VEE OneLab Advanced Techniques

Notice

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

Agilent Technologies makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Agilent Technologies Inc. shall not be liable for errors contained herein or for any direct, indirect, special, incidental, or consequential damages in connection with the furnishing, performance, or use of this material.

This document contains information which is protected by copyright. No part of this manual may be reproduced in any form or by any means (including electronic storage and retrieval or translation into another language) without prior agreement and written consent from Agilent Technologies Inc., as governed by United States and international copyright laws.

U.S. Government Restricted Rights

The Software and Documentation have been developed entirely at private expense. They are delivered and licensed as “Commercial computer software” as defined in DFAR 252.227-7014 (June 1995), as a “commercial item” as defined in FAR 2.101(a), or as “Restricted computer software” as defined in FAR 52.227-19 (June 1987) (or any equivalent agency regulation or contract clause), whichever is applicable. Use, duplication, or disclosure of Software is subject to Agilent Technologies’ standard commercial license terms, and non-DOD Departments and Agencies of the U.S. Government will receive no greater than Restricted Rights as defined in FAR 52.227-19(c)(1-2) (June 1987). U.S. Government users will receive no greater than Limited Rights as defined in FAR 52.227-14 (June 1987) or DFAR 252.227-7015(b)(2) (November 1995), as applicable in any technical data.

Trademark Information

Printing History

Edition 1 - March 2000 Reflects software version 6.0

Conventions Used in This Manual

This manual uses the following typographical conventions:

Getting Started	Italicized text is used for book titles and for emphasis.
Dialog Box	Bold text is used for the first instance of a word that is defined in the glossary.
`File`	Computer font represents text that you will see on the screen, including menu names, features, buttons, or text that you have to enter.
`dir` *filename*	In this context, the text in computer font represents an argument that you type exactly as shown, and the italicized text represents an argument that you must replace with an actual value.
`File ⇒ Open`	The "⇒" is used in a shorthand notation to show the location of Agilent VEE features in the menu. For example, "`File ⇒ Open`" means to select the `File` menu and then select `Open`.
`Sml | Med | Lrg`	Choices in computer font, separated with bars (|), indicate that you should choose one of the options.
Press **Enter**	In this context, bold represents a key to press on the keyboard.
Press **Ctrl + O**	Represents a combination of keys on the keyboard that you should press at the same time.

Contents

1. Introduction

4. Using Transaction I/O

5. Advanced I/O Topics

7. Using VXIplug&play Drivers

8. Data Propagation

9. Math Operations

11. Using Records and DataSets

12. User-Defined Functions/Libraries

13. Using ActiveX Automation Objects and Controls

Figures

Tables

1

Introduction

Introduction

This chapter provides an introduction to this manual and to VEE, including:

- About This Manual
- Configuring VEE
- Using VEE Example Programs
- Supported I/O Interfaces
- Using VEE Execution Modes
- Related Reading

About This Manual

This manual provides detailed information about the advanced features of VEE. Table 1-1 briefly describes the manual contents.

Table 1-1. Manual Contents Descriptions

Chapter	Description
1 - Introduction	Shows how to use VEE example programs and library objects.
2 - Instrument Control Fundamentals	Explains five methods for communicating with instruments.
3 - Configuring Instruments	Explains four methods to configure VEE to communicate with instruments.
4 - Using Transaction I/O	Explains all VEE I/O objects that use transactions.
5 - Advanced I/O Topics	Explains I/O configuration and addressing.
6 - Using Panel Driver and Component Driver Objects	Describes how to use Panel Driver and Component Driver objects with VEE.
7 - Using VXIplug&play Drivers	Explains how to use a VXI*plug&play* driver to communicate with an instrument.
8 - Data Propagation	Describes how to produce programs using data propagation between objects.
9 - Math Operations	Describes math operations on scalars and arrays.
10 - Variables	Describes variables in VEE.
11 - Using Records and DataSets	Describes the Record data type and the DataSet.

Table 1-1. Manual Contents Descriptions

12 - User Defined Functions and Libraries	Describes 19 categories of built-in functions and explains UserFunctions.
13 - Using ActiveX Automation Objects and Controls	Explains how to use ActiveX automation and controls in VEE.

Configuring VEE

This section gives guidelines to configure and customize VEE for your environment by changing Windows options.

Configuring VEE for Windows

VEE for Windows uses the Windows Registry to store VEE environment information. You can change many VEE window properties in the VEE `Default Preferences` dialog box (use `File` ⇒ `Default Preferences`). These properties are saved in the defaults file `VEE.RC` in the following directory:

`%userprofile%\LocalSettings\Application Data\Agilent\VEE OneLab`

Color and Font Settings

In VEE 6.0, the `Save colors & fonts with program` selection no longer appears in the `Default Preferences` dialog box. You can save colors and fonts with the program by choosing File ⇒ Save As. Figure 1-1 shows the new `Save File` dialog box.

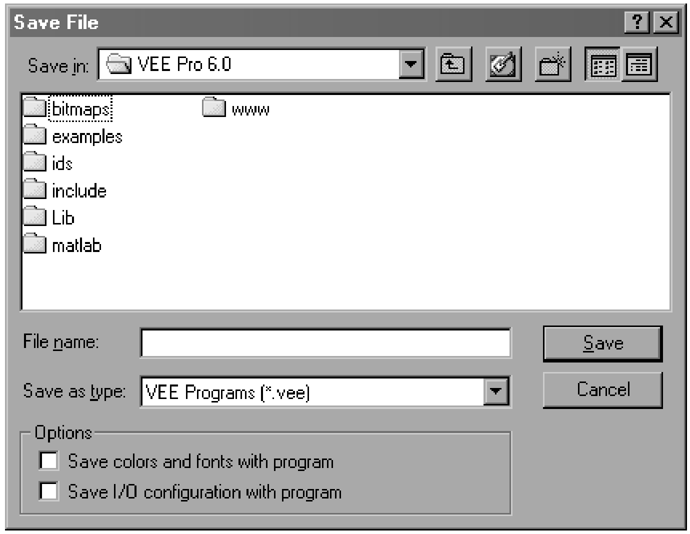


Figure 1-1. The `File ⇒ Save As` Dialog Box

For colors and fonts, only the settings you *change* are saved in the `VEE.RC` defaults file. See `How Do I` in *VEE Help* for more information about changing colors and fonts in VEE.

Customizing Icon Bitmaps

You can change the icon displayed for any iconized object to a bitmap or pixmap. VEE provides many files, or you can create your own. VEE for Windows supports 24-bit `.BMP` bitmap files, `.GIF87`, `.GIF89a`, `.PNG` (Portable Network Graphics), .WMF (Windows Meta Files), and `.ICN` icon files. To select an object's icon, click the object menu's `Properties` feature, then use the `Icon` tab on the `Properties` dialog box.

You can create your own bitmaps for object icons using any editor that outputs graphics formats that VEE supports, such as MS Windows `Paint`. You should specify `48x48` as the size for an icon. Larger icons use more space in the VEE program area while smaller icons are difficult to see. You can also use screen capture utilities such as `Print Screen` with `Paint`.

Selecting a Bitmap for a Panel View

You can select a bitmap to use as the background icon for a panel view. This applies to UserObjects and to VEE programs displayed in their panel views.

Panel view icons must use the formats VEE supports. You can also use icons you create, as described in the previous paragraph.

To select a bitmap as the icon for a *panel view*, enable the panel view so the `Panel` and `Detail` buttons appear in the title bar (by adding an object to the panel). Click the object menu, then click `Properties`. Use the `Panel` tab on the `Properties` dialog box to choose a bitmap.

Using VEE Example Programs

VEE includes many example programs to help you understand how it works. The example programs are installed as part of the VEE installation process.

The Example Directories

The default directory for examples is:

For Windows:

```
C:\Program Files\Agilent\VEE OneLab 6.0\examples\
```

The examples referenced in this manual are included in the `Manual` subdirectory, with file names like `manual01.VEE`, etc. Other examples not referenced in this manual are available in other subdirectories to illustrate specific VEE concepts or to illustrate solutions to engineering problems.

Running the Examples

You can load and run example programs using the `Help` menu, as follows:

1. Click `Help` ⇒ `Open Example` on the menu bar. This presents a list of subdirectories that group similar examples together. (You can also use `File` ⇒ `Open` ⇒ `Examples` to load VEE examples.)
2. Double-click the desired subdirectory to see the programs in that group.
3. Scroll through the list until you find the desired example.
4. Click the example name, then click `OK` to open the program. You are prompted to save the any existing program in the work area.
5. To run the program, press the `Run` button on the tool bar.

Supported I/O Interfaces

Before VEE can communicate with instruments, the computer running VEE must be properly configured and the I/O libraries must be installed as described in *Installing the Agilent IO Libraries - VEE for Windows*. Also, see “Logical Units and I/O Addressing” on page 171 in this manual for **`logical unit`** and I/O addressing information.

Table 1-2 lists the supported I/O interfaces for each platform.

Table 1-2. Instrument I/O Support

Platform	Supported I/O Interfaces
Windows 95/98 (PC, HP 6232, HP 6233, EPC7/8)	GPIB[a] Serial GPIO VXI[b]
Windows NT (PC, HP 6232, HP 6233, EPC7/8)	GPIB[a] Serial GPIO VXI[b]

a. Can address VXI devices using HP E1406 Command Module.

b. Direct backplane access for embedded controllers: HP 6232 or HP 6233 VXI Pentium® Controller, HP RADI-EPC7/8 VXI Controller, or RadiSys EPC7/8 VXI Controller. Direct backplane access for external PCs using VXLink.

Using VEE Execution Modes

This section gives guidelines for using VEE **`Execution Modes`**, including:

- Setting Execution Modes
- Execution Mode Changes: VEE 3 to VEE 4
- Execution Mode Changes: VEE 4 to VEE 5
- Execution Mode Changes: VEE 5 to VEE 6

Setting Execution Modes

Each version of VEE has several Execution Modes (formerly Compatibility Modes). This allows a newer version of VEE to run programs created with an older VEE version exactly the same way the older VEE ran them. This is known as "backwards compatibility", and all version of VEE are 100% backwards compatible using Execution Modes. Version 6.0 of VEE adds the `VEE 6` Execution Mode.

What is an Execution Mode?

VEE Version 4.0 had two Execution Modes: `VEE 3.x` and `VEE 4`. This allowed VEE 4.0 to run old programs created with VEE Version 3.0 (or prior) in the exact same way the programs ran in VEE Version 3. If you want to run a VEE 3 program in Compiled mode, you switch modes and then VEE 4 runs your program using the Compiler.

In the same manner, VEE Version 5.0 had three Execution Modes: `VEE 3.x`, `VEE 4`, and `VEE 5`. In VEE Version 6.0, `VEE 6` Execution Mode is added to the list. When a program created with an older version of VEE is brought into VEE, the program knows what Execution Mode it used. When VEE loads that program, VEE puts itself into that corresponding Execution Mode so the program will run exactly as it did in the older version of VEE.

Once a program is written and saved with a Execution Mode, the program retains that Execution Mode unless changed by the user. If you developed and saved a program in Version 5.0, the program is saved with Execution Mode `VEE 5`. If you then load the program in Version 6.0 and save it, the program still has Execution Mode `VEE 5`. Unless you change the Execution Mode (using the Default Preferences Dialog Box or other means), the

Execution Mode does not change for the program, no matter which version of VEE loads the program.

To change a program's execution mode, open the `File` menu and select `Default Preferences`. When the Default Preferences dialog box opens, click the diamond next to the execution mode you want to apply to the program. Save the program or click `OK`.

Why should I want to change Execution Modes?

You should change Execution Modes if you add new features to an existing program. For example, if you add new features, such as new data types available in VEE 6, to a program written in VEE Version 5.0 (with associated Execution Mode `VEE 5`), you should change the Execution Mode to `VEE 6`. If you change the program but do not change the Execution Mode, the new features added to the program may not run properly.

How do I know when to change Execution Modes?

In most cases, programs written in previous versions of VEE will run 100% as long as the Execution Mode is not changed. However, you may not be able to run new features unless the latest Execution Mode is used.

Table 1-3 shows the combinations of programs that will run for various versions and Execution Modes. Note that old programs will run in any version as long as the Execution Mode is not changed. The only potential problem occurs when the Execution Mode is switched.

Table 1-3. VEE Versions and Execution Modes

	Program Created in Execution Mode:		
Running in VEE:	`VEE 4`	`VEE 5`	`VEE 6`
Version 4.0	runs	CNA*	CNA*
Version 5.0	runs	runs	CNA*
Version 6.0	runs	runs	runs

***CNA means Compatibility Not Assured.** Programs created on later versions of VEE might load and run on earlier versions of VEE, if they do not include any features unique to the later version. Programs which take advantage of newly added features will not run correctly on older versions of

VEE. In some cases, the programs may not even load into older versions of VEE.

The point of Execution Modes is to assure that existing programs will run on newer versions of VEE. There is no assurance that new features will run on old versions of VEE.

Guidelines to Switching Execution Modes

You should use `VEE 6` Execution Mode when you develop new programs in VEE 6.

You can run any existing VEE program by selecting the applicable Execution Mode (`VEE 3` for VEE 3.x programs, `VEE 4` for VEE 4.x programs, or `VEE 5` for VEE 5.x programs). The appropriate mode for older programs is automatically set when the program is loaded.

If you switch to `VEE 6` Execution Mode, an old program may or may not run correctly. Most programs will run correctly. See the following example for a "bug fix" that may cause a program to run differently.

Suppose you have a Version 5.0 program (Execution Mode `VEE 5`) which includes a To File that does a WRITE BINARY BYTE transaction. In Version 5.0 (and prior versions), you could send "300" into this and get "44" written to the file. (This is technically a defect because "300" does not fit into a byte, and this should have errored instead of truncating 300 to 44.)

In Version 6.0 with `VEE 5` and prior Execution Modes, you still get "44" to preserve program compatibility. However, in VEE 6.0 with `VEE 6` Execution Mode you get an error message saying VEE cannot convert 300 to a `UInt8` (out of range). See “Execution Mode Changes: VEE 5 to VEE 6” on page 27 for a list of changes from VEE 5 to VEE 6 execution mode.

About the Compiler

To use the compiler and include ActiveX automation and controls, set `Execution Mode` to `VEE 6`. If you want to convert `VEE 3` programs to `VEE 6` mode, you should make sure they work in `VEE 4` and `VEE 5` modes first, as there are some program execution differences between each mode.

Note It is not necessary to understand the information in this section to use the compiler. This section explains the concepts behind the compiler for your information only. Information about the compiler applies to `VEE 4` and higher modes, except for minor changes.

The compiler works with programs that run in `VEE 4, VEE 5,` or `VEE 6` modes. The VEE compiler converts a VEE program into p-code, but there is no machine language or executable generated.

The compiler allows VEE to:

- Predict at compile time (instead of determining at run time) the order of execution of objects
- Determine what data types will be flowing on certain data lines
- Optimize code generation
- Generate and execute the most optimal p-code for any given VEE object.

VEE programs compile transparently when you press the `Run` button. Stepping and breakpoints are fully supported, as well as `Show Execution Flow`, `Show Data Flow` and `Line Probe`.

Subsequent runs of the same unmodified program do not require recompilation. When a program is modified only the contexts needing recompiling are recompiled (much like an incremental compiler). Most programs benefit from the use of the compiler, though the actual results vary. For example, a program using many levels of nested loops may see a greater speedup than one that does a lot of I/O or screen updates (e.g., displays).

In compiled mode, iterators and formulas gain the most execution speed benefit. A program written with an previous version of VEE may not run exactly the same way with the compiler. This could be due to specific programming techniques, use of undocumented side-effects, or even slight changes in documented behavior.

Execution Mode Changes: VEE 3 to VEE 4

VEE programs written with versions before VEE 4.0 run *exactly* the same as they used to when run in `VEE 3` mode. To ensure this, the interpreter is automatically enabled upon loading of older programs. This section describes the new functions and enhancements in VEE Version 4.0, that is, in VEE 4 mode.

Line Colors in Compiler Mode

In compiler mode, VEE assigns different colors to the data lines that connect objects based on the type of data flowing through the line. The default colors are listed below, along with the names of the color properties. You can change them in the `Default Preferences` dialog box, selected from the `File` menu. Choose the line you want to change in the Screen Element box, click on the Color Value box to open the color palette, and click on the color you prefer. Click `OK` to keep the new color for the selected line type.

- Dark Sky Blue: numeric (Integer or Real type)
- Dark Sky Blue: complex (Complex and PComplex type)
- Med Orange: string (String type)
- Med Dark Gray: sequence out (nil value, usually from a sequence out line)
- Magenta: highlight
- Black: unknown type or type that is not optimized (for example, Record types).

If the data type is an array, VEE displays a wider line. To increase speed, check your program for colored lines. The more non-black lines, the faster the program runs.

Potential Compatibility Problems

Programs written in versions before VEE 4.0 automatically run in `VEE 3` mode. Programs written using VEE 4.x automatically run in `VEE 4` mode. Programs written using VEE 5.x automatically run in `VEE 5` mode. You can, however, change the `Execution Mode` of a program at any time.

Compatibility problems could arise in certain areas when changing an existing program from `VEE 3` to `VEE 4`. The following paragraphs explain the potential problem areas. The information about using older versions of VEE is the same as when using interpreted mode or `VEE 3` mode. (If you are creating new programs, you should use `VEE 6` Execution Mode.)

Time-Slicing UserFunctions. In versions before VEE 4.0, UserFunctions did not time-slice with other parts of the program. In compiled mode, UserFunctions will time-slice when called from separate threads. Be sure to use sequence pins between `Call` objects when parallelism is not desired.

UserFunctions only time-slice when called from `Call`, `Formula`, or `If/Then/Else` objects. Breakpoints also now work in UserFunctions when called from `Call` or the other objects listed above.

UserFunctions will not time-slice, nor will breakpoints work, when called from a `To File`, `To String`, or similar objects or if the formula is supplied via a control pin.

If a UserFunction is executing and gets called again from another part of the program, that call will be blocked until the original call returns.

UserObjects. UserObjects would always time-slice in previous versions, but in compiled mode they will only time-slice when invoked from separate threads.

Function Precedence. The precedence of functions called from the `Formula` object has changed to the following:

1. Internal functions (like `sin()` and `totSize()`)
2. Local UserFunctions
3. Imported UserFunctions
4. Compiled Functions
5. Remote Functions

In `VEE 3` Execution Mode, internal functions are last in precedence. This allowed you to override internal functions such as `totsize()` or `fft()` with your own.

Auto Execute and Start. There are some subtle changes in behavior when using the `Auto Execute` feature of certain objects. In compiled mode, the behavior is as if the object was hooked directly to a `Start` object and that `Start` button was pushed. This change does not affect most programs.

OK Buttons and Wait for Input. Most asynchronous objects like the `OK` object or any object with `Wait for Input` enabled will work better in compiled mode in these two areas:

- Stepping: In previous versions, stepping over such an object would often result in the termination of the program. In compiler mode, stepping works properly.
- CPU usage: In previous versions, executing such an object usually resulted in increased CPU usage. In compiler mode, the CPU stays in an idle state.

Collectors Without Data. In previous versions, pinging the `XEQ` pin of a `Collector` that has never been pinged with data outputs a nil container. In compiler mode, if the data type is known at compile time, you get a zero-element array of that data type. Otherwise, you get a zero-element array of type Integer.

This change allows the type inferences to be more consistent, producing better p-code downstream from the `Collector` object. Note that `totSize()` of a nil produces a one, while `totSize()` of a zero-element array produces a zero.

Sample & Hold Without Data. In previous versions, pinging the `XEQ` pin of a `Sample & Hold` object that has never been pinged with data yields a nil container. In compiler mode, the following error is generated (error number 937):

```
Sample & Hold was not given any data.
```

This change allows the type inferences to be more consistent, producing better p-code downstream from the `Sample & Hold` object.

Timer Object. In previous versions, the `Timer` object output an undefined result if the `Time2` pin (the bottom data input pin) was pinged before the `Time1` pin. In compiler mode, the `Timer` object generates an error if the pins are executed out of sequence.

Feedback Cycles. In compiler mode, a `Junction` object is required inside a feedback cycle. `Start` objects are no longer required. The following error is generated when feedback without a `Junction` is detected (error number 935):

`A Junction` is required inside of feedback cycles. See Figure 1-2 and Figure 1-3.

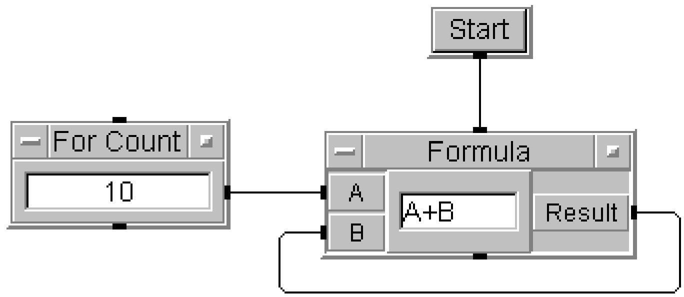


Figure 1-2. Feedback in Previous Versions

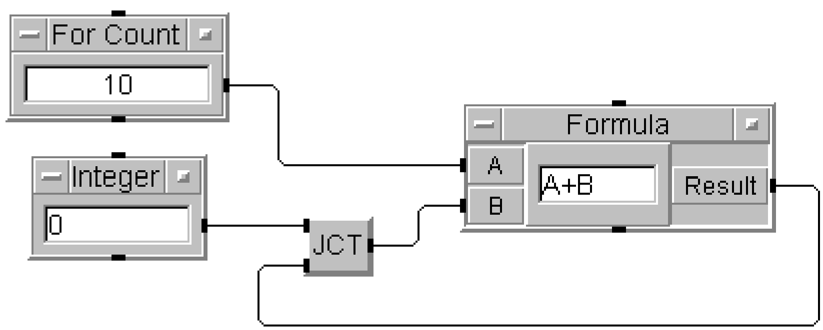


Figure 1-3. Feedback in Compiled Mode

VEE Version 4.0 and higher does not allow invalid connections, such as an object's data input pin connected to its data output pin or, for most objects, connecting a sequence output pin to a data input pin.

Parallel Threads. In `VEE 3` Execution Mode, independent threads would round-robin between each thread, meaning that one object will be executed in one thread, then an object in the other thread, etc. In compiler mode, this behavior is not guaranteed.

Loop Bounds. To increase looping performance, the bounds of iterators (such as the Step field in a For Range object) are examined only at the beginning of the first iteration and not at every iteration. The object's fields are grayed at run time to show the value is not changeable. Data inputs to the iterators will be ignored if the value changes while the loop is running

For example, if the Step value of a For Range object is changed via the data input pin while the loop runs, it is ignored in VEE 4 and higher Execution Mode. In previous versions, the step value would have been checked on every iteration.

UserObjects and Calls With XEQ Pins. In versions before 4.0, you could have an XEQ pin on a UserObject or a Call object run the UserObject or UserFunction before all the data input pins were satisfied. The behavior of objects inside the UserObject or UserFunction connected to those unpurged data inputs was undefined. In VEE 4 and higher Execution Mode, this is not allowed. XEQ pins on those objects will generate an error. You can no longer add an XEQ pin to those objects.

