transaction reads the string name of the waveform.
3. The third through fifth transactions read the specified numeric quantity. Note that HP VEE simply reads and ignores any preceding "extra" characters in the file not needed to build a number.
4. The sixth transaction reads the one-dimensional array of **Y** data using the **ARRAY SIZE** determined by the previous transaction. Note that **Samples** *must* appear as an output terminal to be used in this transaction.

Variable-Format Header. Here is a portion of the data file read by the program in Figure 6-16:

```
First Line Of File
<MARKER1> 1 2 3
<MARKER2> A B C

<DATA>

1    1.1
2    2.2
3    2.9
.
.
.
```

In this case, the exact contents and position of data in the file is not known. The only fact known about this file is that a list of **XY** values follows the special text marker **<DATA>**.

To simplify this example, the program in Figure 6-16 finds only the data associated with **<DATA>**. In your own applications, you might need to search for several markers.

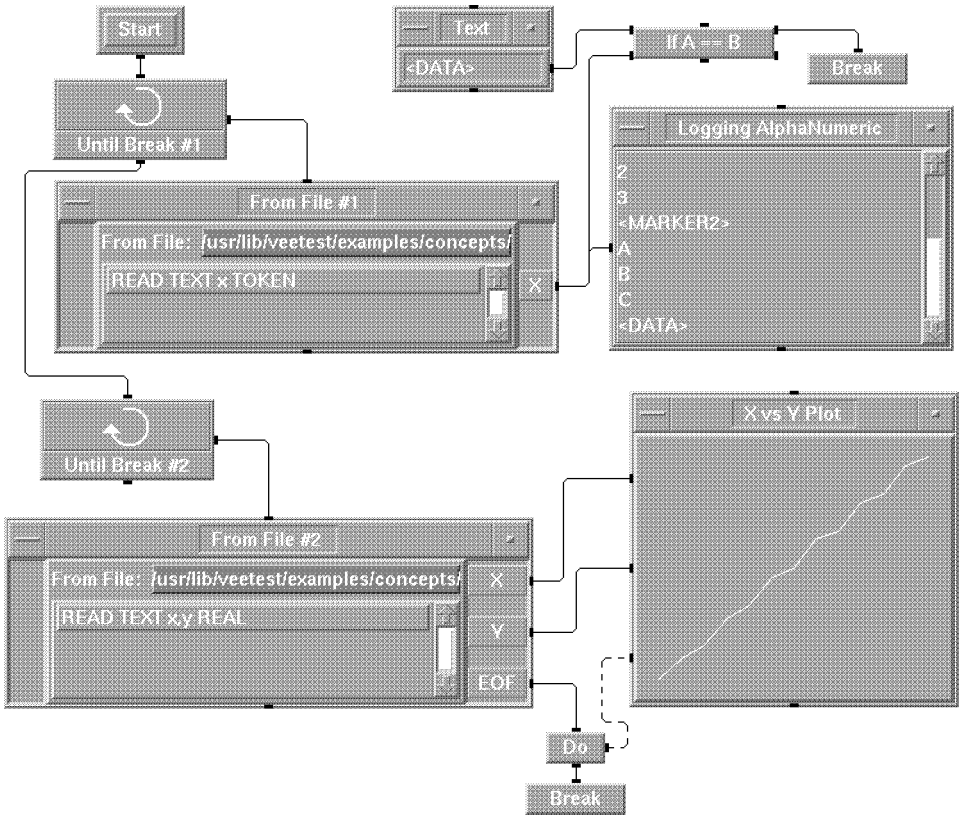


Figure 6-16. Importing a Waveform File

The program shown in Figure 6-16 is saved in the file `manual31.vee` in the Concepts section of your `examples` directory.

`From File #1` reads tokens (words delimited by white space) one at a time, searching for `<DATA>`. Once `<DATA>` is found, `From File` reads `XY` pairs until the end of the file is reached.

Communicating with Programs (UNIX)

Program	Object(s)
Shell command	`Execute Program (UNIX)`
C program	`Execute Program (UNIX)` `To/From Named Pipe (UNIX)`
HP BASIC/UX (HP 300's only)	`Init HP BASIC/UX (UNIX)` `To/From HP BASIC/UX (UNIX)`

Execute Program (UNIX)

At times you may wish to use an HP VEE program to perform a task that you would normally do from the Operating System command line. The `Execute Program (UNIX)` object allows you to do this. You use `Execute Program (UNIX)` to run any executable file including:

- Compiled C programs
- Shell scripts
- UNIX system commands, such as `ls` and `grep`

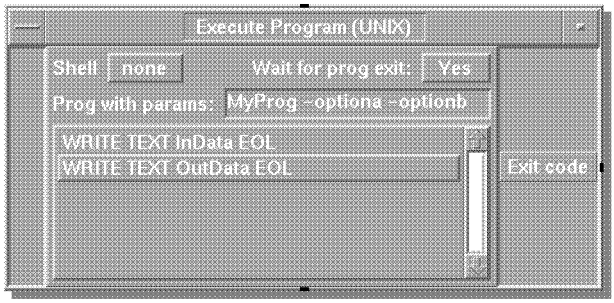


Figure 6-17. The `Execute Program (UNIX)` Object

Execute Program (UNIX) Fields

The following sections explain the fields visible in the open view of `Execute Program (UNIX)`.

Shell. `Shell` specifies the name of an UNIX shell, such as `sh`, `csh`, or `ksh`. If the `Shell` field is set to `none`, the first token in the `Pgm with params` field is assumed to be the name of an executable file, and each token thereafter is assumed to be a command-line parameter. The executable is spawned directly as a child process of HP VEE. All other things being equal, `Execute Program (UNIX)` executes fastest when `Shell` is set to `none`.

If the `Shell` field specifies a shell, HP VEE spawns a process corresponding to the specified shell. The string contained in the `Pgm with params` field is passed to the specified shell for interpretation. Generally, the shell will spawn additional processes.

Wait for Prog Exit. `Wait for prog exit` determines when HP VEE completes operation of the `Execute Program` object and activates any data outputs. If `Wait for prog exit` is set to `Yes`, HP VEE will:

1. Check to see if a child process corresponding to the `Execute Program (UNIX)` object is active. If one is not already active, HP VEE will spawn one.
2. Execute all transactions specified in the `Execute Program` object.
3. Close all pipes to the child process, thus sending an end-of-file (EOF) to the child.
4. Wait until the child process terminates before activating any output pins of the `Execute Program (UNIX)` object. If the `Shell` field is *not* set to `none`, it is the shell that must terminate to satisfy this condition.

If `Wait for prog exit` is set to `No`, HP VEE will:

1. Check to see if a child process corresponding to the `Execute Program (UNIX)` object is active. If one is not already active, HP VEE will spawn one.
2. Execute all transactions specified in the `Execute Program` object.
3. Activate any data output pins on the the `Execute Program` object. The child process remains active and the corresponding pipes still exist.

All other things being equal, `Execute Program (UNIX)` executes fastest when `Wait for prog exit` is set to `No`.

Pgm With Params. `Pgm with params` specifies either:

1. The name of an executable file and command line parameters (`Shell` set to `none`).
2. A command that will be sent to a shell for interpretation (`Shell` *not* set to `none`).

Here are examples of what you typically type into the `Pgm with params` field:

To run a shell command (`Shell` set to `ksh`):

```
ls -t *.dat | more
```

To run a compiled C program (`Shell` set to `none`):

```
MyProg -optionA -optionB
```

If you use shell-dependent features in the `Pgm with params` field, you must specify a shell to achieve the desired result. Common shell-dependent features are:

- Standard input/output redirection (`<` and `>`)
- File name expansion using wildcards (`*`, `?`, and `[a-z]`)
- Pipes (`|`)

Running a Shell Command

`Execute Program (UNIX)` can be used to run shell commands such as `ls`, `mkdir`, and `rm`. Figure 6-18 shows one method for obtaining a list of files in a directory using an HP VEE program.

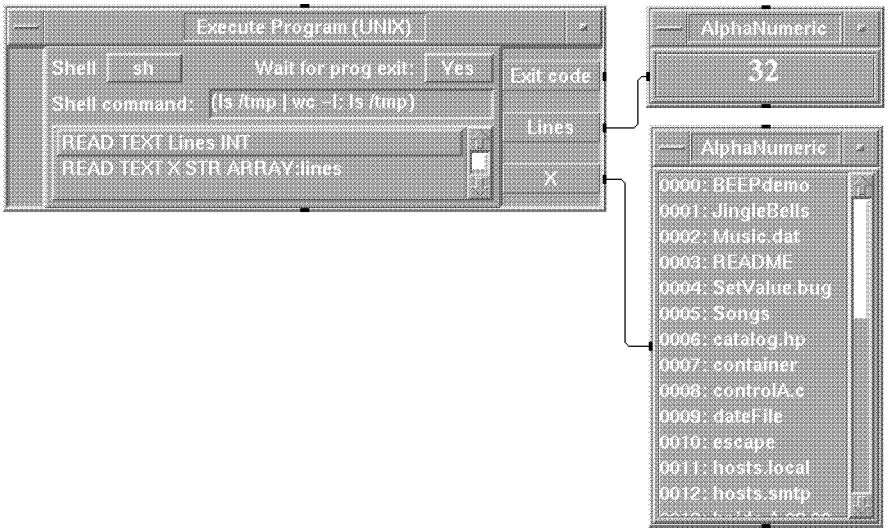


Figure 6-18. `Execute Program (UNIX)` **Running a Shell Command**

The program shown in Figure 6-18 is saved in the file `manual32.vee` in your `examples` directory.

In Figure 6-18, the `Execute Program (UNIX)` determines the number of file names in the `/tmp` directory by listing the names in a single column (`ls -1`) and piping this list to a line counting program (`wc -l`). Because the pipe is used, the command contained in the `Pgm with params` field must be sent to a shell for interpretation. Thus, the `Shell` field is set to `sh`. The number of

lines is read by the `READ TEXT` transaction and passed to the output terminal names `Lines`.

The second transaction reads the list of files in the `/tmp` directory. Note that it reads exactly the number of lines detected in the first transaction. The shell command is separated by a semicolon to tell the shell that it is executing two commands.

In the `Execute Program (UNIX)`, `Wait for prog exit` is set to `Yes`. In this case, this setting is not very important because these shell commands are only executed once. The `No` setting is useful when you want the process spawned by the `Execute Program (UNIX)` to remain active while your HP VEE program continues to execute.

Figure 6-19 shows another method for obtaining a list of files in a directory using an HP VEE program.

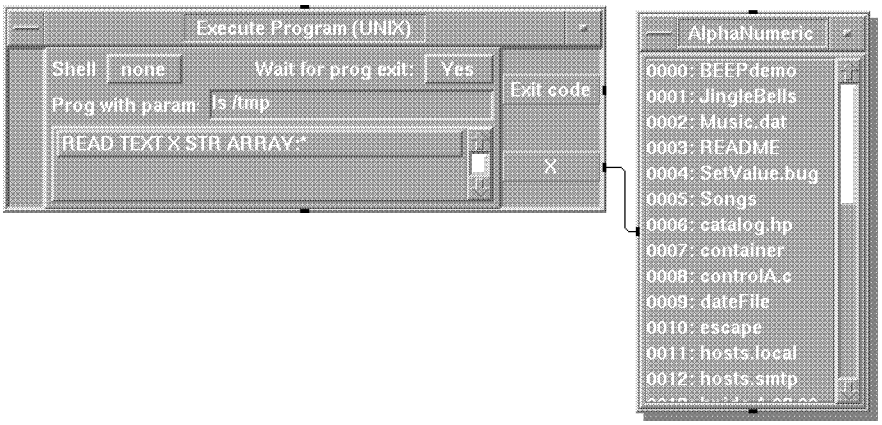


Figure 6-19. `Execute Program (UNIX)` **Running a Shell Command using Read-To-End**

This program is saved in the file `manual50.vee` in your `examples` directory.

In Figure 6-19 the HP VEE program displays the contents of the `/tmp` directory in a simpler fashion than in Figure 6-18.

In Figure 6-19, `Execute Program (UNIX)` has in the `Pgm with params` field the single shell command `ls /tmp`. There is no need to first obtain the number of files in the directory, as was done in the program in Figure 6-18, because the transaction `READ TEXT x STR ARRAY:*` uses the read-to-end

feature discussed in "Reading Data". The shell command, when it is done executing, will close the pipe that `Execute Program (UNIX)` is using to read the list of files. This sends an end-of-file (`EOF`) which terminates the the transaction.

Running a C Program

The program shown in Figure 6-20 illustrates one way to share data with a C program using `stdin` and `stdout` of the C program. In this case, the C program simply reads a real number from HP VEE, adds one to the number, and returns the incremented value.

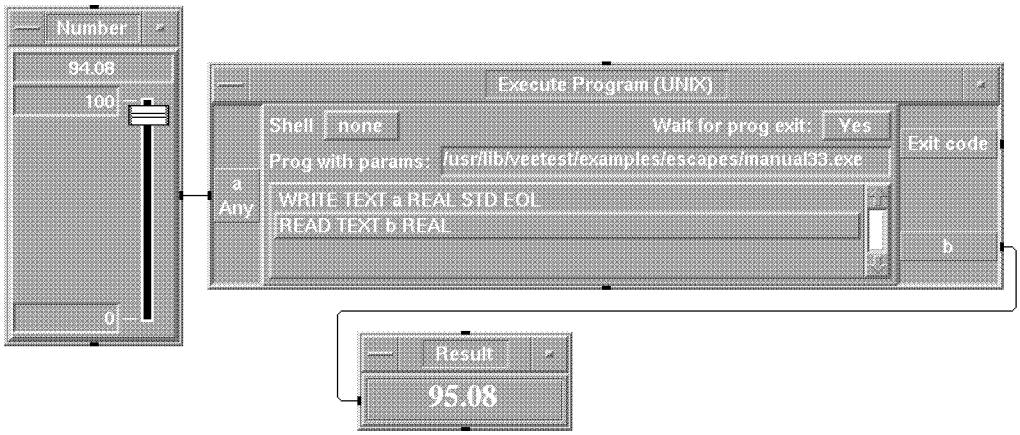


Figure 6-20. `Execute Program` **Running a C Program**

The program shown in Figure 6-20 is saved in the file `manual33.vee` in your `examples` directory.

Figure 6-21 contains a listing of the C program called by the HP VEE program in Figure 6-20.

The program listing in Figure 6-21 uses both `setbuf` and `fflush` to force data through `stdout` of the C program; in practice, either `setbuf` or `fflush` is sufficient. Using `setbuf(`*file*`,NULL)` turns off buffering for all output to *file*. Using `fflush(`*file*`)` flushes any already buffered data to *file*.

```
#include <stdio.h>
main ()
{
   int c;
   double val;
   setbuf(stdout,NULL);   /* turn stdout buffering off */

   while (((c=scanf("%lf",&val)) != EOF) && c > 0){
     fprintf(stdout,"%g\n",val+1);
     fflush(stdout);      /* force output back to VEE*/
   }
   exit(0);
}
```

Figure 6-21. C Program Listing

To/From Named Pipe (UNIX)

`To/From Named Pipe` is a tool for *advanced users* who wish to implement interprocess communication. Using named pipes in UNIX is not a task for casual users; named pipes have some complex behaviors. If you wish to learn more about named pipes and interprocess communication, refer to the section "Related Reading" at the end of this chapter.

All `To/From Named Pipe` objects contain the same default names for read and write pipes. Be certain that you correctly specify the names of the pipes you want to read or write. This can be a problem if you run HP VEE on a diskless

workstation. You must be sure that the named pipes in your program are not being accessed by another user.

HP VEE creates pipes for you as they are needed; you do not need to create them outside the HP VEE environment.

Hints for Using Named Pipes

- Be certain that HP VEE and the process on the other end of the pipe expect to share the same type of data. In particular, be certain that the amount of data sent is sufficient to satisfy the receiver and that unclaimed data is not left in the pipe.
- Use unbuffered output to send data to HP VEE or flush output buffers to force data through to HP VEE. This can be achieved by using non-buffered I/O (`write`), turning off buffering (`setbuf`), or flushing buffers explicitly (`fflush`).

Here are examples of the C function calls used to control buffered output to HP VEE:

`setbuf(`*out_pipe1*`,NULL)` *Turns off output buffering.*

or

`fflush(`*out_pipe1*`)` *Flushes data to HP VEE.*

or

`write(`*out_pipe2*`,data,n)` *Writes unbuffered data.*

where *out_pipe1* is a file pointer and *out_pipe2* is a file descriptor for the `Read Pipe` specified in `To/From Named Pipe`.

Note that HP VEE automatically performs similar flushing operations when writing data to a pipe. HP VEE does the equivalent of an `fflush` when either of these conditions is met:

- The last transaction in the object executes.
- A `WRITE` transaction is followed by a non-`WRITE` transaction.

`To/From Named Pipe` supports read-to-end transactions as described in "Reading Data". `To/From Named Pipe` also supports `EXECUTE CLOSE READ PIPE` and `EXECUTE CLOSE WRITE PIPE` transactions. These transactions can be used for inter-process communications where the amount of data to read and write between HP VEE and the other process is not explicitly known.

For example, suppose that HP VEE is using named-pipes to communicate with another process. If HP VEE is writing data out on a named pipe *and* the amount of data is less than that expected by the reading process, that reading process will hang until such time as there is enough data on the named-pipe.

By using an `EXECUTE CLOSE WRITE PIPE` transaction, the named-pipe is closed when an `EOF` (end-of-file) is sent. Thus, an `EOF` will terminate most read function calls (`read`, `fread`, `fgets`, etc . . .), thereby allowing the reading process to unblock and still obtain the data written by HP VEE into the pipe.

Conversely, if HP VEE is the reading process, a `READ` transaction using the read-to-end feature will allow HP VEE to read an unknown amount of data from the named-pipe *if* the writing process performs a `close()` on the pipe, sending an `EOF`.

HP BASIC/UX Objects (HP VEE-Test, Series 300/400 Only)

The `Init HP BASIC/UX` and `To/From HP BASIC/UX` objects are available in HP VEE only, and are supported only for HP 9000 Series 300/400 systems.

The HP BASIC/UX objects are tools for *advanced users* who wish to communicate with HP BASIC processes. Refer to the section "To/From Named Pipe (UNIX)" earlier in this chapter for general information about using pipes with HP VEE.

Init HP BASIC/UX

`Init HP BASIC/UX` spawns an HP BASIC/UX process and runs a specified HP BASIC program.

Enter the complete path and file name of the HP BASIC program you wish to execute in the `Program` field. The program may be in either STOREd or SAVEd format.

`Init HP BASIC/UX` does not provide any data path to or from the HP BASIC process; use `To/From HP BASIC/UX` for that purpose.

You can use more than one `Init HP BASIC/UX` object in a program, and you can use more than one in a single thread.

Note that there is no direct way to terminate an HP BASIC/UX process from an HP VEE program. In particular, PostRun does not attempt to terminate any HP BASIC/UX processes. PostRun occurs when all threads complete execution or when you press `Stop`. Thus, you must provide a way to terminate the HP BASIC/UX process. Possible ways to do this are:

- Your HP BASIC program executes a QUIT statement when it receives a certain data value from HP VEE.
- An `Execute Program` object kills the HP BASIC/UX process using a shell command, such as `rmbkill`.

If you `Cut` an `Init HP BASIC/UX` while the associated HP BASIC process is active, HP VEE automatically terminates the HP BASIC process. When you `Exit` HP VEE, all HP BASIC processes started by HP VEE are terminated.

To/From HP BASIC/UX

The `To/From HP BASIC/UX` object supports communications between an HP BASIC program and HP VEE using named pipes.

Type in the names of the pipes you wish to use in the `Read Pipe` and `Write Pipe` fields. Be certain that they match the names of the pipes used by your HP BASIC/UX program and that the read and write names are not inadvertently swapped relative to their use in the HP BASIC program. Use different pipes for the `To/From HP BASIC/UX` objects in different threads.

Examples Using To/From HP BASIC/UX

Sharing Scalar Data. Consider a simple case where you wish to:

1. Start HP BASIC.
2. Run a specific HP BASIC program.
3. Send a single number to HP BASIC for analysis.
4. Retrieve the analyzed data.
5. Terminate HP BASIC.

Here are typical `To/From HP BASIC/UX` settings and the corresponding HP BASIC/UX program:

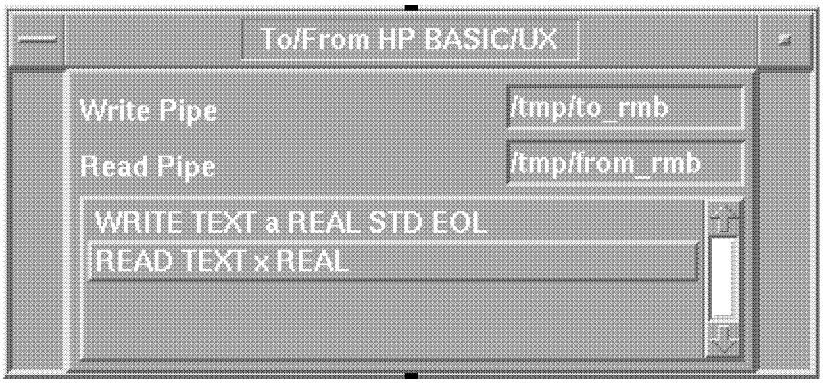


Figure 6-22. `To/From HP BASIC/UX` **Settings**

Here is the HP BASIC/UX program:

```
100  ASSIGN @From_vee TO "/tmp/to_rmb"
110  ASSIGN @To_vee TO "/tmp/from_rmb"
120  ! Your analysis code here
130  ENTER  @From_vee;Vee_data
140  OUTPUT @To_vee;Rmb_data
150  END
```

To view an example program that solves this problem, open this program:

```
/usr/lib/veetest/examples/rmb/manual34.vee
```

Sharing Array Data. To share array data between HP VEE and HP BASIC using `TEXT` encoding, you must modify the default `Array Separator` in `To/From HP BASIC/UX`. To do this, click on `Config` in the `To/From HP BASIC/UX` object menu and set the `Array Separator` field to `", "` (a comma followed by a blank).

Be sure that HP VEE and HP BASIC use the same size arrays.

Note that the order in which HP VEE and HP BASIC read and write array elements is compatible. If HP VEE and HP BASIC share an array using `READ` and `WRITE` transactions in `To/From HP BASIC/UX`, each element will have the same value in HP VEE as in HP BASIC.

To view an example program that shares arrays between HP VEE and HP BASIC, open this program:

```
/usr/lib/veetest/examples/rmb/manual35.vee
```

Sharing Binary Data. It is possible to share numeric data between HP VEE and HP BASIC without converting the numbers to text. To do this, you must select `BINARY` encoding in the `To/From HP BASIC/UX` transactions and `FORMAT OFF` for the `ASSIGN` statements that reference the named pipes in HP BASIC.

There are only two cases where it is possible to share numeric data in binary form:

- HP VEE `BINARY REAL` is equivalent to HP BASIC `REAL`
- HP VEE `BINARY INT16` is equivalent to HP BASIC `INTEGER`

Communicating with Programs (PC)

Program	Object(s)
MS-DOS command	`Execute Program (PC)`
Windows Application	`Execute Program (PC)` `To/From DDE (PC)`

C program	`Execute Program (PC)`
	`Import Library`
	`Call Function`
	`Formula`

Execute Program (PC)

At times you may wish to use an HP VEE program to perform a task that you would normally do from the Operating System command line. The `Execute Program (PC)` object allows you to do this. You use `Execute Program (PC)` to run any executable file including:

- Compiled C programs
- Any MS-DOS program (`*.EXE` or `*.COM` files)
- `.BAT` files
- MS-DOS system commands, such as `dir`

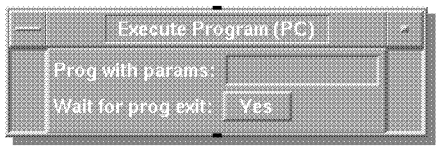


Figure 6-23. The `Execute Program (PC)` Object

Execute Program (PC) Fields

The following sections explain the fields visible in the open view of `Execute Program (PC)`.

Pgm With Params. `Pgm with params` specifies either:

1. The name of an executable file and command line parameters
2. A command that will be sent to MS-DOS for interpretation

Here are examples of what you typically type into the `Pgm with params` field:

To execute a MS-DOS command:

```
COMMAND.COM /C DIR *.DAT
```

To run a compiled C program:

```
MyProg -optionA -optionB
```

Wait for Prog Exit. `Wait for prog exit` determines when HP VEE completes operation of the `Execute Program (PC)` object and activates any data outputs. If `Wait for prog exit` is set to `Yes`, HP VEE will:

1. Execute the command specified in the `Execute Program (PC)` object.
2. Wait until the process terminates before activating any output pins of the `Execute Program (PC)` object.

If `Wait for prog exit` is set to `No`, HP VEE will:

1. Execute the command specified in the `Execute Program (PC)` object.
2. Activate any data output pins on the the `Execute Program (PC)` object.

All other things being equal, `Execute Program (PC)` executes fastest when `Wait for prog exit` is set to `No`.

Using Dynamic Data Exchange (DDE)

Dynamic Data Exchange (DDE) defines a message-based protocol for communication between Windows applications. This communication takes place between a DDE client and a DDE server. The DDE client requests the conversation with the DDE server. The client then requests data and

services from the server application. The server responds by sending data or executing procedures.

A Windows application that supports DDE may act as either a client, a server or both. HP VEE for Windows provides only provides client capabilities. It implements DDE capabilities with the `To/From DDE` object.

The HP VEE for Windows `To/From DDE` object uses four types of transactions:

`READ(REQUEST)`	Reads Data from a DDE transfer.
`WRITE(POKE)`	Writes (pokes) Data to a DDE transfer.
`EXECUTE`	Sends a command to the DDE server that HP VEE for Windows is communicating with. The server then executes the command.
`WAIT`	Waits for the specified amount of time (in seconds).

Note that the `To/From DDE` object initiates and terminates DDE operations as part of its function. You do not need to explicitly perform the initiate and terminate functions.

Key Definitions

- **Application** - The DDE name for the application.
- **Topic** - An application-specific identifier of the kind of data. For example, a word processor's topic would be the document name.
- **Item** - An application-specific identifier for each piece of data. For example, a spreadsheet data item might be a cell location; a word processor data item might be a bookmark name.

`To/From DDE` Object

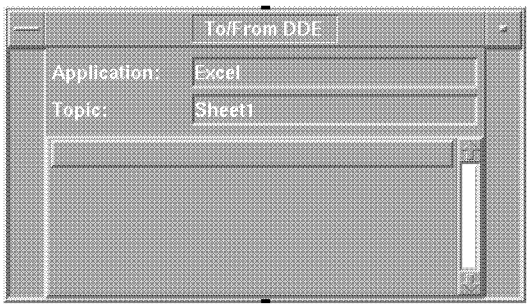


Figure 6-24. The `To/From DDE` Object

The `To/From DDE` object has three main fields. In the `Application` field enter the application name for the Windows application that you want to communicate with. Generally, this is the `.EXE` file name. See the manual for each specific application to determine its DDE application name.

The `Topic` field contains the Topic name for the application.

The last field contains transactions to communicate with the other application. For `READ(REQUEST)` and `WRITE(POKE)` transactions, you must also fill in an `Item` name in the transaction.

For example, the following `To/From` DDE object, communicating with the MS-Windows Program Manager, creates a program group, adds an item to the group, displays it for 5 seconds and then deletes the program group.

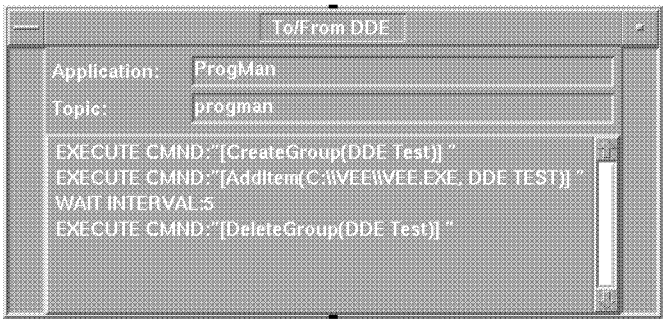


Figure 6-25. The `To/From DDE` Example

Note that if the server DDE application is not currently running, HP VEE will attempt to start that application. This will only be successful if the application's executable filename is the same as the name in the application field. The executable file's directory must also be defined in your `PATH`. Otherwise, use an `Execute Program (PC)` object before the `To/From DDE` object to run the application program, as illustrated in the following example.

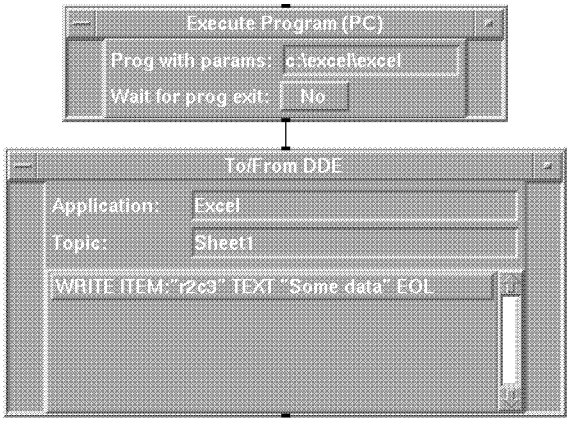


Figure 6-26. Execute PC before To/From DDE

The following example shows the use of input and output terminals with a `To/From DDE` object.

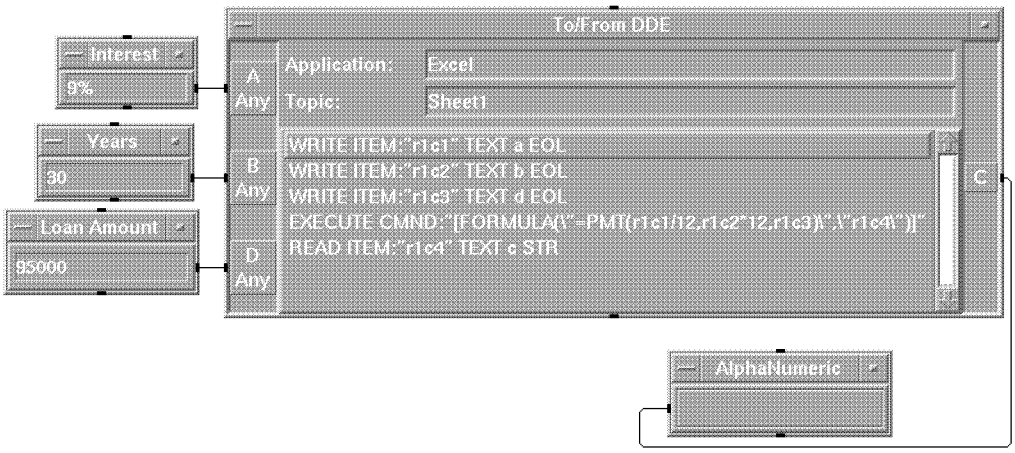


Figure 6-27. I/O Terminals and To/From DDE

DDE Examples

The following figures are examples of how to communicate with various popular Windows software. Read the notepad in each example for important information regarding each example.

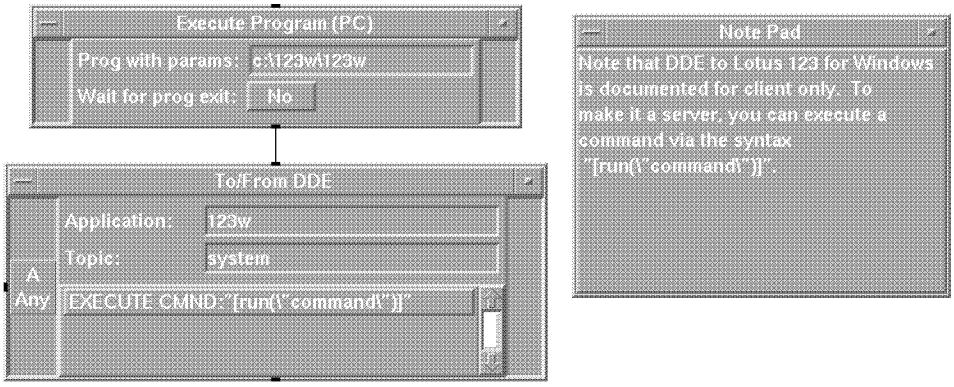


Figure 6-28. Lotus 123 DDE Example

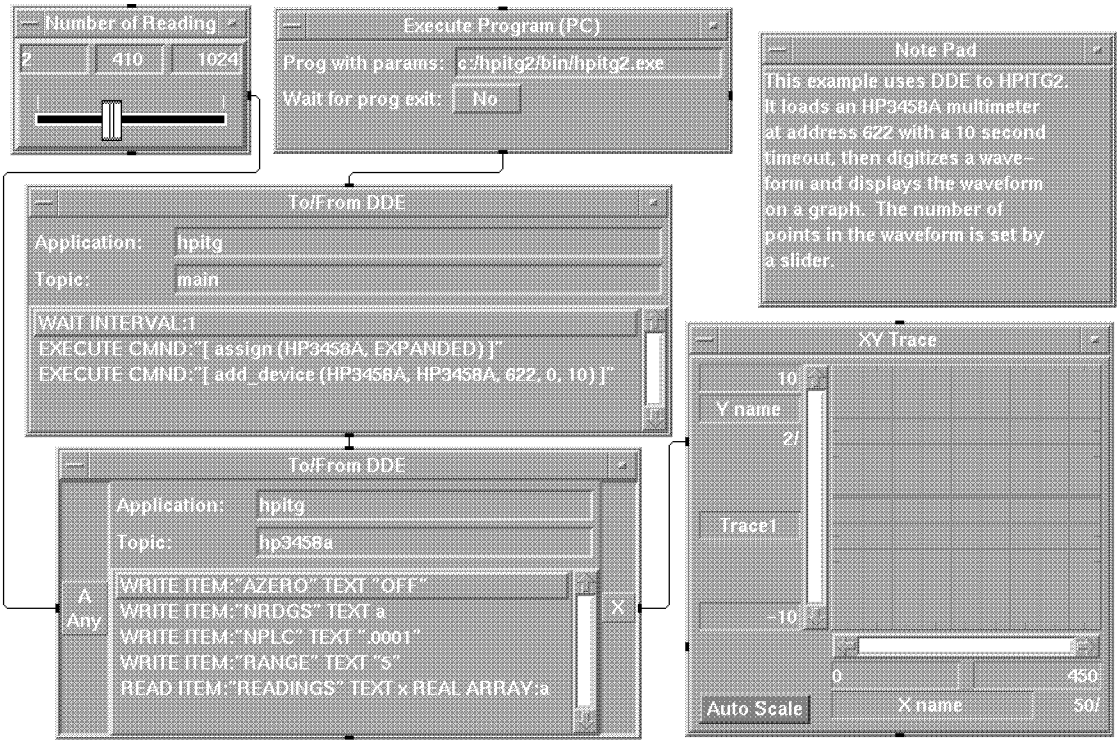


Figure 6-29. HP ITG DDE Example

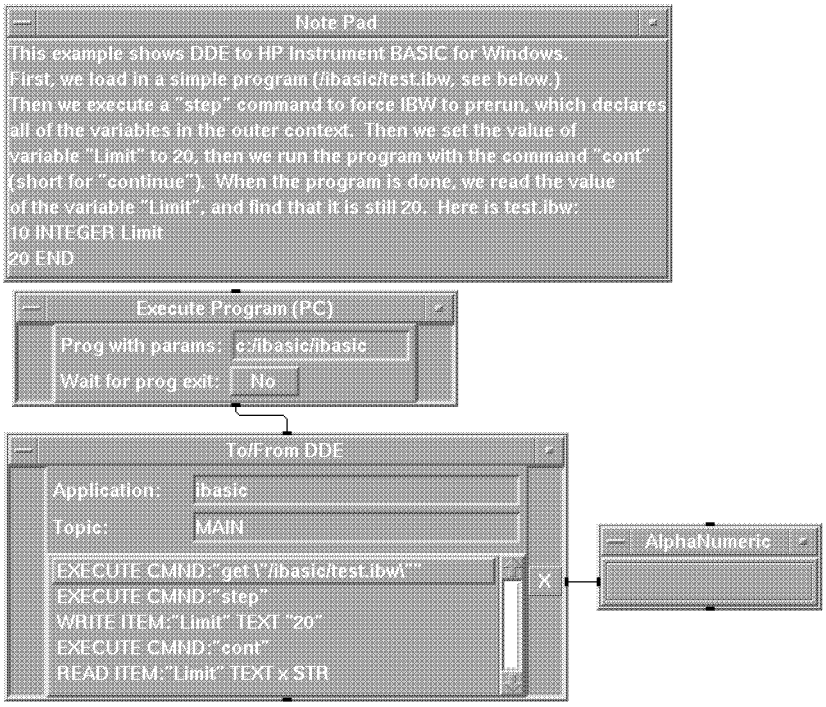


Figure 6-30. Instrument BASIC for Windows DDE Example

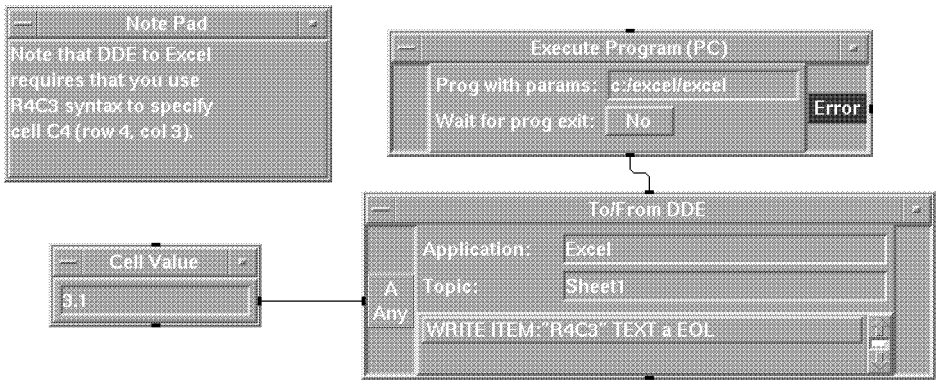


Figure 6-31. Excel DDE Example

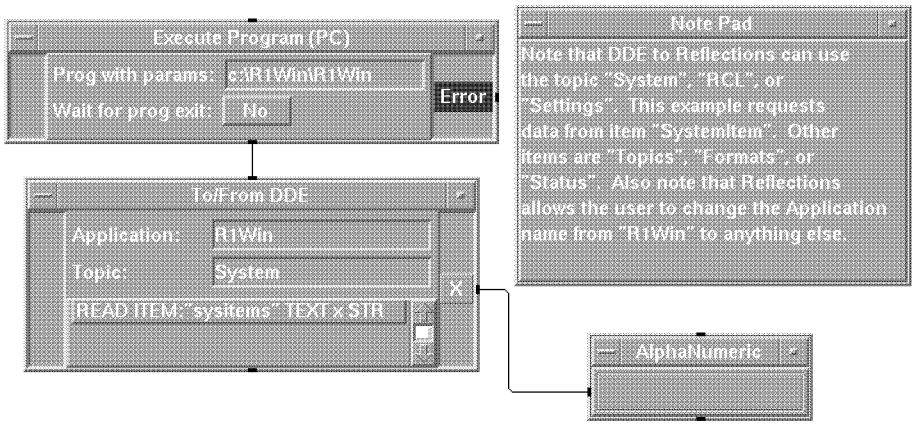


Figure 6-32. Reflections DDE Example

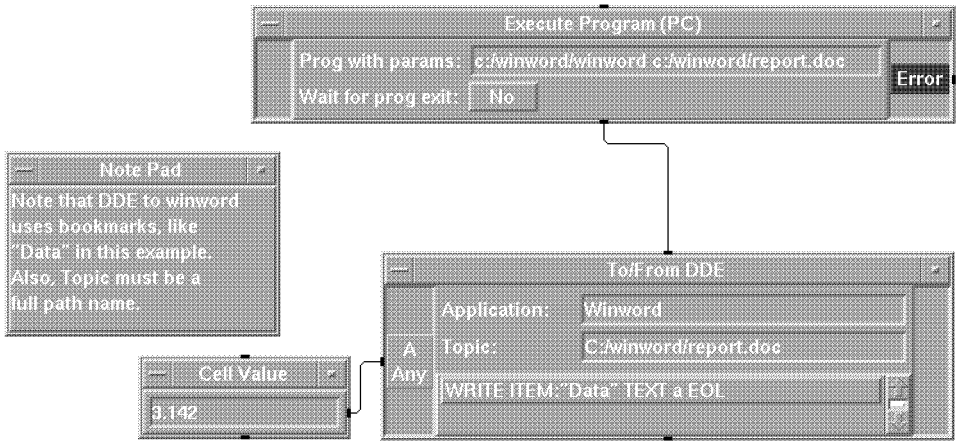


Figure 6-33. Word for Windows DDE Example

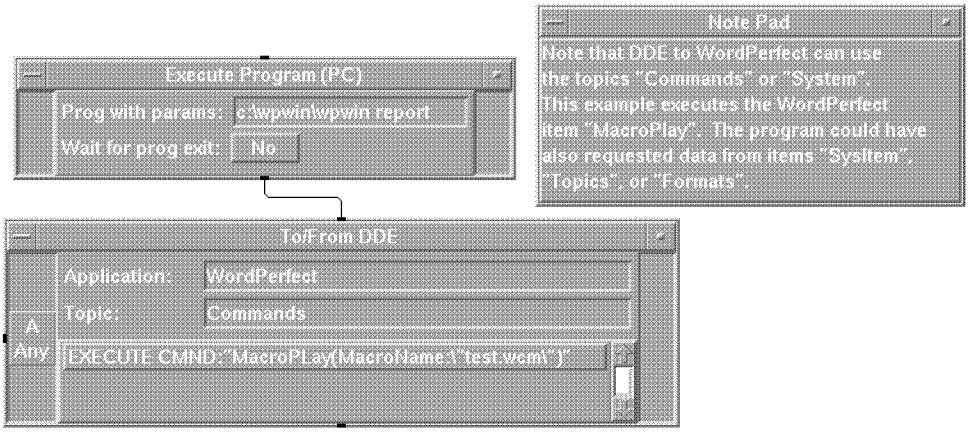


Figure 6-34. WordPerfect DDE Example

Dynamic Linked Libraries (DLL)

For information on using DLL's see "Creating a Compiled Function (MS-Windows)" in Chapter 5.

Communicating with Instruments

Task	Object
Transmit data via HP-IB, VXI, serial, and GPIO interfaces.	`Direct I/O`
Send low-level HP-IB or VXI messages, commands, and data.	`Interface Operations`

NOTE

You must properly configure HP VEE to communicate with instruments before you can use `Direct I/O` objects. Please read "Configuring Instruments" in Chapter 3 for details.

NOTE

HP VEE for Windows does not support GPIO or VXI interfaces. However you can communicate with VXI instruments over HP-IB.

Direct I/O

`Direct I/O` allows you control an instrument directly using the instrument's built-in commands. You do not need an HP VEE driver file to use `Direct I/O` to control an instrument.

Sending Commands

The most important `WRITE` transactions for `Direct I/O` use with HP-IB, message-based VXI, and serial instruments are:

- `WRITE TEXT`
- `WRITE BINBLOCK`
- `WRITE STATE`

`Direct I/O` to GPIO instruments uses only `WRITE BINARY` and `WRITE IOCONTROL`.

`Direct I/O` to register-based and some message-based VXI instruments use `WRITE REGISTER` and `WRITE MEMORY` transactions. These transactions are the *only* method of communicating with register-based VXI instruments. Refer to Appendix E for further information about these transactions.

WRITE TEXT Transactions. Most HP-IB, message-based VXI, and serial instruments use human-readable strings for programming commands. Such commands are easily sent to instruments using `WRITE TEXT` transactions. For example, all instruments conforming to IEEE 488.2 recognize `*RST` as a reset command. Here is the transaction used to reset such an instrument:

```
WRITE TEXT "*RST" EOL
```

Note that instruments often define very precise "punctuation" in their syntax. They may demand that you send specific characters after each command or at the end of a group of commands. In addition, HP-IB instruments vary in their use of the signal line End-Or-Identify (EOI). If you suspect that you are having problems in this area, examine the `END (EOI) on EOL` and `EOL Sequence` fields in the `Direct I/O Configuration` dialog box for the object in question. If you do not know how to access the `Direct I/O Configuration` dialog box, refer to "Configuring Instruments" in Chapter 3.

Please consult your instrument programming manual to determine the proper command syntax for your instrument.

`WRITE TEXT` transactions are all that is needed to set up instruments for the majority of all situations where `Direct I/O` is required. `Direct I/O` allows you to use `WRITE` encodings other than `TEXT` when it is required by the instrument. The encodings other than `TEXT` that are most often useful are `BINBLOCK` and `STATE`.

WRITE BINBLOCK Transactions. `BINBLOCK` encoding writes data to instruments using IEEE-defined block formats. These block formats are typically used to transfer large amounts of related data, such as trace data from oscilloscopes and spectrum analyzers. The block formats supported by HP VEE are discussed in greater detail in Appendix E. Instruments usually require a significant number of commands before accepting `BINBLOCK` data; consult your instrument's programming manual for details.

To use `BINBLOCK` transactions, you *must* properly configure the `Conformance` field (and possibly `Binblock`) in the instrument's `Direct I/O Configuration`. Please refer to "Direct I/O Configuration" in Chapter 3 for more detailed information.

WRITE STATE Transactions. Some HP-IB and message-based VXI instruments support a learn string capability, which allows you to upload all of the instrument settings. Later, you can recall the measurement state of the instrument by downloading the learn string using a `WRITE STATE` transaction. Learn strings are particularly useful when you wish to download measurement states but an instrument driver is unavailable.

Note that `WRITE STATE` transactions are available for HP-IB and message-based VXI instruments only.

Here is the typical procedure for using learn strings:

1. Configure the instrument to the desired measurement state; typically this is done using the instrument front panel.
2. Click on `Upload` in the object menu of a `Direct I/O` object configured for the instrument. The instrument state is now associated with this particular instance of the `Direct I/O` object.
3. Add a `WRITE STATE` transaction to the `Direct I/O` object.

When it is used, `WRITE STATE` is generally the first transaction in a `Direct I/O` object. `WRITE STATE` writes the `Upload`ed learn string to the instrument, thus setting all instrument functions simultaneously. Subsequent `WRITE` transactions can modify the instrument setup in an incremental fashion.

The behavior of `Upload` and `WRITE STATE` for HP-IB and message-based VXI instruments is affected by the `Direct I/O Configuration` settings for `Conformance` and `State (Learn String)`. If `Conformance` is `IEEE 488.2`, HP VEE will automatically handle learn strings using the IEEE 488.2

`*LRN?` definition. If `Conformance` is `IEEE 488`, `Upload String` specifies the command used to query the state, and the `Download String` specifies the command that precedes the string when it is downloaded. Note that message-based VXI instruments are `IEEE 488.2` compliant.

Clicking on `Upload` in the `Direct I/O` object menu has these results:

- The learn string is uploaded *immediately*.
- The learn string remains with that particular `Direct I/O` object as long as it exists, until the next `Upload`. *The learn string is saved with the program*.
- If you clone a `Direct I/O` object, its associated learn string is included in the clone.

Learn String Example. Assume you wish to program the HP 54100A digitizing oscilloscope using learn strings. The oscilloscope's programming manual contains these important facts:

- The oscilloscope conforms to IEEE 488; it does not conform to IEEE 488.2.
- The command used to query the oscilloscope's learn string is `SETUP?`.
- The command that must precede a learn string that is downloaded to the instrument is `SETUP` . Note that a space must come between the `P` in `SETUP` and the first character in the downloaded learn string.

You must use `I/O ⟹ Instruments ...` to specify the proper direct I/O configuration for the oscilloscope. The settings important to learn strings are shown in Figure 6-35.

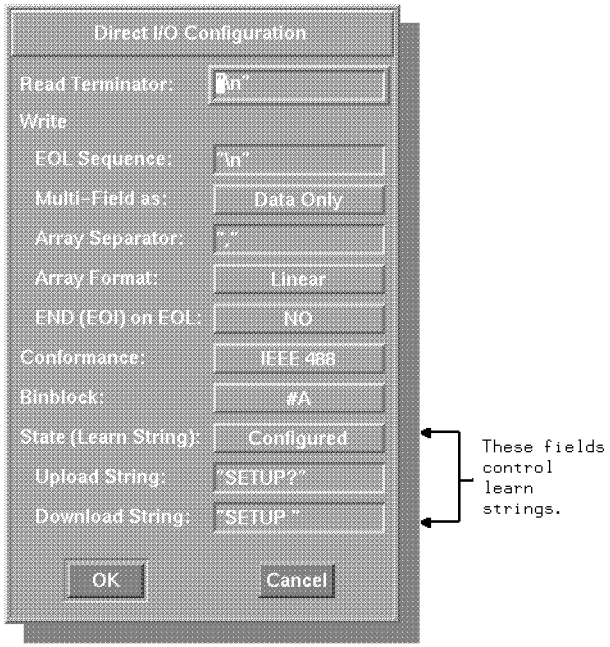


Figure 6-35. Configuring For Learn Strings

To upload a learn string from the oscilloscope, click on `Upload` in the object menu of a `Direct I/O` object that controls the oscilloscope. To download the learn string, use this transaction:

```
WRITE STATE
```

Reading Data

Instruments return data in a variety of formats. In general, you must know *what kind* of data and *how much* data you want HP VEE to read from an instrument. The kind of data determines the encoding and format you must specify in the transaction. The amount of data being read determines the configuration you must use for the `SCALAR` or `ARRAY` fields in the transaction dialog box. `Direct I/O` does not implement read-to-end reads as discussed in "Reading Data". (Note that the size configuration for arrays appears as

`ARRAY` when the settings in the `I/O Transaction` dialog box are copied to the transaction list.)

The most important `READ` transactions for `Direct I/O` use with HP-IB, message-based VXI, and serial instruments are:

- `READ TEXT`
- `READ BINBLOCK`

`Direct I/O` to GPIO instruments uses only `READ BINARY` and `READ IOSTATUS`.

`Direct I/O` to register-based and some message-based VXI instruments use `READ REGISTER` and `READ MEMORY` transactions. These transactions are the *only* method of communicating with register-based VXI instruments. Refer to Appendix E for further information about these transactions.

If you have difficulty reading data from instruments, try using the `Bus I/O Monitor` to examine that data to determine its format. You may wish to consult Appendix E to determine HP VEE's rules for interpreting incoming data.

READ TEXT Transactions. Frequently, the data you read from an instrument as the result of a query is a single numeric value that is formatted as text. For example, a particular voltmeter returns each reading as a single number in exponential notation, such as `-1.234E+00`. Here is the transaction to read a value from the voltmeter:

```
READ TEXT a REAL
```

Some instruments respond to a query with alphabetic information combined with the numeric measurement data. In general, this not a problem; `READ TEXT REAL` transactions throw away preceding alphabetic characters and extracts the numeric value.

Interface Operations

`Interface Operations` allows you to control HP-IB and VXI interfaces and instruments using low-level commands. `Interface Operations` supports two actions that provide this low-level control:

- `EXECTUTE`
- `SEND`

`EXECUTE` commands are of the form:

`EXECUTE` *Command*

where *Command* is one of the bus commands summarized in Table 6-7. While the commands listed in Table 6-7 have the same names as the `EXECUTE` commands in `Direct I/O`, there is an important difference.

- `Direct I/O EXECUTE` commands address an instrument to receive the command.
- `Interface Operations EXECUTE` commands may affect multiple instruments. For HP-IB, these instruments must be addressed to listen.

Please refer to Appendix E for details about the exact bus actions corresponding to each `EXECUTE` command.

Table 6-7.
Summary of `EXECUTE` Commands (`Interface Operations`)

Command	Description
`CLEAR`	Clears all HP-IB devices by sending DCL (Device Clear). For VXI, resets the interface and runs the Resource Manager.
`TRIGGER`	For HP-IB, triggers all devices addressed to listen by sending GET (Group Execute Trigger). For VXI, triggers TTL, ECL, or external triggers.
`LOCAL`	For HP-IB, releases the REN (Remote Enable) line. There is no counterpart for VXI.
`REMOTE`	For HP-IB, asserts the REN (Remote Enable) line. There is no counterpart for VXI.
`LOCAL LOCKOUT`	For HP-IB, sends the LLO (Local Lockout) message. Any device in remote mode at the time LLO is sent will lock out front panel operation. There is no counterpart for VXI.
`ABORT`	Clears the HP-IB interface by asserting the IFC (Interface Clear) line. To clear and reset the VXI interface use `CLEAR`.

`SEND` transactions are of this form:

```
SEND BusCmd
```

BusCmd is one of the bus commands listed in Table 6-8. These messages are defined in detail in IEEE 488.1. *BusCmd* is HP-IB specific only. There are no counterparts for VXI.

Table 6-8. SEND Bus Commands

Command	Description
COMMAND	Sets ATN true and transmits the specified data bytes. ATN true indicates that the data represents a bus command.
DATA	Sets ATN false and transmits the specified data bytes. ATN false indicates that the data represents device dependent information.
TALK	Addresses a device at the specified primary bus address (0-31) to talk.
LISTEN	Addresses a device at the specified primary bus address (0-31) to listen.
SECONDARY	Specifies a secondary bus address following a TALK or LISTEN command. Secondary addresses are typically used by cardcage instruments where the cardcage is at a primary address and each plug-in module is at a secondary address.
UNLISTEN	Forces all devices to stop listening; sends UNL.
UNTALK	Forces all devices to stop talking; sends UNT.
MY LISTEN ADDR	Addresses the computer running HP VEE to listen; sends MLA.
MY TALK ADDR	Addresses the computer running HP VEE to talk; sends MTA.
MESSAGE	Sends a multi-line bus message. Consult IEEE 488.1 for details. The multi-line messages supported by HP VEE are: DCL Device Clear SDC Selected Device Clear GET Group Execute Trigger GTL Go To Local LLO Local Lockout SPE Serial Poll Enable SPD Serial Poll Disable TCT Take Control

Related Reading

1. Haviland, Keith and Salama, Ben, *UNIX System Programming*. (Addison-Wesley Publishing Company, Menlo Park, California, 1987).

This book contains information of general interest to programmers using UNIX. In particular, it contains explanations of interprocess communications and pipes that are applicable to with `To/From Named Pipe`, `To/From HP BASIC/UX`, and `Execute Program`.

7

Using the Sequencer Object

Using the Sequencer Object

The `Sequencer` object is provided by HP VEE-Test, HP VEE for Windows and by HP VEE for Sun SparcStation.

If you are using HP VEE-Engine, please skip this chapter. HP VEE-Engine does not include the `Sequencer` object.

You'll need to understand several topics covered in this and other manuals in order to use the `Sequencer` object effectively. These topics include instrument I/O operations (Chapter 3), UserObjects (See *Using HP VEE*), Records and DataSets (Chapter 4), and User Functions (Chapter 5). Also, for information on how to use a transaction, refer to "Using Transactions" in Chapter 6.

You can use the `Sequencer` object, found under the `Device` menu, to control the order of calling of a series of tests. The `Sequencer` object executes a series of sequence transactions. Each of these transactions evaluates an HP VEE expression, which may contain calls to User Functions, Compiled Functions, Remote Functions, or other HP VEE functions. After evaluating the HP VEE expression, the transaction compares the value returned by that expression against a test specification. Depending on whether the test passes or fails, the transaction then evaluates different expressions and selects the next transaction to be executed. Transactions may optionally log their results to the `Log` output pin, or to a User Function, Compiled Function, or Remote Function specified in the `Logging Config` dialog box.

Sequence Transactions

The `Sequencer` object, in its open view, shows a list of sequence transactions. Each transaction is similar to the other types of transactions shown in Chapter 6. To see how the `Sequencer` uses transactions to execute expressions and call functions, let's look at a simple example.

In the following program there are two User Functions in the background: `myRand1`, which adds a random number from 0 to 1 to the value of its input, and `myRand2`, which adds a random number from 0 to 100 to its input. (Refer to Chapter 5 for further information on creating and using User Functions.)

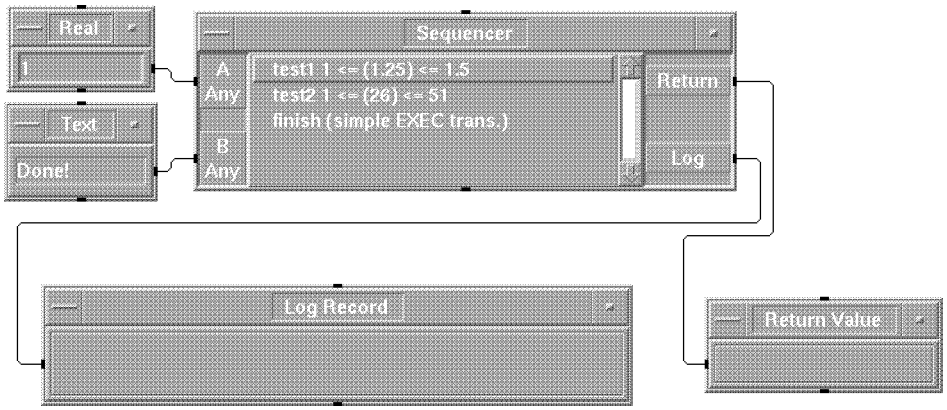
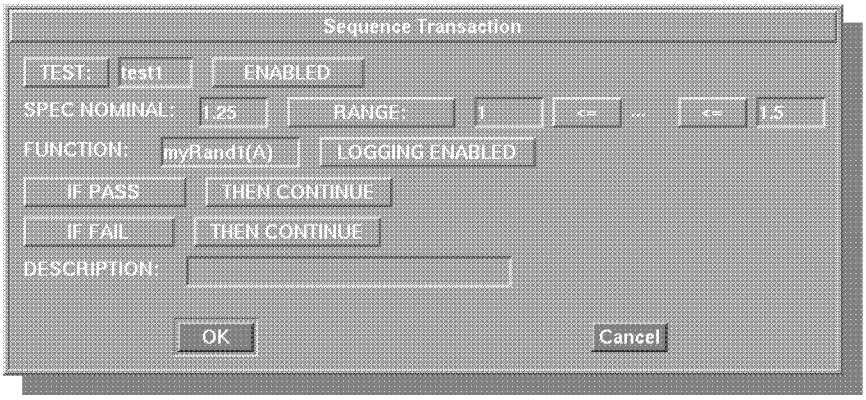


Figure 7-1. A Simple Sequencer Program

When you click on a transaction with the mouse, a dialog box "expands" the transaction so you can view and edit it. The following dialog box shows the first transaction, `test1`:



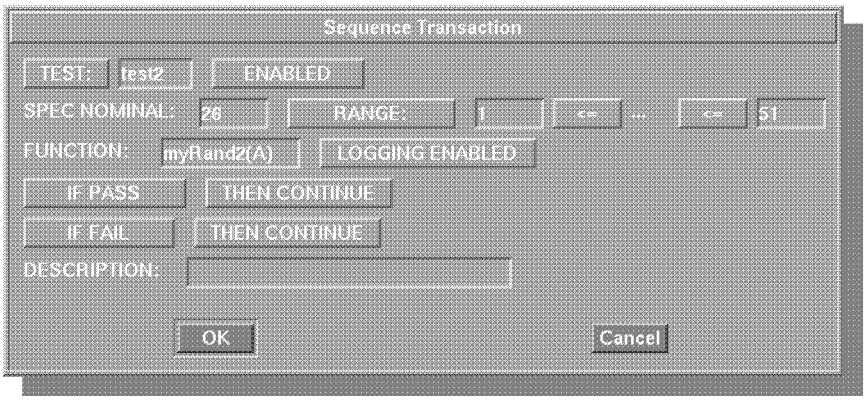
A sequence transaction can either be a TEST transaction or an EXEC transaction. In this transaction, the type is `TEST:`, the name field is `test1`, the nominal specification is `1.25`, a `RANGE:` specification is used, and the range is `1 <= ... <= 1.5`. Thus, only values from 1 to 1.5 will pass the test. The expression `myRand1(A)` calls the user function using the value on the `A` input terminal of the `Sequencer` as its input parameter. The transaction has logging enabled, so a local variable named `Test1` will be automatically created, which contains the log record of the results of this test. This log record will also be available as part of the `Log` output terminal. The `IF PASS` and `IF FAIL` conditions are both `THEN CONTINUE`. This means that, pass or fail, once `test1` is done, the next transaction, `test2`, will be executed.

The `DESCRIPTION` field is simply a comment area for this test.

NOTE

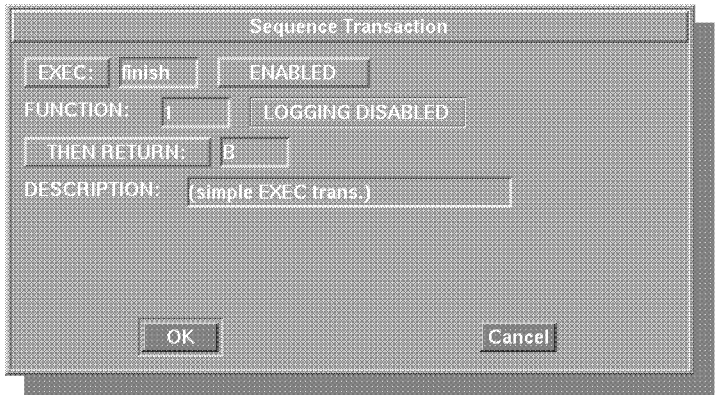
For `RANGE` or `LIMIT` tests, the `SPEC NOMINAL` value is not used, except for "documentation" purposes. However, if you use tests based on `TOLERANCE` or `%TOLERANCE` values, the tolerance will be calculated relative to the `SPEC NOMINAL` value.

The second transaction, `test2`, is also a TEST transaction:



This second test is similar to the first. The User Function `myRand2` is called with the expression `myRand2(A)` and the resulting value is tested to see if it is in the range 1 through 51, with a nominal specification of 26. Again, pass or fail, the `Sequencer` continues to the next transaction.

The third transaction is an EXEC transaction:



An EXEC transaction, unlike a TEST transaction, performs no comparison of the function result to a specification or range. EXEC transactions are used

to perform an action that does not require a pass/fail test. For example, an EXEC transaction could call a routine that sets up an external configuration before a TEST transaction is performed, or it could execute a power down procedure after a series of tests. (An EXEC transaction is a short cut for specifying an "always pass" test condition.)

In our example, the transaction named `finish` returns the value of `B` to the `Return` output terminal of the `Sequencer` object. Since no test is performed, logging does not occur for an EXEC transaction.

Note that you can use the `DESCRIPTION` field to briefly describe any transaction.

When you run the program, the three transactions are executed in sequence:

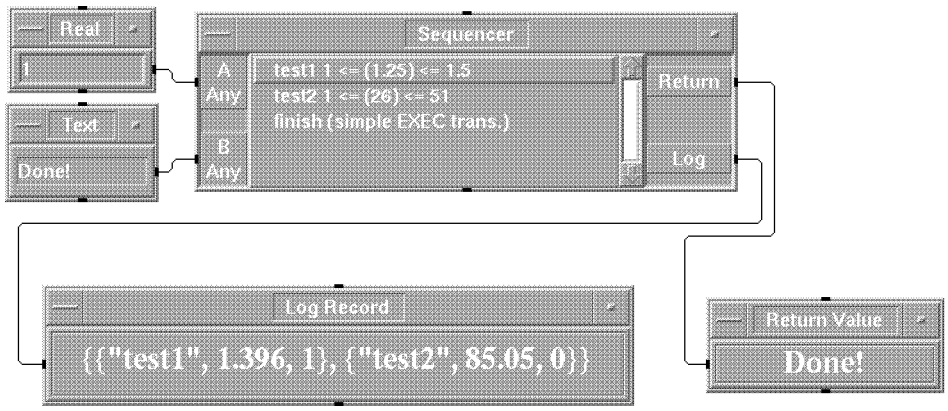


Figure 7-2. Running the Program

The logged test results are output on the `Log` output terminal and displayed. Note that the results are logged as the Record data type, in fact a record of records. In this case, `test1` has passed with a value of `1.396` and `test2` has failed with a value of `85.05`. The third transaction returns the value on the `B` input, which is the string `Done!`.

Let's look more closely at how logging works. Each transaction that has logging enabled creates a log record and attaches it to the transaction name. In our example, logging is enabled for the first two tests, so local variables named `Test1` and `Test2` contain the log records for those transactions. The fields contained in the log records are defined by the `Logging Config` device

menu selection. By default, log records contain `Name`, `Result`, and `Pass` fields.

The `Test1` and `Test2` local variable names can be used in any expression *within* the `Sequencer` to access the results of the current or a previously executed transaction. For example, `Test3` could have called a function with `Test1.Result` as a parameter to pass the result of the first test. Or `Test2.Pass` could be used as an expression, which would evaluate to `1` if `Test2` passed, or `0` if `Test2` failed.