OK Buttons With XEQ Pins. In versions before 4.0, an OK object with an XEQ pin was only executed once, when either the OK button was pressed or when the XEQ pin was sent data. In VEE 4 and higher Execution Mode, the OK button executes every time the XEQ pin is sent data. You can no longer add an XEQ pin to an OK object.

From File With EOF Pins. In versions before 4.0, the data output pin on a From File object was treated differently from other data output pins in VEE. If the From File was in a loop, the data on the output pin remained valid when the EOF data output pin was executed.

In VEE 4 and higher Execution Mode, the data output from a From File object is invalidated each time the loop executes (just like on all other objects). Therefore, when the EOF pin is executed, the data output is already invalid and cannot propagate.

Figure 1-4 illustrates this situation. In versions before 4.0, the data fed into A on the Formula would have remained valid even while another iteration of the loop executed. To get valid data fed into `B` on the `Formula`, the `EOF` pin (on the bottom) executes and then the `Formula` executes.

In `VEE 4` and higher Execution Mode, the data fed into `A` is invalidated as soon as the next iteration of the loop begins. Because `Formula` does not get valid inputs on the same iteration of the loop, it never executes.

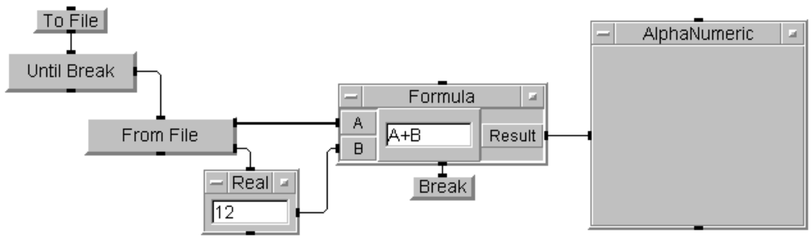


Figure 1-4. EOF Differences

Parallel Junctions. In versions before VEE 4.0, if you had unconstrained objects that were connected in parallel to `Junction` objects, the order that you made the connections affected the execution order. In `VEE 4` and higher Execution Mode, the order of connection does not matter, as Figure 1-5 shows.

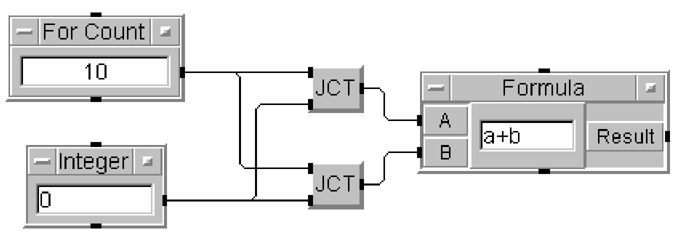


Figure 1-5. Parallel Junctions

Intersecting Loops. In versions before 4.0, you could intersect iteration objects. The execution order was undefined, but was affected by the order the connections were made. In `VEE 4` and higher Execution Mode, only loops that intersect via a `Junction` object are allowed. Any other intersecting loops generate error 938. `VEE was unable to compile this part of the program`. Figure 1-6 shows this situation.

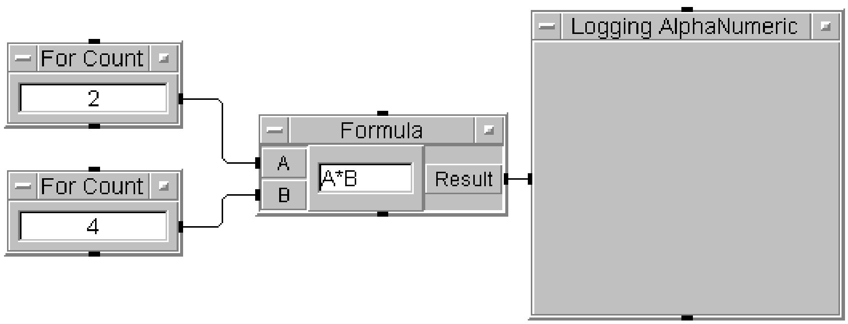


Figure 1-6. Intersecting Loops

Intersecting Loops Via Junctions. In versions before VEE 4.0, the example shown in Figure 1-7 would execute the `Integer` first. When the program encountered the `Break` it would stop. In `VEE 4` and higher Execution Mode, the example below runs the `For Count` objects after the `Integer` objects because the `Break` does not stop the program.

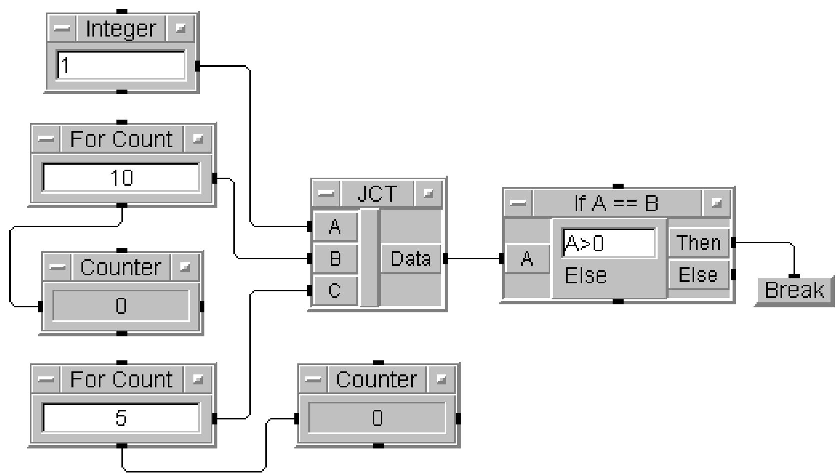


Figure 1-7. Intersecting Loops Via Junctions

Open View Object Changes. In versions before VEE 4.0, you could change the data in open view fields while the program was running or paused. These changes would affect program behavior and the result was not guaranteed. In `VEE 4` and higher Execution Mode, many objects do not allow this type of modification when the program is running or paused (the input fields are grayed). Some examples of this are:

- `Formula` and `If/Then`
- `Collector`
- All Transaction objects' transactions
- `Get Mappings` and `Set Mappings`
- `Get Values` and `Set Values`
- `Constant`'s properties such as setting `Scalar` or `1D Array`, `Wait for Input`, or `Auto Execute`.
- Setting properties like `Clear at PreRun`
- UserObject and UserFunction `Trig Mode`

- `Dialog Boxes` properties

Adding or deleting input or output terminals on objects is grayed at run time (but not when paused). If this action is done at pause time, the program is stopped (as in versions before VEE 4.0).

Array Syntax in Expressions. Expressions with array syntax entered without commas, such as `[1 2 3]`, will be reparsed when the program loads and automatically modified to use commas, as in `[1,2,3]`. This is true for programs in `VEE 3` and `VEE 4` modes.

Execution Mode Changes: VEE 4 to VEE 5

In VEE 6.0, the `VEE 4` and `VEE 3` modes retain their compatibility definitions set in VEE 4.0, which are described in "Execution Mode Changes: VEE 3 to VEE 4" on page 13. There are minor changes that will not affect existing programs that run in their original execution modes (`VEE 3` or `VEE 4`). These changes are important to know if you plan to convert programs from older to newer modes.

About the VEE 5 Execution Mode

The `VEE 5` Execution Mode is a superset of the `VEE 4` mode. The `VEE 5` mode retains the compiler features described previously and introduces significant changes affecting program compatibility. Most of the changes enable support for ActiveX automation and controls.

Other changes may impact your programming techniques if you use any of the features described in this section, even if you do not use ActiveX. For information about using ActiveX in VEE, see Chapter 13, "Using ActiveX Automation Objects and Controls".

Converting Programs to VEE 5 Execution Mode

Old programs will automatically open in the appropriate old execution mode. If you want to change older programs to a newer mode, you must do this manually using `Default Preferences` under the `file` menu. When you change a program to `VEE 5` mode, errors can occur. A list appears explaining problems. You need to fix these errors before VEE 6.0 accepts the switch to `VEE 5` mode. VEE 6.0 does not automatically revise any part of your program to fix the errors.

VEE 5 Execution Mode Changes

To help you know how to fix errors, the `VEE 5` mode compatibility changes are described below.

Note

If you want to change VEE 3.x programs to `VEE 5` mode, you should be sure they work in `VEE 4` mode first and then change them to `VEE 5` mode. See “Execution Mode Changes: VEE 3 to VEE 4” on page 13 for help with that conversion.

Menu Changes. As part of the ActiveX support added to VEE 5.0, the `Device` menu has changed slightly. These new menu items have been added:

```
ActiveX Automation References...
ActiveX Control References...
ActiveX Controls
```

Also, the menu item `Math & Functions` that opened the `Select Functions` dialog box, is now called `Function & Object Browser` and opens the `Function & Object Browser`. You still use it the same way to select math operators and functions for a program, and its expanded functionality supports ActiveX.

Expressions. The following changes affect objects that contain expressions, such as `Formula`:

- SET and ByRef are new keywords that are used for ActiveX automation. They are reserved and cannot be used as names for terminals.
- New syntax is supported for ActiveX automation such as `excel.worksheets(1).cells(1,2) = 2`.
- In `VEE 3` and `VEE 4` modes, expressions with array syntax entered without commas, such as `[1 2 3]`, are reparsed when the program loads and automatically modified to use commas, as in `[1,2,3]`. In VEE 5 and higher modes, entering array syntax without commas, such as `[1 2 3]` will cause an error when `Formula` loses focus.
- A value such as `1` returns an `INT32`, `1.0` returns a `REAL64`. Previously, both returned a `REAL64`.

- There are two new built-in functions for ActiveX automation: `CreateObject()` and `GetObject()`.
- There are two new built-in constants for ActiveX automation: `true` and `false`.

Variables. The following changes affect variables:

- When `Delete Variables at PreRun` is turned on (in `Default Preferences`), global variables are not deleted if they reference ActiveX controls.
- The `Declare Variable` object has a new variable type called Object which is used for ActiveX automation.
- The new Object variable type is also available on input terminals as a `Required Type`, though it cannot be coerced from or to another type.

Global Namespace. Global namespace rules have changed, which affects names given to variables, functions, and libraries in the following ways:

- Local UserFunctions, Library names, global declared and undeclared variables, and local-to-library declared variables are now all in the same name space and must have unique names. This affects existing programs if they contain more than one instance of a name. For example, you cannot have a UserFunction and a declared global variable both named `daily_results`. This will cause an error when you switch the program to `VEE 5` mode.
- Within a Library, local UserFunctions and local-to-library declared variables are in the same namespace and must have unique names. This will cause an error when you switch the program to `VEE 5` mode, or if you import a Library containing conflicting names into a `VEE 5` mode program.

- New syntax is allowed in the `Formula` object in all modes, such as

  ```
  lib.func(a,b) = RightHandExpr
  ```

  This parses correctly in all modes. However, it executes correctly only in `VEE 5` and higher mode and causes a run-time error in `VEE 3` and `VEE 4` modes.

The changes in global namespace rules also change the order of precedence used in `VEE 5` and higher mode to the following order when VEE looks up variable and function names used in a `Formula`:

1. Local input/output terminals.
2. Declared local-to-context variables.
3. Declared local-to-library variables when inside a UserObject context nested in a UserFunction context.
4. Global declared and undeclared variables, local UserFunctions, Library names, which all must be unique names.
5. Built-in functions, such as `sin()` and `totSize()`.
6. ActiveX controls and automation constants depending on which libraries have been referenced using `Instrument` ⇒ `ActiveX Automation References` or `ActiveX Control References.` For example, many constants exist in Excel's automation library, such as `xlMaximized`).
7. Imported UserFunctions, Compiled Functions, and Remote Functions appear in random order. To guarantee getting the correct one, include the imported Library's name, as in `myLib.func()`.

An unlikely example of how this new order can cause an older program to fail might involve a `Formula` containing the expression `sin(90)` with a data input terminal (a variable) named `sin`. In `VEE 3` and `VEE 4` modes, VEE ignores the input terminal name and calls the `sin()` built-in function.

However, VEE 5 and higher mode uses the new precedence order to look up the function and variable names. So VEE 6.0 looks up the terminal name, assumes it has an ActiveX object on the input, and tries to call the object's default method. An expression that calls an ActiveX object's default method, cells(1,1), is similar to sin(90). For information about ActiveX, see Chapter 13, "Using ActiveX Automation Objects and Controls"

READ TEXT Transactions. In VEE 3 and VEE 4 modes, the READ TEXT transaction using the TOKEN format with EXCLUDE CHARS does not advance the read pointer to exclude the specified character. Figure 1-8 shows an example of this in VEE 4 mode:

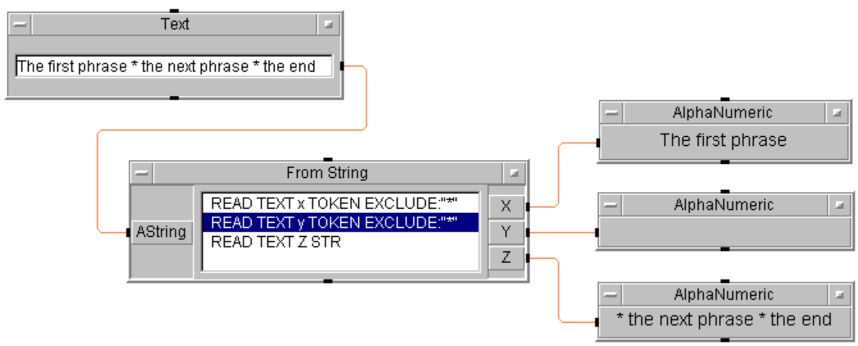


Figure 1-8. READ TEXT Transaction with TOKEN in VEE 4 Mode

This is an unexpected result. An expected result is for each phrase separated by the excluded character "*" to appear in separate AlphaNumeric displays, as shown in the VEE 5 mode example in Figure 1-9.

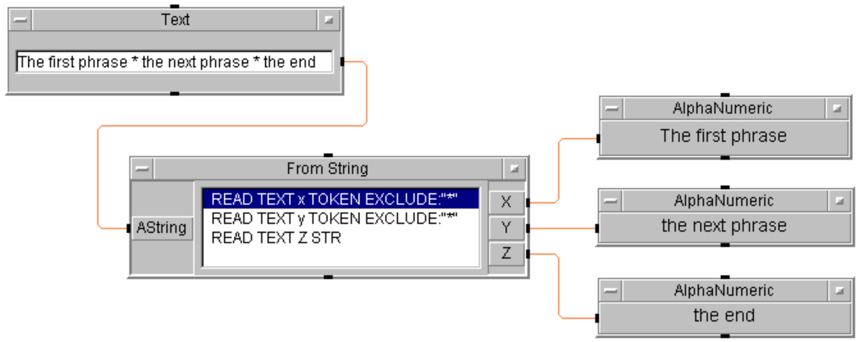


Figure 1-9. READ TEXT Transaction with TOKEN in `VEE 5` Mode

Interaction Between `To/From File` and `To/From DataSet`. In `VEE 3` and `VEE 4` modes, a program using a `To File` or `From File` object with the `EXECUTE REWIND` transaction to access the same data file as a `To DataSet` or `From DataSet` object can cause unexpected interactions. More specifically, if a program uses `From File` (with `EXECUTE REWIND`) to *read* data from a file, then uses `To DataSet` to *write* data back into the same file, the data can be written incorrectly.

A similar interaction can happen when using `From DataSet` with `To File`. In `VEE 5` and higher mode, this unexpected interaction is fixed so the data is written to the file correctly. However, we *still* do not recommend mixing `To/From File` with `To/From DataSet` operations on the same file.

Execution Mode Changes: VEE 5 to VEE 6

In VEE 6.0, the `VEE 4` and `VEE 3` modes retain their compatibility definitions set in VEE 4.0, which are described in “Execution Mode Changes: VEE 3 to VEE 4” on page 13. There are minor changes that will not affect existing programs that run in their original execution modes (`VEE 3` or `VEE 4`). These changes are important to know if you plan to convert programs from older to newer modes. They are described below.

About the VEE 5 Execution Mode

The `VEE 5` Execution Mode is a superset of the `VEE 4` mode. The `VEE 5` mode retains the compiler features described previously and introduces significant changes affecting program compatibility. Most of the changes enable support for ActiveX automation and controls.

Other changes may impact your programming techniques if you use any of the features described in this section, even if you do not use ActiveX. For information about using ActiveX in VEE, see Chapter 13, “Using ActiveX Automation Objects and Controls”.

New Data Types

`Int16`, `Real32`, `Variant`, and `UInt8` are new data types for VEE 6.0. All new data types and new transactions such as WRITE TEXT INT16 appear in all Execution Modes. However, new transactions behave the old way in old modes.

For example, in `VEE 5` mode, WRITE BINARY INT16 actually does a WRITE BINARY INT32 and will not convert the data to an `Int16`. In `VEE 6` mode, WRITE BINARY INT16 does convert data to an `Int16`. See “Setting Execution Modes” on page 10 for ways that the VEE Execution Mode could change program behavior.

Variant to VEE Data Type Conversion - Improved Array Handling

When data are returned from an ActiveX Automation Server (such as Excel) or an ActiveX control, VEE must convert the automation data types to VEE data types. With VEE 5.0, an array of Variants converted into a VEE Record. With VEE 6.0 (in `VEE 6` Execution Mode), an array of Variants converts into a VEE array if all elements are of the same data type. (For mixed data types, there is no change from VEE 5.0 behavior.)

If all elements of an array are of the same data type, mapping of `Variant` data type to VEE array data type is as follows.

Note This feature is available in `VEE 6` Execution Mode only.

Variant array member type	VEE 6.0 data type
VT_UI1	UInt8
VT_BOOL	Int16
VT_I2	Int16
VT_UI2	Int16
VT_I4	Int32
VT_UI4	Int32
VT_R4	Real32
VT_R8	Real64
VT_DATE	Real64
VT_CY	Real64
VT_BSTR	Text
VT_DISPATCH	Object

However, there are still some gaps in this compatibility between new VEE 6 data types and ActiveX automation servers:

- Data types such as `Boolean` (VT_BOOL), `Date` (VT_DATE), `Currency` (VT_CY), and `Error` (VT_ERROR) do not have built-in VEE data type counterparts. Use of these data types with "ByRef" in ActiveX is supported with the Set functions and Query functions described below.
- Certain special Variant values such as `Empty` (VT_EMPTY) and NULL (VT_NULL) have no equivalent and cannot be uniquely identified.

Set Functions. The Set functions tell VEE that during ActiveX automation operations the containers returned by these functions will be given special treatment.The set functions are:

asVariantBool()
asVariantCurrency()
asVariantDate()
asVariantError()
asVariantEmpty()
asVariantNull()

Query Functions. Query functions are used on containers created from the return values of automation methods and properties. The Query functions are:

isVariantBool()
isVariantCurrency()
isVariantDate()
isVariantError()
isVariantEmpty()
isVariantNull()

Updated Functions

The following functions have been updated for VEE 6.0.

whichOS() — updated with return values of "Windows_98" and "Windows_2000".

createObject() — updated with an optional second parameter that specifies the name of a remote host computer.

Related Reading

For more detailed information about instrument control topics discussed in this manual, refer to the following publications.

- *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 a GPIB (IEEE 488.1) interface. This standard contains electrical specifications and information on protocol that is beyond the needs of most programmers.

- *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).

- *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.

- *HP VISA User's Guide* (Hewlett-Packard Company, 1998), part number E2090-90035.

  This document is useful for users who create their own VXI*plug&play* drivers and provides additional information about addressing and using VXI*plug&play* drivers.

2

Instrument Control Fundamentals

Instrument Control Fundamentals

VEE supports five types of objects for controlling instruments. Figure 2-1 shows each of these objects in its open view. Each of these examples communicates with an HP E1410A VXI Multimeter, except the PC PlugIn card driver object.

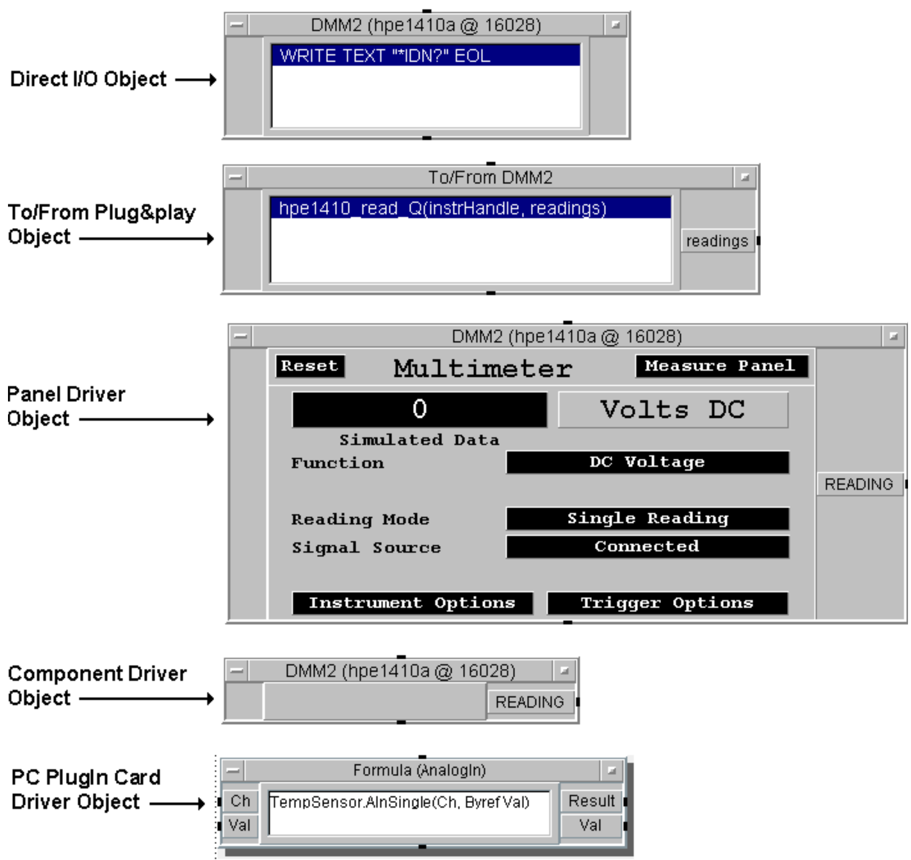


Figure 2-1. VEE Instrument Control Objects

Table 2-1 gives an overview of the differences between these instrument control objects.