There is one more local variable, `thisTest`, available to access the logging records. The value of `thisTest` is always the same as the logging record for the currently executing transaction. This allows you to write transaction expressions that can be used in many transactions without having to include the name of each transaction.

Now let's examine the data structure produced by the `Log` output terminal on the `Sequencer`, which is a record of records:

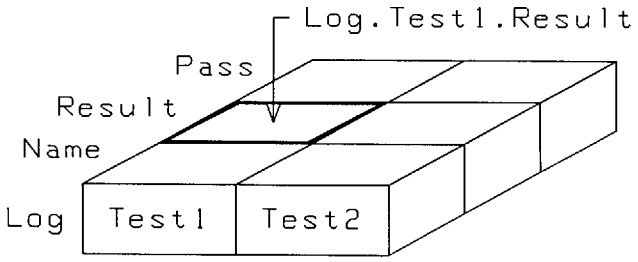


Figure 7-3. A Logged Record of Records

The record produced by the `Log` output pin contains a field for each transaction that has logging enabled — `Test1` and `Test2` in our example. Each of these fields is simply the log record for the specified transaction, containing the fields `Name`, `Result`, and `Pass`. This record of records is available on the `Log` output pin and can be used by other objects by using the record "dot" syntax. For example, the expression `Log.Test1.Result` would, in this case, return the value `1.396` (see Figure 7-2). Likewise, `Log.Test1.Name` would return `test1` and `Log.Test1.Pass` would return `1`.

Note that the data logged on the `Log` output pin is always the data from the *last* execution of each transaction. If you wish to log the results of *every* execution of each transaction, use the `Log Each Transaction To:` option in the `Logging Config` dialog box. This option will call the specified function (or expression) at the completion of every transaction. This option can also be useful if you wish to log test results to a file or printer *as they happen*, rather than waiting until the `Sequencer` has completed. The local variable `thisTest` can be used as a parameter to the logging function to pass the log record of the transaction that has just completed.

Logging Test Results

Now let's look at a more practical example of logging test results, where an iterator causes the `Sequencer` to repeat the tests over and over, and to log the results:

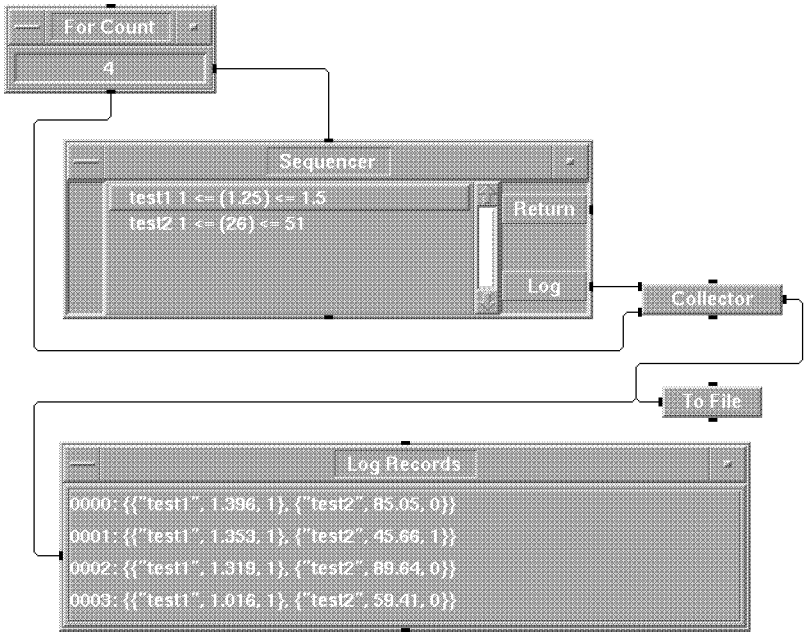


Figure 7-4. A Simple Logging Example

In this example, the `For Count` object causes the `Sequencer` to execute its series of tests (`test1` and `test2` of the previous example) four times. For example, if four "widgets" are being tested on an assembly line, each execution of the `Sequencer` tests one widget. The resulting series of records from the `Log` output terminal is collected by the `Collector` and displayed as an array of records. Note, also, that you can use the `To File` object to output this array to a file using a `WRITE CONTAINER` I/O transaction.

Conceptually, the output of the `Collector` in this example can be viewed as an array of records of records, as shown below:

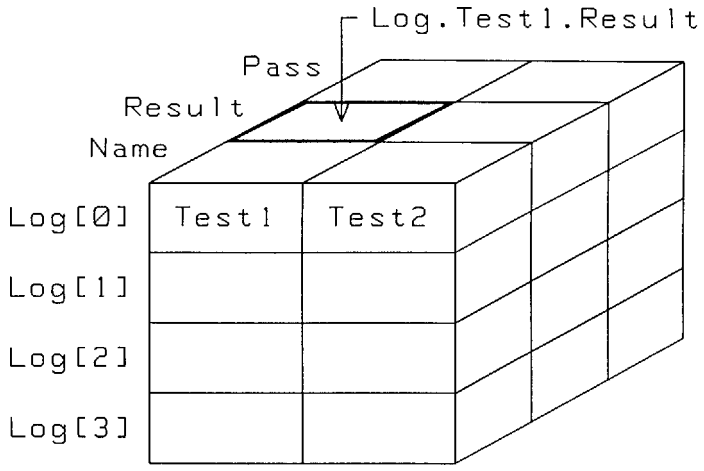


Figure 7-5. A Logged Array of Records of Records

Each array element (`Log[0]`, `Log[1]`, etc.) represents a single iteration of the sequencer, and is a record of records as shown in Figure 7-3. As mentioned before, the logged output is available for analysis in expressions. In this case, `Log.Test1.Result` is a "core sample" from the array. In fact, `Log.Test1.Result` would return an array of values (1.396, 1.353, 1.319, and 1.016 for the example results shown in Figure 7-4).

NOTE

The logged array is *not* a three-dimensional array, but is rather an array that consists of records of records. This is important because the individual fields of a record can be of differing data types. For example, while the `Name` field is Text, the `Result` field could be a Waveform, and so forth. Also, the `Test2.Result` field could be a Waveform, while the `Test1.Result` field is a Real value.

However, each individual field must be of a consistent data type throughout the array. For example, the field `Test1.Result` can't be a Real value for `Log[0]` and a Waveform for `Log[1]`.

Let's extend our example to 10 iterations of the `Sequencer`, and add some analysis of the logged data. In the following example, the expression `log.test1.result` in the `Formula` object returns a 10 element Real Array, which contains the results of `test1`. This array is then statistically analyzed by means of the `min(x)`, `max(x)`, `mean(x)`, and `sdev(x)` objects.

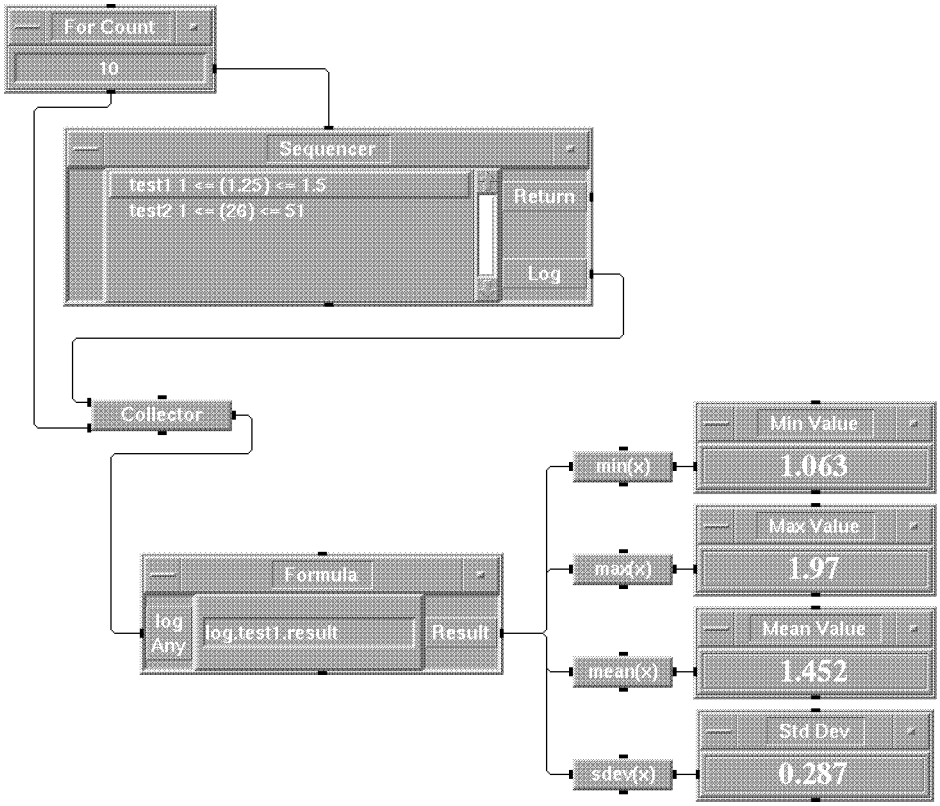


Figure 7-6. Analyzing the Logged Test Results

This example is saved in the file `manual44.vee` in the Manufacturing Test section of your `examples` directory.

Logging to a DataSet

You can use a DataSet to store your logged test results. In the following program, the `Sequencer` object `Log` output terminal is connected to the `To DataSet` object.

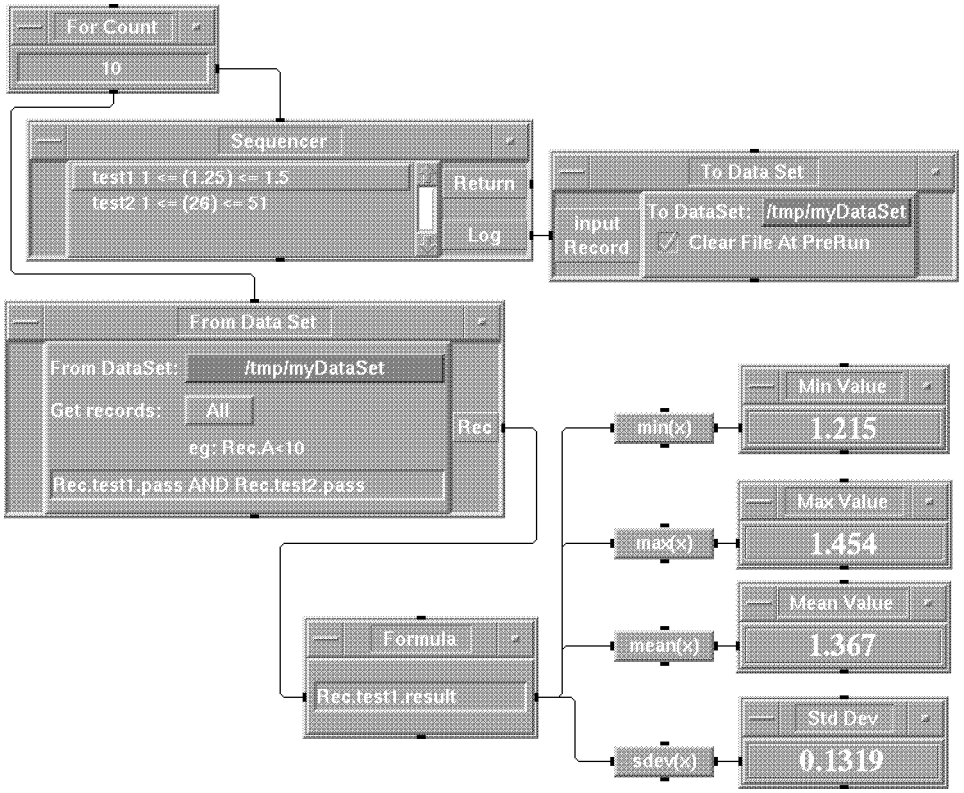


Figure 7-7. Logging to a DataSet

Once the `For Count` object is finished, it causes the `From DataSet` object to retrieve the stored DataSet (`myDataSet`). `From DataSet` is configured to retrieve `ALL` records from `myDataSet`, but to test each record against the condition `Rec.test1.pass AND Rec.test2.pass`. In other words, a

particular record is retrieved only if *both* `test1` and `test2` passed for that record.

Of the retrieved records, if any, the expression `Rec.test1.result` returns all of the `test1.result` record fields, which are then statistically analyzed. (Note that this program will error if none of the records satisfy the expression `Rec.test1.pass AND Rec.test2.pass`.)

This example is saved in the file `manual45.vee` in the Manufacturing Test section of your `examples` directory.

Some Restrictions in Logging Test Results

There are some situations where you must be careful in collecting `Sequencer` log records into an array of records. As explained in Chapter 4, to build an array of records, all of the array elements of a given field must be of the same type, shape, and size. For a record of records, as is generated by the `Log` output terminal of the `Sequencer`, the type, shape, and size of each field must match for sub-records as well.

For example, suppose you are collecting the logged results of several executions of a `Sequencer`, either by using the `Collector` to build an array (see Figure 7-6) or by sending the results to a DataSet (see Figure 7-7). In either case, if any of the logged values of a given transaction were to change type, shape, or size between executions of the `Sequencer`, an error will occur. The error will be generated by the `Collector` or `To DataSet` object because the array of records cannot be built.

This situation could easily occur if a transaction is not executed on every execution of the `Sequencer`; for example, if an `ENABLED IF` condition is specified. If the transaction is not executed, a log record will still be generated, but the `NAME` and `DESCRIPTION` fields will be empty strings and all the other fields will contain a Real scalar value of zero. If the same transaction, on a subsequent execution of the `Sequencer`, is executed and logs a result that is not a Real scalar, an error will occur. You might want to consider, in this situation, just writing each logged record out to a file in container format with `To File`, instead of using `To DataSet`.

An error could also occur if your tests return arrays of different sizes; for example, if the test returns an array of the failed data points. In this case, you might want to design the test so that it pads the array so as to always return the same size array.

A Practical Test Example

So far, we've just looked at how the `Sequencer` works, and how you might store, retrieve, and analyze the logged data. But normally, you'll want to use the `Sequencer` to control a series of "real world" tests. So let's look at a simple practical example.

In the old days, carbon resistors were manufactured by a rather imprecise process, and then tested, sorted, and marked. The trick was that the standard resistance values (for example, 220, 270, and 330 ohms) were chosen to overlap at the 10 percent tolerance. Thus, you didn't have to throw any resistors away. If a resistor was more than 10 percent greater than 220 ohms, it could be labeled as a 270 ohm resistor, and so forth.

So our problem is to construct a program in which the `Sequencer` calls a User Function, which returns a resistance value. The `Sequencer` will then run a series of tests to determine which nominal resistance value and percent tolerance the resistor satisfies. This is a "bin sort" problem. That is, the sequencer returns a result that identifies the bin in which to put the resistor.

One of the big advantages of using the `Sequencer` to call a User Function is that different User Functions can be substituted. For our problem, we'll just use a User Function (`simResist`) that returns a random resistance value in the expected range during development. You can easily substitute another User Function that executes instrument I/O and returns real resistance values once you've tested your solution.

The simplest solution to our problem is to use an extended series of sequence transactions, each testing the resistance value against a nominal value and tolerance.

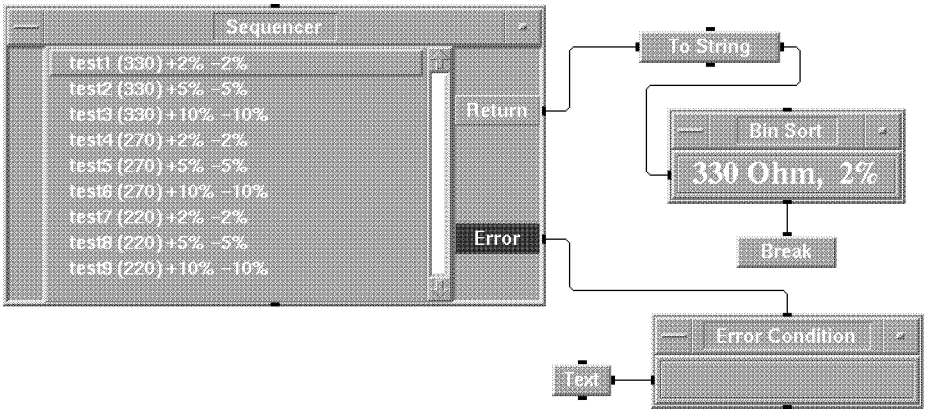
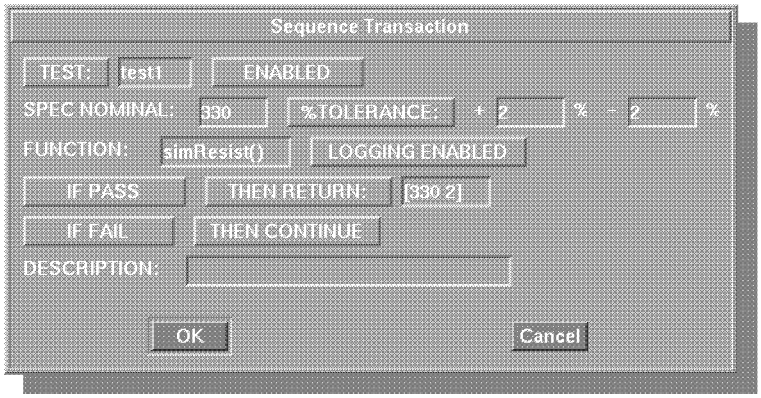


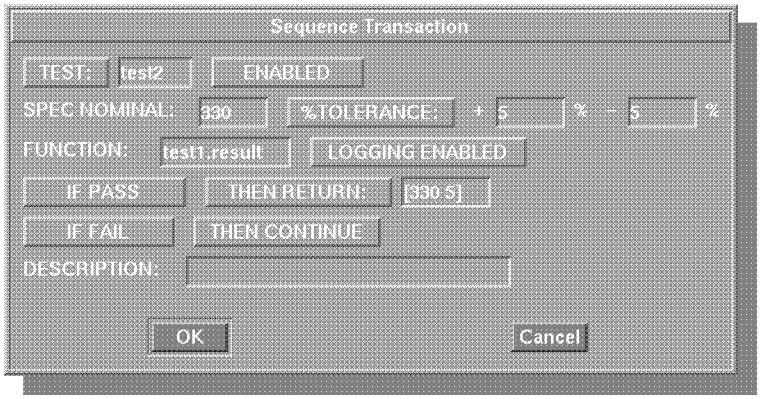
Figure 7-8. Simple Bin Sort Example

In this example, the first sequence transaction (`test1`) calls the User Function `simResist` with the expression `simResist()`. (This User Function requires no inputs.)



Note that `test1` tests to see if the resistance value returned by `simResist` is within ±2 percent of the nominal value 330. If it is, the two-element Real array `[330 2]` is returned on the `Return` output terminal, and the `To String` object converts this value to the string `330 Ohm, 2%`. If the test fails, the `Sequencer` goes on to the next test.

The second transaction, `test2`, works just like the first except that instead of calling `simResist` again, the `FUNCTION` field contains the expression `test1.result`:



Key Idea

Any transaction with logging enabled creates a "local" Record variable with the same name as the test. This record contains the fields specified for the logging record. Thus, for the transaction `test1`, the expression `test1.result` returns the value returned by the function called in `test1`.

There are two reasons for using the expression `test1.result` in our example. First, by using `test1.result` in transactions `test2` through `test9` we can ensure that each transaction uses the same function result, even if we later change `test1` to call a different function. More importantly

in this example, each time you call the User Function, a *new* resistance value will be returned. Instead, we want to continue testing the original resistance value against successive nominal values and tolerances. So the transactions `test2` through `test9` all include the expression `test1.result` in the `FUNCTION` field. These transactions work like the first, returning the appropriate array (`[330 5]`, `[330 10]`, `[270 2]`, and so forth) if passed.

The first eight tests simply continue to the next test if failed. However, an indication is needed if *all* of the tests are failed. Thus, `test9` is configured `IF FAIL THEN ERROR`. The `Error` output terminal causes the `AlphaNumeric` display entitled `Error Condition` to execute, displaying the text `Out of Range`.

Although this approach is simple, it is not very efficient. You would have to create quite a large number of sequence transactions to test several resistance values, with three tolerances in each case. Let's look at an improved version of our "bin sort" example.

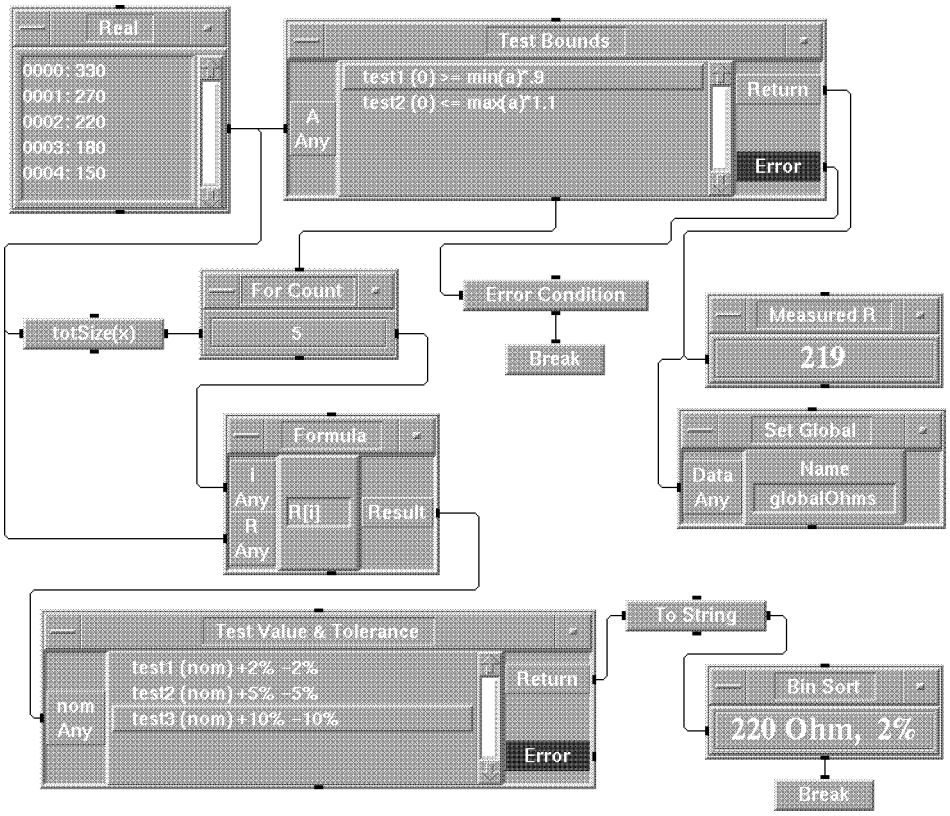
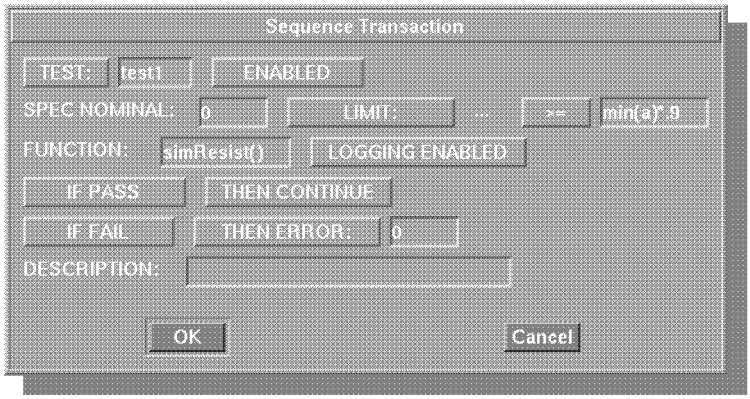


Figure 7-9. Improved Bin Sort Example

This example is saved in the file `manual46.vee` in the Manufacturing Test section of your `examples` directory.

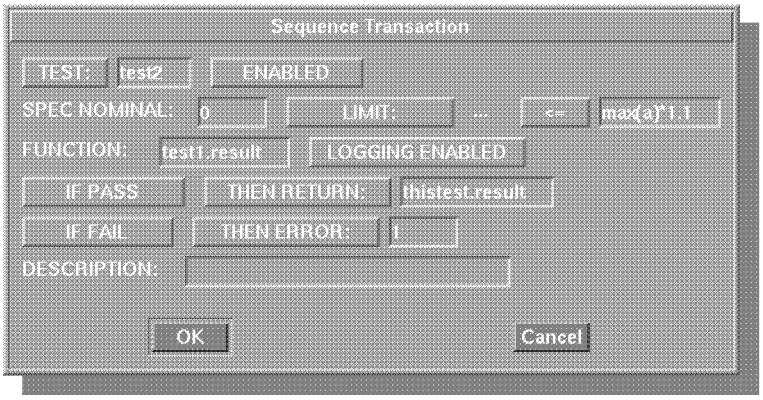
You may want to load this program and explore how it works. Here are some key points:

- This program uses two `Sequencer` objects. The first one (labeled `Test Bounds`) "re-uses" the tests in the second one (labeled `Test Value & Tolerance`).
- The `Real` array in the upper left corner of the program contains five elements, each representing a standard resistance value. However, the list of values is *extensible* in this example. Regardless of the number of array elements, the `TotSize(x)` function returns that number so that the `For Count` object will iterate the correct number of times. The expression `R[i]` in the `Formula` object takes care of the indexing.
- In the `Sequencer` named `Test Bounds`, the first transaction (`test1`) calls the User Function `simResist` with the expression `simResist()`:



A simulated resistance test value is returned and tested to see if it is at least 90 percent of the lowest value (150 Ohms) in the array. (Note that any value field in a sequence transaction can contain an expression such as `min(a)*.9`.)

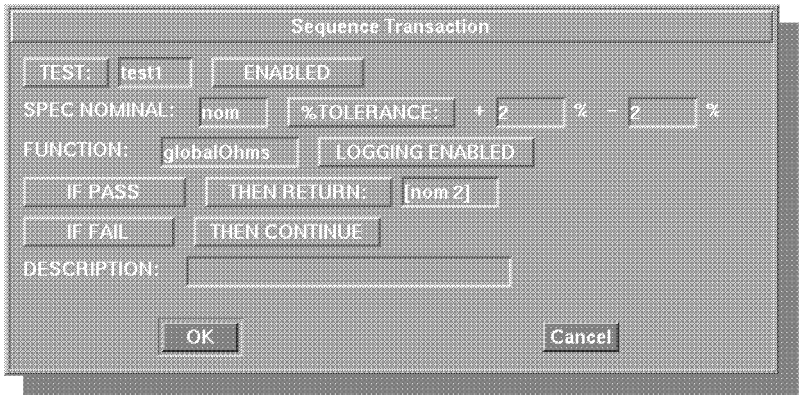
The second transaction (`test2`) tests to see if the value (`test1.result`) is less than or equal to 110 percent of the highest value (330 Ohms) in the array.



If either test fails, an error occurs.

- If an error does occur, the UserObject named `Error Condition` uses a `Triadic` expression to ascertain whether to display `Out of Range: LOW` or `Out of Range: HIGH`. The UserObject is configured as `Show Panel on Exec`, so if either error condition occurs, a display "pops up" to show the error. You'll find that this happens once every few times you run the program because the User Function `simResist` returns random values in the range 100–400. (To continue, just press `OK` in the pop-up box.)
- The transaction `test1` in the first `Sequencer` is the only transaction that calls the User Function `simResist`. (Instead, `test2` includes the expression `test1.result`.) This is necessary in this case because we want to run multiple tests on just one resistance value. Otherwise, a new value would be returned every time the User Function was called. However, there is another reason. Since the User Function `simResist` is only called once, you can easily replace it with a call to a different User Function. The example (`manual46.vee`) contains a second User Function named `measResist`, which uses an HP Instrument Driver to call an HP 3478A Digital Voltmeter configured for resistance measurements. If you have an HP 3478A meter, just connect it to your HP-IB, change the `FORMULA` field in `test1` to the expression `measResist()`, and run the program.

- Regardless of whether simulated or measured resistance values are taken, the `Test Bounds` return value is displayed, and is set as a global variable (`globalOhms`). The three transactions in the `Sequencer` labeled `Test Value & Tolerance` each call this global variable using the expression `globalOhms`, for example:



  If a test passes, the appropriate real array (e.g., `[220 2]`) is output. The `To String` object converts the data to a string (e.g., `220 Ohm, 2%`). The `Sequencer` will be executed as many times as necessary until a `Bin Sort` result is found.

- Note that we are not using the `Log` output terminal in either `Sequencer`, so we've deleted it to speed up execution.
- If you want to see the flow of this program, try running it a few times with `Show Exec Flow` and `Show Data Flow` turned on.

For further information about transaction options, using control pins, and so forth, refer to the `Sequencer` section in the *HP VEE Reference* manual.

For some further examples using the `Sequencer`, look in the directory `mfgtest` in your `examples` directory.

8

Troubleshooting Problems

Troubleshooting Problems

This chapter explains common situations and recovery actions.

Table 8-1. Problems, Causes, and Solutions

Problem	Cause	Solution
Your `UserObject` doesn't operate when you think it should.	You might be crossing the context boundaries with asynchronous data (such as connecting to an XEQ pin on an object inside the `UserObject`).	Possible Solution 1: Move any asynchronous dependencies to outside the `UserObject`. Possible Solution 2: Use `Show Exec Flow` or `Show Data Flow` to view the order of operation in your program.
You want to change the functionality of an object.		Use the object menu features (including control pins).

Table 8-1. Problems, Causes, and Solutions (continued)

Problem	Cause	Solution
You only get one value output from an iterator within a `UserObject`.	A `UserObject` only activates its outputs once.	Take the iterator out of the `UserObject`.
An iterator only operates once.	Your iteration subthread is connected to the sequence output pin, not the data output pin.	Start the iteration subthread from the data output pin.
`For Count` doesn't operate.	The value of `For Count` is 0 or negative.	Change the value; if you need a negative value, negate the output or use `For Range`.
`For Range` or `For Log Range` doesn't operate.	The sign of the step size is wrong. If `From` is less than `Thru`, `Step` must be positive. If `Thru` is less than `From`, `Step` must be negative.	Change `Step`.
You move objects when you try to connect them.	You're clicking too close to the pin. To connect to a pin, the pointer must not be touching the object.	Click just outside the object, near the pin.

Table 8-1. Problems, Causes, and Solutions (continued)

Problem	Cause	Solution
You get the UNIX message **sh:** *name* **- not found**.	You mistyped the name of the executable.	Possible Solution 1: Check your spelling and type it again. Possible Solution 2: Make sure **veeengine** has three *e*'s.
HP VEE appears to hang—the pointer is an hourglass.	Possible Cause 1:HP VEE is rerouting lines because you have **Auto Line Routing** set on and you moved an object. Possible Cause 2: HP VEE is printing the screen or the program. Possible Cause 3: You just **Cut** a large object or a large number of objects. HP VEE is saving the objects to the **Paste** buffer.	Wait. If the pointer doesn't change back to the crosshairs within a few minutes, type CTRL-C (or whatever your **intr** setting is in the terminal window from which you started HP VEE), close the HP VEE window, or kill the HP VEE process.

Table 8-1. Problems, Causes, and Solutions (continued)

Problem	Cause	Solution
You can't **Open** a program, **Cut** objects, or delete a line (the feature is grayed).	The program is still running.	Press **Stop** twice to stop the program, then try the action again.
You can't **Paste** (the feature is grayed).	The **Paste** buffer is empty.	**Cut**, **Copy**, or **Clone** the object(s) again.
You can't **Cut**, **Create UserObject**, or **Add to Panel** (the feature is greyed).	No objects are selected.	Select the objects using **Edit** ⟹ **Select Objects** and try the action again.
A **UserObject** only outputs the last data element generated.	**UserObjects** do not accumulate data in the output terminal buffer. It only holds the last data element received.	Use a **Collector** to gather all of the data generated into an array. Send this data to the output terminal.
You cannot add inputs or modify the condition or formula.	You have a pre-defined object (from **Data** ⟹ **Conditional** ⟹, the **Math** menu or **AdvMath** menu).	Use an **If/Then/Else** or **Formula** object. You can modify them to complete your task.
You can't get out of line drawing mode		Double-click to end line drawing mode.