Table 2-1. Comparing Instrument Control Objects in VEE

VEE Object	Instrument Access	Main Benefits	Supported Interfaces[a]
`Direct I/O`	Communicates directly with any instrument.	Fast I/O. Can control any instrument.	GPIB, Serial, GPIO, VXI, and LAN.
`To/From VXIplug&play`	Requires a VXI*plug&play* driver supplied from the instrument manufacturer specific to each platform. Requires VISA to be installed.	Fast I/O. Drivers can be used by multiple software applications.	GPIB, VXI, and Serial.
`Panel Driver`	Requires an Instrument Panel Driver supplied with VEE.[b]	Easy to use.	GPIB and VXI.
`Component Driver`	Requires an Instrument Panel Driver supplied with VEE.	Faster I/O than `Panel Driver`.	GPIB and VXI.
`PC Plugin Card Driver`	Requires an ODAS driver supplied by the instrument manufacturer.	Fast I/O. Drivers can be used by multiple software applications.	PC plugin slots

a. HP-IB is Hewlett-Packard’s implementation of the IEEE-488 interface bus standard. Other implementations are called GPIB. LAN interface support does not include purely LAN-based instruments.
b. Panel Drivers are also sometimes called "VEE drivers."

The `To/From VXIplug&play`, `Panel Driver`, `Component Driver`, and `PC Plugin Driver` objects 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`.

Note

You can use all five methods to communicate with different instruments within a VEE program. However, do not use VXI*plug&play* drivers along with any of the other methods to communicate with the same instrument in the same program — unexpected results may occur.

Introduction to Direct I/O

Direct I/O objects allow you to read and write arbitrary instrument data in much the same way you read from and write to files. This allows you full access to any programmable feature of any instrument. No instrument driver file is required, but you must have a detailed understanding of your instrument's programming commands to use Direct I/O. In order to use Direct I/O to communicate with GPIB, VXI, or Serial devices, the I/O libraries must be installed as described in *Installing the Agilent I/O Libraries (VEE for Windows)*.

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 VEE. You can recall the measurement state from within your program by using the Direct I/O object.

An Example of Direct I/O

Figure 2-2 shows a Direct I/O object set up to obtain the identification string from an HP 34401A Multimeter:

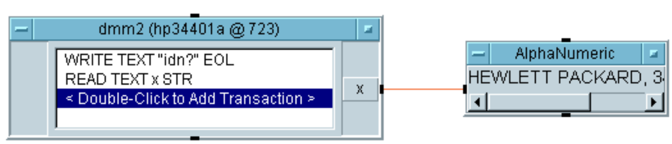


Figure 2-2. Using Direct I/O to Identify an Instrument

The first transaction in the Direct I/O object writes the text string *IDN? to the HP 34401A at GPIB address 722. This causes the HP 34401A to send the identification string, which is read by the second transaction and output to the AlphaNumeric object.

For information about how to *configure* VEE to use Direct I/O, see Chapter 3, "Configuring Instruments". For details about how to *use* the Direct I/O object, see Chapter 4, "Using Transaction I/O".

MultiInstrument Direct I/O

The MultiInstrument Direct I/O object lets you control several instruments from a single object using direct I/O transactions. This object looks the same as the `Direct I/O` object, except that each transaction in the `MultiInstrument Direct I/O` object can address a separate instrument.

The object is a standard transaction object, and works with all interfaces that VEE supports. Since the `MultiInstrument Direct I/O` object does not necessarily control a single instrument, the title does not list an instrument name, address, or live mode condition.

By using the `MultiInstrument Direct I/O`, you can reduce the number of instrument-specific `Direct I/O` objects in your program. The resulting performance increase is especially important for the VXI interface, which is faster than GPIB at instrument control.

Figure 2-3 shows the `MultiInstrument Direct I/O` object and its `I/O Transaction` dialog box. The object is being set up to communicate with an HP E1413B, HP E1328, and HP 3325.

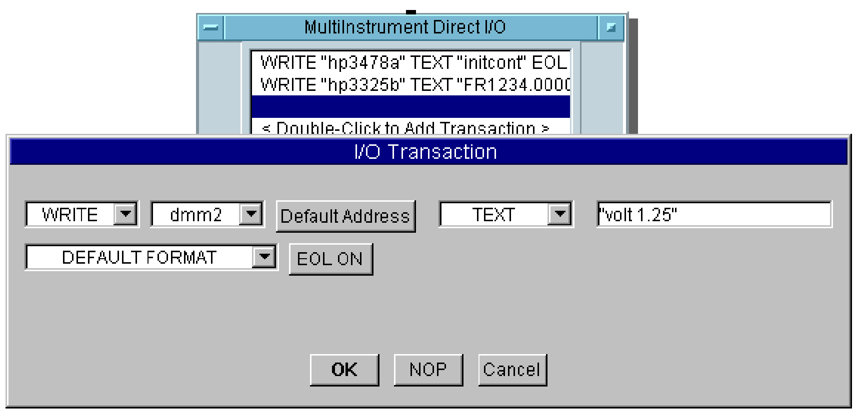


Figure 2-3. MultiInstrument Direct I/O Controlling Several Instruments

For further information about using the `MultiInstrument Direct I/O` object, see “Using the MultiInstrument Direct I/O Object” on page 141.

Introduction to VXI*plug&play*

VXI*plug&play* is an interface specification that allows multiple vendors to supply compatible hardware and software. A VXI*plug&play* driver is a library of functions for controlling a specific instrument. The driver is written by the hardware vendor and shipped with the instrument.

VXI*plug&play* drivers can be written for non-VXI instruments. VEE Version 3.2 and later supports drivers that comply with the WIN95/98, WINNT, VXI*plug&play* specification version 3.0 or later.

Getting Started

Before you can get started with VXI*plug&play*, you must have completed these steps:

1. Install the interface (GPIB or VXI).

2. Install VISA. If you are using an Agilent interface card use VISA as supplied with VEE. See *Installing the Agilent I/O Libraries (VEE for Windows)* for details. Otherwise, you must install VISA as supplied with the interface card.

3. Configure VISA for each hardware interface. If you are using an Agilent interface card follow the instructions in *Installing the Agilent I/O Libraries (VEE for Windows)*. Otherwise, you must configure VISA as specified by the interface manufacturer.

Note

VISA (Virtual Instrument Software Architecture) is an I/O library that VXI*plug&play* drivers use to control instruments. VISA is required for VXI*plug&play* and provides VISA function calls which are used by the VXI*plug&play* drivers.

What You Need

VEE needs these four files for each VXI*plug&play* driver.

- The library file
- The function panel file
- The header file
- The help file

The files installed with each VXI*plug&play* driver always include these files. Other files are also installed.

Note

Not all VXI*plug&play* drivers support all frameworks (platforms). Also, certain versions of VISA may not be supported on all frameworks. Please check with the appropriate vendor.

Installing the VXI*plug&play* Driver Software

To install the set of files needed for each driver, follow the instructions included with the driver by the instrument manufacturer.

Location of Files (WIN95 and WINNT Frameworks)

The VXI*plug&play* files are located under the `WIN95\` or `WIN98\` directory or the `WINNT\` directory. This location is relative to the root drive and directory value stored in the registry by the VISA installation. The default value for the root drive and directory is `C:\VXIPNP`.

Table 2-2 shows the VXI*plug&play* driver files needed by VEE:

Table 2-2. Location of WIN95 and WINNT Framework Driver Files

Filename[a]	Location	Purpose
PREFIX`_32.DLL`	`BIN`	Instrument driver library
PREFIX`.FP`	*PREFIX*	Instrument driver function panel file
PREFIX`.H`	`INCLUDE`	Instrument driver header file
PREFIX`.HLP`	*PREFIX*	Instrument driver help file

a. *PREFIX* refers to the name of the instrument such as `HPE1410`.

Summary of Terminology

Working with VXI*plug&play* drivers is different than using other types of I/O with VEE. Here is a summary of how the different pieces fit together.

- The VEE program calls VXI*plug&play* functions.
- The functions (that have parameters that may be set via function panels) are part of the VXI*plug&play* driver. The functions talk to the instrument through the VISA software.
- The instrument passes data back through VISA and into the function parameters.

A VXIplug&play Example Program

Figure 2-4 shows an example program that uses the `To/From VXIplug&play` object to initiate a voltage measurement and to obtain a reading from the HP E1410A Multimeter.

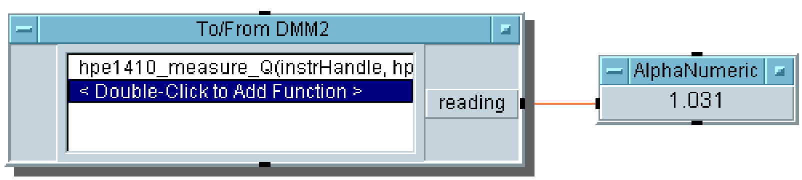


Figure 2-4. Using the To/From VXIplug&play Driver Object

Further Information

For information about how to **configure** VEE to use VXI*plug&play*, see Chapter 3, “Configuring Instruments”. For further information about how to *use* VXI*plug&play* in VEE, see Chapter 7, “Using VXIplug&play Drivers”.

Introduction to Panel Drivers and Component Drivers

`Panel Driver` and `Component Driver` objects can be used for a particular instrument only if there is a **driver file** to support that instrument. The installation procedure for VEE for Windows 95/98 and Windows NT allows you to select which drivers you want to install. Chapter 3, “Configuring Instruments” describes how to select and configure the proper driver files for your instruments. Also, the I/O libraries must be installed as described in *Installing the Agilent I/O Libraries (VEE for Windows).*

Panel Drivers

`Panel Drivers` serve two purposes in VEE:

- They allow you to define a measurement state that specifies all the instrument function settings. When a `Panel Driver` operates, the corresponding physical instrument is automatically programmed to match the settings defined in the `Panel Driver`.
- They act as instrument 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 2-1, the open-view of a `Panel Driver` 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 the fields in the graphical control panel. You can also make measurements and display the results by clicking the 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 benefit if you want to develop programs before instruments are purchased or while they are being used elsewhere.

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

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

Figure 2-5 shows the two `Panel Drivers` that provide the desired signals.

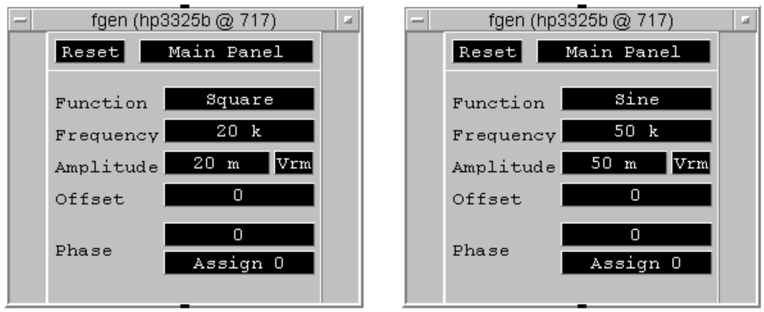


Figure 2-5. Two HP3325B Panel Drivers

Component Drivers

In an 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 a `Panel Driver`, 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 `Panel Drivers`. 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 10 kHz and an amplitude determined by operator input.

2. Sweep the frequency output from 10 kHz to 1 MHz using 5 steps per decade.

In this case, it makes sense to use a `Panel Driver` to perform the initial setup and a `Component Driver` to repeatedly set the output frequency. Figure 2-6 shows a program that does this.

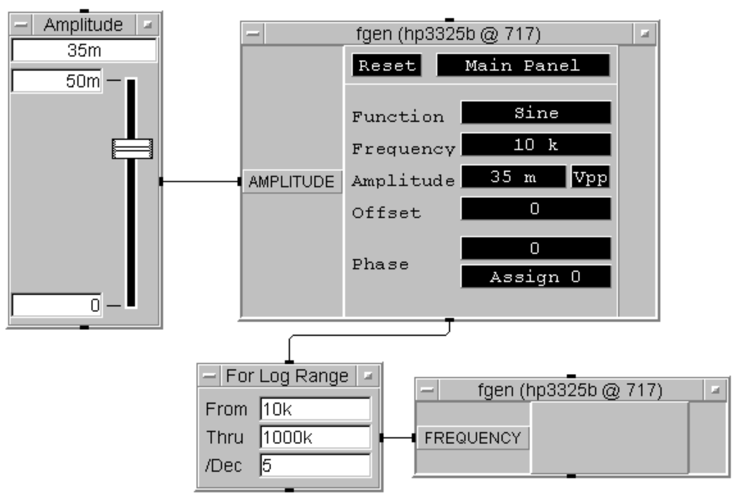


Figure 2-6. Combining Panel Drivers and Component Drivers

Further Information

For information about how to configure VEE, see Chapter 3, “Configuring Instruments”. For further information about how to *use* the `Panel Driver` and `Component Driver` objects, see Chapter 6, “Using Panel Driver and Component Driver Objects”.

Support For Register-Based VXI Devices

When using the instrument control objects to directly address VXI devices on the VXI backplane, you need to know whether devices are **message-based** or **register-based**. VEE communicates with message-based devices by means of SCPI (Standard Commands for Programmable Instruments) messages.

VEE also provides Interpreted SCPI (I-SCPI) support for most Hewlett-Packard and Agilent register-based devices. I-SCPI drivers let you communicate with register-based devices as though they were message-based. This means that a VEE program can communicate with a register-based device using standard SCPI messages, provided there is an I-SCPI driver for that particular device. If no I-SCPI driver is available for a register-based device, VEE must communicate with that device by directly accessing its registers.

The I-SCPI drivers give you the flexibility to use any of the instrument control objects you prefer. You can use the `Panel Driver` for easier programming, or use SCPI commands in `Direct I/O` for faster execution speed. When you program VEE to communicate with a register-based device using SCPI messages, VEE will inform you if the required I-SCPI driver is not available. In that case, you will need to access the device registers directly using `Direct I/O` or `MultiInstrument Direct I/O`.

3

Configuring Instruments

Configuring Instruments

This chapter shows how to configure VEE to communicate with your instruments using the following methods:

1. By means of `Direct I/O` objects (no instrument driver is required).

2. By means of VXI*plug&play* drivers using `To/From VXIplug&play` objects.

3. By means of Agilent Panel Drivers ("IDs") using either `Panel Driver` or `Component Driver` objects.

4. By means of `Formula` objects using ODAS PC PlugIn card drivers. VEE 6.0 supports PC PlugIn cards with ODAS (Open Data Acquisition Standard) compliant software drivers.

The VEE `Instrument Manager` dialog provides a unified method to select and configure all of these instrument-control objects.

For VEE to communicate with instruments, you must first install the Agilent I/O Libraries as described in *Installing the Agilent I/O Libraries (VEE for Windows)*. The Agilent SICL libraries let you use `Panel Driver`, `Component Driver`, or `Direct I/O` objects. The VISA libraries let you use `To/From VXIplug&play` objects.

To use `Panel Driver` or `Component Driver` objects, you must install the appropriate Panel Drivers. For VEE for Windows, you can install any desired selection of Instrument Drivers during the VEE installation. (No instrument drivers are required for `Direct I/O` objects.)

VXI*plug&play* drivers are supplied by the instrument manufacturer with many VXI instruments. To use a `To/From VXIplug&play` object, you must install the appropriate VXI*plug&play* driver files, following the instructions provided with the driver. For further information about VXI*plug&play* drivers, see Chapter 7, “Using VXIplug&play Drivers”.

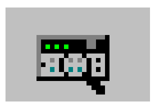
ODAS PCPI card drivers are supplied by the instrument manufacturer with many PC instruments. To use an ODAS PCPI card driver object, you must install the appropriate ODAS PCPI driver files, following the instructions provided with the driver.

Using the Instrument Manager

This section provides an overview of how to use the `Instrument Manager` and the configuration dialog boxes to find and configure instruments in VEE. Some examples are given and, for many applications, you can use the default values for most parameters. However, see “Details of the Properties Dialog Boxes” on page 75 for details of the configuration fields in these dialog boxes.

Overview

To configure an instrument, select `I/O` ⇒ `Instrument Manager` or click on the `Instrument Manager` button in the toolbar.

It looks like this: 

The `Instrument Manager` dialog box appears. It has no instruments until you find and add them, as Figure 3-1 shows.

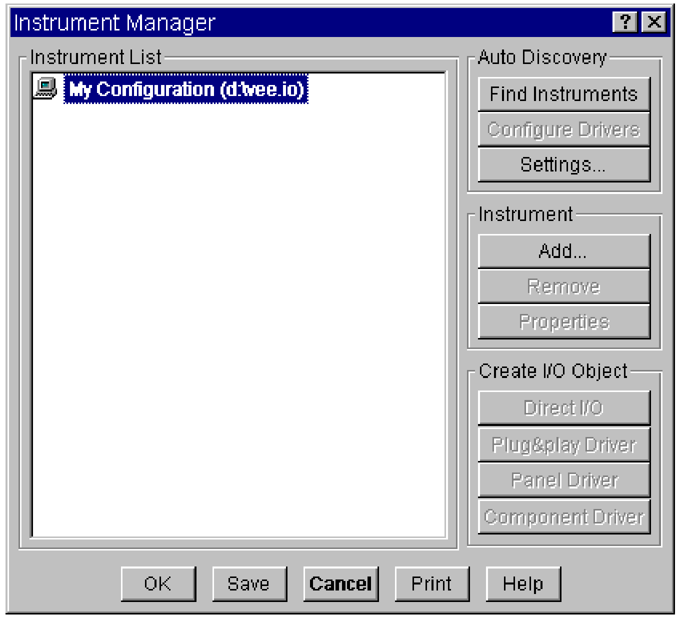


Figure 3-1. The Instrument Manager Dialog Box

The `Instrument Manager` displays four sections:

- `Auto Discovery` buttons allow you to find instruments and configure drivers for them. If you click on the `Find Instruments` button, VEE automatically updates all configured GPIB and VXI instruments and displays any other GPIB and VXI instruments connected to your computer. If you click on the `Settings` button, VEE displays the `Auto Discovery Settings` dialog box, described in the next section. VEE OneLab can find a maximum of four instruments.

- `Instrument List` displays the instruments that are currently configured. This configuration is defined by the I/O configuration file (see “The I/O Configuration File” on page 149 for further information). The default configuration is blank (empty).

- `Instrument` buttons allow you to modify the instrument configuration. The `Instrument` button actions are described in more detail later in this chapter.

- `Create I/O Object` buttons allow you to select `Direct I/O`, `Plug&play Driver`, `Panel Driver`, and `Component Driver` objects and place them in your program.

Auto Discovery

The `Auto Discovery` area contains three buttons: `Find Instruments`, `Configure Drivers`, and `Settings`.

- The `Find Instruments` button updates any existing GPIB and VXI instrument configurations and adds any unconfigured GPIB and VXI instruments connected to your computer to the `Instrument List`. `Find Instruments` also finds and adds any Serial and GPIO interfaces to the `Instrument List`, but not the instruments connected to them.

- The `Configure Drivers` button configures drivers for instruments already found and in the `Instrument List`.

- The `Settings` button allows you to determine how instruments and drivers are configured.

With `My Configuration` highlighted in the `Instrument List`, click `Find Instruments` to update all existing GBIB and VXI instrument configurations and to add any unconfigured GPIB and VXI instruments to the list. Live mode is turned on for instruments that are found and are powered up. (Live mode settings are *not* switched from on to off if configured instruments are not found.)

Next click `Settings`, to bring up the dialog box that allows you to control how instruments and drivers are detected and configured. This box has two sections: `Find Instruments` and `Instrument Identification`.

The `Find Instruments` section has two radio buttons:

- ❑ Detect only
- ❑ Detect, identify, and configure drivers for each instrument.

If "Detect only" is checked, VEE detects all live bus addresses when you click the `Find Instruments` button. If "Detect, identify, and configure drivers for all instruments" is checked, VEE detects all live bus addresses, sends "*IDN?" to all detected instruments, and tries to configure drivers for each instrument.

The lower section controls the `Configure Drivers` button. If "Ask before sending "*IDN?" to each instrument?" box is checked, VEE stops before configuring each driver and asks if you want to proceed. If this box is not checked, VEE automatically configures each driver.

The Instrument List

If `Find Instruments` found one instrument connected to your computer, the `Instrument Manager` might look like Figure 3-2. In this example, `Find Instruments` found a Serial Interface but does not show any instruments that may be connected to it. Newly discovered instruments are named "newInstrument", "newInstrument1", etc. You can give them more descriptive names, as shown later.

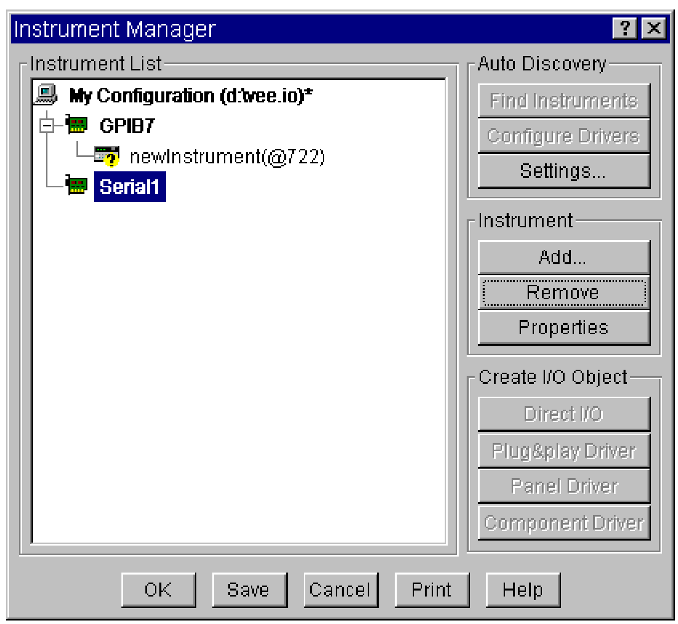


Figure 3-2. The Instrument List

To use the Instrument Manager, click the GPIB7 Interface selection. It becomes highlighted and the Properties button becomes active. Click the [-] icon in front of GPIB7 to "collapse" the selections under it. Figure 3-3 shows the collapsed configuration.

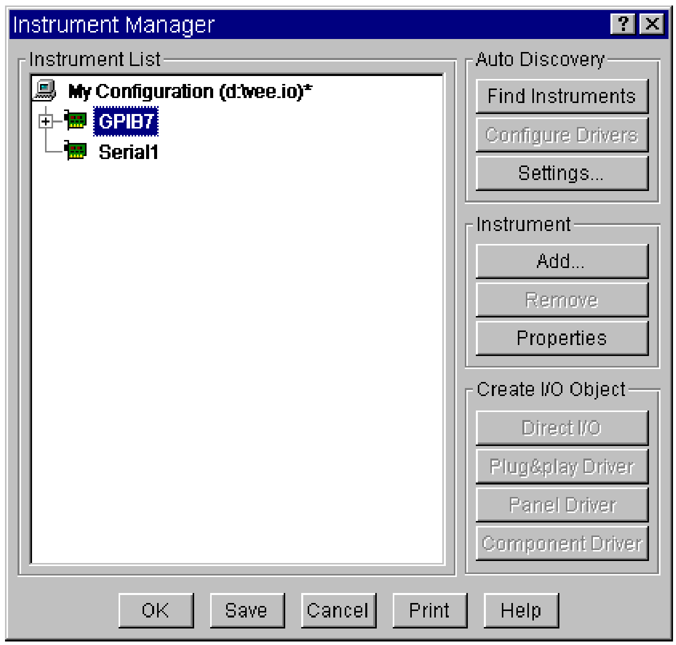


Figure 3-3. Collapsing the GPIB7 Interface Configuration

To "expand" the selections again, click the [+] icon in front of GPIB7. (To expand the entire tree, select My configuration and press the * key.) Now click the selection newInstrument@722 or the "instrument" icon in front of it to highlight it. Figure 3-4 shows how the window looks.