Table 8-1. Problems, Causes, and Solutions (continued)

Problem	Cause	Solution
You get a `Parse Error` object when you `Open` a program.	Possible Cause 1: You `Save`d a program that contained an object with invalid data, such as an `If/Then/Else` object with an invalid expression.	Replace the `Parse Error` object with a new object.
	Possible Cause 2: You have HP VEE-Engine and `Open`ed a program that was created in HP VEE-Test. `Parse Error` replaces objects that are unique to HP VEE-Test.	Run the program in HP VEE-Test.
Your characters are not appearing correctly.	You have a non-USASCII keyboard.	Refer to Appendix A for recovery information.
Your colors outside of HP VEE are changing (although when you're in HP VEE, the HP VEE colors look normal).	Your color map planes are all used.	Refer to Appendix A for recovery information.

A

Configuring HP VEE

Configuring HP VEE

This appendix explains how to configure and customize HP VEE for your environment by changing HP VEE options, and X11 options (in the UNIX environment) or Windows options (in the MS-Windows environment). This appendix discusses the following topics:

- Changing X11 attributes (such as colors, fonts, and window size and placement)
- Changing MS-Windows attributes (such as colors, fonts, and window size)
- Using two (or more) color palettes
- Customizing your icon bitmaps
- Selecting a bitmap for a panel view
- Recovering from X11 color plane limitations
- Using non-USASCII keyboards
- Using HP-GL Plotters
- Using Two-Byte Character Sets

Changing X11 Attributes (UNIX)

HP VEE provides a variety of font and color palette files that you can use to customize the look of HP VEE. The palette files are stored in `/usr/lib/veeengine/config/` or `/usr/lib/veetest/config/`. The beginning of each file contains a short description of the color and font scheme. To use a palette file, you must install it into your X11 resources database.

If you are using `xrdb`, install the palette file by typing `xrdb -merge` *palette_filename* before starting HP VEE.

If you are not using `xrdb`, merge the palette file into your X11 resources file. Your X11 resources file is usually `.Xdefaults in your $HOME` directory, but may be in a file identified with the environment variable `$XENVIRONMENT`.

To change the color and font scheme for all user of HP VEE on your system, your system administrator may replace the file `/usr/lib/X11/app-defaults/Vee` with one of the palette files.

To change other X11 resources, you can change or add to your X11 resources file. For example, to change the default geometry of the HP VEE window so that it always started in the lower right corner of your screen and the window was sized to 640 by 480 pixels, you would add the following line to your X11 resources file (probably `.Xdefaults`): `Vee*geometry: =640x480-0-0`.

For more information about customizing an X11 environment, refer to *Beginner's Guide to the X Window System*.

Configuring HP VEE for Windows

HP VEE for Windows uses the `V.INI` file to control its environment. `V.INI` contains information about default colors, fonts, window size, etc. `V.INI` is located in your HP VEE installation directory (default `C:\VEE`). If you installed HP VEE for Windows in another directory an entry must go into `WIN.INI` that specifies the install directory. For example:

```
[Vee]
InstallDir=D:\VEE
```

General HP VEE Settings

The `Maximized` variable controls whether HP VEE for Windows starts up as a maximized window or not. The value `0` is for not maximized, `1` is for maximized.

The `Geometry` variable controls the initial size of the HP VEE for Windows window. For example:

```
Geometry=630x470
```

You can also override the default Instrument Driver directory (`C:\HPIDS`) and Instrument Help directory (`C:\HPIDS\HELP`) by defining the variables `InstrumentDriverDir` and `InstrumentHelpDir` in the `[Vee]` section of `V.INI`. For example:

```
[Vee]
InstrumentDriverDir=C:\HPITG2\DRIVERS
InstrumentHelpDir=C:\HPITG2\HELP
```

Color Settings

HP VEE for Windows represents colors in the `V.INI` file in two different ways:

1. RGB triplets: `#rrggbb`, where `rr`, `gg`, `bb` are hex numbers from 00 to ff. For example, `#ff0000` is equivalent to full intensity red.
2. MS-Windows system colors (changeable via the Windows Control Panel): `%nn`, , where nn maps to:

   0=scroll bars
   1=desktop
   2=active title bar
   3=inactive title bar
   4=menu bar
   5=window background
   6=window frame
   7=menu text
   8=window text
   9=active title bar text
   10=active border
   11=inactive border
   12=application workspace
   13=highlight
   14=highlighted text
   15=button face
   16=button shadow
   17=disabled text
   18=button text
   19=inactive title bar text
   20=button highlight

The first section about colors in `V.INI`, `[vee-display]` refers to colors on the display device, The second section, `[vee-print]` refers to colors on the print device.

```
[vee-display]

background=%15
foreground=%18
shadowColor=#808080
```

```
workspace.background=%12
workspace.lineColor=%18

terminal.background=%15
controlTerminal.background=%15
errorTerminal.background=#c05050
executeTerminal.background=#00c030

device.execColor=#a8a83b
device.errorColor=#ff0000
device.pinColor=%18
device.control.pinColor=%18
device.error.pinColor=%18
device.control.pinColor=%18
device.foreground=%18
device.background=%15
errorDevice.background=#ff0000

userPanel.background=%12

titleBar.foreground=%9
titleBar.background=%2
toolBar.foreground=%18
toolBar.background=%15

scrollBar.background=%0
scrollBar.elevator.background=%0
scrollBar.elevator.foreground=%0

button.foreground=%18
button.background=%15
dialogBox.foreground=%18
dialogBox.background=%15
dialogBox.titleBar.foreground=%9
dialogBox.titleBar.background=%2
errorBox.foreground=%18
errorBox.background=%15
errorBox.titleBar.foreground=%8
errorBox.titleBar.background=#ff0000
cautionBox.foreground=%18
cautionBox.background=%15
cautionBox.titleBar.foreground=%8
cautionBox.titleBar.background=#808000
```

```
graph.background=#000000
graph.pen0=#000000
graph.pen1=#ffffff
graph.pen2=#ff0000
graph.pen3=#00ee44
graph.pen4=#ffdd00
graph.pen5=#0000ff
graph.pen6=#ffaaaa
graph.pen7=#00ccff
graph.pen8=#ff8700
graph.pen9=#909090


[vee-print]

background=#f0f0f0
foreground=#101010
shadowColor=#808080

workspace.background=#fafafa
workspace.lineColor=#000000

Device.Background=#f0f0f0
Device.Foreground=#000000
Terminal.Background=#f0f0f0

terminal.background=#f0f0f0
controlTerminal.background=#f0f0f0
errorTerminal.background=#800000
executeTerminal.background=#000080

device.execColor=#a8a83b
device.errorColor=#ff0000
device.pinColor=#000000
device.control.pinColor=#000000
device.error.pinColor=#000000
device.control.pinColor=#000000
device.foreground=#000000
device.background=#f0f0f0
errorDevice.background=#ff0000

userPanel.background=#f0f0f0

titleBar.foreground=#000000
titleBar.background=#d0d0d0
```

```
toolBar.foreground=#000000
toolBar.background=#d0d0d0

scrollBar.background=#f0f0f0
scrollBar.elevator.background=#f0f0f0
scrollBar.elevator.foreground=#f0f0f0

button.foreground=#000000
button.background=#f0f0f0
dialogBox.foreground=#000000
dialogBox.background=#f0f0f0
dialogBox.titleBar.foreground=#000000
dialogBox.titleBar.background=#d0d0d0
errorBox.foreground=#000000
errorBox.background=%f0f0f0
errorBox.titleBar.foreground=#000000
errorBox.titleBar.background=#d0d0d0
cautionBox.foreground=#000000
cautionBox.background=#f0f0f0
cautionBox.titleBar.foreground=#000000
cautionBox.titleBar.background=#d0d0d0

graph.background=#f0f0f0
graph.pen0=#f0f0f0
graph.pen1=#000000
graph.pen2=#ff0000
graph.pen3=#00ee44
graph.pen4=#ffdd00
graph.pen5=#0000ff
graph.pen6=#ffaaaa
graph.pen7=#00ccff
graph.pen8=#ff8700
graph.pen9=#909090


printTitle.font is the font used for printout titles.
printTitle.font=arial,18

printText.font is the font used to print out object descriptions, as
 well as the 'To Printer' font.
printText.font=arial,12
```

Fonts

`V.INI` contains two sections that deal with fonts:

`[Vee-lores]` deals with low resolution screens (standard VGA).

`[Vee-hires]` deals with high resolution screens (800x600, or 1024x768).

Fonts can be represented as:

1. fontname,fontsize
2. normal (equivalent to arial,12 in lores, and arial,17 in hires)
3. large (equivalent to arial,16 in lores, and arial,22 in hires)

The normal and large fonts can be redefined by changing the `normal.font` and `large.font` lines.

```
[vee-lores]

normal.font=arial,12
large.font=arial,16

font=normal
titleBar.font=normal
workspace.titleBar.font=arial bold,15
button.font=normal
;device.name.font=normal
;device.value.font=normal
device.textDisplay.value.font=large
;device.titleBar.font=normal
;dialogBox.titleBar.font=normal

textDisplay.scalar.font=large
textDisplay.array.font=normal
lineProbe.textDisplay.scalar.font=large
lineProbe.textDisplay.array.font=normal
help.font=courier new,14
notepad.font=courier new,14
loggingAlpha.font=courier new,14

instrumentDriver.9x15.font=courier new,15
instrumentDriver.15x25.font=courier new,25


[vee-hires]
```

```
normal.font=arial,17
large.font=arial,22

font=normal
titleBar.font=normal
workspace.titleBar.font=arial,20
button.font=normal
;device.name.font=normal
;device.value.font=normal
device.textDisplay.value.font=large
;device.titleBar.font=normal
;dialogBox.titleBar.font=normal

textDisplay.scalar.font=large
textDisplay.array.font=normal
lineProbe.textDisplay.scalar.font=large
lineProbe.textDisplay.array.font=normal
help.font=courier new,17
notepad.font=courier new,17
loggingAlpha.font=courier new,17

instrumentDriver.9x15.font=courier new,15
instrumentDriver.15x25.font=courier new,25
```

Using Multiple Color Sets/Fonts (UNIX)

It is often convenient to keep two (or more) sets of HP VEE color and font combinations. For example, you may want to use a different font and color palette for printing the screen. (The default printer colors are selected to give a desirable printout after gray-scale conversion, and may not be the best choice for a color printer.)

You can use two different options to start HP VEE with a different "instance" name. These are the `-name` option, which controls the palette and font for the screen, and the `-prname` option, which controls the palette for printing.

The default HP VEE resource file contains a `VeePrint` palette for printing, in addition to the `Vee` palette for the screen. HP VEE will automatically read in the `VeePrint*` color resources and use them for printing, but will use the standard `Vee*` colors on the screen. For example, lines will be white on the screen, but will be printed black.

If you want the printer to use the same colors as the screen, start HP VEE with the `-prname` option and the "instance" name `Vee` as follows:

```
veeengine -prname Vee
-- or --
veetest -prname Vee
```

HP VEE will now use the `Vee*` color resources for printing, instead of the `VeePrint*` color resources.

If you want to create a custom palette and font selection for your display, you'll need to merge the appropriate palette into your X11 environment. You'll have to change the application class name from `Vee*` to another name, such as `myVee*`. (Refer to "Changing X11 Attributes" for further information.)

To use the `myVee*` resources instead of `Vee*` for your display screen, execute one of the following:

```
veeengine -name myVee
-- or --
veetest -name myVee
```

To use the `myVee*` resources both for the screen and for printing:

```
veeengine -name myVee -prname myVee
-- or --
veetest -name myVee -prname myVee
```

Several palettes are found in the directory:

```
/usr/lib/veeengine/config
-- or --
/usr/lib/veetest/config
```

Customizing Icon Bitmaps (UNIX & Windows)

You can change the bitmap displayed on any object icon to a bitmap or pixmap of your choice. HP VEE supports the X11 bitmap, xwd, and GIF formats on UNIX platforms. HP VEE for Windows supports `.icon` files (named `.ICN`) and `.BMP` bitmap files.

To create your own bitmaps on UNIX, use the `/usr/bin/X11/bitmap` program. For example, type:

`/usr/bin/X11/bitmap` *name*`.icon` *size*x*size*

name is the name of the file to create or edit. *size* is the number of pixels for the width and height of the bitmap. `48x48` is a good size for an icon. You can also use any other bitmap editor that outputs the X11 bitmap format.

You can also use an X11 Window Dump (xwd) format file. Just use `/usr/bin/X11/xwd` to capture an image from a window on the screen. Or you can use a paint program such as IslandPaint to create a pixmap and save it in a file in xwd format. HP VEE can also import GIF files.

Note that on SunOS systems the `bitmap` and `xwd` programs are found under the directory `/usr/openwin/bin`.

To create your own bitmap files on MS-Windows use the Paintbrush program.

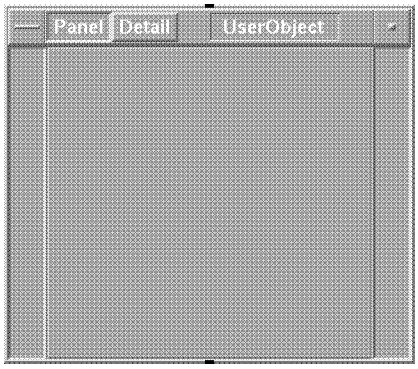
To replace the default bitmap on a single object, select your bitmap from the icon's object menu (`Layout` ⟹ `Select Bitmap`).

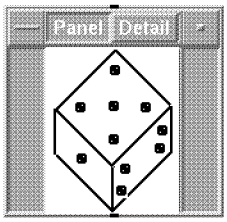
To replace the default bitmap for a category of objects (such as `Formula` objects), rename your bitmap file to the default bitmap file name (such as `formula.icon`) and keep it in your HP VEE startup directory.

Selecting a Bitmap for a Panel View

You can select a bitmap to use as the "background" for a panel view. The bitmap can be an `XWD` or `GIF` file, or any X11 bitmap (or `.ICN` or `.BMP` files for HP VEE for Windows). You can create a bitmap as described in the previous section.

To select a bitmap for the panel view of a UserObject, choose `Select Bitmap` from the object menu. For example, in the illustration below the figure on the left is the panel view of a UserObject, with a blank background. If you choose `Select Bitmap` from the object menu and choose the bitmap named `die5.xwd`, the figure on the right will result (the die will be in color on the screen):





To select a bitmap for the panel view of an HP VEE program, the process is the same except that you choose `Select Bitmap` from the `Edit` menu for the panel view. (Just click on the right mouse button with the pointer in the work area to display the edit menu.)

If You See Colors Changing On Your Screen (UNIX)

Your workstation is equipped with a certain number of color planes (usually 1, 4, 6, or 8). X11 uses the information in these color planes to color your application's window. If you have more than one application running (each in its own window), and you notice the screen colors changing as you move from one application's window to another, then one of two things may be happening. Either all the applications, together, use more colors than your display has available, or one or more of the applications allocates its own private color map (for example, HP BASIC/UX).

HP VEE uses at least 39 colors (this varies depending on how you define the colors and which colors HP VEE actually uses while running), so you may experience this behavior when HP VEE is one of your applications. The symptoms are: when you are in the HP VEE window, the HP VEE colors will be correct for HP VEE, but may be wrong in other application's windows. When you move to another application's window, the colors will be correct for that application, but may be wrong for HP VEE. *This is typical X11 behavior—it is not a problem with HP VEE.*

This behavior does not affect the performance of HP VEE or any other application. However, if it bothers you, there are some things you can do to help, depending on the cause.

There are two causes of this behavior:

- You have requested more colors than your workstation can simultaneously display.
- One of the applications you are running controls a local color map.

Too Many Colors

Your workstation can display some number of colors at one time, based on the number of color planes for your display. This number is:

$$2^{number\ of\ color\ planes}$$

For example, if you have 4 color planes, you can use as many as 16 colors at a time on your display.

$$2^4 = 16$$

If you exceed this number, you may see the screen flashing as you change from one window to another.

If you exceed your total available colors, the first step in eliminating the "flashing" is to reduce your colors to be within the limits of your workstation. Some tips on reducing colors are:

- Remove any "extra" colors. If two applications can use the same color scheme, then customize them to do this.
- Use "reduced-color" color schemes in applications. For example, HP VEE is shipped with several optional color schemes. The color scheme called `Safari` uses the fewest colors in HP VEE (39).
- Stop, or do not even start, any applications that you do not currently need. Often, each application uses its own color scheme. This can quickly increase your requested colors to exceed your color map limit. Once you have stopped other applications, you probably need to stop, then re-start, HP VEE before the behavior goes away.
- Reduce the number of colors allocated by the `xinitcolormap` command. Because these colors remain permanently in the color map, there is room for fewer temporary colors.

Some X11 window managers have a colormap focus directive (for example, `*colormapFocusPolicy`). The value to which this is set may also contribute to how colors are used on the screen. In particular, if you exceed the total number of colors you can simultaneously display, do not set this to be `explicit` or you may not see correct colors in your application's window.

Applications that Use a Local Color Map (UNIX)

Some applications use a local color map. This means that when you run this application, it saves the current color map and switches over to its own, local color map. When this happens you may see the "flashing" between windows.

One way to circumvent this is to pre-allocate the HP VEE colors using the `xinitcolormap` command. To do this, you create an ASCII file listing the colors you wish to pre-allocate. This file is described in the `man` page for `xinitcolormap`. Basically, though, the file cannot contain blank lines, must start with the colors `Black` and `White`, and the color format can be either pre-defined word colors or the actual RGB hex values, preceded by the symbol—`#`. For example, the following two examples contain black, white, and a shade of light gray:

```
Black
White
LightGray
```

Figure A-1. Color Map File Using Words

```
#000000
#ffffff
#a8a8a8
```

Figure A-2. Color Map File Using Hex Numbers

HP BASIC/UX is one application that uses a local color map and recommends that you pre-allocate the HP BASIC/UX colors at startup using the `xinitcolormap` command (refer to the `/usr/lib/rmb/newconfig/rgb.README` file).

Because of this, if you will use HP VEE with HP BASIC/UX (or other applications that allocate colors in the same way HP BASIC/UX does), you need to also pre-allocate HP VEE colors at startup. If you do not, you may see the colors flash on the screen as you move from one window to another.

To do this:

1. Create a "colormap" file that contains all the different HP VEE colors you will use. HP has supplied the following colormap files in HP VEE's configuration directory:
   - `DefColorMap` if you are using the default HP VEE colors.
   - `SafColorMap` if you are using the `Safari` color scheme (use this if you need to use fewer colors).

2. Change to your `$HOME` directory:

   ```
   cd $HOME
   ```

3. Concatenate the HP BASIC/UX and the HP VEE colormap files:

```
cat /usr/lib/rmb/newconfig/xrmbcolormap vee-colormapfile > .xveecolormap
```

   Note that the HP BASIC/UX colors must go first, because HP BASIC/UX assumes that they are the first 16 entries in the colormap. You can mix the word colors and the hex number colors in one file.

4. You must use the `xinitcolormap` command before you allocate any colors for other applications. This means that it should be placed near the beginning of your .x11start file.

   For example, if you use the `.x11start` file, your colors are in `$HOME/.xveecolormap`, and you have 55 colors listed in the file (16 from HP BASIC/UX + 39 from HP VEE), you would add the following line to `.x11start`:

   ```
   /usr/bin/X11/xinitcolormap -c 55 -f $HOME/.xveecolormap
   ```

5. Restart X11. To do this, stop the window manager by pressing the following three keys at the same time: Shift-CTRL-Break, or selecting `Reset` from your root menu (if it is configured for this choice), then type:

   ```
   x11start
   ```

Using Non-USASCII Keyboards (UNIX)

If you are using a non-USASCII keyboard, you need to use a special HP VEE palette so that your characters appear correctly in HP VEE. You need to modify your X11 environment to use the `Iso` or `Katakana` palette. Install the palette using the instructions in "Changing X11 Attributes (UNIX)".

Use the `Iso` palette if you have one of the following keyboards:

- Belgian
- Canadian English
- Canadian French
- Danish
- Dutch
- European Spanish
- Finnish
- French
- German
- Italian
- Latin Spanish
- Norwegian
- Swedish
- Swiss French
- Swiss German
- UK English

Use the `Katakana` or `Kanji` palette if you have one of the following keyboards:

- Katakana
- Kanji

NOTE

If you are accessing data that was created with the `Roman8` character set, you must translate any special characters (above ASCII 127) used.

Your terminal window may use `Roman8`; therefore `TEXT` written to stdout, file names (such as specified by `To File` and `From File`), and programs names must use ASCII characters 0-127 to match with those specified with HP VEE.

Using HP-GL Plotters (UNIX Only)

HP VEE supports graphics output to plotters and files using HP-GL. Before you can send plots to a plotter (either local or networked) your system administrator must add the plotter as a spooled device on your system.

In addition to standard HP-GL plotters such as the HP 7475, the HP ColorPro (HP 7440), or the HP 7550, some printers can be used as plotters, such as the PaintJet XL, and the LaserJet III. The HP ColorPro plotter requires the Graphics Enhancement Cartridge in order to plot polar or Smith Chart graticules, or an Area-Fill line type. The PaintJet XL requires the HP-GL/2 Cartridge in order to make any plots. In order to make plots on the LaserJet III, at least two megabytes of optional memory expansion is required, and the Page Protection configuration option should be enabled. Plots of many vectors, especially with Polar or Smith chart graticules, may require even more optional memory in the LaserJet III. Any plot intended for a printer requires the plotter type to be set to HP-GL/2, which causes the proper HP-GL/2 setup sequence to be included with the plot information.

Any of the following graphical two-dimensional displays can be plotted to an HP-GL or HP-GL/2 plotter, or to a file:

```
XY Trace, Strip Chart, Complex Plane,
X vs Y Plot, Polar Plot, Waveform,
Magnitude Spectrum, Phase Spectrum,
Magnitude vs Phase.
```

You can specify the appropriate plotter configuration by selecting: `File` ⟹ `Preferences` ⟹ `Plotter Config`.

To generate a plot, just select `Plot` on the display's object menu, specify the required parameters in the `Plot Display` dialog box, and then press `OK`. You can also add `Plot` as a control input to generate plots programmatically. The entire view of the display object will be plotted, and scaled to fill the defined plotting area, while retaining the aspect ratio of the original display object. By re-sizing the display object, you can control the aspect ratio of the plotted image. By making the display object larger, you can reduce the relative size of the text and numeric labels around the plot.

For an explanation of the plotter configuration parameters, refer to the `Plotter Config` section in the *HP VEE Reference* manual. Also, refer to the reference sections for the appropriate two-dimensional display devices.

Using Two-Byte Character Sets (UNIX Only)

Two-byte characters (such as Kanji for Japanese) can be entered into any field in HP-VEE where you can enter text. This includes all titles, text (string) constants, input/output pin names, and note pads. To use two-byte characters you must have the UNIX NLIO subsystem installed and initialized, the $LANG shell variable set to your local language (for example: `export LANG=japanese`), and an appropriate set of fonts selected (such as the Kanji app-defaults file):

```
/usr/lib/veeengine/config/Kanji

-- or --

/usr/lib/veetest/config/Kanji
```

Refer to "Changing X11 Attributes" at the beginning of this appendix for further information.

NOTE

The Kanji app-defaults file specifies a two-byte "stroke" font. A stroke font consists of characters drawn as strokes, rather than as a raster image. The stroke font is required for output to most plotters, since most plotters do not directly support Kanji.

The following are some limitations to two-byte character support:

- When using the `Plot` command to send a graphical two-dimensional display (e.g., `XY Trace`) to a plotter or file, you must first specify that labels are to be output using the two-byte "stroke" font. To do this, go to the `Plotter Config` dialog box:

  `File` ⟹ `Preferences` ⟹ `Plotter Config`.

  The `Label Using` field gives you two choices: `Stroke Fonts` and `Plotter ROM`. You must select `Stroke Fonts` in order to output a display with two-byte characters. Otherwise, all two-byte characters in field labels will

be encoded into HP-GL label commands as two-byte characters in the HP15 character set, which is not supported by most plotters.

For further information, refer to the `Plotter Config` section in the *HP VEE Reference* manual.

- When reading text that includes two-byte characters from a `Direct I/O` object, the two-byte character rules are not used when looking for the EOL string. Thus, an EOL character may be incorrectly found in the second byte of a two-byte character. (This is only a problem if an EOL character has an ASCII value greater than 32 decimal.)
- Date/Time parsing and formatting have not been globalized, and continue to only execute in English. To obtain a localized date string, use an `Execute Program` object with a program of "date" and a transaction of `READ TEXT x STR`.

B

Example Programs and Library Objects

Example Programs and Library Objects

HP VEE-Engine, HP VEE-Test and HP VEE for Windows include several examples of HP VEE programs that you can use. A library of objects that you can `Merge` into your programs is also included. The example programs and library objects are installed as part of the normal HP VEE installation process.

Using the Examples

Many of the examples from the manuals are included in the **examples** directories (with file names like **manual01.vee**, etc). Other examples, not referenced in any of the manuals, are included to illustrate specific HP VEE concepts, or to illustrate solutions to engineering problems using HP VEE. To help you find the example you want, the examples are divided into subdirectories.

For HP VEE-Engine the examples are installed in subdirectories under:

/usr/lib/veeengine/examples/

For HP VEE-Test the examples are installed in subdirectories under:

/usr/lib/veetest/examples/

For HP VEE for Windows the examples are installed in subdirectories under:

C:\VEE\EXAMPLES

Under **examples** you will find several subdirectories, each containing a class of example programs. (The exact selection of subdirectories depends on which version of HP VEE you have.)

Once you have selected an example program of interest, just load it with **File** ⟹ **Open** as with any other program. If you want to modify an example program and save it, you'll want to save it in a different directory. (You can't write to the **examples** subdirectories unless you are logged on as "root.")

HP VEE for Windows on-line **Help** provides a description of the example programs as well as an (Open) menu button to quickly load the programs.

Using Library Objects

The object library provides several objects that you can merge into your own program. Just select **Merge** from the **File** menu and a list box will appear for the appropriate library directory:

```
/usr/lib/veeengine/lib/

- or -

/usr/lib/veetest/lib/

- or -

C:\VEE\LIB\
```

To merge an object, just double click on its name and insert the object in your program.

Most of the library objects are actually UserObjects that encapsulate individual objects. You can create your own UserObjects for the library, but you'll need to save them in the **contrib** subdirectory (Unix only):

```
/usr/lib/veeengine/lib/contrib/

- or -

/usr/lib/veetest/lib/contrib/
```

(You can't write to the **lib** directory unless you are logged on as "root" on Unix platforms.)

The **contrib** subdirectory is empty at installation — it provides a place for your own library of "contributed" objects.

There is another subdirectory under **lib**, named **conversions**:

```
/usr/lib/veeengine/lib/conversions/

- or -

/usr/lib/veetest/lib/conversions/
```

```
- or -

C:\VEE\LIB\CONVERT\
```

This subdirectory contains several formula objects that you can `Merge` into your program. Each of these objects performs a useful conversion function such as degrees to radians.

C

ASCII Table

ASCII Table

This appendix contains reference tables of ASCII 7-bit codes.

ASCII 7-bit Codes

	Binary	Oct	Hex	Dec	HP-IB Msg
NUL	0000000	000	00	0	
SOH	0000001	001	01	1	GTL
STX	0000010	002	02	2	
ETX	0000011	003	03	3	
EOT	0000100	004	04	4	SDC
ENQ	0000101	005	05	5	PPC
ACK	0000110	006	06	6	
BEL	0000111	007	07	7	
BS	0001000	010	08	8	GET
HT	0001001	011	09	9	TCT
LF	0001010	012	0A	10	
VT	0001011	013	0B	11	
FF	0001100	014	0C	12	
CR	0001101	015	0D	13	
SO	0001110	016	0E	14	
SI	0001111	017	0F	15	
DLE	0010000	020	10	16	
DC1	0010001	021	11	17	LLO
DC2	0010010	022	12	18	
DC3	0010011	023	13	19	
DC4	0010100	024	14	20	DCL
NAK	0010101	025	15	21	PPU
SYN	0010110	026	16	22	
ETB	0010111	027	17	23	

ASCII 7-bit Codes (continued)

	Binary	Oct	Hex	Dec	HP-IB Msg
CAN	0011000	030	18	24	SPE
EM	0011001	031	19	25	SPD
SUB	0011010	032	1A	26	
ESC	0011011	033	1B	27	
FS	0011100	034	1C	28	
GS	0011101	035	1D	29	
RS	0011110	036	1E	30	
US	0011111	037	1F	31	
space	0100000	040	20	32	listen addr 0
!	0100001	041	21	33	listen addr 1
"	0100010	042	22	34	listen addr 2
#	0100011	043	23	35	listen addr 3
$	0100100	044	24	36	listen addr 4
%	0100101	045	25	37	listen addr 5
&	0100110	046	26	38	listen addr 6
’	0100111	047	27	39	listen addr 7
(	0101000	050	28	40	listen addr 8
)	0101001	051	29	41	listen addr 9
*	0101010	052	2A	42	listen addr 10
+	0101011	053	2B	43	listen addr 11
,	0101100	054	2C	44	listen addr 12
-	0101101	055	2D	45	listen addr 13
.	0101110	056	2E	46	listen addr 14
/	0101111	057	2F	47	listen addr 15

ASCII 7-bit Codes (continued)

	Binary	Oct	Hex	Dec	HP-IB Msg
0	0110000	060	30	48	listen addr 16
1	0110001	061	31	49	listen addr 17
2	0110010	062	32	50	listen addr 18
3	0110011	063	33	51	listen addr 19
4	0110100	064	34	52	listen addr 20
5	0110101	065	35	53	listen addr 21
6	0110110	066	36	54	listen addr 22
7	0110111	067	37	55	listen addr 23
8	0111000	070	38	56	listen addr 24
9	0111001	071	39	57	listen addr 25
:	0111010	072	3A	58	listen addr 26
;	0111011	073	3B	59	listen addr 27
<	0111100	074	3C	60	listen addr 28
=	0111101	075	3D	61	listen addr 29
>	0111110	076	3E	62	listen addr 30
?	0111111	077	3F	63	UNL
@	1000000	100	40	64	talk addr 0
A	1000001	101	41	65	talk addr 1
B	1000010	102	42	66	talk addr 2
C	1000011	103	43	67	talk addr 3
D	1000100	104	44	68	talk addr 4
E	1000101	105	45	69	talk addr 5
F	1000110	106	46	70	talk addr 6
G	1000111	107	47	71	talk addr 7

ASCII 7-bit Codes (continued)

	Binary	Oct	Hex	Dec	HP-IB Msg
H	1001000	110	48	72	talk addr 8
I	1001001	111	49	73	talk addr 9
J	1001010	112	4A	74	talk addr 10
K	1001011	113	4B	75	talk addr 11
L	1001100	114	4C	76	talk addr 12
M	1001101	115	4D	77	talk addr 13
N	1001110	116	4E	78	talk addr 14
O	1001111	117	4F	79	talk addr 15
P	1010000	120	50	80	talk addr 16
Q	1010001	121	51	81	talk addr 17
R	1010010	122	52	82	talk addr 18
S	1010011	123	53	83	talk addr 19
T	1010100	124	54	84	talk addr 20
U	1010101	125	55	85	talk addr 21
V	1010110	126	56	86	talk addr 22
W	1010111	127	57	87	talk addr 23
X	1011000	130	58	88	talk addr 24
Y	1011001	131	59	89	talk addr 25
Z	1011010	132	5A	90	talk addr 26
[	1011011	133	5B	91	talk addr 27
\	1011100	134	5C	92	talk addr 28
]	1011101	135	5D	93	talk addr 29
^	1011110	136	5E	94	talk addr 30
_	1011111	137	5F	95	UNT

ASCII 7-bit Codes (continued)

	Binary	Oct	Hex	Dec	HP-IB Msg
‘	1100000	140	60	96	secondary addr 0
a	1100001	141	61	97	secondary addr 1
b	1100010	142	62	98	secondary addr 2
c	1100011	143	63	99	secondary addr 3
d	1100100	144	64	100	secondary addr 4
e	1100101	145	65	101	secondary addr 5
f	1100110	146	66	102	secondary addr 6
g	1100111	147	67	103	secondary addr 7
h	1101000	150	68	104	secondary addr 8
i	1101001	151	69	105	secondary addr 9
j	1101010	152	6A	106	secondary addr 10
k	1101011	153	6B	107	secondary addr 11
l	1101100	154	6C	108	secondary addr 12
m	1101101	155	6D	109	secondary addr 13
n	1101110	156	6E	110	secondary addr 14
o	1101111	157	6F	111	secondary addr 15
p	1110000	160	70	112	secondary addr 16
q	1110001	161	71	113	secondary addr 17
r	1110010	162	72	114	secondary addr 18
s	1110011	163	73	115	secondary addr 19
t	1110100	164	74	116	secondary addr 20
u	1110101	165	75	117	secondary addr 21
v	1110110	166	76	118	secondary addr 22
w	1110111	167	77	119	secondary addr 23

ASCII 7-bit Codes (continued)

	Binary	Oct	Hex	Dec	HP-IB Msg
x	1111000	170	78	120	secondary addr 24
y	1111001	171	79	121	secondary addr 25
z	1111010	172	7A	122	secondary addr 26
{	1111011	173	7B	123	secondary addr 27
|	1111100	174	7C	124	secondary addr 28
}	1111101	175	7D	125	secondary addr 29
~	1111110	176	7E	126	secondary addr 30
[del]	1111111	177	7F	127	

D

HP VEE Utilities

HP VEE Utilities

HP VEE provides some utility programs. You can access these programs from a UNIX command line in any X11 window or from the HP VEE for Windows program group in MS-Windows.