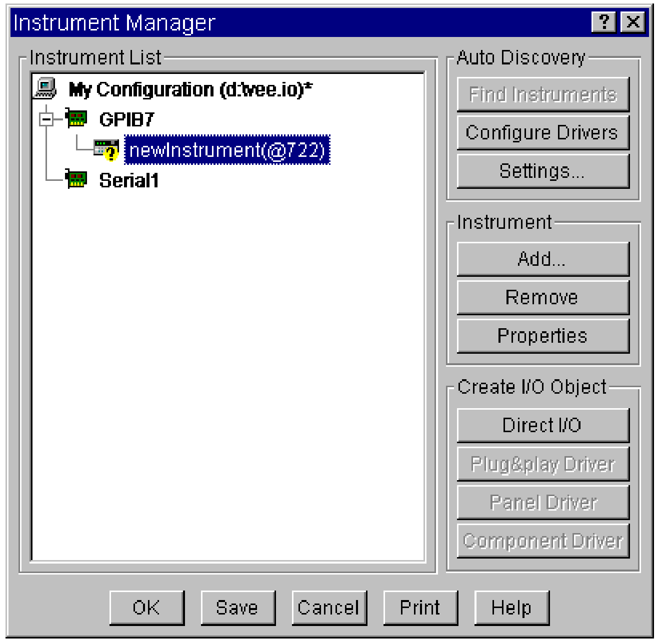


Figure 3-4. Selecting an Instrument for Configuration

Instrument Configuration

Note that all of the buttons under Instrument are now active, including `Properties`. This means you can delete, edit, or manually configure the configuration of the existing instrument or add a new instrument to the list.

Also, note that one of the buttons under `Create I/O Object` is now active. This means you can select and place a `Direct I/O Object` for the instrument. With other instrument configurations, the `Plug&play Driver`, `Panel Driver`, and `Component Driver` buttons may be active at this point, depending on what drivers you have installed.

Click on the `Configure Drivers` button to update the instrument configuration. The Identify Instrument dialog box appears asking if you want to send the `*IDN?` (identification) message to the instrument. Figure 3-5 shows this dialog box.

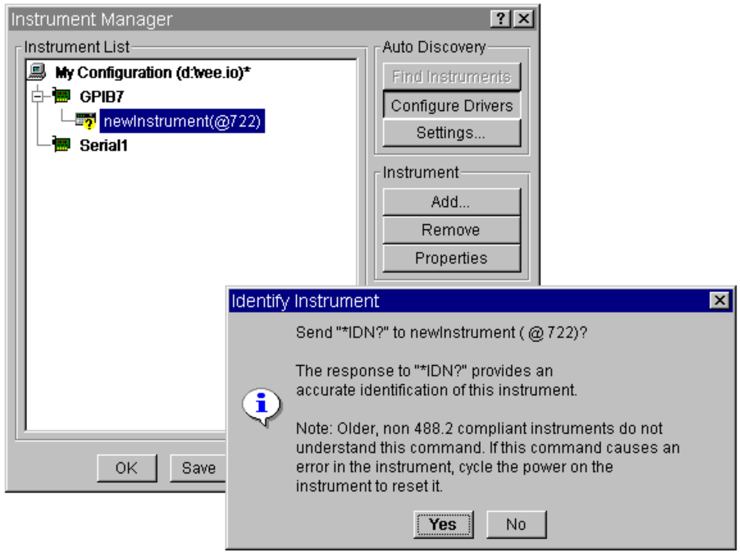


Figure 3-5. Updating the Instrument Configuration

Click `OK`. If the instrument connected to the GPIB Interface is turned on, the instrument will respond. In this example, the instrument is an HP 34401A and is turned on. Figure 3-6 shows how the `Instrument List` looks at this point.

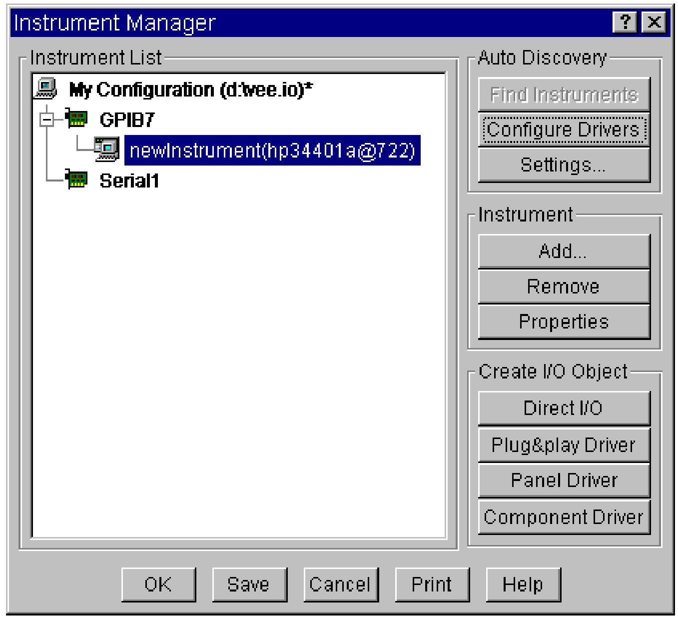


Figure 3-6. The Instrument List after Configuring Drivers

Note that two changes have occurred:

1. The instrument identification has changed to `newInstrument(hp34401a@722)`.

2. The "instrument" icon in front of `newInstrument(hp34401a@722)` has changed to show that the instrument is connected to the computer.

(If the instrument is not powered up, the identification and the icon will not change.)

Renaming an Instrument

When the instrument has been identified, you can give it a more meaningful name in the Instrument List. Click the `Properties` button to do this. When the `Properties` dialog box appears, click in the Name field and type the name you prefer. Figure 3-7 shows the name "dmm" entered to replace "newInstrument" for the HP 34401A.

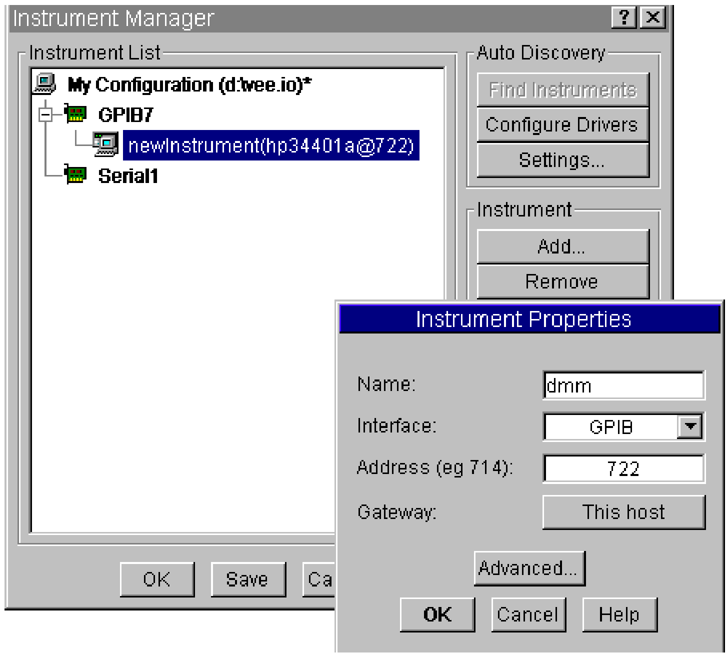


Figure 3-7. Changing an Instrument Name

Clicking OK completes the change. Figure 3-8 shows the `Instrument List` with the new name for the HP 34401A.

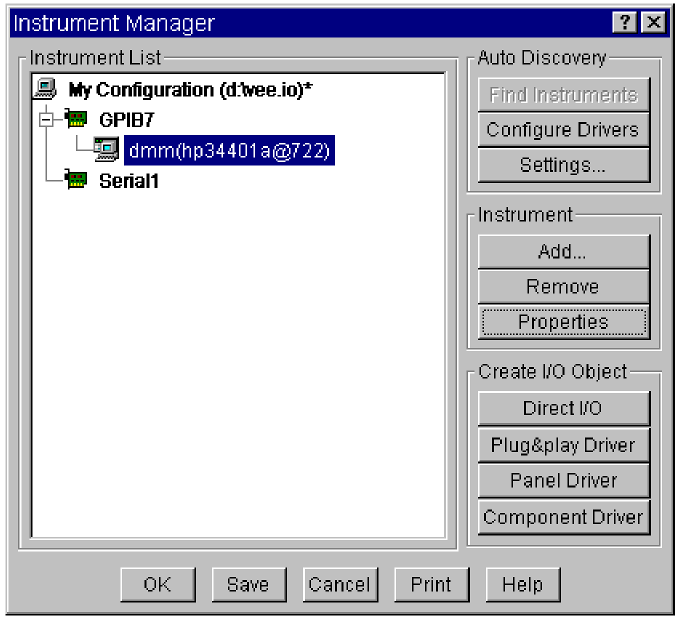


Figure 3-8. The Renamed Instrument

Adding an Instrument Configuration

To add an instrument, click the `Add...` button. The Instrument Properties dialog box appears as shown in Figure 3-9.

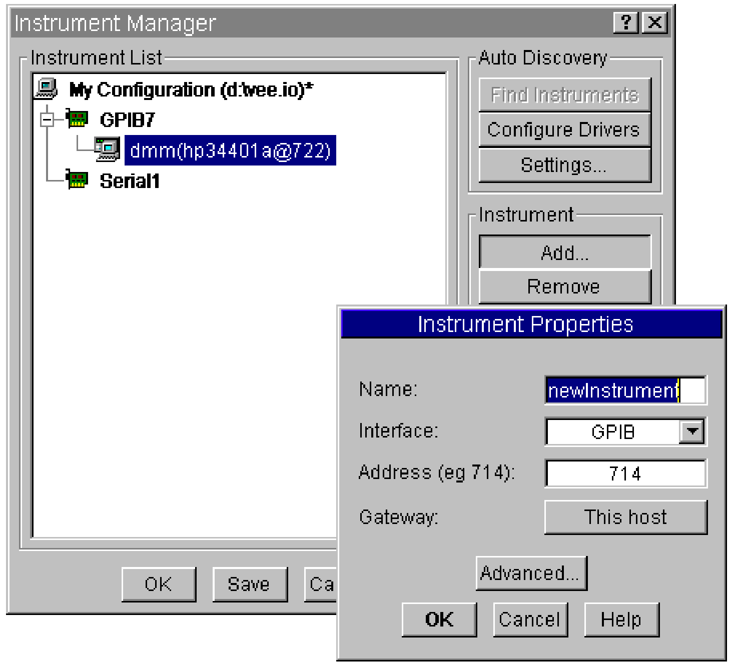


Figure 3-9. Adding an Instrument

By default, the new configuration displays the name `newInstrument`. You can type in a new name, such as `dmm2`. Leave the `Interface` field with `GPIB` selected. (If you want to change the type of interface, click the arrow to the right of `GPIB` to display the drop-down list.) Then, click the address field and change the address to `723`. Figure 3-10 shows the Instrument Properties dialog box with these changes.

Note

To move from field to field in the dialog box, click the desired field, or use the **Tab** key. If you press **Enter** or **Return**, the dialog box will exit.

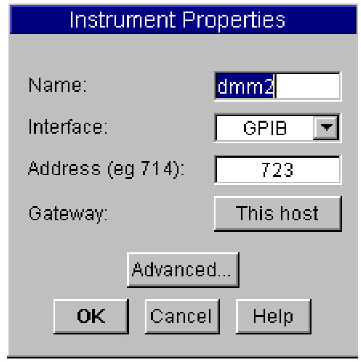


Figure 3-10. Changing the Name and Address Fields

Now click the `Advanced`... button to display the `Advanced Instrument Properties` dialog box in Figure 3-11.

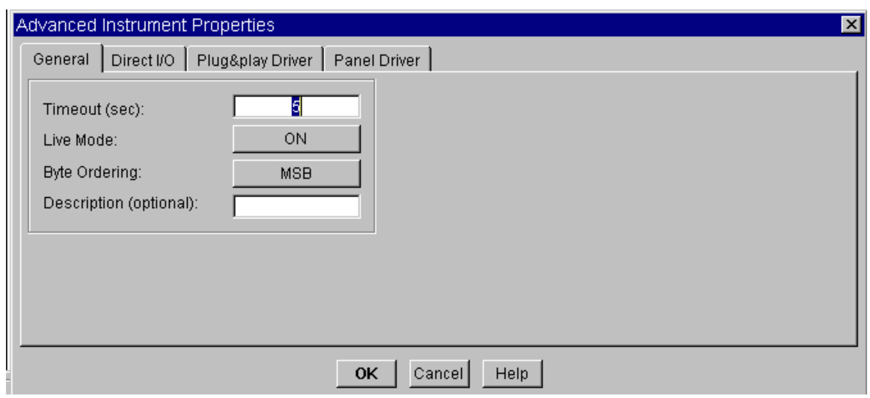


Figure 3-11. The Advanced Instrument Properties Dialog Box

The `General` tab of this dialog box allows you to specify a timeout value, to turn live mode on or off, to select byte ordering, and to add a description. Click the `Description` field and enter `hp34401a`.

Note

For further information about the individual fields in the Instrument Properties and `Advanced Instrument Properties` dialog boxes, see "Details of the Properties Dialog Boxes" on page 75.

The tabs and fields displayed in the `Advanced Instrument Properties` dialog box depend on the interface you have selected.

Now select the `Panel Driver` tab to display the dialog box shown in Figure 3-12.

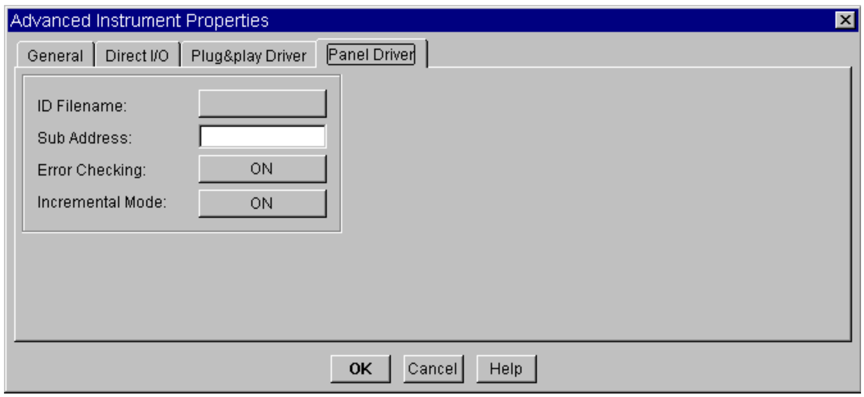


Figure 3-12. The Panel Driver Tab

Click the `ID Filename` field. You are prompted to select an Instrument Driver file. (The Windows dialog is shown in Figure 3-13.)

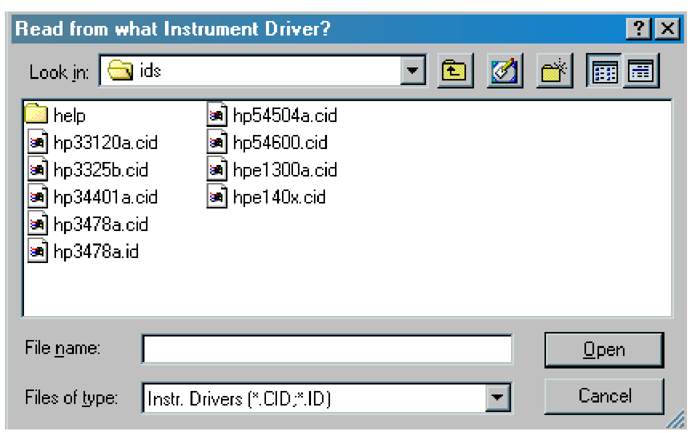


Figure 3-13. Selecting an Instrument Driver File

Double-click `hp34401a.cid` to select that file, as shown in Figure 3-14.

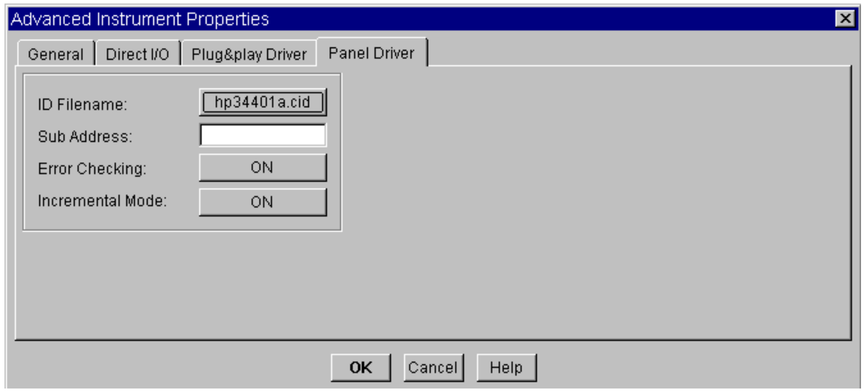


Figure 3-14. The Selected ID Filename

Now click `OK` on each dialog box to return to the `Instrument Manager` as shown in Figure 3-15.

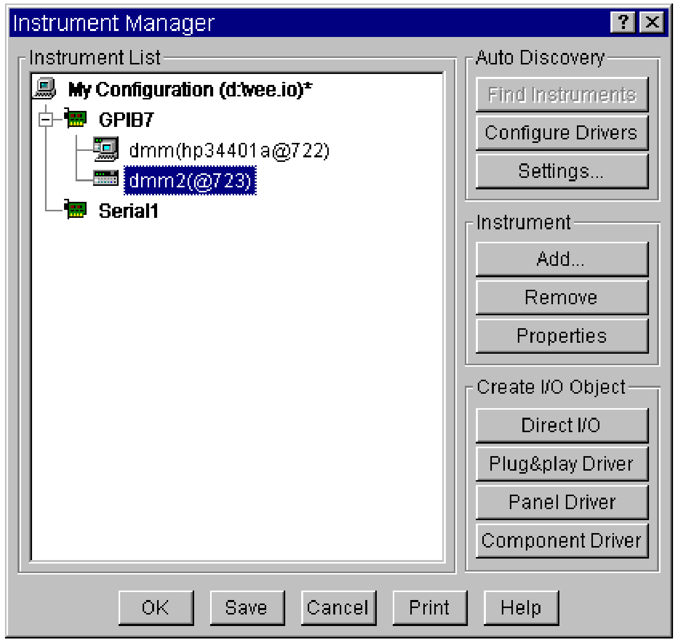


Figure 3-15. The New Configuration

At this point you can save the new configuration by clicking the `Save` button.

Adding a Panel Driver or Component Driver

When you have saved your new configuration, you can add either a `Panel Driver` object or a `Component Driver` object for `dmm2`. Select `I/O` ⇒ `Instrument Manager` to redisplay the `Instrument Manager`, as shown in Figure 3-15. Click `dmm2(@723)` if it is not already highlighted and then click the `Component Driver` button. Move the outline to the desired position in the work area, and click the mouse button to place the `Component Driver` object. The object appears as an icon as shown in Figure 3-16.

Figure 3-16. The Component Driver Object

In the same manner, if you had clicked on the `Panel Driver` button, a `Panel Driver` object would have appeared.

Editing an Instrument Configuration

You can edit an existing instrument configuration, also using the `Instrument Properties` and `Advanced Instrument Properties` dialog boxes. To edit the configuration for the HP 34401A Digital Multimeter, select `dmm(hp34401a@722)` in the `Instrument List`, and then click the `Properties`... button. The `Instrument Properties` dialog appears as shown in Figure 3-17.

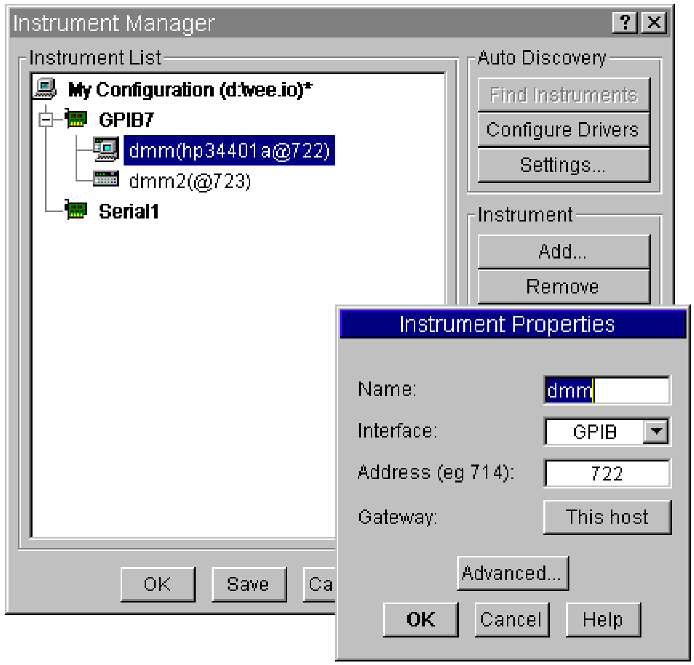


Figure 3-17. Editing the dmm Configuration

To change the configuration, modify the fields in the properties dialog boxes as described previously in “Adding an Instrument Configuration” on page 57.

Editing an Interface Configuration

You can also edit an entire Interface configuration, affecting multiple instruments. To do this, select the Interface in the `Instrument List`, and then click the `Properties` button. For example, select `GPIB7` and click the `Properties` button to get the display shown in Figure 3-18.

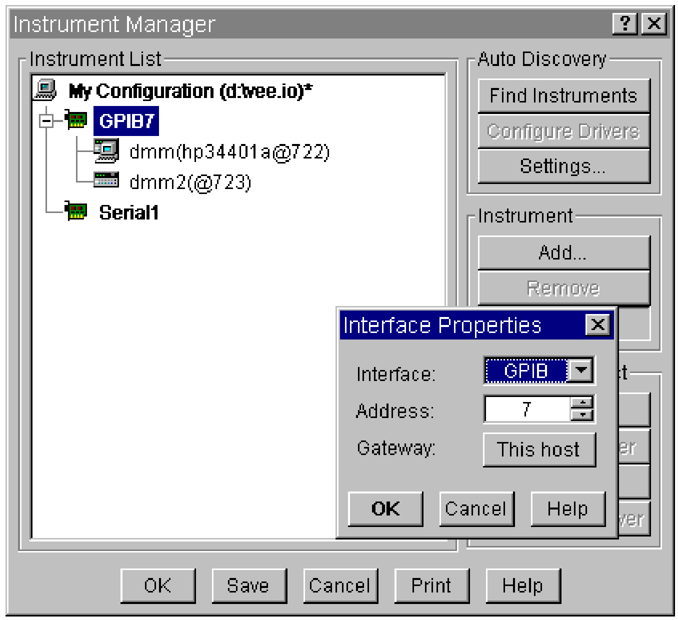


Figure 3-18. Editing the GPIB7 Configuration

Press `Cancel` to make no changes, retaining the `GPIB7` configuration for use in examples.