The veedoc Utility for Documenting Programs

HP VEE includes a utility program **veedoc**, accessible from a UNIX or MS-DOS command line, which extracts information from a program created with HP VEE. The **veedoc** utility prints a line for every object or local User Function in your program. Each line contains an identification number and the name of the object or User Function. This identification number denotes the relative "nesting" position of the object in the program and is unique to a particular object. The identification number, once assigned, will not change and will not be reused. The **veedoc** utility also extracts the **Show Description** information for the root context level (accessible through the **File** ⟹ **Show Description** menu), all objects, and all local User Functions. If a **Note Pad** object is present, its content is also extracted.

To use this utility, go to a UNIX or MS-DOS command line and execute:

```
/usr/lib/veetest/veedoc filename [ ... ]
- or -
/usr/lib/veeengine/veedoc filename [ ... ]
- or -
C:\VEE\VEEDOC filename [ ... ]
```

where *filename* is the name of your program, including path. For example, to run **veedoc** on the example program **mfgtest.vee**, execute:

```
veedoc examples/mfgtest/mfgtest.vee  (UNIX)
- or -
VEEDOC \VEE\EXAMPLES\MFGTEST\MFGTEST.VEE  (MS-DOS)
```

To print this same information, execute:

```
veedoc examples/mfgtest/mfgtest.vee | lp  (UNIX)
- or -
VEEDOC \VEE\EXAMPLES\MFGTEST\MFGTEST.VEE > PRN
```

Additional information can be found by looking at the UNIX manual page for **veedoc**, which was added when you installed HP VEE. This **man** page is included below for your convenience.

NOTE

The `veedoc` utility is compatible with programs created with HP VEE Release A.00.01 and later versions. If you want to use `veedoc` with programs created with Release A.00.00, you must first load the program into the current version of HP VEE and then re-save it.

```
VEEDOC(1)                                                    VEEDOC(1)


NAME
     veedoc - veedoc is a utility to extract information about a program
     created with HP's visual engineering environment (HP VEE).

SYNOPSIS
     veedoc [filename]

REMARKS
     This command requires installation of optional HP VEE-Test or HP VEE-
     Engine software (not included with the standard HP-UX operating
     system) before it can be used.

HP-UX COMPATIBILITY
     Versions: HP-UX 8.0 and later.

DESCRIPTION
     HP VEE-Test is a visual engineering environment that runs under the X
     Window System (X11). veedoc is a utility that extracts the
     identification numbers, names and the Show Description information from
     objects and User Functions within an HP VEE program.  Note Pad contents
     are also extracted.

     Two current features of HP VEE are the ability to "comment" each
     object (using Show Description) for another developer, and the ability
     to create custom HP VEE objects (UserObjects) or functions (User
     Functions).  When an HP VEE program is saved to disk, it creates a file
     that contains all information about the program. veedoc accesses this
     file and provides documentation of the objects and User Functions in
     the program, along with their "comments".

     First veedoc lists the program file name, the HP VEE version used to
     create the file and the date of the last revision.  Then the title of
     the program is printed, followed by the top-level Show Description
     information if it exists.  The User Functions are documented next,
     followed by the objects.  Each object or User Function is documented
     by one line containing its identification number and its name as shown
     in the title bar.  If the object or User Function has information
     entered under Show Description, it is printed next.  If the object is
     a Note Pad, its contents follow.

     Identification numbers are assigned to the object when the object is
```

```
     created, and are saved in the program file.  The identification number
     never changes nor is reused.  Moving an object from one context to
     another is effectively a delete followed by an add.  Thus the object
     receives a new identification number.  The identification numbers
     reflect the nesting of the UserObjects. User Function identification
     numbers begin with the character 'F'.

     This utility works with programs created using HP VEE-Test or HP VEE-
     Engine.

RETURN VALUE
     veedoc returns 0 (zero) if successful, or non-zero if an error was
     encountered.

FILES
     /usr/lib/veetest/veedoc            executable veedoc file

      or

     /usr/lib/veeengine/veedoc

AUTHOR
     veedoc was developed by the Hewlett-Packard Company.

SEE ALSO
     Using HP VEE-Engine and HP VEE-Test,
     HP VEE-Engine and HP VEE-Test Reference, and
     X(1).
```

The HP Driver Writer Tool

The HP Driver Writer Tool (HP DWT) is a utility program that allows you to create your own HP Instrument Driver (or ID). This utility is installed as part of the normal HP VEE-Test and HP VEE for Windows installation, but it is not shipped with HP VEE-Engine.

To start the HP DWT on a UNIX platform, execute the following command from the UNIX command line in an X11 window.

```
/usr/lib/veetest/dwt
```

To start the HP DWT on an MS-Windows platform, click on the Driver Writer Tool icon in the HP VEE application window.

A menu driven window, similar to the HP VEE window, will appear. The pull-down menus `File`, `Edit`, `Create`, `Other`, and `Help` allow you to navigate within the HP DWT. For instructions on how to create, edit, save, compile, and use an ID, refer to the online `Help` for the HP DWT. For the advanced ID programmer, the ID syntax is described in the *HP Instrument Driver Language Reference* manual. Ordering information for this manual is given in the online Help.

The DWT is sufficient to write component drivers to control instruments. If you need to write full state drivers or complete instrument panel drivers, you will need to obtain the *HP Instrument Driver Language Reference* manual (p/n E2001-90004) and become more familiar with the driver writing process.

The HP Instrument Driver Compiler

After writing an Instrument Driver you must also compile the driver. This is normally done as part of the driver writing process. In fact, the Driver Writer Tool asks you if you want to compile the driver when you save the driver file. Unless you learn how to write a driver using a text editor, you do not need to use the ID Compiler. The ID Compiler is included only in the MS-Windows application window for your convenience. For more help on its operation see the on-line help in the Driver Writers Tool.

The Instrument Finder (MS-Windows Only)

The Instrument Finder locates all interfaces and instruments connected to your computer. Click on the **Instrument Finder** icon in the HP VEE application window. It will display the connected devices. For more information on each device, click on the listing for the device and then press the More Info button in the **Instrument Finder** Window. The Print... button will print the information displayed in this window to your printer.

Install Drivers (PC)

The HP VEE for Windows Driver Installation tool helps you install additional drivers on your system. You can start this application from the HP VEE application group window or as part of the installation process. For more information see the *Getting Started with HP VEE for Windows* manual.

Configure I/O Utility

The Hewlett Packard `Configure I/O` Utility for Windows allows you to configure the `sicl.ini` file for the SICL (Standard Instrument Control Library) Windows drivers. This utility can configure the following interface cards:

- HP 82990 HP-IB Card
- HP 82335 HP-IB Card
- HP 82341 HP-IB Card

Also, the `Configure I/O` utility will configure the serial `COM` ports indentified in `win.ini`.

This utility maps each HP-IB card to a select code (logical unit number) and a symbolic name. This mapping is determined by the I/O table defined for HP VEE for Windows. See Appendix G for more information about PC select codes.

You cannot change the select code or the symbolic name of the interface card. However, you can change other interface attributes such as interrupt line or for the serial port, reception queue size and baud rate.

E

I/O Transaction Reference

I/O Transaction Reference

This appendix contains details about the behavior of all I/O transaction actions, encodings, and formats. This appendix is organized by the transaction actions summarized in Table E-1. For example, if you need detailed information about `TEXT` encoding, do this:

- Look in the `WRITE` section for details about `WRITE TEXT` transactions.
- Look in the `READ` section for details about `READ TEXT` transactions.

Table E-1. Summary of Transaction Types

Action	Description
`WRITE`	Writes data to the destination specified in the object.
`READ`	Reads data from the source specified in the object.
`EXECUTE`	Executes low-level commands to control the file, device, or interface associated with the object. `EXECUTE` is used to adjust file pointers, to close pipes and files, and to provide low-level control of devices and hardware interfaces.
`WAIT`	Waits for the specified number of seconds before executing the next transaction. For `Direct I/O`[1] objects, `WAIT` can also wait for a specific serial poll response, or for specific values in accessible VXI device registers.
`SEND`[1]	Sends IEEE 488-defined bus messages (bus commands and data) to an HP-IB interface.
`READ(REQUEST)`[2]	Reads DDE data from another application.
`WRITE(POKE)`[2]	Writes DDE data to another application.

1 HP VEE-Test only.

2 HP VEE for Windows only.

Table E-2. Summary of I/O Transaction Objects

Objects	Supported Transactions				
	EXECUTE	WAIT	READ	WRITE	SEND[1]
To File	X	X		X	
From File	X	X	X		
To Printer		X		X	
To String		X		X	
From String		X	X		
To StdOut		X		X	
From StdIn		X	X		
To StdErr		X		X	
Execute Program (UNIX)[2]	X	X	X	X	
To/From Named Pipe	X	X	X	X	
Direct I/O[1]	X	X	X	X	
Interface Operations[1]	X				X
To/From HP BASIC/UX[1]	X	X	X	X	
To/From DDE[3]	X	X	X	X	

1 HP VEE-Test only.

2 **Execute Program (PC)** is not transaction based.

3 HP VEE for Windows only.

WRITE Transactions

This section is organized by the `WRITE` encodings summarized in Table E-3. Topics that apply to all `WRITE` encodings are summarized at the beginning of this section.

Path-Specific Behaviors

Some `WRITE` transactions behave differently depending on the I/O path of the destination. For example, `WRITE TEXT HEX` transactions format hexadecimal numbers differently depending on whether the destination is a UNIX file or an instrument. To distinguish these behaviors, this section uses the following terms:

Term	Meaning
UNIX paths	Any destination other than an instrument, such as a UNIX file, a string, the printer, or a UNIX pipe.
MS-DOS paths	Any destination other than an instrument, such as an MS-DOS file, a string, or the printer.
direct I/O paths	Any instrument accessed using `Direct I/O`.

The behaviors described in the following sections apply to all paths, except as specifically noted.

Table E-3. WRITE Encodings and Formats

Encodings	Formats
TEXT	DEFAULT STRING QUOTED STRING INTEGER OCTAL HEX REAL COMPLEX PCOMPLEX COORD TIME STAMP
BYTE	Not Applicable
CASE	Not Applicable
BINARY	STRING BYTE INT16 INT32 REAL32 REAL64 COMPLEX PCOMPLEX COORD
BINBLOCK	BYTE INT16 COMPLEX INT32 PCOMPLEX REAL32 REAL64 COORD

Table E-3. WRITE Encodings and Formats (continued)

Encodings	Formats
CONTAINER	DEFAULT STRING INTEGER REAL COMPLEX PCOMPLEX COORD WAVEFORM SPECTRUM
STATE[1]	Not Applicable
REGISTER[2]	BYTE WORD16 WORD32 REAL32
MEMORY[2]	BYTE WORD16 WORD32 REAL32
IOCONTROL[3]	Not Applicable

1 Direct I/O to HP-IB only (HP VEE-Test only).

2 Direct I/O to VXI only (HP VEE-Test only).

3 Direct I/O to GPIO only (HP VEE-Test only).

TEXT Encoding

WRITE TEXT transactions are of this form:

WRITE TEXT *ExpressionList [Format]*

ExpressionList is a single expression or a comma-separated list of expressions.

Format is an optional setting that specifies one of the formats listed in Table E-4.

Table E-4. Formats for `WRITE TEXT` Transactions

Format	Description
`DEFAULT`	HP VEE automatically determines an appropriate text representation based on the data type of the item being written.
`STRING`	Writes Text data without any conversion. Writes numeric data types as Text with maximum numeric precision.
`QUOTED STRING`	Writes data in the the same format as `STRING`, except the data is surrounded by double quotes (ASCII 34 decimal).
`INTEGER`	Writes data as a 32-bit two's complement integer in decimal form.
`OCTAL`	Writes data as a 32-bit two's complement integer in octal form.
`HEX`	Writes data as a 32-bit two's complement integer in hexadecimal form.
`REAL`	Writes data as a 64-bit floating point number in a variety of notations including fixed decimal and scientific notation.
`COMPLEX`	Writes a comma-separated pair of 64-bit floating point numbers that represent a complex number. The first number represents the real part and the second number represents the imaginary part.
`PCOMPLEX`	Writes a comma-separated pair of 64-bit floating point numbers that represent a complex number. The first number represents the magnitude and the second number represents the phase angle in the phase units specified in the transaction.
`COORD`	Writes a comma-separated series of 64-bit floating point numbers that represent a rectangular coordinate.
`TIME STAMP`	Converts a real number (for example, the output of the `now()` function) to a meaningful form and writes it in a variety of combinations of year, month, day, and time.

DEFAULT Format

`WRITE TEXT` (default) transactions are of this form:

`WRITE TEXT` *ExpressionList*

ExpressionList is a single expression or a comma-separated list of expressions.

The transaction converts each item in *ExpressionList* to a meaningful string and writes it. Consider the simple case of writing the scalar variable `X`:

```
WRITE TEXT X
```

Figure E-1. A `WRITE TEXT` Transaction

If `X` in Figure E-1 contains text, such as:

```
bird cat dog
```

then no conversion is performed and the transaction writes exactly 12 characters.

If `X` in Figure E-1 contains a scalar Integer, such as:

```
8923      the value of X (decimal notation)
```

then the numeric value is converted to text and HP VEE writes exactly four characters.

If `X` in Figure E-2 contains a scalar real value, such as:

```
1.2345678901234567
```

Figure E-2. Numeric Data

then each significant digit up to 16 significant digits is written. (The least significant digit is approximate because of the conversion between HP VEE's internal binary form and decimal notation).

For example, if you write the data in Figure E-2 using this transaction:

```
WRITE TEXT a EOL
```

then HP VEE writes this:

```
1.234567890123457
```

If the absolute value of the number is sufficiently large or small, exponential notation is used. The Reals that form the sub-elements of Coord, Complex, and PComplex behave the same way.

If `EOL ON` is specified for any `WRITE TEXT DEFAULT` transaction, the character specified in the `EOL Sequence` field for that object is written following the last character in *ExpressionList*.

STRING Format

`WRITE TEXT STRING` transactions are of this form:

`WRITE TEXT` *ExpressionList* `STR`

ExpressionList is a single expression or a comma-separated list of expressions.

`WRITE TEXT STRING` transactions behave basically the same as `WRITE TEXT` (default) transactions (one exception will be discussed). The significant difference is that `STRING` allows you to specify additional details about output formatting including field width, justification, and number of characters.

Field Width and Justification. If a transaction specifies `DEFAULT FIELD WIDTH`, only those characters resulting from the conversion of items within *ExpressionList* to Text are written.

If a transaction specifies `FIELD WIDTH:` *F*, then the converted Text is written right- or left-justified within a space *F* characters wide.

The transactions in Figure E-3 specify that all characters are to be written within a field of twenty characters with left justification.

```
WRITE TEXT X STR FW:20 LJ EOL
WRITE TEXT Y STR FW:20 LJ EOL
```

Figure E-3. Two `WRITE TEXT STRING` Transactions

If `X` and `Y` in Figure E-3 have these values:

```
bird cat dog          the Text value of X
12345678901234567     the Real value of Y
```

then HP VEE writes this:

```
bird cat dog
12345678901234567
^                  ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the right of `dog` and to the right of the second `7` are spaces (ASCII 32 decimal).

If justification is changed to `RIGHT JUSTIFY`, then the transactions appear as shown in Figure E-4.

```
WRITE TEXT X STR FW:20 RJ EOL
WRITE TEXT Y STR FW:20 RJ EOL
```

Figure E-4. Two `WRITE TEXT STRING` Transactions

If `X` and `Y` in Figure E-4 have these values:

```
bird cat dog          the Text value of X
12345678901234567     the Real value of Y
```

then HP VEE writes this:

```
          bird cat dog
   12345678901234567
^                  ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the left of `bird` and to the left of the first `1` are spaces (ASCII 32 decimal).

If the length of a string exceeds the specified field width, the entire string is written. The field width specification never truncates; only `MAX NUM CHARS` can truncate characters.

The transaction in Figure E-5 specifies that all characters are to be written in a field width of four characters with left justification.

```
WRITE TEXT X STR FW:4 LJ
```

Figure E-5. A `WRITE TEXT STRING` Transaction

If `X` in Figure E-5 has this value:

`bird cat dog` *the Text value of X, 12 characters*

then HP VEE writes this:

`bird cat dog` *all 12 characters*

Even though the specified field width is four characters, the transaction writes all twelve characters of the string.

Number of Characters. If you specify `ALL CHARS`, then all of the characters generated by the conversion of each item in *ExpressionList* are written. If you specify `MAX NUM CHARS:` *M*, then only the first *M* characters of each item in *ExpressionList* are written.

The transactions in Figure E-6 specify that a maximum of seven characters are written in each field, the field width is twenty characters, and field entries are left justified.

```
WRITE TEXT X STR:7 FW:20 LJ EOL
WRITE TEXT Y STR:7 FW:20 LJ EOL
```

Figure E-6. Two `WRITE TEXT STRING` Transactions

If `X` and `Y` in Figure E-3 have these values:

```
bird cat dog          the Text value of X
12345678901234567     the Real value of Y
```

then HP VEE writes this:

```
bird ca
1234567
^                   ^
```

Notice that the numeric value of `Y` is first converted to Text and characters are truncated. Numeric values are not rounded by `MAX NUM CHARS`.

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the right of `bird` and to the right of the first `1` are spaces (ASCII 32 decimal).

Writing Arrays with Direct I/O. `WRITE TEXT STR` transactions that write arrays to direct I/O paths ignore the `Array Separator` setting for the `Direct I/O` object. These transactions always use linefeed (ASCII decimal 10) to separate each element of an array (which is a string) as it is written. This behavior is consistent with the needs of most instruments.

NOTE

This special behavior for arrays does not apply to any other types of transactions.

QUOTED STRING Format

`WRITE TEXT QUOTED STRING` transactions are of this form:

```
WRITE TEXT ExpressionList QSTR
```

ExpressionList is a single expression or a comma-separated list of expressions.

In general, the behaviors previously discussed for the `STRING`

format apply to `QUOTED STRING` format. There are two differences between `STRING` and `QUOTED STRING`:

- For `QUOTED STRING`, a double quote (ASCII 34 decimal) is added to the beginning and the end of the string. Note that the double quotes are applied before any padding spaces are added to justify the string within the specified field width.
- Control characters (ASCII 0-31 decimal), escape characters (Table E-5), and the characters ' (ASCII 39 decimal) and " (ASCII 34 decimal) embedded inside a double-quoted string receive special treatment.

Field Width and Justification. If you specify `DEFAULT FIELD WIDTH`, only those characters resulting from the conversion of items within *ExpressionList* to Text and the surrounding double quotes are written.

If you specify `FIELD WIDTH:` *F*, then the converted Text and the surrounding quotes are written right or left justified within a space *F* characters wide.

The transactions in Figure E-7 specify that all characters are to be written as quoted strings in a field 20 characters wide with left justification.

```
WRITE TEXT X QSTR FW:20 LJ EOL
WRITE TEXT Y QSTR FW:20 LJ EOL
```

Figure E-7. Two `WRITE TEXT QUOTED STRING` Transactions

If `X` and `Y` in Figure E-7 have these values:

```
bird cat dog          the Text value of X
12345678901234567     the Real value of Y
```

then HP VEE writes this:

```
"bird cat dog"
"12345678901234567"
^                  ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the right of `dog"` and to the right of `7"` are spaces (ASCII 32 decimal).

If justification is changed to `RIGHT JUSTIFY`, then the transactions appear as shown in Figure E-8.

```
WRITE TEXT X QSTR FW:20 RJ EOL
WRITE TEXT Y QSTR FW:20 RJ EOL
```

Figure E-8. Two `WRITE TEXT QUOTED STRING` Transactions

If `X` and `Y` in Figure E-8 have these values:

```
bird cat dog          the Text value of X
12345678901234567     the Real value of Y
```

then HP VEE writes this:

```
       "bird cat dog"
 "12345678901234567"
^                  ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the left of `"bird` and to the left of `"1` are spaces (ASCII 32 decimal).

If the length of a string exceeds the specified field width, the entire string is output. The field width specification never truncates strings that are written; only `MAX NUM CHARS` can truncate characters.

The transactions in Figure E-9 that specifies that all characters are to be written within a field of four characters with left justification.

```
WRITE TEXT X QSTR FW:4 LJ
```

Figure E-9. A `WRITE TEXT QUOTED STRING` Transaction

If `X` in Figure E-9 has this value:

`bird cat dog` *the Text value of X, 12 characters*

then HP VEE writes this:

`"bird cat dog"` *all 12 characters*

Number of Characters. If you specify `ALL CHARS`, then all of the characters generated by the conversion of each item in *ExpressionList* as well as the surrounding double quotes are written. If you specify `MAX NUM CHARS:` *M*, then only the first *M* characters of each item in *ExpressionList plus* the surrounding double quotes are written. In other words, a total of *M*+2 characters are written for each item in *ExpressionList*.

The transaction in Figure E-10 that specifies `MAX NUM CHARS:7` (field width 20, left justified).

```
WRITE TEXT X QSTR:7 FW:20 LJ EOL
WRITE TEXT Y QSTR:7 FW:20 LJ EOL
```

Figure E-10. Two `WRITE TEXT QUOTED STRING` Transactions

If `X` and `Y` in Figure E-10 have these values:

```
bird cat dog          the Text value of X
12345678901234567     the Real value of Y
```

then HP VEE writes this:

```
"bird ca"
"1234567"
^                   ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the right of `ca"` and to the right of `7"` are spaces (ASCII 32 decimal).

Embedded Control and Escape Characters. In this discussion, the terms **control character** and **escape character** have specific meaning. A control character is a single byte of data corresponding to one of the ASCII characters 0-31 decimal. For example, linefeed is ASCII 10 decimal and the symbol **<LF>** denotes linefeed character in this discussion. The string **\n** is a human-readable escape character representing linefeed that is recognized by HP VEE. HP VEE uses escape characters to represent control characters within quoted strings.

Table E-5. Escape Characters

Escape Character	ASCII Code (decimal)	Meaning
`\n`	10	Newline
`\t`	9	Horizontal Tab
`\v`	11	Vertical Tab
`\b`	8	Backspace
`\r`	13	Carriage Return
`\f`	12	Form Feed
`\"`	34	Double Quote
`\'`	39	Single Quote
`\\`	92	Backslash
`\`*ddd*		The ASCII character corresponding to the three-digit octal value *ddd*.

Consider the effects of various embedded escape characters on the transaction in Figure E-11.

```
WRITE TEXT X QSTR EOL
```

Figure E-11. A `WRITE TEXT QUOTED STRING` Transaction

If `X` in Figure E-11 has this value:

```
bird\ncat dog
```

then HP VEE writes this to UNIX paths:

```
"bird\ncat dog"
```

For the same transaction and data, HP VEE-Test writes this to direct I/O paths:

```
"bird<LF>cat dog"
```

Note that `<LF>` means the single character, linefeed (ASCII 10 decimal).

If `X` in Figure E-11 has this value:

```
bird \"cat\" dog
```

then HP VEE writes this to UNIX paths and Direct I/O paths for serial interfaces:

```
"bird \"cat\" dog"
```

For the same transaction and data, HP VEE-Test writes this to direct I/O paths for HP-IB interfaces:

```
"bird ""cat"" dog"
```

This unique behavior for HP-IB interfaces is provided to support the requirements of IEEE 488.2.

INTEGER Format

`WRITE TEXT INTEGER` transactions are of this form:

`WRITE TEXT` *ExpressionList* `INT`

ExpressionList is a single expression or a comma-separated list of expressions.

The type of integer generated by this transaction is a 32-bit two's complement integer. The range of these integers is `2 147 483 647` to `-2 147 483 648`. The only characters written to represent these numbers are `+-0123456789`.

HP VEE attempts to convert each item in *ExpressionList* to the Int32 data type before converting it to Text for final formatting. HP VEE follows the usual conversion rules; refer to the `Data Type Conversions` appendix in the *HP VEE Reference* manual for more details.

If a Real is written using `INTEGER` format:

- Real values outside the valid range of Int32 generate an error.
- Real values within the valid range of Int32 are converted by truncating the fractional portion of the Real.

Number of Digits. If you specify `DEFAULT NUM DIGITS`, the transaction writes only the digits required to express the value of the integer; leading zeros are not used.

If you specify `MIN NUM DIGITS:` *M*, the transaction pads the output with leading zeros as required to give a total of exactly *M* digits.

Consider the two transactions in Figure E-12 which differ only in their specification for the number of output digits.

```
WRITE TEXT X INT EOL      default number of digits
WRITE TEXT X INT:6 EOL    six digits
```

Figure E-12. Two `WRITE TEXT INTEGER` Transactions

If `X` in Figure E-12 has this value:

```
4567
```

then HP VEE writes this:

```
4567
004567
```

`MIN NUM DIGITS` never causes truncation of the output string. The transaction in Figure E-13 specifies the minimum number of digits to be 1.

```
WRITE TEXT X INT:1 EOL
```

Figure E-13. A `WRITE TEXT INTEGER` Transaction

If `X` in Figure E-13 has a value of:

```
12345678
```

then HP VEE writes this:

```
12345678    all eight digits
```

Sign Prefixes. You may optionally specify one of the sign prefixes listed in Table E-6 as part of a `WRITE TEXT INT` transaction.

Table E-6. Sign Prefixes

Prefix	Description
`/-`	Positive numbers are written with no prefix, neither a `+` nor a space. All negative numbers are written with a `-` prefix.
`+/-`	All positive numbers are written with a `+` prefix. All negative numbers are written with a `-` prefix.
`" "/-`	All positive numbers are written with a space (ASCII 32 decimal) prefix. All negative numbers are written with a `-` prefix.

Any prefixed signs do not count towards `MIN NUM DIGITS`. The transaction shown in Figure E-14 specifies explicit leading signs for positive and negative numbers.

```
WRITE TEXT X INT:6 SIGN:"+/-" EOL
WRITE TEXT Y INT:6 SIGN:"+/-" EOL
```

Figure E-14. Two `WRITE TEXT INTEGER` Transactions

If `X` and `Y` in Figure E-14 have values of:

```
123     the Integer value of X
-123    the Integer value of Y
```

then HP VEE writes this:

```
+000123     six digits plus sign
-000123
```

OCTAL Format

`WRITE TEXT OCTAL` transactions are of this form:

```
WRITE TEXT ExpressionList OCT
```

ExpressionList is a single expression or a comma-separated list of expressions.

The type of integer written by this transaction is a 32-bit two's complement integer. The range of these integers is `2 147 483 647` to `-2 147 483 648`. The only characters written to represent these octal numbers are `01234567`. An optional prefix may be specified which may include other characters.

HP VEE attempts to convert any data written using `OCTAL` format to the Int32 data type before converting it to Text for final formatting. The usual HP VEE conversion rules are followed.

If a Real is written using `OCTAL` format:

- Real values outside the valid range of Int32 generate an error.
- Real values within the valid range of Int32 are converted by truncating the fractional portion of the Real.

Number of Digits. The behavior of `DEFAULT NUM DIGITS` and `MIN NUM DIGITS` is the same as described previously in the "Number of Digits" section for `WRITE TEXT INTEGER` transactions.

Octal Prefixes. You may specify one of the prefixes listed in Table E-7 as part of a `WRITE TEXT OCTAL` transaction.

Table E-7. Octal Prefixes

Prefix	Description
`NO PREFIX`	HP VEE writes each octal number without any prefix; only the digits `01234567` appear in the output.
`DEFAULT PREFIX`	For direct I/O paths, HP VEE prefixes each octal number with `#Q`. This supports the octal Non-Decimal Numeric data format defined by IEEE 488.2. For UNIX paths, HP VEE prefixes each octal number with a `0` (zero). If leading zeros are added to achieve the specified `MIN NUM DIGITS`, `DEFAULT PREFIX` will not add additional leading zeros.
`PREFIX:`*string*	HP VEE prefixes each octal number with the characters specified in *string*.

The transaction in Figure E-15 specifies the default prefix and six digits:

```
WRITE TEXT X OCT:6 PREFIX EOL
```

Figure E-15. A `WRITE TEXT OCTAL` Transaction

If `X` in Figure E-15 has this value:

`15` *the value 15 decimal*

then HP VEE-Test writes this to direct I/O paths:

`#Q000017` *exactly six digits plus prefix*

Using the same transaction and data, HP VEE writes this to UNIX paths:

`000017` *exactly six digits*

The transaction in Figure E-16 specifies a custom prefix and ten digits:

```
WRITE TEXT X OCT:10 PREFIX:"oct>" EOL
```

Figure E-16. A `WRITE TEXT OCTAL` Transaction

If `X` in Figure E-16 has this value:

`15` *the Integer value 15 decimal*

then HP VEE writes this to UNIX paths and direct I/O paths:

```
oct>000017
```

Note that the prefix written by `DEFAULT PREFIX` depends on the destination, but the prefix written by `PREFIX:` *string* is independent of the destination.

HEX Format

`WRITE TEXT HEX` transactions are of this form:

`WRITE TEXT` *ExpressionList* `HEX`

The type of integer written by this transaction is a 32-bit two's complement integer. The range of these integers is `2 147 483 647` to `-2 147 483 648`. The only characters written to represent these hexadecimal numbers are `0123456789abcdef`. An optional prefix may be specified that may include other characters.

The behavior of `WRITE TEXT HEX` is nearly identical to that of `WRITE TEXT OCTAL`. The only difference is the set of prefixes available and the behavior of `DEFAULT PREFIX`.

Hexadecimal Prefixes. You may specify one of the prefixes listed in Table E-8 as part of a `WRITE TEXT HEX` transaction.

Table E-8. Hexadecimal Prefixes

Prefix	Description
`NO PREFIX`	HP VEE writes each hexadecimal number without any prefix; only the digits `0123456789abcdef` appear in the output.
`DEFAULT PREFIX`	For direct I/O paths, HP VEE prefixes each hexadecimal number with `#H`. This supports the hexadecimal Non-Decimal Numeric data format defined by IEEE 488.2. For UNIX paths, HP VEE prefixes each hexadecimal number with `0x`.
`PREFIX:` *string*	HP VEE prefixes each hexadecimal number with the characters specified in *string*.

The transaction in Figure E-17 specifies the default prefix and six digits:

```
WRITE TEXT X HEX:6 PREFIX EOL
```

Figure E-17. A `WRITE TEXT HEX` Transaction

If `X` in Figure E-15 has this value:

`15` *the Integer value 15 decimal*

then HP VEE-Test writes this to direct I/O paths:

`#H00000f` *exactly six digits plus prefix*

Using the same transaction and data, HP VEE this to UNIX paths:

`0x00000f` *exactly six digits plus prefix*

The transaction in Figure E-18 specifies a custom prefix and three digits:

```
WRITE TEXT X HEX:3 PREFIX:"hex>" EOL
```

Figure E-18. A `WRITE TEXT HEX` Transaction

If `X` in Figure E-18 has this value:

`15` *the Integer value 15 decimal*

then HP VEE writes this to UNIX paths and direct I/O paths:

`hex>00f` *exactly three digits plus prefix*

Note that the prefix written by `DEFAULT PREFIX` depends on the destination, but the prefix written by `PREFIX:` *string* is independent of the destination.

REAL Format

`WRITE TEXT REAL` transactions are of this form:

```
WRITE TEXT ExpressionList REAL
```

The type of Real number generated by this transaction is a 64-bit IEEE 754 floating-point number. The range of these numbers is:

```
-1.797 693 134 862 315E+308
-2.225 073 858 507 202E-307
 0
 2.225 073 858 507 202E-307
 1.797 693 134 862 315E+308
```

The only characters written to represent these numbers are `+-.0123456789E`.

Notations and Digits. You may optionally specify one of the notations in Table E-9 as part of a `WRITE TEXT REAL` transaction.

Table E-9. `REAL` Notations

Notation	Description
`STANDARD`	HP VEE automatically determines whether each Real value should be written in fixed-point notation (decimal points as required, no exponents) or in exponential notation. Non-significant zeros are never written.
`FIXED`	HP VEE writes each Real value as a fixed-point number. Numbers with fractional digits are automatically rounded to fit the number of fractional digits specified by `NUM FRACT DIGITS`. Trailing zero digits are added as required to give the specified number of fractional digits.
`SCIENTIFIC`	HP VEE writes each Real value using exponential notation. Each exponent includes an explicit sign (`+` or `−`) and the upper-case `E` is always used. Numbers with fractional digits are automatically rounded to fit the number of fractional digits specified by `NUM FRACT DIGITS`. Trailing zero digits are added as required to give the specified number of fractional digits.

The transactions in Figure E-19 specify `STANDARD` notation and four significant digits.