Note

From the `Interface Properties` dialog box, you can change the Interface type from `GPIB` to `VXI`, the address from `7` to some other unused logical unit, and you can configure a LAN gateway. Any changes will affect all of the instruments (`dmm`, `dmm2`, etc.) currently under `GPIB7`. For further information, see “Details of the Properties Dialog Boxes” on page 75.

Configuring for a Direct I/O Object

The following example shows how to configure a `Direct I/O` object. In this example, we configure a Serial Instrument at logical unit 1 (COM1) for direct I/O.

1. Select My Configuration

2. Click on Find Instruments

3. When Find Instruments is finished, select Serial1 and click on Add...

4. You should see the dialog box shown in Figure 3-19.

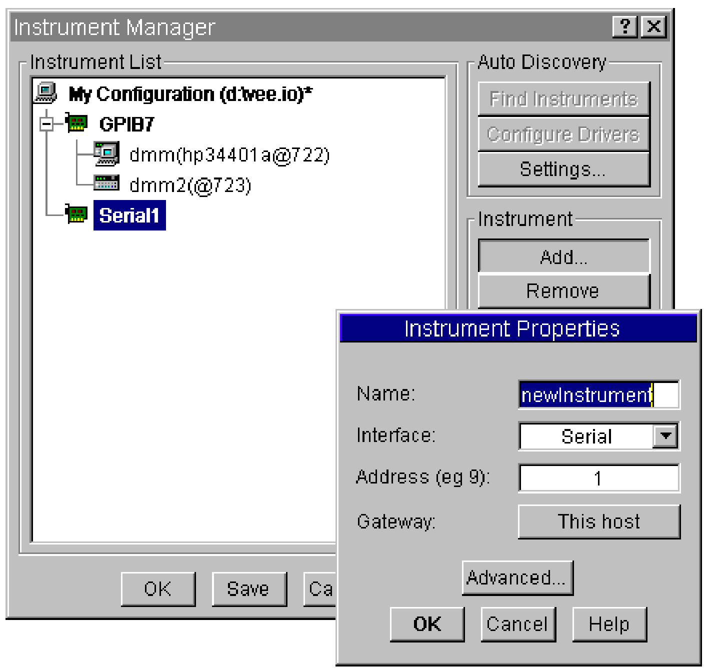


Figure 3-19. Configuring a Serial Device

The `Instrument Properties` dialog box allows you to select the name and address of the new instrument. Change the name to Serial1.

Click `Advanced...` to display the `Advanced Instrument Properties` dialog box in Figure 3-20. There are two tabs of interest.

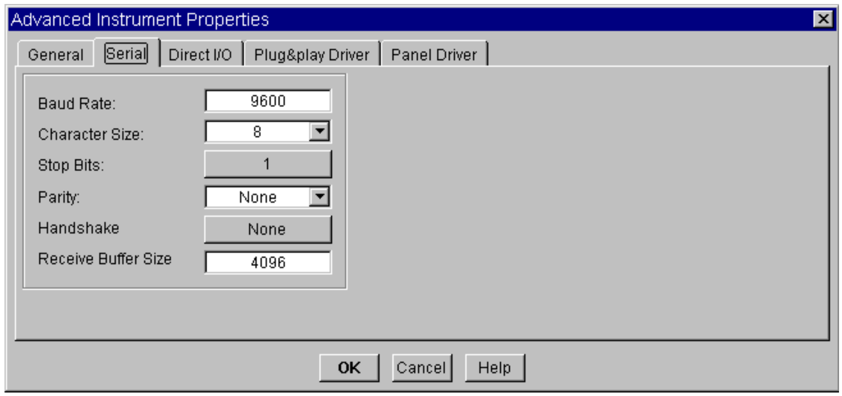


Figure 3-20. The Serial Tab

The `Serial` tab allows you to specify the serial parameters such as baud rate. See "Details of the Properties Dialog Boxes" on page 75 for further information about the individual parameters and fields. You can use the defaults for many applications.

The `Direct I/O` tab, shown in Figure 3-21, allows you to specify a number of parameters for direct I/O, including the EOL sequence. You can use the defaults for most applications.

Note

The selection of fields displayed by the `Direct I/O` tab depends on the Interface that you have selected. In addition, for VXI only there are two additional tabs, `A16 Space` and `A24/A32 Space`.

These tabs allow you to configure a VXI device's registers for `WRITE` or `READ` transactions in a `Direct I/O` object. See "Details of the Properties Dialog Boxes" on page 75 for further information about the parameters and fields displayed by each tab.

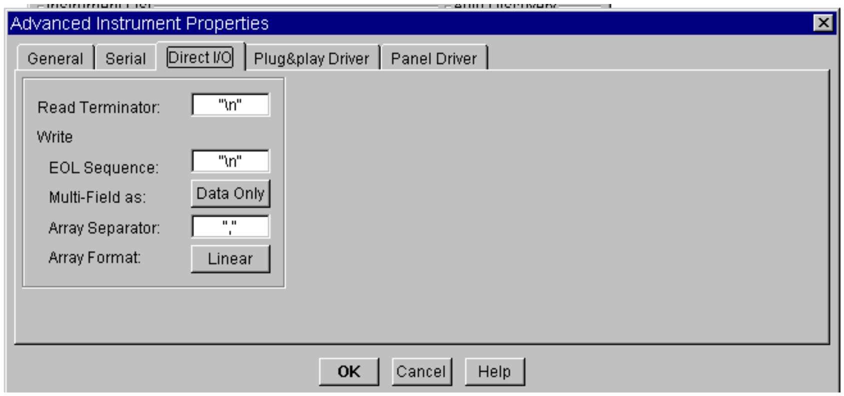


Figure 3-21. The Direct I/O Tab

Click `OK` (or `Cancel` to make no changes) on each dialog box to return to the `Instrument Manager`. In this example, a new instrument, `Serial1`, has been added under the `Serial1` interface. To add a `Direct I/O` object to the work area, click the `Direct I/O` button, place the object, and click again for the display in Figure 3-22.

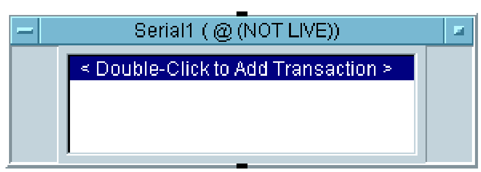


Figure 3-22. The Direct I/O Object

Note

`Direct I/O` objects use transaction-based I/O to communicate with instruments, without using an instrument driver. See Chapter 4, “Using Transaction I/O” for further information.

Configuring for a VXI*plug&play* Driver

The procedure to configure for a `To/From VXIplug&play` object is very similar to the procedures for `Panel Driver`, `Component Driver`, and `Direct I/O` objects. However, you must first install the appropriate VXI*plug&play* driver files as described in “Installing the VXIplug&play Driver Software” on page 39.

Note If you are using the Windows operating system, the VISA Assistant utility provides helpful information about VXI*plug&play* drivers. The information helps you determine valid addresses required for VXI*plug&play* driver configuration. Look for VISA Assistant in the Windows `Start` menu `Program Files` ⇒ `Agilent I/O Libraries` ⇒ `VISA Assistant`

For example, we will add a VXI*plug&play* configuration for the HP E1410A 6.5-Digit VXI Multimeter. Select `I/O` ⇒ `Instrument Manager`, and click `Add...`. The `Instrument Properties` dialog box appears. Change the name to `vxiDevice` and select `VXI` for the Interface type, as shown in Figure 3-23.

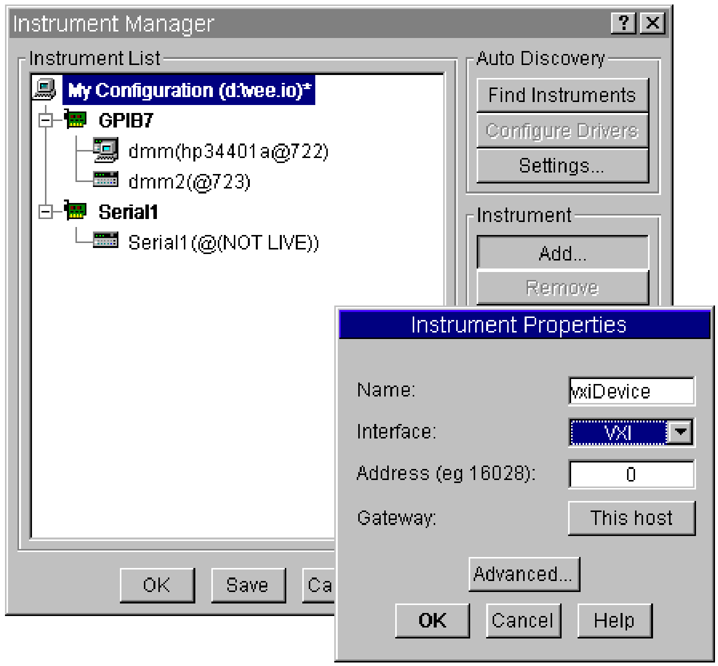


Figure 3-23. Adding a VXI Device

The `Address` field is not used for VXI*plug&play* drivers. Click `Advanced...` to display the `Advanced Instrument Properties` dialog box, and then select the `Plug&play Driver` tab.

Next, select the driver named `HPE1410` from the `Plug&play Driver Name` drop-down list, as shown in Figure 3-24. You will not be able to select the VXI*plug&play* driver unless you have previously installed the driver as described in “Installing the VXIplug&play Driver Software” on page 39.

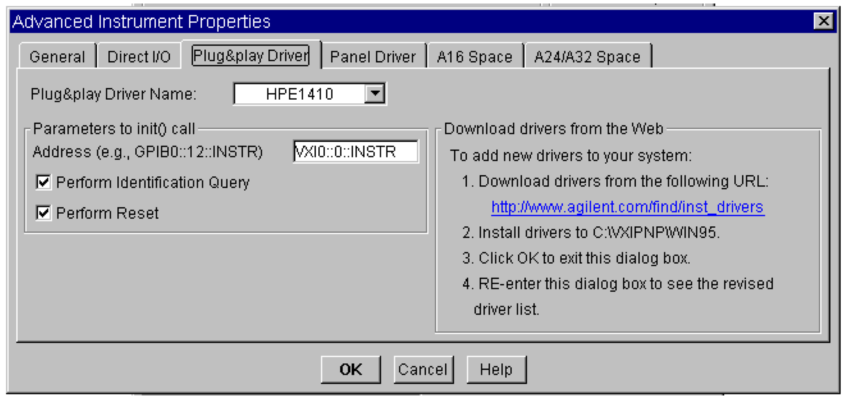


Figure 3-24. The Plug&play Driver Tab

By default, the `Address` field displays `VXI0::0::INSTR`, which assumes a VXI logical address of `0` for the instrument. Generally, you will need to supply the correct logical address. For example, if the logical address of the HP E1410A is `24`, change the `Address` field to `VXI0::24::INSTR`. For further information about the fields in the `Plug&play Driver` tab, see “Details of the Properties Dialog Boxes” on page 75.

Note

Only the `Plug&play Driver` tab applies to configuring VXI*plug&play* drivers. The `General`, `Direct I/O`, `Panel Driver`, `A16 Space`, and `A24/A32 Space` tabs have no effect on a VXI*plug&play* configuration. For example, the `Live Mode` setting on the `General` tab is ignored since a VXI*plug&play* device is always considered live.

When you have configured the instrument, click `OK` on each dialog box to return to the `Instrument Manager`, which will show the added instrument, as in Figure 3-25.

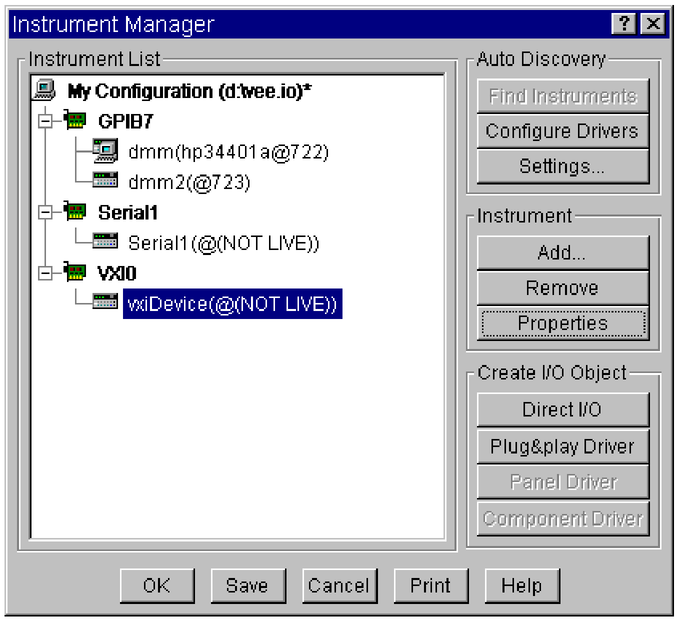


Figure 3-25. The VXI Configuration

Click the `Plug&play Driver` button to add a `To/From VXIplug&play` object as shown in Figure 3-26.

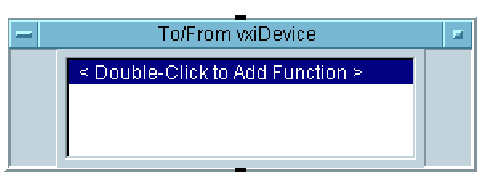


Figure 3-26. The To/From VXIplug&play Object

See “Using the To/From VXIplug&play Object” on page 195 for information about using the `To/From VXIplug&play` object.

Configuring for a PC PlugIn Card

VEE supports ODAS (Open Data Acquisition Standard) compatible PC PlugIn cards through ActiveX automation. Follow the manufacturer's instructions to install and configure these cards.

In the Instrument Manager, click Find Instruments. If the PC PlugIn hardware and software have been configured correctly you see a configuration similar to Figure 3-27:

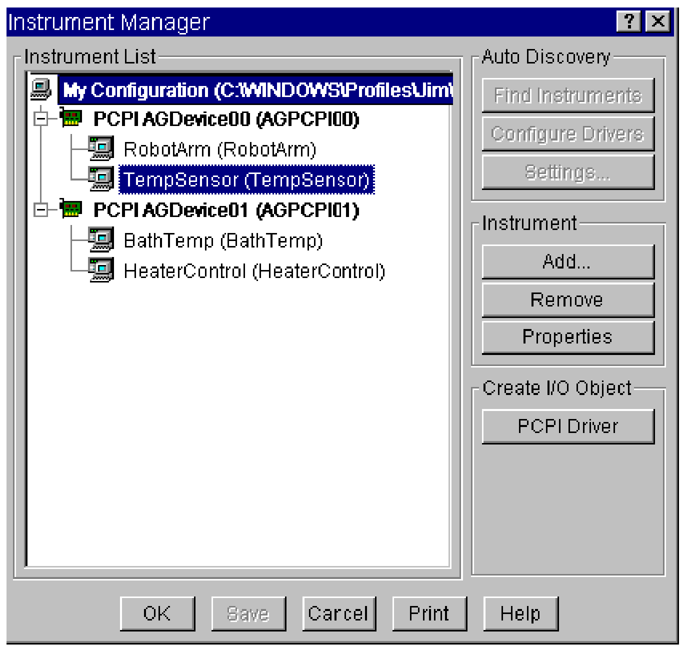


Figure 3-27. Example PC PlugIn Configuration

Click on the `PCPI Driver` button to get a formula object similar to Figure 3-28:

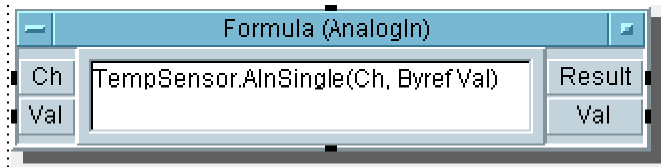


Figure 3-28. Formula Object Created by VEE

This is a formula object with a call to AInSingle method. VEE will automatically create an object for this method (in this example ‘TempSensor’) so you don’t have to call CreateObject() to create it. All the properties and methods supported for this objects are listed in the Function & Object browser under ActiveX Objects.

Details of the Properties Dialog Boxes

This section provides a detailed description of the `Instrument Properties` dialog box, each tab of the `Advanced Instrument Properties` dialog box, and the `Interface Properties` dialog box. For an overview of using `Instrument Manager` and these dialog boxes, see "Using the Instrument Manager" on page 48.

Instrument Properties Dialog Box

The `Instrument Properties` dialog box appears when you select an instrument and click either the `Add...` button or the `Properties` button in the `Instrument Manager`. See Figure 3-29 for an example of this dialog box:

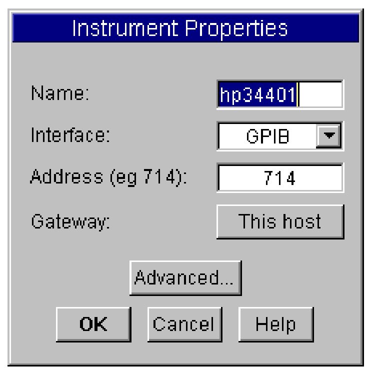


Figure 3-29. The Instrument Properties Dialog Box

The following sections describe the individual fields.

Name Field

VEE Instrument control objects require that 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`.

`Name` must be a valid VEE variable name if you want to programatically get/set its properties. The name must start with an alpha character, followed by alphanumeric characters or underscores.

`Name`s must be unique. For example, 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 `Panel Driver`s or VXI*plug&play* drivers 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 Properties in the Instrument Control object menu, (e.g. Direct I/O, MultiInstrument I/O).

Note

It is very important that you use `Name`s correctly. This section discusses only the more common situations. For more details about how VEE uses names, see “The Importance of Names” on page 187.

Interface Field

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

Address Field

The `Address` field specifies the address of the instrument. For instruments using GPIO or Serial Interfaces, the address is the same as the interface logical unit. An interface logical unit is a number used by the computer to identify a particular interface.

For instruments using GPIB Interfaces, the address is of the form *xxyyzz*, where:

- `xx` is the one- or two-digit interface logical unit. The factory default logical unit for most GPIB 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. Secondary addresses are typically used by cardcage-type instruments that use multiple plug-in modules. Secondary addresses are used to access devices through a command module in a C size VXI mainframe, and to address devices in a B size VXI mainframe.

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 `Advanced Instrument Properties` dialog box. Subaddresses are a *driver-related* feature and are used *very rarely.*

For instruments using VXI Interfaces (connected to embedded controllers or controllers with direct access to the VXI backplane), the address is of the form `xxyyy`, where:

- `xx` is the one- or two-digit logical unit of the VXI backplane interface of an embedded or external controller.
- `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 there is no physical instrument matching this device description connected to the computer. An address of `0` automatically sets `Live Mode` to `OFF`.

GPIB Address Example 1. To control a GPIB instrument at bus address 9 using a GPIB interface card with logical unit 7, the `Address` field setting for the instrument is `709`. See “Logical Units and I/O Addressing” on page 171 for information about the recommended logical units.

GPIB Address Example 2. To control an instrument at bus address 12 using a GPIB interface card with logical unit 14, the `Address` field setting is `1412`.

VXI Address Example 1. To control a VXI instrument with logical address 28 using an embedded VXI controller with logical unit 16, the `Address` field setting is `16028`. See “Logical Units and I/O Addressing” on page 171 for information about recommended logical units. Logical addresses for VXI instruments are 1 - 255, inclusive.

VXI Address Example 2. To address a VXI instrument with logical address 24 using an HP E1406 GPIB Command Module with bus address 9 via a GPIB Interface at logical unit 7, the `Address` field setting is `70903`.

For an HP E1406 Command Module, use a secondary address for the VXI instrument equal to the instrument’s logical address divided by 8. For logical address 24, the secondary address is 3. Thus, the complete address is `70903`.

Serial Address Example. To control an instrument using the COM1 serial port with logical unit 9, the `Address` field setting for the instrument is `9`. See “Logical Units and I/O Addressing” on page 171 for information about recommended logical units.

GPIO Address Example. To control a custom-built instrument using a GPIO Interface with logical unit 13, the `Address` field setting for the instrument is `13`. See “Logical Units and I/O Addressing” on page 171 for information about recommended logical units.

Gateway Field

Use the `Gateway` field set to the name of the LAN gateway used during a remote process. See “LAN Gateways” on page 153 for further information.

Advanced... Button

Click the `Advanced...` button to go to the `Advanced Instrument Properties` dialog box.

Advanced Instrument Properties Dialog Box: General Tab

Figure 3-30 shows an example of the `General` tab of the `Advanced Instrument Properties` dialog box:

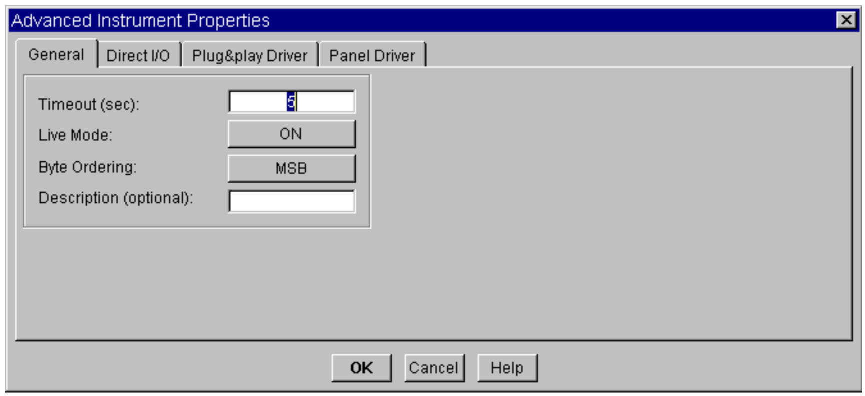


Figure 3-30. The General Tab

The following sections describe the individual fields.

Note

The parameters specified in the `General` tab apply to `Direct I/O`, `Panel Driver`, and `Component Driver` objects, but not to `To/From VXIplug&play` objects.

Timeout (sec) Field

The `Timeout` field specifies how many seconds 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, VEE will *never* detect a timeout. Certain `Direct I/O` transactions for register or memory access of VXI devices do not support a timeout.

Live Mode Field

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

If `Live Mode` is `OFF` for a particular instrument, you can run programs containing `Panel Drivers`, `Component Drivers`, or `Direct I/O` objects that would otherwise read and write to that instrument. However, no instrument communication actually takes place. This behavior can be useful if you want to develop or debug portions of a program while instruments are not available.

Byte Ordering Field

Use this field to specify the order the device uses for reading and writing binary data. VEE uses the value in this field to determine if byte swapping is necessary. Click 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. See your device manual for specific information.

Description (optional) Field

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

Advanced Instrument Properties Dialog Box: Direct I/O Tab

Figure 3-31 shows an example of the `Direct I/O` tab of the `Advanced Instrument Properties` dialog box (shown for the GPIB Interface):

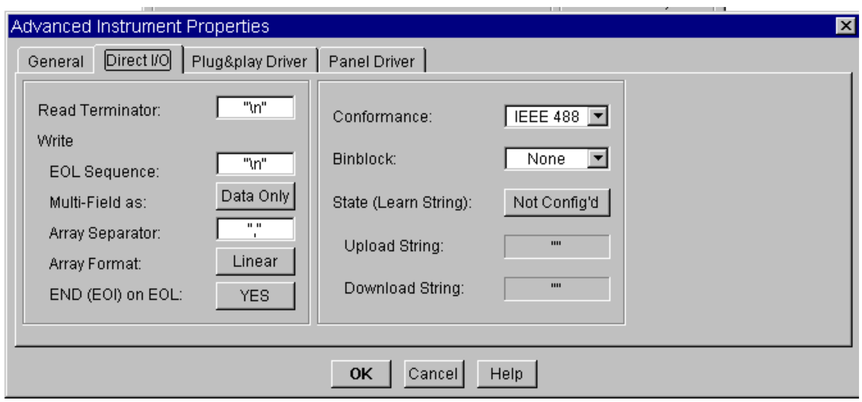


Figure 3-31. The Direct I/O Tab

The following sections describe the individual fields.

Note

When addressing VXI devices directly on the VXI backplane, you can use SCPI messages to control register-based devices, if I-SCPI drivers exist for them. VEE will inform you if required I-SCPI drivers are not available.

If I-SCPI drivers are not available, you must control register-based devices by direct read/write access to device registers or device memory. See “Advanced Instrument Properties Dialog Box: A16 Space (VXI Only) Tab” on page 94 or “Advanced Instrument Properties Dialog Box: A24/A32 Space (VXI Only) Tab” on page 98 for details.

Read Terminator Field

The `Read Terminator` field specifies the character that terminates `READ` transactions. The entry in this field must be a single character surrounded by double quotes. "Double quote" means ASCII 34 decimal. VEE recognizes any ASCII character as a `Read Terminator` as well as the escape characters shown in Table 3-1.

The character you should specify is determined by the design of your instrument. Most GPIB instruments send `Newline` after sending data to the computer. See your instrument programming manual for details.

Table 3-1. 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*.

Write EOL Sequence Field

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. VEE recognizes any ASCII characters within `EOL Sequence` including the escape characters shown in Table 3-1.

Write Multi-field As Field

The `Multi-field As` field specifies the formatting style for multi-field data types for `WRITE TEXT` transactions. The multi-field data types in VEE are Coord, Complex, PComplex, and Spectrum. Other data types and other formats are not affected 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+2j` could be written as `(2,2)` using `(...) Syntax` or as `2,2` using `Data Only` syntax.

Write Array Separator Field

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 quotes" means ASCII 34 decimal. VEE recognizes any ASCII character as an `Array Separator` as well as the escape characters shown in Table 3-1.

`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.

VEE allows arrays of multi-field data types. For example, you can create an array of Complex data. In this 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.

Write Array Format Field

The `Array Format` field 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
```

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
4 5 6
7 8 9
```
Block Array Format

```
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. *(This special behavior for arrays does not apply to any other type of transaction.)*

Write END (EOI) On EOL Field (GPIB 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. See your instrument's programming manual for details.

Conformance Field

`Conformance` specifies whether an instrument conforms to the IEEE 488.1 or IEEE 488.2 standard. See 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 GPIB 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, as well as register-based VXI instruments supported by I-SCPI drivers.

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, as described in the following sections.

Binblock Field

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 (Learn String) Field

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 want to configure the instrument for use with learn strings, click 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 Field

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. See your instrument programming manual for details. You must surround the command with double quotes.

Download String Field

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. See your instrument programming manual for details.

Advanced Instrument Properties Dialog Box: Plug&play Driver Tab

Figure 3-32 shows an example of the `Plug&play Driver` tab of the `Advanced Instrument Properties` dialog box (shown for the GPIB Interface):

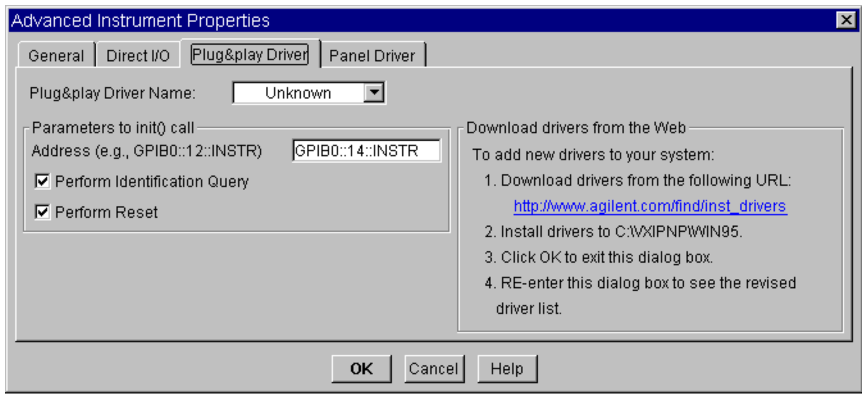


Figure 3-32. The Plug&play Driver Tab

The `Plug&play Driver` tab is the *only* tab of the `Advanced Instrument Properties` dialog box that applies to VXI*plug&play* driver configurations.

The following sections describe the individual fields.

Plug&play Driver Name Field

This field specifies the name of the VXI*plug&play* driver. You must select a driver name, as this parameter is required. The drop-down list displays all VXI*plug&play* drivers installed. If there are no entries in the list, either you do not have any VXI*plug&play* drivers installed or your registry entry or the environment variable may not be set correctly. See “Introduction to VXIplug&play” on page 38 for further information.

Parameters to init() call Field

Address. Enter the address that identifies the instrument. The address format depends on the interface to which the instrument is connected:

- VXI address string (*embedded VXI, VXLink, or MXIbus controller*).

  For a VXI instrument with an embedded, VXLink, or MXIbus controller, the address string takes the form

  `VXI[`*`board`*`]::`*`VXI logical address`*`[::INSTR]`

  An example is `VXI::24::INSTR` for an instrument at logical address 24.

  The *`board`* number is optional for the first board (`VXI::24::INSTR` is equivalent to `VXI0::24::INSTR`). However, the *`board`* number is required for subsequent boards (`VXI1`, `VXI2`, and so forth).

- GPIB-VXI address string (*command module*).

  For a VXI instrument that is being controlled from a GPIB card connected to a command module, the address string takes the form

  `GPIB-VXI[`*`board`*`]::`*`VXI logical address`*` [::INSTR]`

  An example is `GPIB-VXI::24::INSTR` (or `GPIB-VXI0::24::INSTR`) for an instrument at VXI logical address 24.

- GPIB address string (*GPIB instruments*).

  For a non-VXI instrument being controlled from a GPIB card, the address string takes the form

  `GPIB[`*`board`*`]::`*`GPIB primary address`*`::[`*`GPIB secondary address`*`][::INSTR]`

  An example is `GPIB::23::INSTR` (or `GPIB0::23::INSTR`) for a GPIB instrument at primary address 23. (The optional secondary address is rarely used.)

Perform Identification Query. Select this check box if you want the driver to query the instrument for its identification the first time a function panel for this driver is executed. You generally want to select the check box, except in the rare case that your instrument does not support this operation.

Perform Reset. Select this check box if you want a reset sent to the instrument the first time a function panel for this driver is executed. You generally want to select the check box, except in the rare case that your instrument does not support this operation. Note that all VXI instruments support this operation.

Download Drivers. If you need a new driver or to update a driver, click on the URL in the `Advanced Instrument Properties` dialog box.

Advanced Instrument Properties Dialog Box: Panel Driver Tab

Figure 3-33 is an example of the `Panel Driver` tab of the `Advanced Instrument Properties` dialog box:

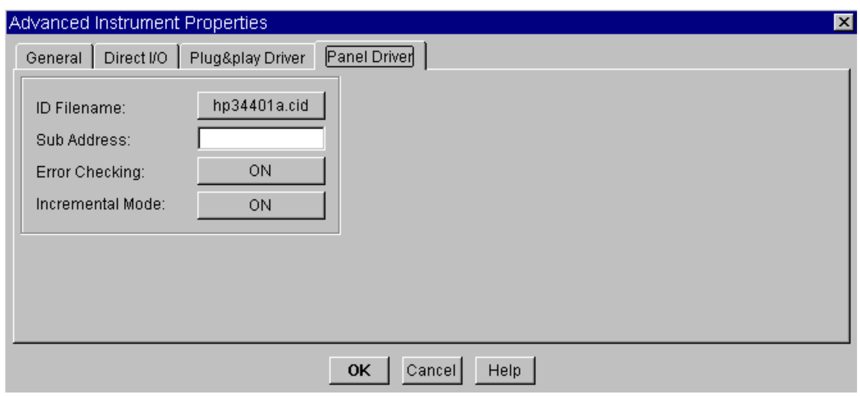


Figure 3-33. The Panel Driver Tab

Note You can configure register-based VXI devices as message-based only if they are supported by I-SCPI drivers.

This tab is used to configure both `Panel Driver` and `Component Driver` objects. The following sections describe the individual fields.

ID Filename Field

The `ID Filename` field specifies the file that contains the desired Panel Driver. Click the field to display the `Read from what Instrument Driver?` dialog box and choose a file. Files are named according to instrument model number.

Be certain to choose the name corresponding to the exact model number you are using, as there are similar file names such as `hp3325a.cid` and `hp3325b.cid`.

Sub Address Field

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 almost all situations.

If you *are* configuring a driver for one of these plug-ins, see online help for the instrument driver to determine if and how subaddresses are used.

Note

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

Error Checking Field

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

Incremental Mode Field

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

Note

The proper setting for `Incremental Mode` is `ON` in almost all situations.

When Incremental Mode is set to ON, VEE automatically minimizes the number of commands sent to the instrument to change its state. To do this, VEE compares its record of the current state the physical instrument to the new state specified in the Panel Driver.

VEE determines which component settings are different and 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, since this mode provides the best execution speed.

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

The Incremental Mode setting affects the operation of Panel Driver objects, but not Component Driver objects. These things *do* suggest setting Incremental Mode to OFF:

- Allowing front panel operation of an instrument while a VEE program is also controlling the instrument.
- Changing instrument settings outside of the VEE environment through C programs, Rocky Mountain Basic programs, or shell commands while a VEE program is also controlling the instrument.

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

Advanced Instrument Properties Dialog Box: Serial Tab

Figure 3-34 is an example of the `Serial` tab of the `Advanced Instrument Properties` dialog box (valid for serial interfaces only):

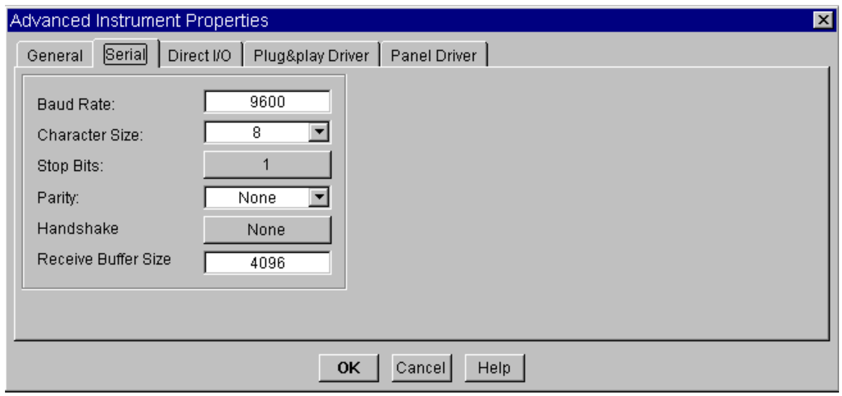


Figure 3-34. The Serial Tab

You can set the following fields for the serial (RS-232) interface:

- `Baud Rate` – The default is `9600` (bits per second).
- `Character Size` – The default is `8` (bits). Allowed values are `5`, `6`, `7`, `8`, and `None`.
- `Stop Bits` – The default is `1`. Allowed values are `1` and `2`.
- `Parity` – The default is `None`. Allowed values are `None`, `Odd`, `Even`, `Mark`, and `Space`.
- `Handshake` – The default is `None`. Allowed values are `None` and `Xon/Xoff`.
- `Receive Buffer Size` – The default is `4096` (bytes).

Advanced Instrument Properties Dialog Box: GPIO Tab

Figure 3-35 is an example of the `GPIO` tab of the `Advanced Instrument Properties` dialog box (valid for GPIO interfaces only):

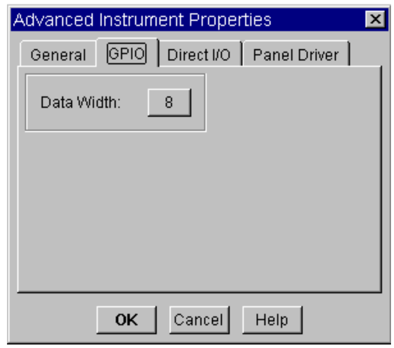


Figure 3-35. The GPIO Tab

The `GPIO` tab has only one field, `Data Width`. 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.

Advanced Instrument Properties Dialog Box: A16 Space (VXI Only) Tab

Figure 3-36 is an example of the `A16 Space` tab of the `Advanced Instrument Properties` dialog box. This tab appears only for the VXI Interface, and is used only for register-based `Direct I/O` transactions.

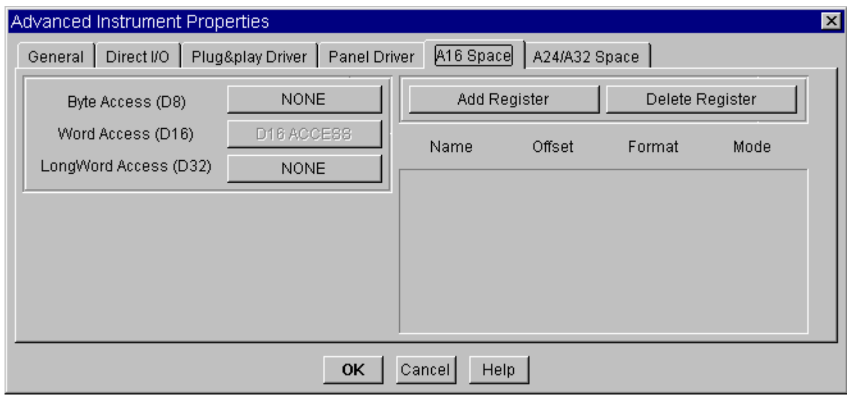


Figure 3-36. The A16 Space Tab

The following sections describe the individual fields.

Byte Access (D8) Field

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 (D16) Field

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

LongWord Access (D32) Field

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 Field

When you click the Add Register field, it adds a row of fields to 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. 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. 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` - Specifies 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 Field

When you click 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.

An Example

Figure 3-37 shows the `A16 Space` tab with the register configuration of an HP E1411B VXI Multimeter. Note that the list of registers scrolls as additional registers are added using `Add Register`.

Note

An extended (`A24/A32 Space`) memory configuration would be similar, but would consist of memory "locations," rather than "registers."

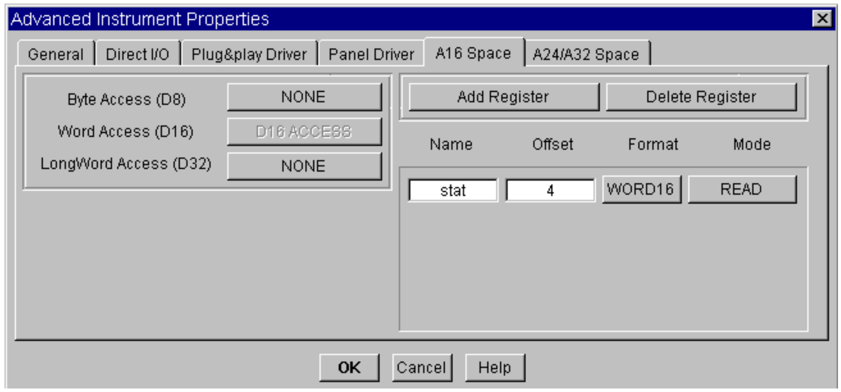


Figure 3-37. The A16 Configuration for the HP E1411B Multimeter

The `Offset` field is configured with the offset in bytes of each register from the *relative* base of the device's A16 space. The `status` register (`Name: = stat` in the figure) is configured with a 4-byte offset and is configured for `READ` mode.

The `control` register is not shown in the figure, but typically would be configured for a 4-byte offset in `WRITE` mode. While two separate register locations could have the same mode, the `Name` field must be unique. However, it would be possible for the register at byte location 4 to be named `statuscontrol` with a mode of `READ/WRITE`.

Advanced Instrument Properties Dialog Box: A24/A32 Space (VXI Only) Tab

Figure 3-38 is an example of the `A24/A32 Space` tab of the `Advanced Instrument Properties` dialog box. This tab appears only for the VXI Interface, and is used only for register-based `Direct I/O` transactions.

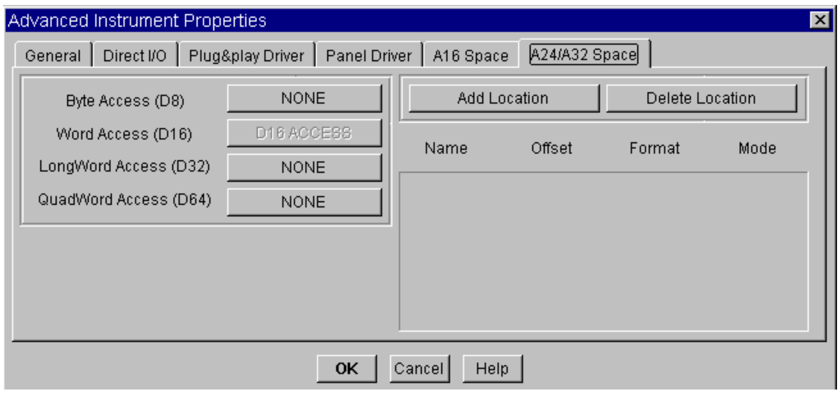


Figure 3-38. The A24/A32 Space Tab

The following sections describe the individual fields.

Note

The term "extended memory" indicates either A24 or A32 memory in a VXI device. (A VXI device can implement either A24 or A32 memory, but not both.)

Byte Access (D8) Field

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 (D16) Field

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

LongWord Access (D32) Field

The `LongWord Access` field 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.

QuadWord Access (D64) Field

The `QuadWord Access` field specifies whether the VXI device supports 64-bit extended memory access. The possible choices are:

- `NONE` - Device does not support 64-bit access.
- `D64 ACCESS` - Device supports 64-bit memory access.

Agilent I/O Libraries G.02.02 supports 64-bit access to some VXI I/O instruments' memory space. This feature enables VEE programs to read/write memory in 64-bit units for VXI instruments that support this mode. If you have version G.02.02 installed, you can use the A24/A32 Space Tab on the `Advanced Instrument Configuration` dialog box to enable this access mode.

To enable this mode, first enable `QuadWord access (D64)` Access and choose format `WORD32*2`, or `REAL64`. If you choose the `WORD32*2` format, a 64-bit value is read into two adjacent numbers of the INT32 array.

<u>Location:</u> I/O ⇒ Instrument Manager (select VXI instrument) ⇒ Edit Instrument ⇒ Advanced I/O Config... A24/A32 Space Tab.

Add Location Field

When you click the `Add Location` field, it adds a row of fields to 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. 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. 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.
  - WORD32*32 - Read or write a 64-bit word as two 32-bit words (as two Int32). QuadWord Access must be enabled.
  - REAL64 - Read or write a 64-bit word as a REAL64. QuadWord Access must be enabled.
- Mode - Specify what I/O mode the location will support. The choices are:
  - READ - This location will appear as a choice in a READ MEMORY transaction only.
  - WRITE - This location will appear as a choice in a WRITE MEMORY transaction only.

- ❑ READ/WRITE - This location will appear as a choice in both a READ MEMORY and WRITE MEMORY transaction.

Delete Location Field

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

Interface Properties

The Interface Properties dialog box appears only when you select an Interface in the Instrument Manager's instrument list, and then click the Properties button. Figure 3-39 is an example of this dialog box:

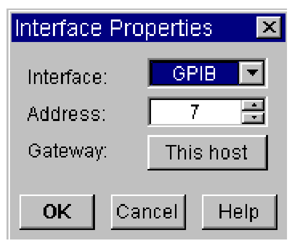


Figure 3-39. The Interface Properties Dialog Box

The following sections describe the individual fields.

Interface Field

The Interface field specifies the type of hardware interface. You can interchange GPIB with VXI (both are multiple-instrument buses), or Serial with GPIO (both are single-instrument interfaces).

Address Field

The Address field specifies the logical unit for the Interface, affecting all instruments connected to it. Use the up and down arrows to change the Address – only the logical units without conflicts will appear.

Gateway Field

Use the Gateway field set to the name of the LAN gateway used during a remote process. See "LAN Gateways" on page 153 for further information.

4

Using Transaction I/O

Using Transaction I/O

VEE for Windows provides a means to communicate with files, printers, programs, and hardware interfaces and the instruments connected to them.

I/O objects control this communication using **transactions**. This chapter explains general concepts common to all objects using transactions, including:

- Creating and Reading Transactions
- Using Transaction-Based Objects
- Choosing Correct Transactions
- Communicating With Files
- Using Transactions in Direct I/O and Interface Operations

It also explains how to use transactions in Direct I/O and Interface Operations.

Creating and Reading Transactions

All I/O objects discussed in this chapter contain transactions. A transaction specifies 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 `I/O` ⇒ `To` ⇒ `String` to create a `To String` object. Figure 4-1 shows this object.

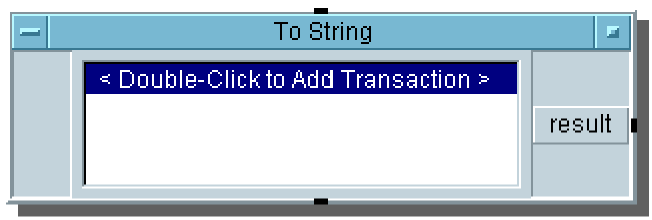


Figure 4-1. Default Transaction in `To String` Object

To add a transaction, double click in the object.

Figure 4-2 shows a simple program using the `To String` object to illustrate how transactions operate. The program uses two transactions, one to write a string literal and one to write a number in fixed decimal format.

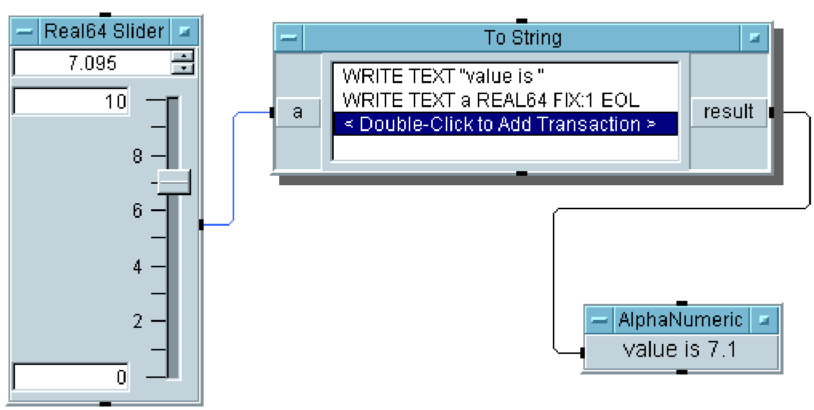


Figure 4-2. A Program Using `To String` Object

You generally need to do at least two things with a transaction-based object:

1. Add additional transactions as required.

2. Add input terminals, output terminals, or both. Most terminals will be automatically added as needed—as you add or edit transactions.

Creating and Editing Transactions

Editing with Mouse and Keyboard

Table 4-1 describes briefly how to edit transactions with a mouse.

Table 4-1. Editing Transactions With a Mouse

To Do This...	Click This...
Add another transaction to the end of the list.	`Double click in the object, or add Trans` in the object menu.
Move the highlight bar to 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 transaction.	Double-click the transaction.

Table 4-2 describes briefly how to edit transactions with the keyboard.

Table 4-2. Editing Transactions With the Keyboard

To Do This...	Press This Key...
Move the highlight bar to the next transaction.	**CTRL+N**
Move the highlight bar to the previous transaction.	**CTRL+P**
Move the highlight bar to a different transaction.	↑, ↓, **Home**
Insert a transaction above the highlighted transaction.	**Insert line** or **CTRL+O**
Cut (delete) the highlighted transaction, saving it to the transaction "cut-and-paste" buffer.	**Delete line** or **CTRL+K**
Paste the transaction currently in the buffer above the highlighted transaction.	**CTRL+Y**
Edit the highlighted transaction.	**space bar**

To edit the fields within a transaction, double-click the transaction to expand it to an `I/O Transaction` dialog box, as shown in Figure 4-3.

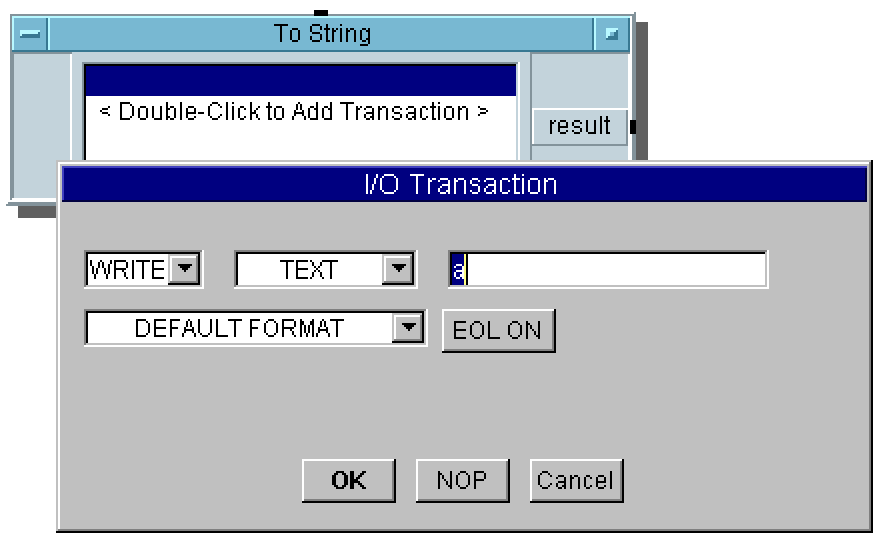


Figure 4-3. Editing the Default Transaction in `To String` Object

The fields shown in the `I/O Transaction` dialog box are different for different types of I/O operations. To change information in a field, click on the arrow and select from the list that appears. Fields without an arrow require you to enter text. Click `OK` to accept the selections and return to the I/O object.

Clicking `NOP` saves the latest settings shown in the dialog box, and makes that transaction a "no operation" or a "no op." Its effect is the same as commenting out a line of code in a text-based computer program.

Input and output terminals are added automatically as needed. You can also use the Object menu to add or delete terminals.

Editing the Data Field

The data field requires you to enter text. Figure 4-4 shows an example of a `READ` Transaction and what you might enter in the data field.

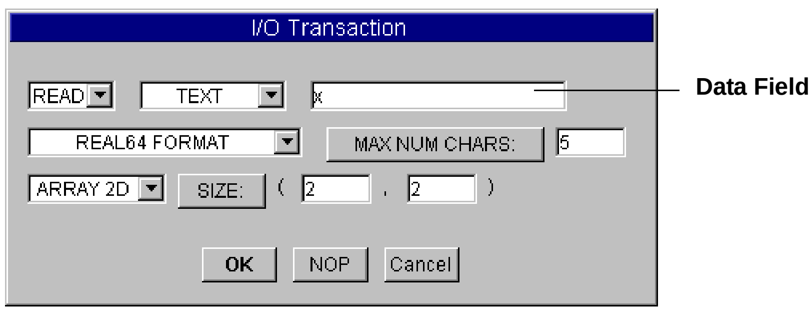


Figure 4-4. `READ` **Transaction Using a Variable in the Data Field**

Figure 4-5 shows an example of a `WRITE` Transaction and what you might enter in the data field.

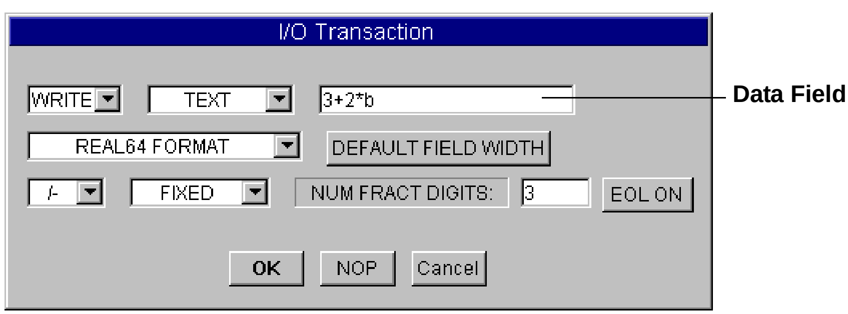


Figure 4-5. `WRITE` **Transaction Using an Expression in the Data Field**

`WRITE` transactions allow you to specify an expression list (variables, constants and operators), but `READ` allows only a variable list. Table 4-3 lists typical entries for the data field.

Table 4-3. Typical Data Field Entries

Data Field Entry	Meaning

`X`
 	
`A`
	(READ) Read data into the variable `X`.
	
	(WRITE) Write the value of the variable `A`.

`X,Y`
 	
`A,B`
	(READ) Read data into the variable `X` and then read data into the variable `Y`.
	
	(WRITE) Write the value of the variable `A` and then write the value of the variable `B`.

`null`
 	
`A,A*1.1`
	(READ only) Read the specified value and throw it away. `null` is a special variable defined by VEE.
	
	(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 VEE writes `FR 1.234 MHZ`.

You may include the escape characters shown in Table 4-4 in any field that accepts text input as a string delimited by double quotes.

Note

`READ` transactions allow a `null` variable in the data field. Reading data into the null variable throws the data away. This is useful for removing unneeded data.

Table 4-4. 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

Vee automatically adds input and output terminals as needed. To add one manually, click on "Add Terminal" in the object menu, or use the keyboard short cut **CTRL**+**A**.

`WRITE` transactions transfer data from VEE to the destination associated with the object and require a data input terminal. A `WRITE` transaction can also write data from a global or an expression such as "abs(globalA)"

`READ` transactions transfer data from the source associated with the object to VEE and require a data output terminal.

Variable names that appear on the terminal must match the variable names in the transaction specification, as shown in Figure 4-6.

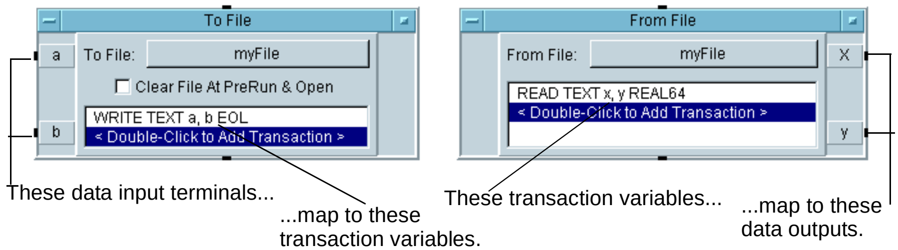


Figure 4-6. Terminals Correspond to Variables

To edit a terminal variable name, do the following:

1. Double click the terminal to expand it into a `Terminal Information` dialog box.

2. Edit the `Name` field in the dialog box.

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

Reading Transaction Data

To read data into a variable, VEE must know either the number of data elements to read or the specific terminating condition. READ transactions look for either a specified number of data elements or an end-of-file (EOF) indication. Specify this in the last field of the I/O Transaction dialog box.

Transactions that Read a Specified Number of Data Elements

The last field in the transaction dialog box has the default value `SCALAR`. This specifies that the `READ` transaction is to read only one element. To change this, click the `SCALAR` field and choose from a list of available choices, as shown in Figure 4-7.

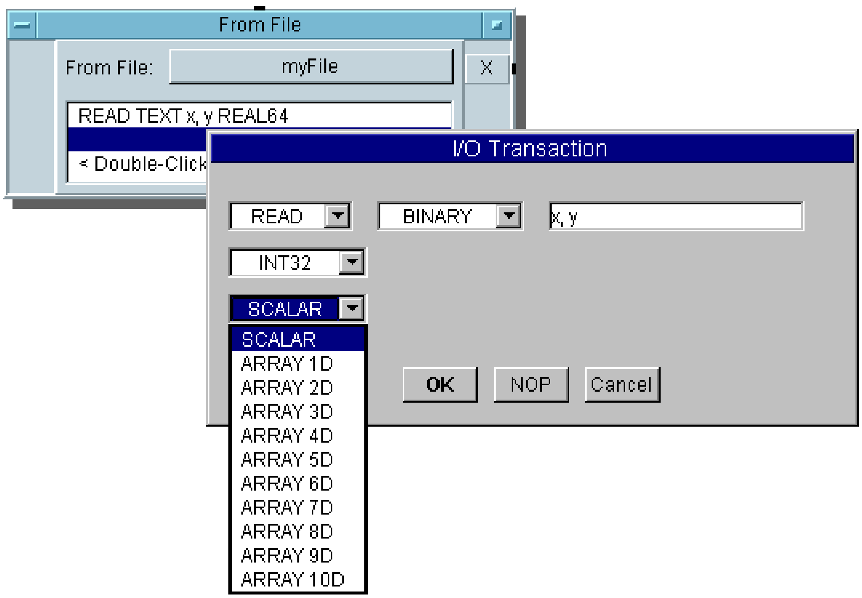


Figure 4-7. Select Read Dimension from List

The choices in the list 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, etc.

When you select a dimension, the transaction dialog box shows a fill-in field for each dimension specified. Figure 4-8 shows a 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.)

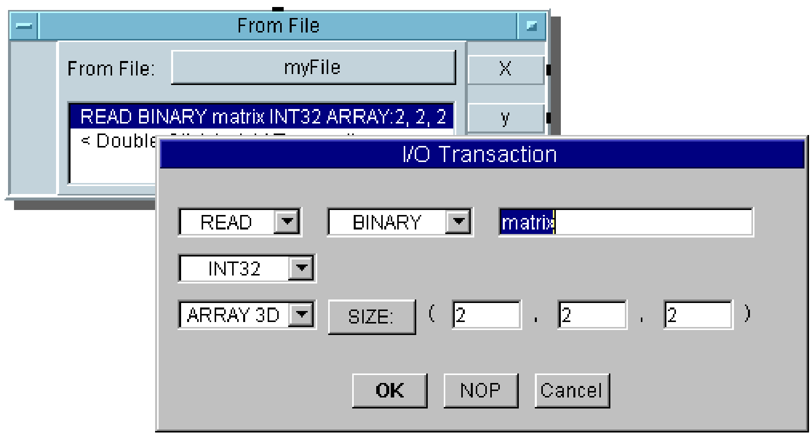


Figure 4-8. Transaction Dialog Box for Multi-Dimensional Read

When more than one dimension is specified, the rightmost or "innermost" dimension is filled first. In this example, the elements are read in the following 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 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 VEE objects support `READ` transactions that read to the end-of-file (`EOF`). This makes it possible to read the contents of a file with a single transaction. Such transactions are called **read-to-end transactions**. `EOF`, besides indicating end-of-file for a standard disk file, can also indicate closure of a named-pipe or pipe.

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

- `From File`
- `From String`

Figure 4-9 shows the transaction dialog box of a `From File` object, reading a three dimensional array of binary integers, but configured for read-to-end:

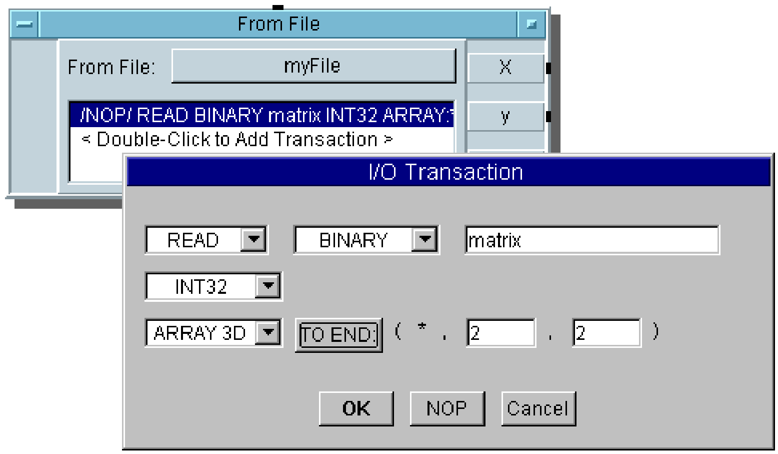


Figure 4-9. Transaction Dialog Box for Multi-Dimensional Read-To-End

Read-to-end transactions are not supported for scalars. The transaction must be configured for at least a one-dimensional array to be configured as read-to-end. If a VEE object supports read-to-end, the `SIZE:` field appears as a button in the transaction dialog box. Clicking the `SIZE:` field enables read-to-end, and the field appears as `TO END:`.

If a one-dimensional array is read to the end, the number of elements in the array is unknown until `EOF` is found. The unknown size of the array is denoted by an asterisk (`*`) in the transaction.

When reading a multi-dimensional array is read to the end, the number of elements must be supplied for each dimension except the left-most or "outer" dimension. Figure 4-9 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.

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.

In read-to-end transactions of dimensions greater than an `ARRAY 1D,` the number of total elements read has to be evenly divisible by the product of the known dimensions. For example, if the read-to-end example of a three-dimensional array is from a file with 16 total elements, 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`.

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

Suggestions for Developing Transactions

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 program in Figure 4-10 shows one way you might do this.

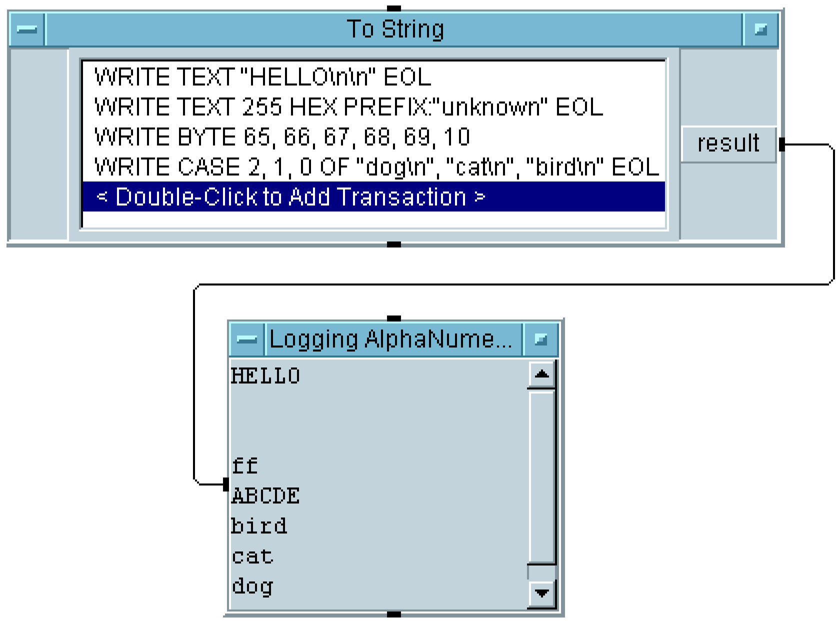


Figure 4-10. Example: Using `To String`

Using Transaction-Based Objects

This section gives guidelines for using transaction-based objects, including execution rules and Object configuration.

Execution Rules

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

1. Transactions execute beginning with the top-most 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 or the same thread within the same program.

   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 `Properties` in the object menu

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

Transaction-based I/O objects that write data (except `Direct I/O`) include an additional tab in the `Properties` dialog box that lets you edit the data format. The resulting dialog box allows you to view and edit various settings.

Note

`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` ⇒ `Instrument Manager`.

Clicking `Properties` in the object menu of a transaction I/O object yields a `Properties` dialog box like the one in Figure 4-11.

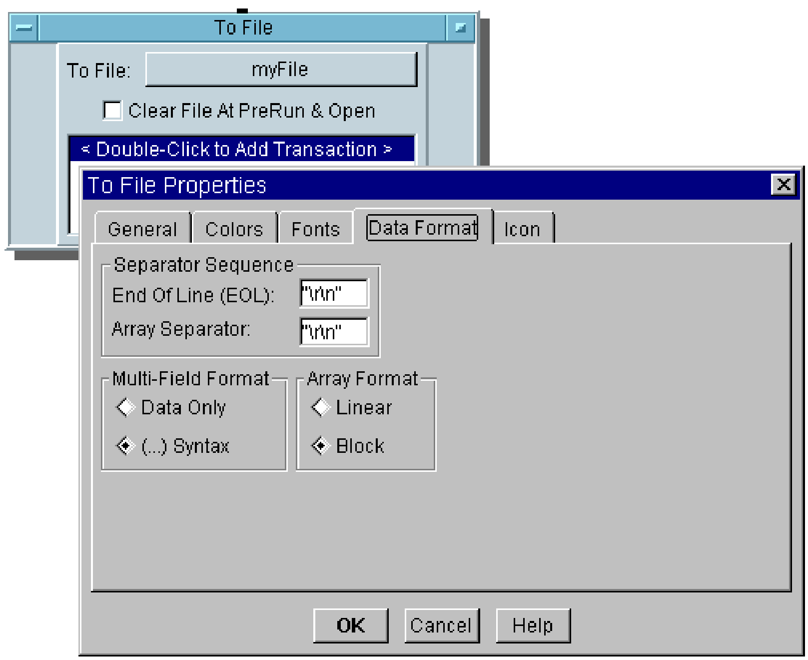


Figure 4-11. The `Properties` Dialog Box

The Properties dialog box has a Data Format tab containing settings that affect the way certain data is written by WRITE transactions. The End Of Line (EOL) affects any WRITE in which EOL ON is set. The remaining Data Format fields affect only WRITE TEXT transactions.

The following sections explain the fields in the Data Format tab in detail.

End Of Line (EOL) Field

The End Of Line (EOL) 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. VEE recognizes any ASCII characters within End Of Line (EOL) including the escape characters shown in Table 4-4.

Array Separator Field

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. VEE recognizes any ASCII character as an Array Separator as well as the escape characters shown in Table 4-4.

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.

Multi-Field Format Field

The Multi-Field Format field specifies the formatting style for multi-field data types for WRITE TEXT transactions. The multi-field data types in 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.

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 Field

The `Array Format` field 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
```