```
WRITE TEXT X REAL STD:4 EOL
WRITE TEXT Y REAL STD:4 EOL
WRITE TEXT Z REAL STD:4 EOL
```

Figure E-19. Three `WRITE TEXT REAL` Transactions

If `X`, `Y`, and `Z` in Figure E-19 have these values:

`1.23456E2`	*the Real value of X*
`1.23456E09`	*the Real value of Y*
`1.23`	*the Real value of Z*

then HP VEE writes this:

`123.5`	*mantissa rounded as required*
`1.235E+09`	*large numbers in exponential notation*
`1.23`	*never any trailing zeros*

The transactions in Figure E-20 specify `FIXED` notation and four fractional digits.

```
WRITE TEXT X REAL FIX:4 EOL
WRITE TEXT Y REAL FIX:4 EOL
WRITE TEXT Z REAL FIX:4 EOL
```

Figure E-20. Three `WRITE TEXT REAL` Transactions

If `X`, `Y`, and `Z` in Figure E-20 have these values:

`1.2345678E2`	*the Real value of X*
`1.2345678E-09`	*the Real value of Y*
`1.23`	*the Real value of Z*

then HP VEE writes this:

`123.4568`	*mantissa rounded as required*
`0.0000`	*small numbers round to zero*
`1.2300`	*trailing zeros added as required*

The transactions in Figure E-21 specify `SCIENTIFIC` notation and four fractional digits.

```
WRITE TEXT X REAL SCI:4 EOL
WRITE TEXT Y REAL SCI:4 EOL
WRITE TEXT Z REAL SCI:4 EOL
```

Figure E-21. Three `WRITE TEXT REAL` Transactions

If `X`, `Y`, and `Z` in Figure E-21 have these values:

`1.2345678E2`	*the Real value of X*
`-1.2345678E-09`	*the Real value of Y*
`0`	*the Real value of Z*

then HP VEE writes this:

`1.2346E+02`	*exponent is E plus two signed digits*
`-1.2346E-09`	*last digit rounded as required*
`0.0000E+00`	*trailing zeros padded as required*

COMPLEX, PCOMPLEX, and COORD Formats

`COMPLEX`, `PCOMPLEX`, and `COORD` correspond to the HP VEE multi-field data types with the same names. The behavior of all three formats is very similar. The behaviors described in this section apply to all three formats except as noted.

Just as the HP VEE data types Complex, PComplex, and Coord are composed of multiple Real numbers, the `COMPLEX`, `PCOMPLEX`, and `COORD` formats are essentially compound forms of the `REAL` format. Each constituent Real value of the multi-field data types is written with the same output rules that apply to an individual `REAL` formatted value.

The final output of transactions involving multi-field formats is affected by the `Multi-Field Format` setting for the object in question. `Multi-Field Format` is accessed via `I/O ⟹ Instruments ...` for `Direct I/O` objects and via `Config` in the object menu for all other objects. The two possible settings for `Multi-Field Format` are:

- `Data Only`. This writes multi-field data formats as a list of comma-separated numbers *without* parentheses.
- `( ... ) Syntax`. This writes multi-field data formats as a list of comma-separated numbers grouped by parentheses.

Subsequent examples will illustrate these behaviors.

COMPLEX Format. `WRITE TEXT COMPLEX` transactions are of this form:

`WRITE TEXT` *ExpressionList* `CPX`

The transaction in Figure E-22 specifies a fixed-decimal notation, explicit leading signs, a field width of 10 characters, and right justification.

```
WRITE TEXT X CPX FIX:3 SIGN:"+/-" FW:10 RJ EOL
```

Figure E-22. A `WRITE TEXT COMPLEX` Transaction

If the `Multi-Field Format` is set to `( ... ) Syntax`, and `X` in Figure E-22 has this value:

```
( -1.23456 , 9.8 )     the Complex value of X
```

then HP VEE writes this:

```
(    -1.235 ,     +9.800)
 ^         ^   ^         ^
```

If the `Multi-Field Format` is set to `Data Only` and `X` in Figure E-22 has the same value, then HP VEE writes this:

```
    -1.235,    +9.800
^         ^ ^         ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the left of **+** are spaces (ASCII 32 decimal).

Note that with `( . . . ) Syntax`, a space-comma-space sequence separates the ten-character wide fields that contain the real and imaginary parts of the Complex number. With either `Multi-Field Format` there is a separate ten-character field for both the real and the imaginary part. Neither parentheses nor the separating comma and spaces are included in the field.

PCOMPLEX Format.

`WRITE TEXT PCOMPLEX` transactions are of this form:

`WRITE TEXT` *ExpressionList* `PCX`

`PCOMPLEX` format allows you to specify the phase units for the polar complex number it writes. Note that phase units are independent of the units set by `File` ⟹ `Preferences` ⟹ `Trig Mode`.

Table E-10. `PCOMPLEX` Phase Units

Unit	Description
`DEG`	Degrees
`RAD`	Radians
`GRAD`	Gradians

The first transaction in Figure E-23 specifies phase measurement in degrees, and the second transaction specifies phase measurement in radians.

```
WRITE TEXT X PCX:DEG STD EOL
WRITE TEXT X PCX:RAD STD EOL
```

Figure E-23. Two `WRITE TEXT PCOMPLEX` Transactions

If the `Multi-Field Format` is set to `Data Only`, and `X` in Figure E-23 has this value:

```
(-1.23456 , @90)     the PComplex value of X, phase in degrees
```

then HP VEE writes this:

```
1.23456,-90
1.23456,-1.570796326794897
```

The transaction in Figure E-24 specifies phase measurement in radians, fixed-decimal notation, three fractional digits, explicit leading signs, a field width of ten characters, and right justification.

```
WRITE TEXT X PCX:RAD FIX:3 SIGN:"+/-" FW:10 RJ EOL
```

Figure E-24. A `WRITE TEXT PCOMPLEX` Transaction

If the `Multi-Field Format` is set to `( ... ) Syntax`, and `X` in Figure E-24 has this value:

```
(-1.23456 , @9.8)    the PComplex value of X,  angle in radians
```

then HP VEE writes this:

```
(    +1.235 , @    +0.375)
 ^         ^    ^         ^
```

Note that HP VEE normalizes all PComplex numbers to yield a positive magnitude and a phase between $+\pi$ and $-\pi$.

If the `Multi-Field Format` is set to `Data Only`, and `X` in Figure E-24 has the same value, then HP VEE writes this:

```
    +1.235,    +0.375
^         ^ ^         ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the left of `-` and to the left of `+` are spaces (ASCII 32 decimal).

COORD Format. `WRITE TEXT COORD` transactions are of this form:

```
WRITE TEXT ExpressionList COORD
```

`COORD` format has all the same behaviors of `COMPLEX` format. The only difference is that `COORD` may contain an arbitrary number of fields while `COMPLEX` has exactly two fields.

TIME STAMP Format

`WRITE TEXT TIME STAMP` transactions are of this form:

`WRITE TEXT` *ExpressionList* *[*`DATE:` *DateSpec]* *[*`TIME:` *TimeSpec]*

ExpressionList is a single expression or a comma-separated list of expressions.

DateSpec is one of the following pre-defined date and time combinations:

- `Date`
- `Time`
- `Date&Time`
- `Time&Date`
- `Delta Time`

If you specify a transaction that includes `Date`, you may also specify a *DateSpec* of `Weekday DD/Month/YYYY` or `DD/Month/YYYY`.

If you specify a transaction that includes `Time`, you may also specify a *TimeSpec*. *TimeSpec* is a combination of the following pre-defined time formats:

- `HH:MM` (hours and minutes)
- `HH:MM:SS` (hours, minutes, and seconds)
- `12 HOUR`
- `24 HOUR`

Each item in *ExpressionList* is converted to a Real and interpreted as a date and time. This Real number represents the number of seconds that have elapsed since midnight, January 1, AD 1 UTC. The most common source for this Real number is the output of a `Time Stamp` object. You use the `TIME STAMP` format to convert this Real number to a meaningful string that contains a human-readable date and/or time.

TIME STAMP supports a variety of notations for writing dates and times. If a Real variable contains this value:

```
62806574669.31164
```

then TIME STAMP can write it using any of these Time and Date notations:

Notation	Result
Date with Weekday DD/Month/YYYY	Thu 04/Apr/1991
Time with HH:MM:SS and 24 HOUR	15:44:29
Date&Time with Weekday DD/Month/YYYY, HH:MM:SS, and 24 HOUR	Thu 04/Apr/1991 15:44:29
Time&Date with HH:MM:SS, 24 HOUR, and Weekday DD/Month/YYYY	15:44:29 Thu 04/Apr/1991
Delta Time with HH:MM:SS	17446270:44:29
Date with Weekday DD/Month/YYYY	Thu 04/Apr/1991
Date with DD/Month/YYYY	04/Apr/1991
Time with HH:MM:SS and 24 HOUR	15:44:29
TIME with HH:MM and 24 HOUR	15:44
TIME with HH:MM:SS and 24 Hour	15:44:29
TIME with HH:MM:SS and 12 Hour	3:44:29 PM

BYTE Encoding

`BYTE` transactions are of this form:

```
WRITE BYTE ExpressionList
```

ExpressionList is a single expression or a comma-separated list of expressions.

HP VEE converts each item in *ExpressionList* to an Int16 (16-bit two's complement integer) and writes the least-significant 8-bits. This is a transaction for writing single characters to a device. Each expression in *ExpressionList* must be a scalar.

The transactions in Figure E-25 produce the output shown in Figure E-26.

```
WRITE BYTE 65,66,67
WRITE BYTE 65+1024,65+2048
```

Figure E-25. Two `WRITE BYTE` Transactions

```
ABCAA
```

Figure E-26. Character Data

CASE Encoding

`WRITE CASE` transactions are of this form:

`WRITE CASE` *ExpressionList1* `OF` *ExpressionList2*

ExpressionList is a single expression or a comma-separated list of expressions.

HP VEE converts each item in *ExpressionList1* to an integer and uses it as an index into *ExpressionList2*. The indexed item(s) in *ExpressionList2* are written in a string format that is the same as `WRITE TEXT` (default).

Note that the indexing of items in *ExpressionList2* is zero-based.

The transactions in Figure E-27 illustrate the behavior of `CASE` format.

```
WRITE CASE 2,1  OF "Str0","Str1","Str2"
WRITE CASE X  OF  1,1+A,3+A
```

Figure E-27. Two `WRITE CASE` Transactions

If the variables in Figure E-27 have these values:

`2` *the Real value of X*
`0.1` *the Real value of A*

then HP VEE writes this:

```
Str2Str1
3.1
```

BINARY Encoding

`WRITE BINARY` transactions are of this form:

`WRITE BINARY` *ExpressionList DataType*

ExpressionList is a single expression or a comma-separated list of expressions.

*DataType*s is one of the following pre-defined HP VEE data types:

- `BYTE` - 8-bit byte
- `INT16` - 16-bit two's complement integer
- `INT32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating-point number
- `REAL64` - 64-bit IEEE 754 floating-point number
- `STRING` - null terminated string
- `COMPLEX` - equivalent to two `REAL`s
- `PCOMPLEX` -equivalent to two `REAL`s
- `COORD` - equivalent to two or more `REAL`s

NOTE

HP VEE stores and manipulates all integer values as the `INT32` data type, and all real numbers as the `Real` data type, also known as `REAL64`. Thus, the `INT16` and `REAL32` data types are provided for I/O only. HP VEE-Test performs the following data-type conversions for instrument I/O:

- On an output transaction `INT32` values are individually converted to `INT16` values, which are output to the instrument. However, since the `INT16` data type has a range of -32768 to 32767, values outside this range will be truncated to 16 bits.
- On an output transaction `REAL64` values are individually converted to `REAL32` values, which are output to the instrument. However, since the `REAL32` data type has a smaller range than `REAL64` data type, values outside this range cannot be converted to `REAL32` and will result in an error.

`BINARY` encoded transactions convert each of the values specified in *ExpressionList* to the HP VEE data type specified by *DataType*. Each converted item is then written in the specified binary format. However, since the binary data written is a copy of the representation in computer memory, it is not easily shared by different computer architectures or hardware.

`BINARY` encoded data has the advantage of being very compact. `READ BINARY` transactions can read any corresponding `WRITE BINARY` data.

Note that `BINARY` encoding writes only the numeric portion of each data type. For example, the parentheses and comma that can be included when writing Complex and Coord data with `TEXT` encoding are never written with `BINARY` encoding. Similarly, when writing arrays, `BINARY` encoding does not write any `Array Separators`. `WRITE BINARY` transactions do allow you to specify `EOL ON`. There is rarely a need to write `EOL` with `BINARY` transactions because numeric data types are of fixed length and strings are null-terminated.

BINBLOCK Encoding

`WRITE BINBLOCK` transactions are of this form:

`WRITE BINBLOCK` *ExpressionList DataType*

ExpressionList is a single expression or a comma-separated list of expressions.

DataType is one of these pre-defined HP VEE data types:

- `BYTE` - 8-bit byte
- `INT16` - 16-bit two's complement integer
- `INT32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating-point number
- `REAL64` - 64-bit IEEE 754 floating-point number
- `COMPLEX` - equivalent to two `REAL`s
- `PCOMPLEX` -equivalent to two `REAL`s
- `COORD` - equivalent to two or more `REAL`s

`BINBLOCK` writes *each item* in *ExpressionList* as a separate data block. The block header used depends on the type of object performing the `WRITE` and the object's configuration.

Non-HP-IB BINBLOCK

If the object is *not* `Direct I/O` to HP-IB, a `WRITE BINBLOCK` always writes an IEEE 488.2 Definite Length Arbitrary Block Response Data block. This data format is primarily used for communicating with HP-IB instruments using `Direct I/O`, although it is supported by other objects.

Each Definite Length Arbitrary Block is of the form:

```
#<Num_digits><Num_bytes><Data>
```

where:

`#` is literally the `#` character as shown.

`<Num_digits>` is an ASCII character that is a single digit (decimal notation) indicating the number of digits in `<Num_bytes>`.

`<Num_bytes>` is a list of ASCII characters that are digits (decimal notation) indicating the number of bytes that follow in `<Data>`.

`<Data>` is a sequence of arbitrary 8-bit data bytes.

HP-IB BINBLOCK

If the object is `Direct I/O` to HP-IB, the behavior of `WRITE BINBLOCK` transactions depends upon the `Direct I/O Configuration` settings for `Conformance` and `Binblock`; these settings are accessed via the `I/O` ⟹ `Instruments` . . . menu selection.

If `Conformance` is set to `IEEE 488.2`, then `WRITE BINBLOCK` *always* writes an IEEE 488.2 Definite Length Arbitrary Block Response Data block.

If `Conformance` is set to `IEEE 488`, then the type of header used depends on `Binblock`. `Binblock` may specify IEEE 728 `#A`, `#T`, or `#I` block headers. If `Binblock` is `None`, `WRITE BINBLOCK` writes an IEEE 488.2 Definite Length Arbitrary Block Response Data block.

IEEE 728 block headers are of the following forms:

```
#A<Byte_Count><Data>
#T<Byte_Count><Data>
#I<Data><END>
```

where:

`#` is the character as shown.

`A`,`T`, `I` are the characters as shown.

`<Byte_Count>` consists of two bytes which together form a 16-bit unsigned integer that specifies the number of bytes that follow in `<Data>`. (HP VEE calculates this automatically.)

`<Data>` is a stream of arbitrary bytes.

`<END>` indicates that EOI is asserted with the last data byte transmitted.

CONTAINER Encoding

`WRITE CONTAINER` transactions are of this form:

```
WRITE CONTAINER ExpressionList
```

ExpressionList is a single expression or a comma-separated list of expressions.

A `WRITE CONTAINER` transaction writes each item in *ExpressionList* using a special HP VEE text representation.

This representation retains all the HP VEE attributes associated with the data type written, such as shape, size, and name. Any `WRITE CONTAINER` data can be retrieved without any loss of information using `READ CONTAINER`.

For example, this transaction:

```
WRITE CONTAINER 1.2345
```

writes this:

```
(Real
 (data 1.2345)
)
```

If `DEFAULT` formatting is specified, HP VEE writes a container of the same data type as each item in *ExpressionList*.

If a format is specified, HP VEE converts each item in *ExpressionList* to the specified format using the standard data type conversion rules. The converted data is written as a container of the specified type. Please refer to `Converting Data Types` in the on-line `Help` for details about automatic conversion of data types.

In general, it is best to use `DEFAULT` formatting. If a format is specified, conversion of incoming data may produce undesirable results.

STATE Encoding

`STATE` encoding is supported by HP VEE-Test only.

`WRITE STATE` transactions are of the form:

`WRITE STATE` *[DownloadString]*

DownloadString is an optional string that allows you to specify a download string if you have not previously specified one in the direct I/O configuration for the corresponding instrument. This explained in greater detail in the sections that follow.

`WRITE STATE` transactions are used by `Direct I/O` objects to download a learn string to an instrument. There is exactly one learn string associated with each instance of a `Direct I/O` object. This learn string is uploaded by clicking on `Upload` in the `Direct I/O` object menu. The learn string contains the null string before `Upload` is selected for the first time.

The behavior of `WRITE STATE` is affected by the `Direct I/O Configuration` settings for `Conformance` and `Download String`. These settings are accessed via the `I/O` ⟹ `Instruments` . . . menu selection. If `Conformance` is `IEEE 488`, the `WRITE STATE` transaction writes the `Download String` followed by the learn string. If `Conformance` is `IEEE 488.2`, the learn string is downloaded without any prefix as defined by IEEE 488.2. Please refer to "WRITE STATE Transactions" in Chapter 6 for a detailed example using learn strings.

REGISTER Encoding

`REGISTER` encoding is supported by HP VEE-Test only.

`WRITE REGISTER` is used to write values into a VXI device's A16 memory.

`WRITE REGISTER` transactions are of this form:

```
WRITE REG: SymbolicName ExpressionList INCR
-or-
WRITE REG: SymbolicName ExpressionList
```

where:

SymbolicName is a name defined during configuration of a VXI device. The name refers to a specific address within a device's register space. Specific data types for `WRITE REGISTER` transactions are:

- `BYTE` - 8 bit byte
- `WORD16` - 16-bit two's complement integer
- `WORD32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating point number

These data types are also specified during configuration of a VXI device and do not appear in the transaction.

ExpressionList is a single expression or a comma-separated list of expressions.

`INCR` specifies that array data is to be written incrementally starting at the register address specified by *SymbolicName*. The first element of the array is written at the starting address, the second at that address plus an offset equal to the length in bytes of the data type, and so forth until all array elements have been written. If `INCR` is not specified in the transaction, the entire array is written to the single location specified by *SymbolicName*.

MEMORY Encoding

`MEMORY` encoding is supported by HP VEE-Test only.

`WRITE MEMORY` is used to write values into a VXI device's A24 or A32 memory.

`WRITE MEMORY` transactions are of this form:

```
WRITE MEM: SymbolicName ExpressionList INCR
-or-
WRITE MEM: SymbolicName ExpressionList
```

where:

SymbolicName is a name defined during configuration of a VXI device. The name refers to a specific address within a device's extended memory. Specific data types for `WRITE MEMORY` transactions are:

- `BYTE` - 8 bit byte
- `WORD16` - 16-bit two's complement integer
- `WORD32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating point number

These data types are also specified during configuration of a VXI device and do not appear in the transaction.

ExpressionList is a single expression or a comma-separated list of expressions.

`INCR` specifies that array data is to be written incrementally starting at the memory location specified by *SymbolicName*. The first element of the array is written at that location, the second at that location plus an offset equal to the length in bytes of the data type, and so forth until all array elements have been written. If `INCR` is not specified in the transaction, the entire array is written to the single memory location specified by *SymbolicName*.

IOCONTROL Encoding

`IOCONTROL` encoding is supported by HP VEE-Test only.

`WRITE IOCONTROL` transactions are of this form:

```
WRITE IOCONTROL CTL ExpressionList
-or-
WRITE IOCONTROL PCTL ExpressionList
```

ExpressionList is a single expression or a comma-separated list of expressions.

`IOCONTROL` encoding is used only for `Direct I/O` to GPIO interfaces.

This transaction sets the control lines of a GPIO interface:

```
WRITE IOCONTROL CTL a
```

HP VEE-Test converts the value of `a` to an Integer. The least *X* significant bits of the Integer value are mapped to the control lines of the interface, where *X* is the number of control lines.

For example, the HP 98622A GPIO interface uses two control lines, `CTL0` and `CTL1`.

Value Written	CTL1	CTL0
0	0	0
1	0	1
2	1	0
3	1	1

In the preceding table, `1` indicates that a control line is asserted, a `0` indicates that it is cleared.

This transaction controls the computer-driven handshake line of a GPIO interface:

```
WRITE IOCONTROL PCTL a
```

If the value of `a` is non-zero, the PCTL line is set. If the value is zero, no action is taken. PCTL is cleared automatically by the interface when the peripheral meets the handshake requirements.

Table E-11. READ **Encodings and Formats**

Encodings	Formats
TEXT	CHAR TOKEN STRING QUOTED STRING INTEGER OCTAL HEX REAL COMPLEX PCOMPLEX COORD TIME STAMP
BINARY	STR BYTE INT16 INT32 REAL32 REAL64 COMPLEX PCOMPLEX COORD

Table E-11. READ Encodings and Formats (continued)

Encodings	Formats
BINBLOCK	BYTE INT16 INT32 REAL32 REAL64 COMPLEX PCOMPLEX COORD
CONTAINER	Not Applicable
IOSTATUS	Not Applicable
REGISTER[1]	BYTE WORD16 WORD32 REAL32
MEMORY[1]	BYTE WORD16 WORD32 REAL32

1 Direct I/O to VXI only (HP VEE-Test only).

TEXT Encoding

`READ TEXT` transactions are generally very easy to use. This is because they are able to read and discard what is irrelevant and selectively read what is important. This works well most of the time, but occasionally you must analyze very carefully what HP VEE considers to be irrelevant and what it considers to be important. This will rarely (if ever) be a problem if you are reading text files written by HP VEE, as long as you read them using the same format used to write them. Problems are most likely to occur when you are trying to import a file from another software application.

Table E-12 describes `READ TEXT` behavior in a general way only; be sure to read all the sections that follow to understand all the possible variations.

Table E-12. Formats for `READ TEXT` Transactions

Format	Description
`CHAR`	Reads *any* 8-bit character.
`TOKEN`	Reads a contiguous list of characters as a unit; this unit is called a token. Tokens are separated by specified delimiter characters (you specify the delimiters). For example, in normal written English, words are tokens and spaces are delimiters.
`STRING`	Reads a list of 8-bit characters as a unit. Most control characters are read and discarded. The end of the string is reached when the specified number of characters has been read, or when a newline character is encountered.
`QSTRING`	Reads a list of 8-bit characters that conform to the IEEE 488.2 arbitrary length string defined by a starting and ending double quote character (ASCII 34). Control characters are not discarded. Escaped characters are expanded to a corresponding control character. The end of the string is reached when the double quote character (ASCII 34) has been read.
`INTEGER`	Reads a list of characters and interprets them as a decimal or non-decimal representation of an integer. The only characters considered to be part of a decimal `INTEGER` are `0123456789-+`. HP VEE recognizes the prefix `0x` (hex) and all the Non-Decimal Numeric formats specified by IEEE 488.2: `#H` (hex), `#Q` (octal), `#B` (binary).
`OCTAL`	Reads a list of characters and interprets them as the octal representation of an integer. The characters considered to be part of an `OCTAL` are `01234567`. HP VEE also recognizes the IEEE 488.2 Non-Decimal Numeric prefix `#Q` for octal numbers.

Table E-12. Formats for READ TEXT Transactions (continued)

Format	Description
HEX	Reads a list of characters and interprets them as the hexadecimal representation of an integer. The only characters considered to be part of a **HEX** are 0123456789abcdefABCDEF. The character combination **0x** is the default prefix; it is not part of the number and is read and ignored. HP VEE also recognizes **0x** and the IEEE 488.2 Non-Decimal Numeric prefix **#H** for hexadecimal numbers.
REAL	Reads a list of characters and interprets them as the decimal representation of a Real (floating-point) number. All common notations are recognized including leading signs, signed exponents, and decimal points. The characters recognized to be part of a **REAL** are 0123456789-+.Ee. HP VEE also recognizes certain characters as suffix multipliers for Real numbers (refer to Table E-13).
COMPLEX	Reads the equivalent of two **REAL**s and interprets them as a complex number. The first number read is the real part and the second number read is the imaginary part.
PCOMPLEX	Reads the equivalent of two **REAL**s and interprets them as a complex number in polar form. Some engineering disciplines refer to this as "phasor notation". The first number read is considered to be the magnitude and the second is the angle. You may specify units of measure for phase in the transaction.
COORD	Reads the equivalent of two or more **REAL**s and interprets them as rectangular coordinates.
TIME STAMP	Reads one of the specified HP VEE time stamp formats which represent the calendar date and/or time of day.

General Notes for READ TEXT

Read to End. The READ TEXT formats support a choice between reading a specified number of elements or reading until EOF is encountered. In a transaction, *NumElements* is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the first expression is an asterisk (*), the transaction will read data until an EOF is encountered. Read to end is supported only for **From File**, **From String**, **From StdIn**, **Execute Program**, **To/From Named Pipe**, and **To/From HP BASIC/UX** transactions.

Only the first dimension can have an asterisk rather than a number.

For example, the following transaction, reading from a file:

```
READ TEXT a REAL ARRAY:*,10
```

will read until EOF is encountered resulting in a two dimensional array with ten columns. The number of rows is dependent on the amount of data in the file. The total number of data elements read must be evenly divisible by the product of the known dimension sizes, in this example: 10. If this criteria is not met, an error will occur.

Number of Characters Per READ.

These `READ TEXT` formats support a choice between `DEFAULT NUM CHARS` and `MAX NUM CHARS`:

```
STRING
INTEGER
OCTAL
HEX
REAL
```

This section discusses the effects of `DEFAULT NUM CHARS` and `MAX NUM CHARS` on these formats.

The basic difference between `DEFAULT NUM CHARS` and `MAX NUM CHARS` is this:

- `DEFAULT NUM CHARS` causes HP VEE to read and ignore most characters that do not appear to be part of the number or string it expects.
- `MAX NUM CHARS` allows you to read *up to* the specified number of 8-bit characters in an attempt to build the type of number or string specified. HP VEE stops reading characters as soon as the `READ` is satisfied. All characters are read and HP VEE attempts to convert them to the data type specified in the transaction.

If you specify `DEFAULT NUM CHARS`, the transaction reads as many characters as it requires to fill each variable. Characters that are not meaningful to the specified data type are read and ignored.

If you specify `MAX NUM CHARS`, HP VEE makes no attempt to sort out characters that are not meaningful to the data type specified. If non-meaningful characters are encountered, they are read and may later generate an error.

In either case, newline and end-of-file are recognized as terminators for strings or numbers. For numeric formats, white space encountered before any significant characters (digits) is read and ignored; after reading significant characters, white space or other non-numeric characters terminate the current `READ`. These are the general behaviors; read the examples that follow for additional detail.

Consider this example that distinguishes between the behaviors of

`DEFAULT NUM CHARS` and `MAX NUM CHARS` using `INTEGER` format. Assume that you are trying to read a file containing this data:

```
bird dog cat 12345 horse
```

It is impossible to extract the integer `12345` from this data with a `READ TEXT INTEGER` transaction using `MAX NUM CHARS` no matter how many characters are read. This is because the characters `bird dog cat` are always read before the digits, they cannot be converted to an Integer, and this generates an error.

`DEFAULT NUM CHARS` will extract the integer `12345` by reading and ignoring `bird dog cat` and treating the white space following `5` as a delimiter.

Effects of Quoted Strings. The presence of quoted strings affects the behavior of `READ TEXT QSTR` and `READ TEXT TOKEN` for all I/O paths and `READ TEXT STRING` for instrument or interface I/O. In this discussion, a quoted string means a set of characters beginning and ending with a double quote character and no embedded (non-escaped) double quote characters. The double quote character is ASCII 34 decimal. The presence of double quotes affects the way that these `READ` transactions group characters into strings and tokens, and how embedded control and escape characters are handled.

In this discussion, the terms **control character** and **escape character** have specific meaning. A control character is a single byte of data corresponding to one of the ASCII characters 0-31 decimal. For example, linefeed is ASCII 10 decimal and the symbol `<LF>` denotes linefeed character in this discussion. The string `\n` is a human-readable escape character representing linefeed that is recognized by HP VEE.

The behavior of certain transactions when dealing with quoted strings is dependent on the particular I/O path. For all I/O paths except instrument I/O, `READ TEXT QSTR` treats quoted strings specially. For all I/O paths except instrument I/O, `READ TEXT STRING` does not recognize quoted strings. For instrument I/O there is no `READ TEXT QSTR` transaction. Instead, `READ TEXT STRING` recognizes quoted stings and deals with them accordingly. This is done since quoted strings have special meaning in the IEEE 488.2 specification. For all I/O paths including instruments, `READ TEXT TOKEN` treats quoted strings specially. In the following discussions, we will assume the I/O path to be file I/O.

When a string does not begin and end with double quotes, control characters other than linefeed are read and discarded by `READ TEXT STRING` transactions and by `READ TEXT TOKEN` transactions that specify `SPACE DELIM`. In both `STRING` and `TOKEN` transactions, linefeed terminates the `READ`. Escape character sequences, such as `\n` (newline) are simply read as the two characters `\` and `n`.

Within double quoted strings, `READ TEXT QSTR` and `READ TEXT TOKEN` will read all enclosed characters (including control characters) store them in the input variable. Embedded linefeeds are read and treated like any other character; they do not terminate the current `READ`. Escape character sequences are read and translated to their single-character counterpart.

Grouping effects are best explained by using an example. For the discussion in the rest of this section, the data being read is a file with the contents shown in Figure E-28.

```
"This is in quotes."  This is not.
```

Figure E-28. Quoted and Non-Quoted Data

Assume that you read the file shown in Figure E-28 using `From File` with these transactions:

```
READ TEXT x QSTR
READ TEXT y QSTR
```

After reading the file, the results are:

```
x = This is in quotes.
y = This is not.
```

Note that the double quotes are interpreted as delimiters and do not appear in the input variable.

Now assume that you read the file shown in Figure E-28 using `From File` with these transactions:

```
READ TEXT x QSTR MAXFW:4
READ TEXT y QSTR
```

After reading the file, the results are:

```
x = This
y = This is not.
```

Here the double quotes are still acting a delimiters; the first transaction reads from double quote to double quote and assigns the first four characters to `x`. This leaves the file's read pointer positioned before the second occurrence of `This`. The second transaction reads the same string as before.

Next, assume that you read the file shown in Figure E-28 using `From File` with these transactions:

```
READ TEXT x TOKEN
READ TEXT y QSTR
```

Now after reading the file, the results are:

```
x = This is in quotes.
y = This is not.
```

Here the double quotes effectively make the entire first sentence into a single token. Even though default `TOKEN` delimiter is white space, the entire quoted string is treated as a single token. In addition, `TOKEN` reads and discards the double quote characters.

CHAR Format

`READ TEXT CHAR` transactions are of this form:

`READ TEXT` *VarList* `CHAR:`*NumChar* `ARRAY:`*NumStr*

VarList is a single Text variable or a comma-separated list of Text variables.

NumChar specifies the number of 8-bit characters that must read to fill each element of each variable in *VarList*.

NumStr is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

`CHAR` format is useful when you wish to simply read one character at a time, or when you need to read *every* character without ignoring any incoming data.

This transaction reads two two-dimensional Text arrays; each element in each array contains two characters.