VEE writes the same matrix in one of two ways, depending on the setting of `Array Format`. In the two examples that follow, `End Of Line (EOL)` 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 `End Of Line (EOL)` character.

In the more general case (arrays greater than two dimensions), `Block Array Format` outputs an `End Of Line (EOL)` 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 `End Of Line (EOL)` character 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 Character>
A[0,1,0]  A[0,1,1]<EOL Character>
<EOL Character>
A[1,0,0]  A[1,0,1]<EOL Character>
A[1,1,0]  A[1,1,1]<EOL Character>
```

After A[0,1,1] is written, x and y change simultaneously and consequently two <EOL Character>s are written.

Choosing Correct Transactions

This section summarizes various I/O objects and 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, see Appendix A, "I/O Transaction Reference". Figure 4-5 and Figure 4-6 summarize transaction-based objects available in VEE and the actions they support.

Table 4-5. Summary of Transaction-Based Objects

Object	Description

`To File`
 	
`From File`
	Writes data to a file.
	
	Reads data from a file.

`To String`
 	
`From String`
	Writes text to a VEE container.
	
	Reads text from a VEE container.

`To Printer`	Writes text to the VEE text printer.

`Direct I/O`
 	
`MultiInstrument Direct I/O`
 	
`Interface Operations`
	Communicates directly with GPIB, VXI, serial, or GPIO instruments.
	
	Communicates directly with multiple GPIB, VXI, serial, or GPIO instruments in the same object.
	
	Transmits low-level bus commands and data bytes on an GPIB or VXI interface.

Table 4-6. Summary of Transaction Types

Action	Description
EXECUTE	Executes low-level commands to control the file, instrument, 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. For Direct I/O to GPIB, message-based and I-SCPI-supported register-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 a GPIB interface.

Selecting Correct Objects and Transactions

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 4-5.

3. Determine the correct type of transaction using Table 4-6.

4. To determine the remaining specifications for the transaction, such as encodings and formats, see Appendix A, “I/O Transaction Reference”.

Example: Selecting an Object and Transaction

For example, 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. A partial listing of the contents of the file is:

```
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, the steps to solve this problem are:

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. From Table 4-5, the object used to read a file is `From File`.

3. From Table 4-6, the type of transaction used to read data from a file is `READ`.

4. The required 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 a VEE container.

Use `From String` to read formatted Text from a 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`, see Appendix A, "I/O Transaction Reference".

`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 an `EOL` or will consume the `EOL` and continue with the read. This is dependent on the format. For details about formats, see Appendix A, "I/O Transaction Reference".

Note READ and WRITE Compatibility

In general, you must know how data was written 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.

Communicating With Files

This section gives guidelines for communicating with files, including using file pointers and importing data.

Using File Pointers

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. Figure 4-7 shows objects and source/destination files.

Table 4-7. Objects and Sources/Destinations

Source or Destination	Object
Data Files	`To File`, `From File`

The position 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 file name. 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.

Note

The `To DataSet` and `From DataSet` objects also share one read and one write pointer per file with the `To File` and `From File` objects. However, mixing `To DataSet` and `From DataSet` operations with `To File` and `From File` operations on the same file is not recommended.

Closing Files

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.

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.

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 want to read a file of unknown length.