```
READ TEXT X,Y CHAR:2 ARRAY:2,2
```

If a file read by the previous transaction contains these characters:

```
<space>ABCDEFG"AB"<LF>'CD
```

then the variables `X` and `Y` contain these values after the `READ`:

```
X [0 0] = <space>A
X [0 1] = BC
X [1 0] = DE
X [1 1] = FG

Y [0 0] = "A
Y [0 1] = B"
Y [1 0] = <LF>'
Y [1 1] = CD
```

The symbol `<space>` means the single character, space (ASCII 32 decimal). The symbol `<LF>` means the single character, linefeed (ASCII 10 decimal). Note that space, linefeed, and double quotes are read without any special consideration or interpretation.

TOKEN Format

`READ TEXT TOKEN` transactions are of this form:

`READ TEXT` *VarList* `TOKEN` *Delimiter*
`ARRAY:`*NumElements*

VarList is a single Text variable or a comma-separated list of Text variables.

Delimiter specifies the combinations of characters that terminate (delimit) each token.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

TOKEN format allows you to define the delimiter (boundary) for tokens using one of these choices for *Delimiter*:

- SPACE DELIM
- INCLUDE CHARS
- EXCLUDE CHARS

The following discussion of delimiters explains how the choice of delimiters affects reading a file with these contents:

```
A phrase.
"A phrase."
Tab follows     .
XOXXOOXXXOOOXXXX
XAXXBCXXXDEF
```

Figure E-29. Data for READ TOKEN

The file contains only the letter O, not the digit zero.

Note that there is an invisible linefeed character at the end of each of the first four lines of the file in Figure E-29. The figure shows the file as it would appear in a text editor like vi.

SPACE DELIM. If you use SPACE DELIM, tokens are terminated by any white space. White space includes spaces, tabs, newline, and end-of-file. This corresponds roughly to words in written English. Using SPACE DELIM, you could read a file containing a paragraph of prose and separate out individual words.

Note that double quoted strings receive special treatment. Double quoted strings are read as a single token and the double quotes are stripped away. Control characters (ASCII 0-31 decimal) embedded in double-quoted strings are returned in the output variable. Escape characters (such as \n) embedded in double-quoted strings are converted into their equivalent control characters. This special treatment of double-quoted strings applies only to SPACE DELIM transactions; INCLUDE CHARS and EXCLUDE CHARS treat double quotes, escapes, and control characters the same as any other character.

If you read the data shown in Figure E-29 using `SPACE DELIM` with this transaction:

```
READ TEXT a TOKEN ARRAY:8
```

then the variable `a` contains these values:

```
a[0] = A
a[1] = phrase.
a[2] = A phrase.
a[3] = Tab
a[4] = follows
a[5] = .
a[6] = XOXXOOXXXOOOXXXX
a[7] = XAXXBCXXXDEF
```

INCLUDE CHARS. If you use `INCLUDE CHARS`, you can specify a list of characters to be "included" in tokens returned by the `READ`. These specified characters will be the *only* characters returned in any token. Any character other than the specified `INCLUDE` characters terminates the current token. The terminating characters *are not* included in the token and are stripped away.

If HP VEE reads the data shown in Figure E-29 using `INCLUDE CHARS` with this transaction:

```
READ TEXT a TOKEN INCLUDE:"X" ARRAY:7
```

then the variable `a` contains these values:

```
a[0] = X
a[1] = XX
a[2] = XXX
a[3] = XXXX
a[4] = X
a[5] = XX
a[6] = XXX
```

If HP VEE reads the data shown in Figure E-29 using `INCLUDE CHARS` with this transaction:

```
READ TEXT a TOKEN INCLUDE:"OXZ" ARRAY:4
```

then the variable `a` contains these values:

```
a[0] = XOXXOOXXXOOOXXXX
a[1] = X
a[2] = XX
a[3] = XXX
```

Note that the first character in the `INCLUDE` list is the letter `O`, not the digit zero.

Assume that you are trying to read a file containing the data in Figure E-30.

```
111 222 333 444 555
```

Figure E-30. Data for `READ TOKEN`

If you try to read the file in Figure E-30 using this transaction:

```
READ TEXT x,y,z TOKEN INCLUDE:"1234567890"
```

then the Text variables `x`, `y`, and `z` will contain these values:

```
x = 111
y = 222
z = 333
```

Another way to do this is to specify an **ARRAY** greater than one and read data into an array. For example, if you read the data in Figure E-30 using this transaction:

```
READ TEXT x TOKEN INCLUDE:"1234567890" ARRAY:3
```

then the Text variable **x** contains these values:

```
x[0] = 111
x[1] = 222
x[2] = 333
```

EXCLUDE CHARS. If you use **EXCLUDE CHARS**, you can specify a list of characters, any one of which will terminate the current token. The terminating characters *are not* included in the token. they are read and discarded.

If you read the data shown in Figure E-29 using **EXCLUDE** with this transaction:

```
READ TEXT a TOKEN EXCLUDE:"X" ARRAY:8
```

then the variable **a** contains these values:

```
a[0] = A phrase.<LF>"A phrase."<LF>Tab follows     .<LF>
a[1] = 0
a[2] = 00
a[3] = 000
a[4] = <LF>
a[5] = A
a[6] = BC
a[7] = DEF
```

Assume the data shown in Figure E-31 is sent to HP VEE from an instrument.

```
++1.23++4.98++0.45++2.34++0.01++23.45++12.2++
```

Figure E-31. Data for `READ TOKEN`

If HP VEE reads the data in Figure E-31 with this transaction:

```
READ TEXT x TOKEN EXCLUDE:"+" ARRAY:7
```

then the variable **x** will contain these values:

```
x[0] = null string (empty)
x[1] = 1.23
x[2] = 4.98
x[3] = 0.45
x[4] = 2.34
x[5] = 0.01
x[6] = 23.45
```

Note that even though seven "numbers" were available, only six were read. At the end of this transaction, HP VEE has read seven tokens terminated by the **+**, including the first character which was terminated before it was filled with any data.

STRING Format

`READ TEXT STRING` transactions are of this form:

`READ TEXT` *VarList* **`STR ARRAY:`***NumElements*
`-or-`
`READ TEXT` *VarList* **`STR MAXFW:`***NumChars* **`ARRAY:`***NumElements*

VarList is a single Text variable or a comma-separated list of Text variables.

NumChars specifies the maximum number of 8-bit characters that can be read in an attempt to build a string.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the **`ARRAY`** keyword does not appear in the transaction. Note that **`ARRAY:1`** is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

This transaction reads all incoming characters and returns strings. The following discussion pertains to instrument I/O paths only, such as HP-IB or VXI. All other I/O paths, such as files or named-pipes, will not treat Quoted Strings specially. Please refer to the section "Effects of Quoted Strings" earlier in this chapter for details about the effects of double quoted strings on `READ TEXT STRING`.

Effects of Control and Escape Characters. In this discussion, the terms **control character** and **escape character** have specific meaning. A control character is a single byte of data corresponding to one of the ASCII characters 0-31 decimal. For example, linefeed is ASCII 10 decimal and the symbol `<LF>` denotes linefeed character in this discussion. The string `\n` is a human-readable escape character representing linefeed that is recognized by HP VEE. HP VEE uses escape characters to represent control characters within quoted strings.

Control characters and escape characters are handled differently depending on whether or not they appear within double quoted strings.

Outside double quoted strings, control characters other than linefeed are read and discarded. Linefeed terminates the current string. Escape characters, such as `\n`, are simply read as two individual characters (`\` and `n`).

Within double quoted strings, control characters and escape characters are read and included in the string returned by the `READ`. A linefeed within a double quoted string does *not* terminate the current string. Escape characters, such as `\n`, are interpreted as their single character equivalent (`<LF>`) and are included in the returned string as a control character.

Assume you wish to read the data in Figure E-32 using `READ TEXT STRING` transactions.

```
Simple string.
Random \n % $ * 'A'
"In quotes."
"In quotes
with control."
"In quotes\nwith escape."
```

Figure E-32. String Data

If you read the data in Figure E-32 using this transaction:

```
READ TEXT x STR ARRAY:5
```

then the variable x contains these values:

```
a[0] = Simple string.
a[1] = Random \n % $ * 'A'
a[2] = In quotes.
a[3] = In quotes<LF>with control.
a[4] = In quotes<LF>with escape.
```

If you read the same data in Figure E-32 using this transaction:

```
READ TEXT x STR MAXFW:16 ARRAY:5
```

then the variable x contains these values:

```
a[0] = Simple string.
a[1] = Random \n % $ *
a[2] = 'A'
a[3] = In quotes.
a[4] = In quotes<LF>with c
```

Note that the transaction terminates the current `READ` whenever 16 characters have been read (`a[1]`) or when a non-quoted `<LF>` (`a[2]`) is read. Double quoted strings are read from double quote to double quote and the first 16 delimited characters are returned (`a[4]`).

QUOTED STRING Format

`READ TEXT QUOTED STRING` transactions are of this form:

`READ TEXT` *VarList* `QSTR ARRAY:`*NumElements*
`-or-`
`READ TEXT` *VarList* `QSTR MAXFW:`*NumChars* `ARRAY:`*NumElements*

VarList is a single Text variable or a comma-separated list of Text variables.

NumChars specifies the maximum number of 8-bit characters that can be read in an attempt to build a string.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

This transaction reads all incoming characters and returns strings. The following discussion pertains to all non-instrument I/O paths. Instrument I/O paths do not implement the `READ TEXT QSTR` transaction. Please refer to the section "Effects of Quoted Strings" earlier in this chapter for details about the effects of double quoted strings on `READ TEXT STRING`.

Effects of Control and Escape Characters. In this discussion, the terms **control character** and **escape character** have specific meaning. A control character is a single byte of data corresponding to one of the ASCII characters 0-31 decimal. For example, linefeed is ASCII 10 decimal and the symbol `<LF>` denotes linefeed character in this discussion. The string `\n` is a human-readable escape character representing linefeed that is recognized by HP VEE. HP VEE uses escape characters to represent control characters within quoted strings.

Control characters and escape characters are handled differently depending on whether or not they appear within double quoted strings.

Outside double quoted strings, control characters other than linefeed are read and discarded. Linefeed terminates the current string. Escape characters, such as `\n`, are simply read as two individual characters (`\` and `n`).

Within double quoted strings, control characters and escape characters are read and included in the string returned by the `READ`. A linefeed within a double quoted string does *not* terminate the current string. Escape characters, such as `\n`, are interpreted as their single character equivalent (`<LF>`) and are included in the returned string as a control character.

Assume you wish to read the data in Figure E-33 using `READ TEXT QUOTED STRING` transactions.

```
Simple string.
Random \n % $ * 'A'
"In quotes."
"In quotes
with control."
"In quotes\nwith escape."
```

Figure E-33. String Data

If you read the data in Figure E-33 using this transaction:

```
READ TEXT x QSTR ARRAY:5
```

then the variable x contains these values:

```
a[0] = Simple string.
a[1] = Random \n % $ * 'A'
a[2] = In quotes.
a[3] = In quotes<LF>with control.
a[4] = In quotes<LF>with escape.
```

If you read the same data in Figure E-33 using this transaction:

```
READ TEXT x QSTR MAXFW:16 ARRAY:5
```

then the variable x contains these values:

```
a[0] = Simple string.
a[1] = Random \n % $ *
a[2] = 'A'
a[3] = In quotes.
a[4] = In quotes<LF>with c
```

Note that the transaction terminates the current `READ` whenever 16 characters have been read (`a[1]`) or when a non-quoted `<LF>` (`a[2]`) is read. Double quoted strings are read from double quote to double quote and the first 16 delimited characters are returned (`a[4]`).

INTEGER Format

`READ TEXT INTEGER` transactions are of this form:

```
READ TEXT VarList INT ARRAY:NumElements
-or-
READ TEXT VarList INT MAXFW:NumChars
ARRAY:NumElements
```

VarList is a single Integer variable or a comma-separated list of Integer variables.

NumChars specifies the maximum number of 8-bit characters that can be read in an attempt to build a number.

NumStr is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

`READ TEXT INTEGER` transactions interpret incoming characters as 32-bit, two's complement integers. The valid range for these integers is 2 147 483 647 to -2 147 483 648. Any numbers outside this range wrap around so there is never an overflow condition. For example, 2 147 483 648 is interpreted as -2 147 483 648. As it starts to build a number, HP VEE discards any leading characters that are not recognized as part of a number. Once HP VEE starts building a number, any character that is not recognized as part of a number terminates the `READ` for that number. These are the only combinations of characters that are recognized as part of an `INTEGER`:

Notation	Characters Recognized
Decimal	Valid characters are `+-0123456789`. Leading zeros are *not* interpreted as an octal prefix as they are in HP VEE data entry fields.
HP VEE hexadecimal	HP VEE interprets `0x` as a prefix for a hexadecimal number. Valid characters following the prefix are `0123456789aAbBcCdDeEfF`.
IEEE 488.2 binary	HP VEE interprets `#b` or `#B` as a prefix for a binary number. Valid characters following the prefix are `0` and `1`.
IEEE 488.2 octal	HP VEE interprets `#q` or `#Q` as a prefix for an octal number. Valid characters following the prefix are `01234567`.
IEEE 488.2 hexadecimal	HP VEE interprets `#h` or `#H` as a prefix for a hexadecimal number. Valid characters following the prefix are `0123456789aAbBcCdDeEfF`.

All of the following notations are interpreted as the Integer value 15 decimal:

```
15
+15
015
0xF
0xf
#b1111
#Q17
#hF
```

OCTAL Format

`READ TEXT OCTAL` transactions are of this form:

```
READ TEXT VarList OCT ARRAY:NumElements
-or-
READ TEXT VarList OCT MAXFW:NumChars
ARRAY:NumElements
```

VarList is a single Integer variable or a comma-separated list of Integer variables.

NumChars specifies the number of 8-bit characters that can be read in an attempt to build a number.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

`READ TEXT OCTAL` transactions interpret incoming characters as octal digits representing 32-bit, two's complement integers. The valid range for these integers is 2 147 483 647 decimal to -2 147 483 648 decimal.

If the transaction specifies a `MAX NUM CHARS` (`MAXFW`), the octal number read may contain more than 32 bits of data. For example, assume HP VEE reads the data in Figure E-34 using this transaction:

```
READ TEXT x OCT MAXFW:21
```

```
377237456214567243777
```

Figure E-34. Octal Data

HP VEE reads all the digits in Figure E-34, but uses only the last 11 digits (`14567243777`) to build a number for the value of `x`. This is because each digit corresponds to 3 bits and the octal number must be stored in an HP VEE Integer, which contains 32 bits. Eleven octal digits yield 33 bits; the most significant bit is dropped to fit the value in an HP VEE Integer. There is no possibility of overflow.

If the transaction specifies `DEFAULT NUM CHARS`, it will continue to read characters until it builds enough numbers to fill each variable in *VarList*. Linefeed characters will not terminate number building early. For example, this transaction:

```
READ TEXT x OCT ARRAY:4
```

interprets each line of data in Figure E-35 as the same set of four octal numbers.

```
0345 067 003<LF>0377<LF>
345 67 3 377<EOF>
345,67,3,377,45,67<EOF>
```

Figure E-35. Octal Data

The symbol `<LF>` represents the single character linefeed (ASCII 10 decimal). The symbol `<EOF>` represents the end-of-file condition.

HEX Format

`READ TEXT HEX` transactions are of this form:

`READ TEXT` *VarList* `HEX ARRAY:`*NumElements*
`-or-`
`READ TEXT` *VarList* `HEX MAXFW:`*NumChars*
`ARRAY:`*NumElements*

VarList is a single Integer variable or a comma-separated list of Integer variables.

NumChars specifies the number of 8-bit characters that can be read in an attempt to build a number.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

`READ TEXT HEX` transactions interpret incoming characters as hexadecimal digits representing 32-bit, two's complement integers. The valid range for these integers is 2 147 483 647 decimal to -2 147 483 648 decimal.

If the transaction specifies a `MAX NUM CHARS` (`MAXFW`), the hexadecimal number read may contain more than 32 bits of data. For example, assume HP VEE reads the data in Figure E-36 using this transaction:

```
READ TEXT x HEX MAXFW:21
```

```
ad2469Ff725BCdef37964
```

Figure E-36. Hexadecimal Data

HP VEE reads all the digits in Figure E-36, but uses only the last 8 digits (`def37964`) to build a number for the value of `x`. This is because each digit corresponds to 4 bits and the hexadecimal number must be stored in an HP VEE Integer, which contains 32 bits. Eight hexadecimal digits yields exactly 32 bits. There is no possibility of overflow.

Assume HP VEE reads the same data in Figure E-36, but with a different `MAX NUM CHARS`, as in this transaction:

```
READ TEXT x HEX MAXFW:3 ARRAY:7
```

In this case, the transaction reads the same data and interprets it as seven Integers, each comprised of three hexadecimal digits.

If the transaction specifies `DEFAULT NUM CHARS`, it will continue to read characters until it builds enough numbers to fill each variable in *VarList*. Each number will read *exactly* 8 hexadecimal digits. Linefeed characters will not terminate number building early.

Assume HP VEE reads the same data in Figure E-36, but with `DEFAULT NUM CHARS`, as in this transaction:

```
READ TEXT x HEX ARRAY:2
```

In this case, the transaction reads the same data and interprets it as two Integers, each comprised of eight hexadecimal digits. The last five digits (`37946`) are not read.

REAL Format

`READ TEXT REAL` transactions are of this form:

```
READ TEXT VarList REAL ARRAY:NumElements
-or-
READ TEXT VarList REAL MAXFW:NumChars
ARRAY:NumElements
```

VarList is a single Real variable or a comma-separated list of Real variables.

NumChars specifies the maximum number of 8-bit characters that can be read in an attempt to build a number.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

The decimal number read by this transaction is interpreted as an HP VEE Real which is a 64-bit IEEE 754 floating-point number. The range of these numbers is:

```
-1.797 693 134 862 315E+308
-2.225 073 858 507 202E-307
 0
 2.225 073 858 507 202E-307
 1.797 693 134 862 315E+308
```

If the transaction specifies a `MAX NUM CHARS` (`MAXFW`), the Real number read may contain more than 17 digits of data. For example, assume HP VEE reads the data in Figure E-37 using this transaction:

```
READ TEXT x REAL MAXFW:19
```

```
1.234567890123456789
```

Figure E-37. Real Data

HP VEE reads all the digits in Figure E-37, but uses only the 17 most-significant digits of the mantissa to build a number for the value of `x`. This is because each Real contains a 54-bit mantissa, which is equivalent to more than 16 but less than 17 decimal digits. As a result, `x` has the value

`1.2345678901234567`. Text to Real conversions are not guaranteed to yield the same value to the least-significant digit. Comparisons of the two least-significant bits is unadvisable.

Assume HP VEE reads the same data in Figure E-37, but with a different `MAX NUM CHARS`, as in this transaction:

```
READ TEXT x REAL MAXFW:6 ARRAY:3
```

In this case, the transaction reads the same data and interprets it as 3 Real numbers, each comprised of six decimal characters. The last two characters are not read.

If the transaction specifies `DEFAULT NUM CHARS`, it will continue to read characters until it builds enough numbers to fill each variable in *VarList*. Each number will read at most 17 decimal digits. Linefeed characters, white space and other non-numeric characters will terminate number building before 17 digits have been read.

`READ TEXT REAL` transactions recognize most commonly used decimal notations for Real numbers including leading signs, decimal points, and signed exponents. The characters `+-.0123456789Ee` are recognized as valid parts of a Real number by *all* `READ TEXT REAL` transactions. If the transaction specifies `DEFAULT NUM CHARS`, the suffix characters shown in Table E-13 are also recognized. The suffix character must immediately follow the last digit of the number with no intervening white space.

Table E-13. Suffixes for REAL Numbers

Suffix	Multiplier
P	10^{15}
T	10^{12}
G	10^{9}
M	10^{6}
k or K	10^{3}
m	10^{-3}
u	10^{-6}
n	10^{-9}
p	10^{-12}
f	10^{-15}

The Text data in Figure E-38 represents six real numbers.

```
1001
+1001.
1001.0
1.001E3
+1.001E+03
1.001K
```

Figure E-38. Example of Real Notations

If HP VEE reads the data in Figure E-38 with this transaction:

```
READ TEXT x REAL ARRAY:6
```

then each element of the Real variable `x` contains the value `1001`.

If HP VEE reads the data in Figure E-38 with this transaction:

```
READ TEXT x REAL MAXFW:20 ARRAY:6
```

then the first five elements of the Real variable `x` contain the value `1001` and the sixth element contains the value `1.001`.

COMPLEX, PCOMPLEX, and COORD Formats

`COMPLEX`, `PCOMPLEX`, and `COORD` correspond to the HP VEE multi-field data types with the same names. The behavior of all three `READ` formats is very similar. The behaviors described in this section apply to all three formats except as noted.

Just as the HP VEE data types Complex, PComplex, and Coord are composed of multiple Real numbers, the `COMPLEX`, `PCOMPLEX`, and `COORD` formats are compound forms of the `REAL` format. Each constituent Real value of the mult-field data types is read using the same rules that apply to an individual `REAL` formatted value.

COMPLEX Format. `READ TEXT COMPLEX` transactions are of this form:

`READ TEXT` *VarList* `CPX ARRAY:`*NumElements*

Each `READ TEXT COMPLEX` transaction reads the equivalent of two `REAL` formatted numbers. The first number read is interpreted as the real part and the second number read is interpreted as the imaginary part.

PCOMPLEX Format. `READ TEXT PCOMPLEX` transactions are of this form:

`READ TEXT` *VarList* `PCX:`*PUnit* `ARRAY:`*NumElements*

PUnit specifies the units of angular measure in which the phase of the PComplex is measured.

Each `READ TEXT PCOMPLEX` transaction reads the equivalent of two `REAL` formatted numbers. The first number read is interpreted as the magnitude and the second number read is interpreted as the phase.

If any transaction reading `COMPLEX`, `PCOMPLEX`, or `COORD` formats encounters an opening parenthesis, it expects to find a closing parenthesis.

Assume you wish to read a file containing the data shown in Figure E-39.

```
(1.23 , 3.45  (6.78 , 9.01) (1.23 , 4.56)
```

Figure E-39. Data Containing Parentheses

If HP VEE reads the data in Figure E-39 with this transaction:

```
READ TEXT x,y CPX
```

then the variables `x` and `y` contain these Complex values:

```
x = (1.23 , 3.45)
y = (1.23 , 4.56)
```

Note that the transaction read past `6.78` and `9.01` to find the closing parenthesis. If parentheses had been omitted from the data entirely, `y` would have the value `(6.78 , 9.01)`.

COORD Format. `READ TEXT COORD` transactions are of this form:

`READ TEXT` *VarList* `COORD:`*NumFields* `ARRAY:`*NumElements*

VarList is a single Coord variable or a comma-separated list of Coord variables.

NumFields is a single variable or expression that specifies the number of rectangular dimensions in each Coord value. This value must be `2` or more for the `READ` to execute without error.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

BINARY Encoding

`READ BINARY` transactions are of this form:

`READ BINARY` *VarList DataType* `ARRAY:`*NumElements*

VarList is a single variable or a comma-separated list of variables.

DataType is one of the following pre-defined formats corresponding to the HP VEE data type with the same name:

- `BYTE` - 8-bit byte
- `INT16` - 16-bit two's complement integer
- `INT32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating-point number
- `REAL64` - 64-bit IEEE 754 floating-point number
- `STRING` - null terminated string
- `COMPLEX` - equivalent to two `REAL`s
- `PCOMPLEX` -equivalent to two `REAL`s
- `COORD` - equivalent to two or more `REAL`s

NOTE

HP VEE stores and manipulates all integer values as the `INT32` data type, and all real numbers as the `Real` data type, also known as `REAL64`. Thus, the `INT16` and `REAL32` data types are provided for I/O only. HP VEE-Test performs the following data-type conversions for instrument I/O:

- On an input transaction `INT16` values from an instrument are *individually* converted to `INT32` values by HP VEE. This conversion assumes that the `INT16` data was *signed* data. If you need the resulting `INT32` data in *unsigned* form, simply pass the data through a formula object with the formula

  ```
  BITAND(a, 0xFFFF)
  ```

- On an input transaction `REAL32` values from an instrument are *individually* converted to `REAL64` values by HP VEE.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the first expression is an asterisk (`*`), the transaction will read data until an EOF is encountered. Read to end is supported only for `From File`, `From String`, `From StdIn`, `Execute Program`, `To/From Named Pipe`, and `To/From HP BASIC/UX` transactions.

Only the first dimension can have an asterisk rather than a number. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

For example, the following transaction, reading from a file:

```
READ BINARY a REAL64 ARRAY:*,10
```

will read until EOF is encountered, resulting in a two dimensional array with 10 columns. The number of rows is dependent on the amount of data in the file. The total number of data elements read must be evenly divisible by the product of the known dimension sizes, in this example: 10.

`READ BINARY` transactions expect that incoming data is in *exactly* the same format that would be produced by an equivalent `WRITE BINARY` transaction.

`BINARY` encoded data has the advantage of being very compact, but it is not easily shared with non-HP VEE applications.

BINBLOCK Encoding

`READ BINBLOCK` transactions are of this form:

```
READ BINBLOCK VarList DataType
```

VarList is a single variable or a comma-separated list of variables.

DataType is one of these pre-defined HP VEE data types:

- `BYTE` - 8-bit byte
- `INT16` - 16-bit two's complement integer
- `INT32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating-point number
- `REAL64` - 64-bit IEEE 754 floating-point number
- `COMPLEX` - equivalent to two `REAL`s
- `PCOMPLEX` -equivalent to two `REAL`s
- `COORD` - equivalent to two or more `REAL`s

You do not need to specify any additional information about the format of incoming data; the block header contains sufficient information. `READ BINBLOCK` can read any of the block formats previously described previously with `WRITE BINBLOCK` transactions.

This transaction reads two traces from an oscilloscope that formats its traces as IEEE 488.2 Definite Length Arbitrary Block Response Data:

```
READ BINBLOCK a,b REAL
```

CONTAINER Encoding

`READ CONTAINER` transactions are of the form:

```
READ CONTAINER VarList
```

VarList is a single variable or a comma-separated list of variables.

`READ CONTAINER` transactions reads data stored in the special special text representation written by `WRITE CONTAINER` transactions. No additional specifications, such as format, need to be specified with `READ CONTAINER` since that information is part of the container.

REGISTER Encoding

`REGISTER` encoding is supported by HP VEE-Test only.

`READ REGISTER` is used to read values from a VXI device's A16 memory.

`READ REGISTER` transactions are of this form:

```
READ REG: SymbolicName ExpressionList INCR
ARRAY:NumElements
-or-
READ REG: SymbolicName ExpressionList
ARRAY:NumElements
```

where:

SymbolicName is a name defined during configuration of a VXI device. The name refers to a specific address within a device's register space. Specific data types for `READ REGISTER` transactions are:

- `BYTE` - 8 bit byte
- `WORD16` - 16-bit two's complement integer
- `WORD32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating point number

These data types are also specified during configuration of a VXI device and do not appear in the transaction.

ExpressionList is a single expression or a comma-separated list of expressions.

`INCR` specifies that array data is to be read from the register incrementally starting at the address specified by *SymbolicName*. The first element of the array is read from the starting address, the second from that address plus an offset equal to the length in bytes of the data type, and so forth until all array elements have been read. If `INCR` is not specified in the transaction, the entire array is read from the single location specified by *SymbolicName*.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element.

HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

MEMORY Encoding

`MEMORY` encoding is supported by HP VEE-Test only.

`READ MEMORY` is used to read values from a VXI device's A24 or A32 memory.

`READ MEMORY` transactions are of this form:

```
READ MEM: SymbolicName ExpressionList INCR
ARRAY:NumElements
-or-
READ MEM: SymbolicName ExpressionList
ARRAY:NumElements
```

where:

SymbolicName is a name defined during configuration of a VXI device. The name refers to a specific address within a device's extended memory. Specific data types for `READ MEMORY` transactions are:

- `BYTE` - 8 bit byte
- `WORD16` - 16-bit two's complement integer
- `WORD32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating point number

These data types are also specified during configuration of a VXI device and do not appear in the transaction.

ExpressionList is a single expression or a comma-separated list of expressions.

`INCR` specifies that array data is to be read from the memory location incrementally starting at the location specified by *SymbolicName*. The first element of the array is read from the starting location, the second from that location plus an offset equal to the length in bytes of the data type, and so forth until all array elements have been read. If `INCR` is not specified in the

transaction, the entire array is read from the single memory location specified by *SymbolicName*.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

IOSTATUS Encoding

`READ IOSTATUS` transactions are of this form:

```
READ IOSTATUS STS Bits VarList
-or-
READ IOSTATUS DATA READY VarList
```

VarList is a single Integer variable or a comma-separated list of Integer variables.

`READ IOSTATUS` transactions are used by `Direct I/O` for GPIO interfaces, `From StdIn`, `To/From Named Pipe`, and `To/From HP BASIC/UX`.

`READ IOSTATUS` transactions for GPIO reads the peripheral status bits available on the interface. The number of bits read is dependent on the model number of the interface. A single integer value is returned that is the weighted sum of all the status bits.

For example, the HP 98622A GPIO interface supports two peripheral status lines, STI0 and STI1. Table E-14 illustrates how to interpret the value of `x` in this transaction:

```
READ IOSTATUS STS Bits a
```

Table E-14. `IOSTATUS` Values

Value Read	STI1	STI0
0	0	0
1	0	1
2	1	0
3	1	1

`READ IOSTATUS` transactions read the instantaneous values of the status lines; the status line are not latched or buffered in any way.

`READ IOSTATUS` transactions for `To/From Named Pipe`, `To/From HP BASIC/UX` and `From StdIn` returns a Boolean YES (1) if there is data ready to read. If no data is present, a Boolean NO (0) is returned. The `READ IOSTATUS` transaction can be used to avoid a read that will block indefinitely.

EXECUTE Transactions

EXECUTE transactions send low-level commands to control the file, instrument, or interface associated with a particular object. **EXECUTE** is used to adjust file pointers, clear buffers, and provide low-level control of hardware interfaces. The various **EXECUTE** commands available are summarized in Table E-15.

Table E-15. Summary of EXECUTE Commands

Commands	Description
To File, From File	
REWIND	Sets the read pointer (From File) or write pointer (To File) to the beginning of the file without changing the data in the file.
CLEAR	(**To File** only). Erases existing data in the file and sets the write pointer to the beginning of the file.
CLOSE	Explicitly closes the file. Useful when multiple processes are reading and writing the same file.
DELETE	Explicitly deletes the file. Useful for deleting temporary files.
Interface Operations (HP VEE-Test and HP VEE for Windows only)	
CLEAR	For **HP-IB** clears all devices by sending DCL (Device Clear). For **VXI**, resets the interface and runs the resource manager
TRIGGER	For **HP-IB** triggers all devices addressed to listen by sending GET (Group Execute Trigger). For **VXI** triggers specified backplane trigger lines or external triggers on an embedded controller.
LOCAL	For **HP-IB** releases the REN (Remote Enable) line.
REMOTE	For **HP-IB** asserts the REN (Remote Enable) line.
LOCAL LOCKOUT	For **HP-IB** sends the LLO (Local Lockout) message. Any device in remote at the time LLO is sent will lock out front panel operation.
ABORT	Clears the HP-IB interface by asserting the IFC (Interface Clear) line.

Table E-15. Summary of EXECUTE Commands (continued)

Commands	Description
Direct I/O to HP-IB (HP VEE-Test and HP VEE for Windows only)	
CLEAR	Clears the HP-IB interface associated with the **Direct I/O** object by asserting the IFC (Interface Clear) line.
TRIGGER	Triggers the device at the address of a **Direct I/O** object by addressing it to listen and sending GET (Group Execute Trigger).
LOCAL	Places the device at the address of the **Direct I/O** object in the local state.
REMOTE	Places the device at the address of the **Direct I/O** object in the remote state.
RESET	Resets the HP-IB interface associated with the **Direct I/O** object. For HP series 300 computers, **RESET** conducts the following sequence: 1. Clears the REN (Remote Enable) line. 2. Pulses the IFC (Interace Clear) line. 3. Asserts the REN (Remote Enable) line.
Direct I/O to GPIO (HP VEE-Test only)	
RESET	Resets the GPIO interface associated with the **Direct I/O** object by pulsing the PRESET line (Peripheral Reset).
Direct I/O to message based VXI (HP VEE-Test only)	
CLEAR	Clears the VXI device associated with the **Direct I/O** object by sending the word-serial command Clear (0xffff).
TRIGGER	Triggers the VXI device associated with the **Direct I/O** object by sending the word-serial command Trigger (0xedff).
LOCAL	Places the VXI device associated with the **Direct I/O** object into local state by sending the word-serial command Clear Lock (0xefff).
REMOTE	Places the VXI device associated with the **Direct I/O** object into local state by sending the word-serial command Set Lock (0xeeff). in the remote state.

Table E-15. Summary of `EXECUTE` Commands (continued)

Commands	Description
Direct I/O to Serial Interfaces (HP VEE-Test and HP VEE for Windows only)	
`RESET`	Resets the serial interface associated with the `Direct I/O` object.
`BREAK`	Transmits a signal on the Data Out line of the serial interface associated with the `Direct I/O` object as follows: 1. A logical High for 400 milliseconds 2. A logical Low for 60 milliseconds
Execute Program, To/From Named Pipe, To/From HP BASIC/UX	
`CLOSE READ PIPE`	Closes the read named pipe associated with the (To/From) object or the stdin pipe associated with the (Execute Program).
`CLOSE WRITE PIPE`	Closes the write named pipe associated with the (To/From) object or the stdout pipe associated with the (Execute Program).

Details About HP-IB

The `EXECUTE` commands used by `Direct I/O` to HP-IB devices and `Interface Operations` are similar but different.

- `Direct I/O EXECUTE` commands address an instrument to receive the command.
- `Interface Operations EXECUTE` commands may affect multiple instruments already addressed to listen.

The following series of tables indicate the exact bus actions conducted by `Direct I/O` and `Interface Operations EXECUTE` transactions.

Table E-16. `EXECUTE ABORT` HP-IB Actions

`Direct I/O`	`Interface Operations`
Not applicable.	IFC ($\geq$ 100 μsec) REN $\overline{\text{ATN}}$

Table E-17. `EXECUTE CLEAR` HP-IB Actions

`Direct I/O`	`Interface Operations`
ATN	ATN
MTA	DCL
UNL	
LAG	
SDC	

Table E-18. `EXECUTE TRIGGER` HP-IB Actions

`Direct I/O`	`Interface Operations`
ATN	ATN
MTA	GET
UNL	
LAG	
GET	

Table E-19. EXECUTE LOCAL **HP-IB Actions**

Direct I/O	Interface Operations
ATN	$\overline{\text{REN}}$
MTA	$\overline{\text{ATN}}$
UNL	
LAG	
GTL	

Table E-20. EXECUTE REMOTE **HP-IB Actions**

Direct I/O	Interface Operations
REN	REN
ATN	$\overline{\text{ATN}}$
MTA	
UNL	
LAG	

Table E-21. EXECUTE LOCAL LOCKOUT **HP-IB Actions**

Direct I/O	Interface Operations
Not applicable.	ATN
	LLO

Details About VXI

The `EXECUTE` commands used by `Direct I/O` to VXI devices and `Interface Operations` are similar, but different.

- `Direct I/O EXECUTE` commands address a message based VXI device to receive a word-serial command.
- `Interface Operations EXECUTE` commands affect the VXI interface directly and may affect VXI devices within the interfaces servant area.

`EXECUTE TRIGGER` transactions for the `Interface Operations` object are of the form:

`EXECUTE TRIGGER` *TriggerType Expression TriggerMode*

TriggerType specifies which trigger group will be used by the

`EXECUTE TRIGGER` transaction. The groups are:

- `TTL` - Specifies the eight TTL trigger lines on the VXI backplane.
- `ECL` - Specifies the four ECL trigger lines on the VXI backplane.
- `EXT` - Specifies the external triggers on a embedded VXI controller.

Expression evaluates to a single Integer variable that represents a bit pattern indicating which trigger lines for a particular *TriggerType* are to be triggered. A value of 5, represented in binary as `101`, indicates that TTL lines 0 and 2 are to be triggered. A value of 255 triggers all eight TTL lines.

TriggerMode indicates the way the trigger lines are to be asserted:

- `PULSE` - Lines are to be pulsed for a discreet time limit (*TriggerType* dependent).
- `ON` - Asserts the trigger lines and leaves them asserted.
- `OFF` - Removes the assertion from trigger lines that were asserted by a previous `ON` transaction.

The following series of tables indicate the exact bus actions conducted by `Direct I/O` and `Interface Operations` `EXECUTE` transactions.

Table E-22. `EXECUTE CLEAR` **VXI Actions**

Direct I/O	Interface Operations
Word-serial command Clear(0xffff)	Pulse SYSRESET line, rerun Resource Manager

Table E-23. `EXECUTE TRIGGER` **VXI Actions**

Direct I/O	Interface Operations
Word-serial command Trigger(0xedff)	Triggers either the TTL or ECL trigger lines in the backplane, or the external trigger(s) on the embedded VXI controller. You can specify which lines are to be triggered for each trigger type.

Table E-24. `EXECUTE LOCAL` **VXI Actions**

Direct I/O	Interface Operations
Word-serial command Set Lock(0xeeff)	Not applicable.

Table E-25. `EXECUTE REMOTE` **VXI Actions**

Direct I/O	Interface Operations
Word-serial command Clear Lock(0xefff)	Not applicable.

WAIT Transactions

There are four types of WAIT transactions:

- WAIT INTERVAL
- WAIT SPOLL (Direct I/O to HP-IB and message based VXI devices only)
- WAIT REGISTER (Direct I/O to VXI devices only)
- WAIT MEMORY (Direct I/O to VXI devices only)

WAIT INTERVAL transactions simply wait for the specified number of seconds before executing the next transaction listed in the open view of the object. For example, this transaction waits for 10 seconds:

```
WAIT INTERVAL:10
```

WAIT SPOLL transactions are of the form:

WAIT SPOLL *Expression Sense*

Expression is an expression that evaluates to an integer. The integer will be used as as a bit mask.

Sense is a field with two possible values.

- ANY SET
- ALL CLEAR

WAIT SPOLL transactions wait until the serial poll response byte of the associated instrument meets a specific condition. The serial poll response is tested by bitwise ANDing it with the specified mask and ORing the resulting bits into a single test bit. The transaction following WAIT SPOLL executes when one of the following conditions is met:

- The transaction specifies ANY (ANY SET) and the test bit is true (1).
- The transaction specifies CLEAR (ALL CLEAR) and the test bit is false (0).

The following transactions show how to use WAIT SPOLL:

```
WAIT SPOLL:256 ANY      Wait until any bit is set.
WAIT SPOLL:256 CLEAR    Wait until all are clear.
WAIT SPOLL:0x40 ANY     Wait until bit 6 is set.
WAIT SPOLL:0x40 CLEAR   Wait until bit 6 is clear.
```

`WAIT REGISTER` and `WAIT MEMORY` transactions are of the form:

```
WAIT REG:SymbolicName MASK:Expression Sense
[Expression]
-or-
WAIT MEM:SymbolicName MASK:Expression Sense
[Expression]
```

where:

SymbolicName is a name defined during configuration of a VXI device. The name refers to a specific address within a device's A16 or extended memory.

`MASK:`*Expression* is an expression that evaluates to an integer. The integer will be used as as a bit mask. The size in bytes of this mask value depends on the data type for which *SymbolicName* has been configured.

Sense is a field with three possible values.

- `ANY SET`
- `ALL CLEAR`
- `EQUAL`

[*Expression*] is an optional compare value that evaluates to an integer. The integer is used only when *Sense* is `EQUAL`.

`WAIT REGISTER` or `MEMORY` transactions wait until the value read from the register or memory location specified by *SymbolicName*s in the associated VXI device meets a certain condition. The value read is logically ANDed with the bit mask specified in `MASK:`*Expression*, resulting in a test value. The size of the test value is dependent on the data type configured for the specified register or memory location. The transaction following `WAIT SPOLL` executes when one of the following conditions is met:

- The transaction specifies `ANY` (`ANY SET`) and the test value has at least one bit true (1).
- The transaction specifies `CLEAR` (`ALL CLEAR`) and the test value has all bits false (0).
- The transaction specifies `EQUAL` and the test value is equal bit-for-bit with the compare value specified in [*Expression*].

SEND Transactions

`SEND` transactions are supported by HP VEE-Test and HP VEE for Windows only.

`SEND` transactions are of this form:

`SEND` *BusCmd*

BusCmd is one of the bus commands listed in Table E-26.

`SEND` transactions are used within `Interface Operations` objects to transmit low-level bus messages via an HP-IB interface. These messages are defined in detail in IEEE 488.1.

Table E-26. `SEND` Bus Commands

Command	Description
`COMMAND`	Sets ATN true and transmits the specified data bytes. ATN true indicates that the data represents a bus command.
`DATA`	Sets ATN false and transmits the specified data bytes. ATN false indicates that the data represents device-dependent information.
`TALK`	Addresses a device at the specified primary bus address (0-31) to talk.
`LISTEN`	Addresses a device at the specified primary bus address (0-31) to listen.
`SECONDARY`	Specifies a secondary bus address following a TALK or LISTEN command. Secondary addresses are typically used by cardcage instruments where the cardcage is at a primary address and each plug-in module is at a secondary address.
`UNLISTEN`	Forces all devices to stop listening; sends UNL.

Table E-26. SEND **Bus Commands (continued)**

Command	Description
UNTALK	Forces all devices to stop talking; sends UNT.
MY LISTEN ADDR	Addresses the computer running HP VEE to listen; sends MLA.
MY TALK ADDR	Addresses the computer running HP VEE to talk; sends MTA.
MESSAGE	Sends a multi-line bus message. Consult IEEE 488.1 for details. The multi-line messages supported by HP VEE-Test are: DCL Device Clear SDC Selected Device Clear GET Group Execute Trigger GTL Go To Local LLO Local Lockout SPE Serial Poll Enable SPD Serial Poll Disable TCT Take Control

WRITE(POKE) Transactions

The WRITE(POKE) transaction is very similar to the WRITE transaction, except that it applies only to the To/From DDE object. The main difference of WRITE(POKE) is that you must specify an item name. For example:

```
WRITE ITEM:"r2c3" TEXT a EOL
```

WRITE(POKE) transactions are supported by HP VEE for Windows only.

The following encodings are allowed:

- TEXT
- BYTE
- CASE
- CONTAINER

For more specific information about these formats see the WRITE transaction.

READ(REQUEST) Transactions

The READ(REQUEST) transaction is very similar to the READ transaction, except that it applies only to the To/From DDE object. The main difference of READ(REQUEST) is that you must specify an item name. For example:

```
READ ITEM:"r2c3" TEXT a EOL
```

READ(REQUEST) transactions are supported by HP VEE for Windows only.

The following encodings are allowed:

- TEXT
- CONTAINER

For more specific information about these formats see the READ transaction.

F

HP VEE-Test and HP VEE for Windows Differences

HP VEE-Test and HP VEE for Windows Differences

In general, programs written for HP VEE will work on any supported platform. Difficulties arise when you use programs that access features specific to the underlying platform, such as DLL's on PC's or named pipes on UNIX. This appendix contains information on the differences between HP VEE on UNIX and PC platforms.

I/O

Since HP VEE for Windows does not support GPIO or the VXI backplane, programs written for these interfaces will not work. You can access VXI through HP-IB and the VXI Command Module, but you would need to modify your program to access this different interface.

If you load a program that accesses a non-supported interface, HP VEE will prompt you to add the interface. You will then need to configure a supported interface with proper parameters. You will probably also need to modify the program appropriately.

Execute Program

There is an **Execute Program** object for both the UNIX and PC platforms. Note that you can determine which platform you are executing on by using the **whichplatform()** or **whichos()** functions in a **formula** box. You can then programmatically determine which **Execute Program** object to use.

DLL versus Shared Library

There are several differences that must be noted when creating DLL's and Shared Libraries for Compiled Functions.

I/O	From a Shared Library do I/O through SICL, DIL or `TERMIO`. For DLL's use the HP-IB Command Library or other similar products.
Graphics	Shared Libraries use X11 graphics while DLL's use Microsoft Windows GDI calls.
File I/O	This is similar between Shared Libraries and DLL's with the exception noted above regarding standard in, out and error.

Data Files

No binary files will work across platforms since byte ordering is reversed between UNIX and PC platforms. However, ASCII data files written using `To File` objects should be readable by `From File` objects on other platforms. Also, HP VEE program files should be compatible since they are stored in ASCII. Note that you must be careful even when moving ASCII files from one platform to another in non-binary formats. For example, UNIX files use the linefeed character to terminate lines while MS-DOS uses the carriage return/linefeed sequence to terminate lines.

Memory Usage

Due to the limitations of Microsoft Windows memory capabilities, i.e., it cannot access more than 16 Mbytes of RAM for all processes, some larger programs may not be able to execute on the PC platform. By eliminating networking or other memory resident software you may be able to free up enough RAM for large HP VEE programs to execute.

G

HP VEE for Windows Instrument I/O Select Codes

HP VEE for Windows Instrument I/O Select Codes

I/O Select Code Map

HP VEE for Windows uses the following select codes to communicate with various interfaces on your PC.

Table G-1. HP VEE for Windows I/O Select Codes

Select Code	Description
1*	HP 82341 HP-IB card
2*	HP 82341 HP-IB card
3	HP 82341 or HP 82335 HP-IB card
4	HP 82341 or HP 82335 HP-IB card
5	HP 82341 or HP 82335 HP-IB card
6	HP 82341 or HP 82335 HP-IB card
7 (default)	HP 82341 or HP 82335 HP-IB card
8*	HP 82341 HP-IB card
9 (default)	COM1 serial port
10	COM2 serial port
11	COM3 serial port
12	COM4 serial port
13	unused
14	GPIB0 for National GPIB cards
15	GPIB1 for National GPIB cards
16	Reserved for VXI
17	GPIB2 for National GPIB cards
18	GPIB3 for National GPIB cards

* Select Codes 1, 2, and 8 are not recommended for the HP 82335 HP-IB card.

Glossary

Glossary

This Glossary defines several terms used to name or describe HP VEE features.

Activate

1. To send a container to a terminal. See also "Container" and "Terminal."

2. The action that resets the context of a `UserObject` before it operates each time. See also "Context" and "PreRun."

Array

A data shape that contains a systematic arrangement of data items in one or more dimensions. The data items are accessed via indexes. See also "Data Shape."

Asynchronous

In asynchronous operation, a device operates without a common signal to synchronize events. Rather, the events occur at unspecified times. HP VEE control pins are asynchronous.

Auto Execute

An option on the object menus of the data constant objects. When `Auto Execute` is set, the object operates when its value is edited.

Bitmap

A bit pattern or picture. In HP VEE you can display a bitmap on an icon.

Buffer

An area in memory where information is stored temporarily.

Button

1. A button on a mouse.

2. A graphical object in HP VEE that simulates a real-life pushbutton and appears to pop out from your screen. When you "press" a button in HP VEE, by clicking on it with the mouse, an action occurs.

Cascading Menu

A sub-menu on a pull-down or pop-up menu that provides additional selections.

Checkbox

A recessed square box on HP VEE menus and dialog boxes that allows you to select a setting. To select a setting, click on the box and a checkmark appears in the box to indicate a selection has been made. To cancel the setting, simply click on the box again.

Click

To press and release a mouse button quickly. Clicking usually selects a menu feature or object in the HP VEE window. See also "Double-Click" and "Drag."

Compiled Function

A user-defined function created by dynamically linking a program, written in a programming language such as C, into the HP VEE process. For UNIX systems, the user must create a shared library file and a definition file for the program to be linked. For Windows, the user must create a DLL (Dynamically Linked Library) file and a definition file. The `Import Library` object attaches the shared library or DLL to the HP VEE process and parses the definition file declarations. The Compiled Function can then be called with the `Call Function` object, or from certain expressions. See also "User Function" and "Remote Function."

Component

A single instrument function or measurement value in an HP VEE instrument panel or component driver. For example, a voltmeter driver contains components that record the range, trigger source, and latest reading. See also "Component Driver," "Driver Files," "State," and "Instrument Panel."

Component Driver

An instrument control object that reads and writes values to components you specifically select. Use component drivers to control an instrument using a driver by setting the values of only a few components at a time. (Component drivers do not support coupling.)

Composite Data Type

A data type that has an associated shape. See also "Data Shape" and "Data Type."

Container

The package that is transmitted over lines and is processed by objects. Each container contains data, the data type, and the data shape.

Context
A level of the work area that can contain other levels of work areas (such as nested `UserObjects`), but is independent of them.

Control Pin
An asynchronous input pin that transmits data to the object without waiting for the object's other input pins to contain data. For example, control pins in HP VEE are commonly used to clear or autoscale a display.

Coupling
The inter-relationship of certain functions in an instrument. If, in an instrument panel, functions A and B are coupled, changing the value of A may automatically change the value of B, even though you do not change B explicitly.

Cursor
A pointer in an entry field that shows where alphanumeric data will appear when you type information from the keyboard.

Data Field
The field within a transaction specification in which you specify either the expression to be written (WRITE transactions), or the variable to receive data that is read (READ transactions). See also "Transactions."

Data Flow
The flow of data through and between HP VEE objects. Data flows from left to right through objects, but an object does not execute until it has data on all of its data input pins. Data is propagated from the data output pin of one object to the data input pin of the next object. Data flow is the chief factor that determines the execution of a HP VEE program.

Data Input Pin
A connection point on the left side of an object that permits data to flow into the object.

Data Output Pin
A connection point on the right side of an object that propagates data flow to the next object and passes the results of the first object's operation on to the next object.

DataSet
A collection of "Record" containers saved into a file for later retrieval. The `To DataSet` object collects Record data on its input and writes that data to a named file (the DataSet). The `From DataSet` object retrieves Record data from the named file (the DataSet) and outputs that data as Record containers on its `Rec` output pin. See also "Record."

Data Shape
A pre-defined structure that defines how data is grouped together (for example, an array).

Data Type
A pre-defined structure that determines how data is organized and treated by HP VEE (for example, Real or Complex).

DDE (Dynamic Data Exchange)
A communication mechanism that allows HP VEE for Windows to communicate with other Windows applications that support DDE. HP VEE can send data to, and receive data from, such applications. Also, HP VEE can execute commands in the other application. Examples of Windows applications that support DDE are Microsoft Excel and Microsoft Word for Windows.

Default
A value or action that HP VEE automatically selects.

Default Button
The button in a dialog box that is activated by default if Enter or Return is pressed, or the selection is double-clicked. The default button has a recessed border.

Demote
To convert from a data type that contains more information to one that contains less information. See also "Data Type" and "Promote."

Detail View
The view of an HP VEE program that shows all the objects and the lines between them.

Device
An instrument attached to or plugged into an HP-IB, RS-232, GPIO, or VXI interface. Specific HP VEE objects such as the `Direct I/O` object send and receive information to a device.

Device Driver
See "Interface Driver."

Dialog Box
A secondary window displayed when HP VEE requires information from you before it can continue. For example, a dialog box may contain a list of files from which you may choose.

Direct I/O Object
An instrument control object that allows HP VEE to directly control an instrument without using an HP Instrument Driver.

DLL (Dynamically Linked Library)
A collection of functions written in C that can be called from HP VEE for Windows. DLLs can be created by experienced C programmers using tools available from Microsoft and Borland. DLLs in the Windows environment are similar to shared libraries in the UNIX environment.

Double-Click
To press and release a mouse button twice in rapid succession. Double-clicking is usually a short-cut to selecting and performing an action. For example, double-clicking on a file name from `File` ⟹ `Open` will select the file and open it.

Drag
To press *and continue to hold down* a mouse button while moving the mouse. Dragging moves something (for example, an object or scroll slider).

Driver
Software that allows a computer to communicate with other software or hardware. See also "Component Driver," "Driver Files," "Interface Driver," and "Instrument Panel."

Driver Files
A set of files included with HP VEE that contains the information needed to create instrument panel and component driver objects for instrument control.

Entry Field
A field that is typically part of a dialog box or an editable object, and which is used for text entry. An entry field appears recessed. For

example, the open view of the `For Range` object has entry fields where you type values that specify the beginning, ending, and step values.

Error Message
Information that appears in an error dialog box, explaining that a problem has occurred.

Error Pin
A pin that traps any errors that occur in an object. Instead of getting an error message, the error number is output on the error pin. When an error is generated, the data output pins are not activated.

Execute
The action of a program, or parts of a program, running.

Execution Flow
The order in which objects operate. See also "Data Flow."

Expression
An equation in an entry field that can contain the input terminal names and any `Math` or `AdvMath` functions. An expression is evaluated at run-time. Expressions are allowed in `Formula`, `If/Then/Else`, `Get Values`, `Get Field`, `Set Field`, and `Sequencer` objects, and in I/O transaction objects.

Feature
An item on a menu that you select to cause a particular action to occur (for example, to open a file), or to get a particular object.

Feedback
A continuous thread path of sequence and/or data lines that uses values from the previous execution to change values in the current execution.

Flow
See "Data Flow" and "Execution Flow."

Function
The name and action of objects where the output is a function of the input. These objects are located under `Math` or `AdvMath` menus and may be used in the `Formula` object. For example `sqrt(x)` is a function; `+` is not.

Global Variable

A named variable that is set globally, and which can be used by name in any context of an HP VEE program. For example, a global variable can be set with `Set Global` in the root context of the program, and can be accessed by name with `Get Global` or from certain expressions within the context of a `UserObject`. However, a local variable with the same name as the global variable takes precedence in an expression.

Grayed Feature

A menu feature that is displayed in gray rather than black, indicating that the feature is not active or not available.

Group Window

A group window in Microsoft Windows is a window that contains icons for a group of applications. Each icon starts an application in the group.

Highlight

1. The colored band or shadow around an object that provides a visual cue to the status of the object.

2. The change of color on a menu feature that indicates you are pointing to that feature.

Host

To begin a thread or subthread. For example, the subthread that is hosted by `For Count` is the subthread that iterates.

HP-UX

Hewlett-Packard Company's enhanced version of the UNIX operating system.

Hypertext

A system of linking topics so that you can jump to a related topic when you want more information. In online help systems, typically hypertext links are designated with underlined text. When you click on such text, related information is presented.

Icon

1. The small, graphical representation of an HP VEE object, such as the representation of an instrument, a control, or a display.

2. The small, graphical representation of a Microsoft Windows application within a group window. See "Group Window."

Instrument Driver
See "Driver Files," "Component Driver," and "Instrument Panel."

Instrument Panel
An instrument control object that forces all the function settings in the corresponding physical instrument to match the settings in the control panel displayed in the open view of the object.

Interface
HP-IB, RS-232, GPIO, and VXI are referred to as interfaces used for I/O. Specific HP VEE objects, such as the `Interface Event` object can only send commands to an interface.

Interface Driver
Software that allows a computer to communicate with a hardware interface, such as HP-IB or RS-232. Also called *device driver* in the UNIX operating system, interface drivers are configured into the kernel of the operating system.

Interrupt
A signal that requires immediate attention that may suspend a process, such as the execution of a computer program. An interrupt is usually caused by an event external to that process. After the interrupt is serviced, the process may be resumed.

Label
The text area or name on an icon or button that identifies that object or button.

Library
A collection of often-used objects or small programs grouped together for easy access.

Line
A link between two objects in HP VEE that transmits data containers to be processed. See also "Subthread" and "Thread."

Main Menu
The menus located in the HP VEE menu bar. The main menus may be opened by clicking or dragging on the menu titles in the menu bar.

Main Work Area
The area where you create a program. The main work area is the parent context of all other contexts.

Mapping
To associate a set of independent values with an array, when the array is a function of the values.

Maximize
To enlarge a window to its maximum size. In HP VEE, the `UserObject` has a maximize button.

Menu
A collection of features that are presented in a list. See also "Cascading Menu," "Main Menu," "Object Menu," "Pop-Up Menu," and "Pull-Down Menu."

Menu Bar
A rectangular bar at the top of the HP VEE window that contains titles of the pull-down, main menus from which you select features.

Menu Title
The name of a menu within the HP VEE menu bar. For example, `File` or `Edit`.

Minimize
1. To reduce an open view of an object to its smallest size—an icon.
2. To reduce a window to its smallest size—an icon.

Mouse
A pointing device that you move across a surface to move a pointer within the HP VEE window.

Mouse Button
One of the buttons on a mouse that you can click, double-click, or drag to perform a particular action with the corresponding pointer in the HP VEE window.

Network
A group of computers and peripherals linked together to allow the sharing of data and work loads.

Object
A graphical representation of an element in a program, such as an instrument, control, display, or mathematical operator. An object is placed in the work area and connected to other objects to create a program. Objects can be displayed as icons or as open views.

Object Menu
The menu associated with an object that contains features that operate on the object (for example, moving, sizing, copying, and deleting the object).

Open
To start an action or begin working with a text, data, or graphics file. When you select `Open` from HP VEE, a program is loaded into the work area.

Open View
The representation of an HP VEE object that is more detailed than an icon. Within the open view, you can modify the operation of the object and change the object's title.

Operate
The action of an object processing data and outputting a result. An object operates when its data and sequence input pins have been activated. See "Activate."

Outline Box
A box that represents the outer edges of an object or set of objects and indicates where the object(s) will be placed in the work area.

Palette
A set of colors and fonts that is supplied with HP VEE and used in your HP VEE environment.

Panel
Information displayed in the center of the object's open view. In a `UserObject`, the panel contains a work area. In a `For Count` object, the panel contains an entry field. Compare with "Panel View."

Panel View
The view of a program in HP VEE that shows only those objects needed for the user to run the program and view the resultant data. You can create a panel view to meet the needs of your users.

Pin
An external connection point on an object to which you can attach a line.

Pointer
The graphical image that maps to the movement of the mouse. A pointer allows you to make selections and provides you feedback on a particular process underway. HP VEE has pointers of different shapes that correspond to process modes, such as an arrow, crosshairs, and hourglass.

Pop-Up Menu
A menu that is raised by clicking the right mouse button. For example, you can raise the `Edit` menu by clicking the right mouse button in an empty area within the work area. Or you can raise the object menu by clicking the right mouse button on an inactive area of an object.

PostRun
The set of actions that are performed when the program is stopped.

PreRun
The set of actions that resets the program and checks for errors before the program starts to run.

Priority Thread
A priority thread executes to completion blocking all other parallel threads from executing. Certain of the I/O objects for devices and interfaces will host a priority thread.

Program
In HP VEE, a graphical program that consists of a set of objects connected with lines. The program typically represents a solution to an engineering problem.

Promote
To convert from a data type that contains less information to one that contains more information. See also "Data Type" and "Demote."

Propagation
The rules that objects and programs follow when they operate or run. See also "Data Flow."

Pterodactyl
Any of various extinct flying reptiles of the order Pterosauria of the Jurassic and Cretaceous periods. Pterodactyl are characterized by wings

consisting of a flap of skin supported by the very long fourth digit on each front leg.

Pull-Down Menu
A menu that is pulled down from the menu bar when you position the pointer over a menu title and click the left mouse button.

Radio Button
A diamond-shaped button in HP VEE dialog boxes that allows you to select a setting that is mutually exclusive with other radio buttons in that dialog box. To select a setting, click on the radio button. To remove the setting, click on another radio button in the same dialog box.

Record
A data type that has named data fields which can contain multiple values. Each field can contain another Record container, a Scalar, or an Array. The Record data type has the highest precedence of all HP VEE data types. However, data cannot be converted to and from the Record data type through the automatic promotion/demotion process. Records must be built/unbuilt using the using `Build Record` and `UnBuild Record` objects.

Remote Function
A User Function running on a remote host computer, which is callable from the local host. (Remote functions are supported only on UNIX systems.) The `Import Library` object starts the process on the remote host and loads the Remote File into the HP VEE process on the local host. You can then call the Remote Function with the `Call Function` object, or from certain expressions. See also "User Function" and "Compiled Function."

Resource Manager
A program which exists on VXI controllers that runs at start-up and after a VXI system reset. This program initializes and manages the instruments in a VXI card cage.

Restore
To return a minimized window or an icon to its full size as a window or open view by double-clicking on it.

Run
To start the objects on a program or thread operating.

Save
To write a file to a storage device, such as a hard disk, for safekeeping.

Scalar
A data shape that contains a single value. See also "Data Shape."

Schema
The structure or framework used to define a data record. This includes each field's name, type, shape (and dimension sizes), and mapping.

Screen Dump
A graphical printout of a window or part of a window.

Scroll
The act of using a scroll bar either to move through a list of data files or other choices in a dialog box, or to pan the work area.

Scroll Arrow
An arrow that, when clicked on, moves you through a list of data files or other choices in a dialog box, or moves the work area.

Scroll Slider
A rectangular bar that, when dragged, moves you through a list of data files or other choices in a dialog box, or moves the work area.

Select
To choose an object, an action to be performed, or a menu item. Usually you select by clicking with your mouse.

Select Code
A number used to identify the logical address of a hardware interface. For example, the factory default select code for most HP-IB interfaces is 7.

Selection
1. A menu selection (feature).
2. An object or action you have selected in the HP VEE window.

Sequence Input Pin
The top pin of an object. When connected, execution of the object is held off until the pin receives a container.

Sequence Output Pin
The bottom pin of an object. When connected, this output pin is activated when the object and all data propagation from that object finishes executing.

Sequencer
An object that controls execution flow through a series of sequence transactions, each of which may call a "User Function," "Compiled Function," or "Remote Function." The sequencer is normally used to perform a series of tests by specifying a series of sequence transactions.

Shared Library
A collection of functions, written in a programming language such as C, that can be called from HP VEE running on a UNIX system. Shared libraries can be created by experienced programmers. Shared libraries in the UNIX environment are similar to DLLs in the Windows environment.

Shell
In a UNIX system, the program that interfaces between the user and the operating system.

Shell Prompt
In a UNIX system, the character or characters that denote the place where you type commands while at the operating system shell level. The prompt you see displayed depends upon the type of shell you are running, such as a **#** prompt for the Bourne shell.

Sleep
An object sleeps during execution when it is waiting for an operation or time interval to complete, or for an event to occur. A sleeping object will allow other parallel threads to run concurrently. Once the event, time interval, or operation occurs, the object will execute, allowing execution to continue.

Startup Directory
The directory from which you start HP VEE on a UNIX system. This directory determines the default paths for most file actions including `Save` and `Open`.

State
A particular set of values for all of the components related to an HP VEE instrument panel, which represents the measurement state of an instrument. For example, a digital multimeter uses one state for

high-speed voltage readings and a different state for high-precision resistance measurements. See also "Instrument Panel."

Step
The action of operating one object at a time. An arrow points to the object that will operate next.

Terminal
The internal representation of a pin that displays information about the pin and the data container held by the pin. Double-click on the terminal to view the container information.

Thread
A set of objects connected by solid lines in an HP VEE program. A program with multiple threads can run all threads simultaneously.

Title Bar
The rectangular bar at the top of the open view of an object or window, which shows the title of the object or window.

Tool Bar
The rectangular bar at the top of the HP VEE window which provides the `Run`, `Stop`, `Cont`, and `Step` buttons to control HP VEE programs. The tool bar also displays the title of a program.

Transaction
The specifications for input and output (I/O) used by certain objects in HP VEE. These include the `To File`, `From File`, `Direct I/O`, and `Sequencer` objects. Transactions appear as English-like phrases listed in the open view of these objects.

User-Defined Function
HP VEE allows three types of user-defined functions: the "User Function," "Compiled Function," and "Remote Function."

User Function
A user-defined function created from a "UserObject" by executing `Make UserFunction`. The User Function exists in background, but provides the same functionality as the original UserObject. You can call a User Function with the `Call Function` object, or from certain expressions. A User Function can be created and called locally, or it can be saved in a library and imported into an HP VEE program with `Import Library`. See also "Compiled Function," "Remote Function," and "UserObject."

User Interface
The part of an application that permits a user and the application to communicate with each other to perform certain tasks. HP VEE uses a graphical user interface, which includes windows, menus, dialog boxes, and objects.

UserObject
An object that can encapsulate a group of objects to perform a particular purpose within a program. A UserObject allows you to use top-down design techniques when building a program, and to build user-defined objects that can be saved in a library and reused.

View
See "Detail View," "Icon," "Open View," and "Panel View."

Wait
See Sleep.

Window
A rectangular area on the screen that contains a particular application program, such as HP VEE.

Work Area
The area within the HP VEE window or the open view of a `UserObject` where you group objects together. When you `Open` a program, it is loaded into the main work area.

X Window System (X11)
An industry-standard windowing system used on UNIX computer systems.

X11 Resources
A file or set of files that define your X11 environment in a UNIX system.

XEQ Pin
A pin that forces the operation of the object, even if the data or sequence input pins have not been activated. See also "Control Pin," "Data Input Pin," and "Sequence Input Pin."