The read-to-end feature, discussed in "Reading Transaction Data" on page 111, also provides a means of reading a file of unknown length. However, the contents of the file will be in a single 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 4-12 illustrates a typical use of `EOF`. The file being read contains a list of `X-Y` data of unknown length. Typical contents of the file are:

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

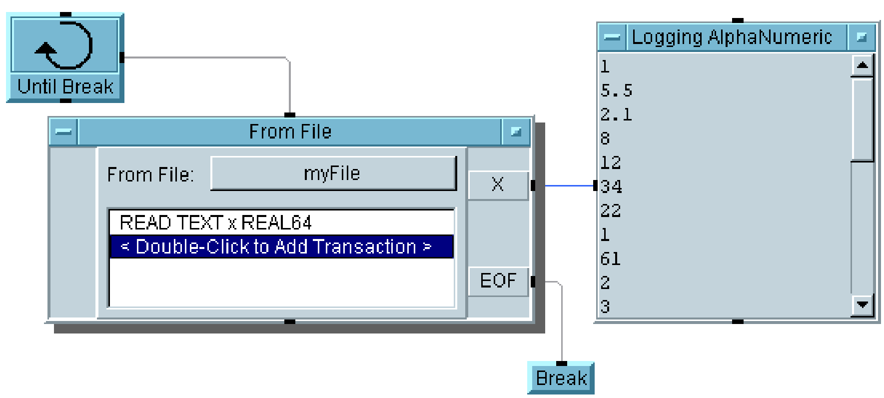


Figure 4-12. Typical Use of EOF to Read a File

Importing Data

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

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 in Figure 4-13 solves this problem.

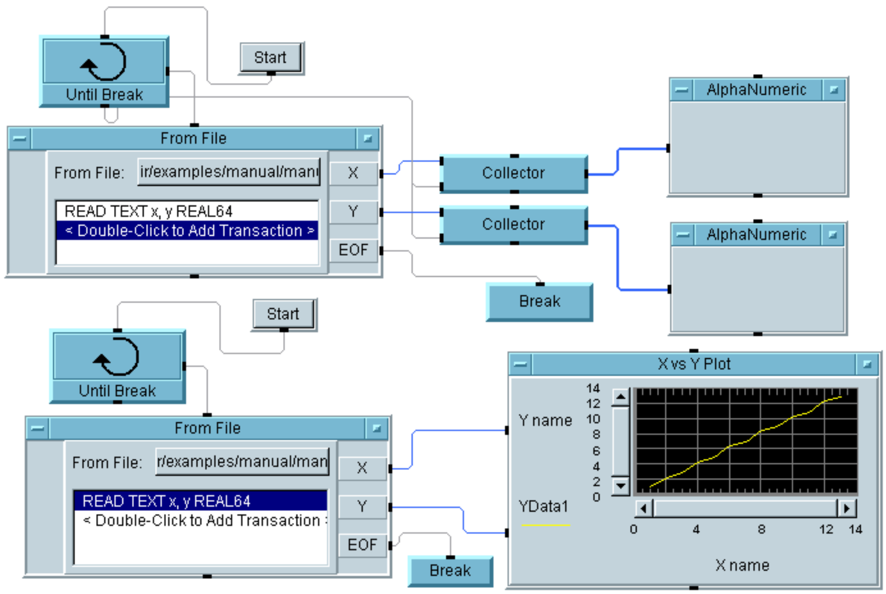


Figure 4-13. Importing `XY` Values

The program in Figure 4-13 is saved in the file `manual29.vee` in the `examples` directory.

The `READ TEXT REAL64` transaction easily handles all the different notations used for `Y` values, including signs, decimals and exponents. A portion of the data file is:

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

Importing Waveforms

Other software applications have many different conventions 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. A portion of the data file read by the program in Figure 4-14 is:

```
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 4-14 shows a program that reads the waveform data file.

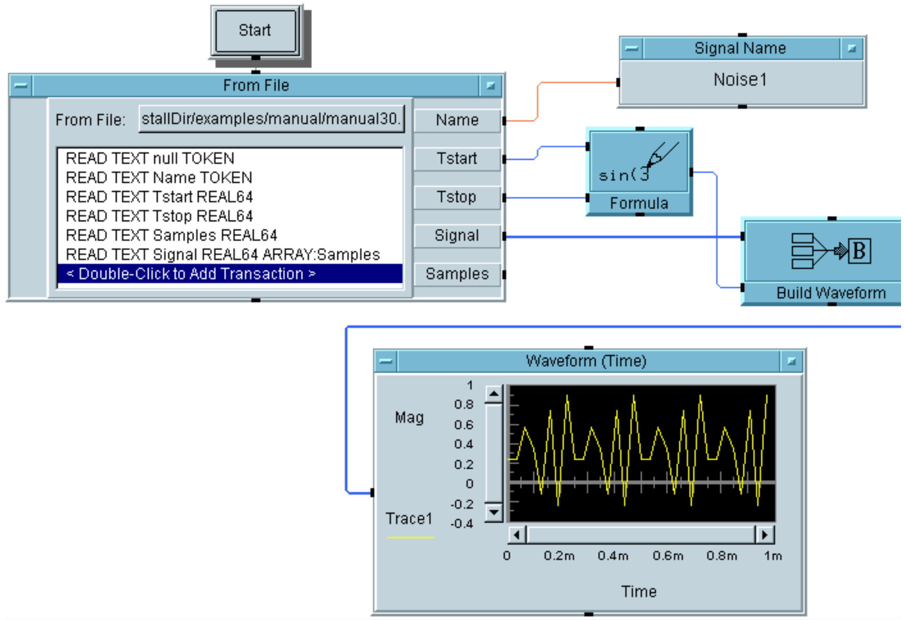


Figure 4-14. Importing a Waveform File

The program in Figure 4-14 is saved in the file `manual30.vee` in 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 `NAME` and `Noise1` will 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. VEE 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. `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 4-15:

```
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 are 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 4-15 finds only the data associated with `<DATA>`. In your own applications, you might need to search for several markers.

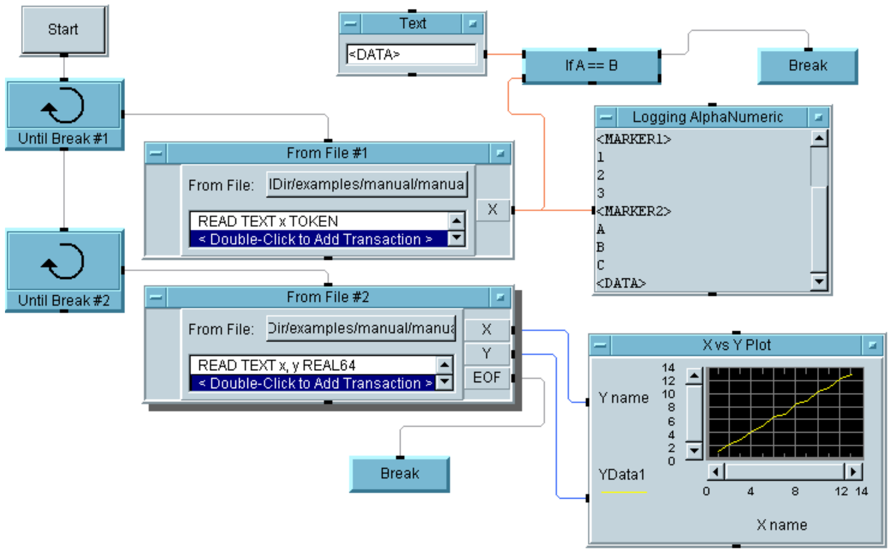


Figure 4-15. Importing a Waveform File

The program in Figure 4-15 is saved in the file `manual31.vee` in 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.

Using Transactions in Direct I/O and Interface Operations

Three VEE objects allow you to communicate with instruments using **I/O transactions**:

- The `Direct I/O` object allows you to transmit data to and from instruments via the GPIB, VXI, serial, and GPIO interfaces and via a LAN connection.
- The `MultiInstrument Direct I/O` object allows you to perform direct I/O transactions to multiple instruments from a single object.
- The `Interface Operations` object allows you to send low-level GPIB or VXI messages, commands, and data.

Note Register-based VXI devices can be used as message-based only if supported by I-SCPI drivers.

For any of these objects, the messages are "constructed" and sent by means of I/O transactions. This chapter describes some techniques for using I/O transactions in the `Direct I/O`, `MultiInstrument Direct I/O`, and `Interface Operations` objects.

Note You must properly configure VEE to communicate with instruments before you can use the `Direct I/O`, `MultiInstrument Direct I/O`, and `Interface Operations` objects. See Chapter 3, “Configuring Instruments” for details.

Using the Direct I/O Object

The `Direct I/O` object allows you control an instrument directly using the instrument's built-in commands. You do not need a VEE instrument driver (ID) or VXI*plug&play* driver to use `Direct I/O` to control an instrument.

Sending Commands

Use `WRITE` transactions to send commands to an instrument using `Direct I/O`. The most important `WRITE` transactions for sending commands to GPIB, message-based VXI, register-based VXI supported by I-SCPI, and serial instruments are:

- `WRITE TEXT`
- `WRITE BINBLOCK`
- `WRITE STATE`

`Direct I/O` uses only `WRITE BINARY` and `WRITE IOCONTROL` transactions to send commands to GPIO instruments.

`Direct I/O` uses `WRITE REGISTER` and `WRITE MEMORY` transactions to send commands to register-based and some message-based VXI instruments. These transactions are the *only* method of communicating with register-based VXI instruments not supported by I-SCPI drivers.

WRITE TEXT Transactions. `WRITE TEXT` transactions are all you need to set up instruments for the majority of all situations where `Direct I/O` is required. Most GPIB, message-based VXI, and serial instruments use human-readable text 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. The transaction used to reset such an instrument is:

```
WRITE TEXT "*RST" EOL
```

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, GPIB instruments vary in their use of the signal line End-Or-Identify (EOI).

If you suspect you are having problems in this area, examine the `END(EOI)on EOL` and `EOL Sequence` fields in the `Direct I/O` tab of the `Advanced Instrument Properties` dialog box. See Chapter 3, "Configuring Instruments". See your instrument programming manual to determine the proper command syntax for your instrument.

`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. Instruments usually require a significant number of commands before accepting `BINBLOCK` data. See your instrument's programming manual for details.

To use `BINBLOCK` transactions, you *must* properly configure the `Conformance` field (and possibly `Binblock`) in the `Direct I/O` tab of the `Advanced Instrument Properties` dialog box. See Chapter 3, "Configuring Instruments".

WRITE STATE Transactions. Some GPIB 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 `WRITE STATE` transactions are available for GPIB and message-based VXI instruments only.

A typical procedure for using learn strings is:

1. Configure the instrument to the desired measurement state. Typically, this is done using the instrument front panel.

2. Click `Upload State` 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 Uploaded learn string to the instrument, 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 GPIB and message-based VXI instruments is affected by the Direct I/O tab settings for Conformance and State (Learn String).

If Conformance is IEEE 488.2, VEE automatically handles 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.

Message-based VXI instruments and register-based VXI instruments supported by I-SCPI are IEEE 488.2 compliant.

Clicking Upload State 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. For example, suppose you want to program the HP 54100A digitizing oscilloscope using learn strings. Important facts for the oscilloscope are:

- 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 SETUP command must precede a learn string that is downloaded to the instrument. A space is required between the P in SETUP and the first character in the downloaded learn string.

You must use the Instrument Manager (see Chapter 3, “Configuring Instruments”) to specify the proper direct I/O configuration for the oscilloscope. Figure 4-16 shows settings for learn strings

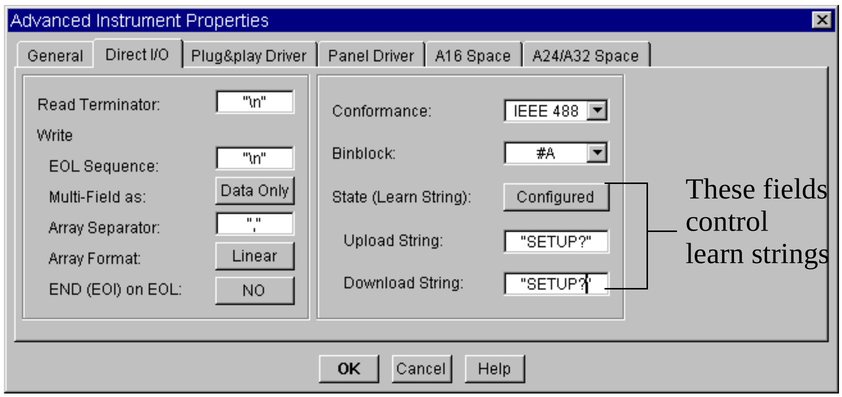


Figure 4-16. Configuring for Learn Strings

To upload a learn string from the oscilloscope, click 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

To read data from an instrument using Direct I/O, you can use READ transactions.

Note

Instruments return data in a variety of formats. In general, you must know *what kind* of data and *how much* data you want 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.

The most important READ transactions for Direct I/O use with GPIB, message-based VXI, and serial instruments are:

- ❑ READ TEXT
- ❑ READ BINBLOCK

Direct I/O uses only READ BINARY and READ IOSTATUS transactions to read data from GPIO instruments.

Direct I/O uses READ REGISTER and READ MEMORY transactions to read data from register-based and some message-based VXI instruments. These transactions are the *only* method of communicating with register-based VXI instruments not supported by I-SCPI.

Note If you have difficulty reading data from instruments, try using the Bus I/O Monitor to examine the data format.

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 voltmeter returns each reading as a single number in exponential notation, such as -1.234E+00. The transaction to read a value from the voltmeter is:

```
"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 since READ TEXT REAL transactions discard preceding alphabetic characters and extract the numeric value.

Note When reading numeric data from an instrument, the data type of the instrument data is automatically converted, if necessary, according to the rules listed in Appendix C, “Instrument I/O Data Type Conversions”.

Using the MultiInstrument Direct I/O Object

The MultiInstrument Direct I/O object (I/O ⇒ Advanced I/O ⇒ MultiInstrument Direct I/O) lets you control several instruments from a single object using direct I/O transactions. The object is a standard transaction object and works with all interfaces that VEE supports.

It appears the same as the `Direct I/O` object, except each transaction in `MultiInstrument Direct I/O` can address a separate instrument. Since the `MultiInstrument Direct I/O` object does not necessarily control a particular instrument as the `Direct I/O` object does, the title does not list an instrument name, address, or live mode condition.

By using the `MultiInstrument Direct I/O`, you can reduce the number of instrument-specific `Direct I/O` objects in your program, which optimizes icon-to-icon interpretation time. This performance increase is especially important for the VXI interface, which is faster than GPIB at instrument control.

Figure 4-17 shows the `MultiInstrument Direct I/O` object and its `I/O Transaction` dialog box configured to communicate with four instruments.

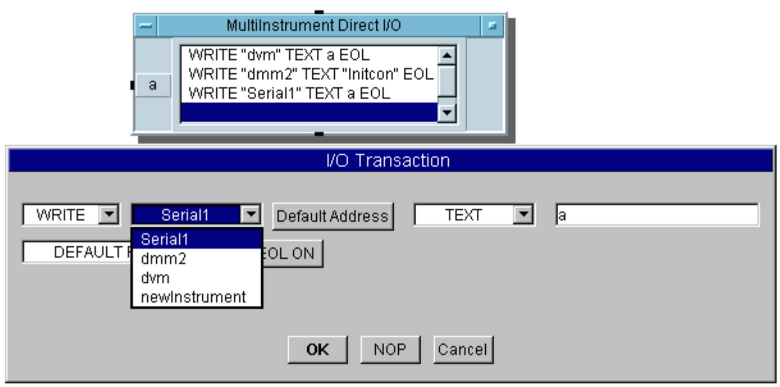


Figure 4-17. MultiInstrument Direct I/O Controlling Several Instruments

Transaction Dialog Box

The `I/O Transaction` dialog box is similar to the one used by `Direct I/O`, except it contains two additional fields. The common fields work the same way. The following sections describe the two additional fields.

Instrument Field. The `Instrument Field` contains the name of any of the currently configured instruments. Clicking the down arrow presents a list of available configured instruments. You can select a different instrument for each transaction.

Address Field. The `Address Field` specifies the address of the device showing in the `Instrument Field`. The `Address Field` has two modes: `Default Address` and `Address:`.

`Default Address` sets VEE to use the address entered when the instrument was originally configured. `Address:` includes a text box that lets you enter a different address.

You can enter a specific numeric value, a variable name, or an expression. The entry must evaluate to a valid address. The value entered for `Address:` will change the device's address when the object executes, which is like the address control pin action. Figure 4-18 shows the `I/O Transaction` dialog box using `Address:`.

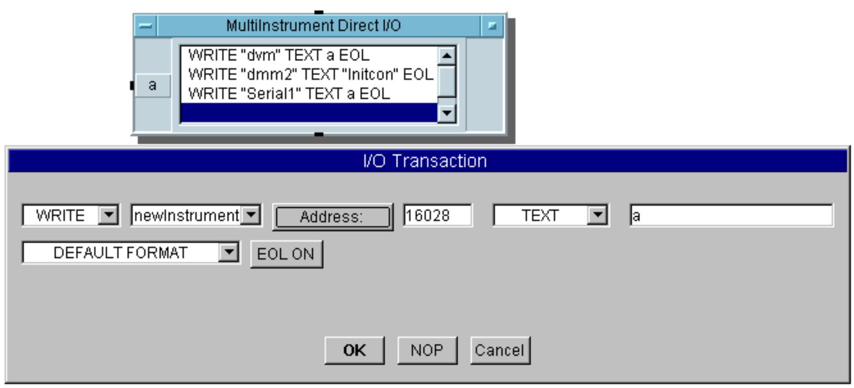


Figure 4-18. Entering an Instrument Address as a Variable

Editing Transactions

As you edit transactions using the `I/O Transaction` dialog box, only those transactions allowed by the type of instrument are accepted. For example, if the name showing in the `Instrument Field` is configured as a VXI device controlled via the VXI backplane, you can configure a `REGISTER` or `MEMORY` access transaction.

If the `I/O Transaction` dialog box is configured for a particular type of transaction and you change the `Instrument Field` name, the transaction must remain correct for the different instrument. If the transaction is incorrect, entries in the `I/O Transaction` dialog box will change to the last valid transaction for that instrument type. A `REGISTER` access transaction for a VXI device will be incorrect if you change the `Instrument Field` name to a non-VXI instrument.

Object Menu

The object menu for `MultiInstrument Direct I/O` is similar to that of the `Direct I/O` object. The `MultiInstrument Direct I/O` menu does not include the `Show Config...` or `Upload State` menu choices. These menu choices are for specific instrument configurations. Use the `Direct I/O` object to show an instrument's configuration or to upload a physical instrument's settings.

There is no live mode indicator for any of the possible devices in the transactions. To control live mode for an instrument, click `I/O` ⇒ `Instrument Manager...`, and then edit the selected instrument configuration.

Using the Interface Operations Object

The `Interface Operations` object (`I/O` ⇒ `Advanced I/O` ⇒ `Interface Operations`) allows you to control GPIB, VXI, and serial instruments using low-level commands. `Interface Operations` supports two types of transactions that provide this low-level control: `EXECUTE` and `SEND`.

The EXECUTE Transaction

`EXECUTE` transactions are of the form:

`EXECUTE` *`Command`*

where *`Command`* is one of the bus commands summarized in Table 4-8. While the commands listed in Table 4-8 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 GPIB, these instruments must be addressed to listen.

Table 4-8. Summary of EXECUTE Commands (Interface Operations)

Command	Description
ABORT	Clears the GPIB interface by asserting the IFC (Interface Clear) line. To clear and reset the VXI interface use CLEAR
CLEAR	Clears all GPIB devices by sending DCL (Device Clear). For VXI, resets the interface and runs the Resource Manager.
TRIGGER	For GPIB, triggers all devices addressed to listen by sending GET (Group Execute Trigger). For VXI, triggers TTL, ECL, or external triggers.
REMOTE	For GPIB, asserts the REN (Remote Enable) line. There is no counterpart for VXI.
LOCAL	For GPIB, releases the REN (Remote Enable) line. There is no counterpart for VXI.
LOCAL LOCKOUT	For GPIB, 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.
LOCK INTERFACE	In a multi-process system with shared resources, lets one process lock the resources for its own use during a critical section to prevent another process from trying to use them.
UNLOCK INTERFACE	In a multi-process system where a process has locked shared resources for its own use, unlocks the resources to allow other processes access to them.
PASS CONTROL	Passes control to a GPIB device at the specified address, provided the device is capable of becoming the active controller. There is no counterpart for VXI.

The SEND Transaction

SEND transactions are of this form:

```
SEND BusCmd
```

where *BusCmd* is one of the bus commands listed in Table 4-9. These messages are defined in detail in IEEE 488.1. *BusCmd* is GPIB specific only. There are no counterparts for VXI.

Table 4-9. 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 (0-30) to talk.
LISTEN	Addresses a device at the specified primary bus address (0-30) to listen.
SECONDARY	Specifies a secondary bus address following a TALK or LISTEN command. Secondary addresses are typically used by card cage instruments where the card cage 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 VEE to listen; sends MLA.
MY TALK ADDR	Addresses the computer running VEE to talk; sends MTA.
MESSAGE	Sends a multi-line bus message. Consult IEEE 488.1 for details. The multi-line messages supported by 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

5

Advanced I/O Topics

Advanced I/O Topics

This chapter covers the following advanced instrument I/O topics:

- I/O Configuration Techniques
- I/O Control Techniques
- Logical Units and I/O Addressing

I/O Configuration Techniques

This section provides information about instrument configuration with VEE. Agilent is making the instruments formerly made by HP. In general, instrument model numbers will remain the same but be preceded by Agilent, instead of HP. Because so many VEE users have instruments with the HP brand, we often use that nomenclature in this manual to avoid confusion.

The I/O Configuration File

The I/O configuration for each program can be embedded in the program file (recommended) or stored as a separate file. If it is stored as a separate file, it is the VEE.IO file. This file is stored in the following path:

```
%userprofile%\Local Settings\Application
Data\Agilent\VEE One Lab
```

When you configure instruments in a new program that does not contain an embedded configuration, the new settings are saved in memory for the remainder of your work session and in the `VEE.IO` file.

When the I/O configuration is saved with the program, the `Save` button in Instrument Manager is disabled. To keep the configuration, click OK and save the program. This saves the updated configuration with the program.

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 Properties` dialog box, not the text in the object's title bar.

If the object is a `Panel Driver` 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 settings may affect how the programs *run*.

Changing the Configuration File

Generally, VEE takes care of the `VEE.IO` file. However, there are situations when you may want to erase, update, or copy this file outside the VEE environment

If you want to run an instrument control program developed by someone else, but the I/O configuration is not embedded with the program, you need the I/O configuration that program uses. There are three ways to get it:

1. You can manually add all of the instruments to your configuration using the `Instrument Manager` and configuration dialog boxes.

2. You can copy the `VEE.IO` file for that program to your Agilent directory, using the path given at the beginning of this section.

   If you use the file copying method, save a copy of your original `VEE.IO` file to another name (such as `VEEIO.OLD`) in case you need it later.

3. You can save the program with the embedded configuration. Use the Save As option and be sure the "Save I/O configuration with program" option is checked.

Note Example programs were saved with embedded I/O configuration so the I/O configuration is self-contained. They do not depend on an external I/O configuration file.

Programmatic I/O Configuration

You can programmatically modify your instrument configuration. The preferred way of programing the I/O configuration is to use the programmatic instrument configuration in the Function and Object Browser. Figure 5-1 shows the browser window with Instruments selected in the Type window. Selecting Instruments activates the Create Set Formula button at the bottom of the window.

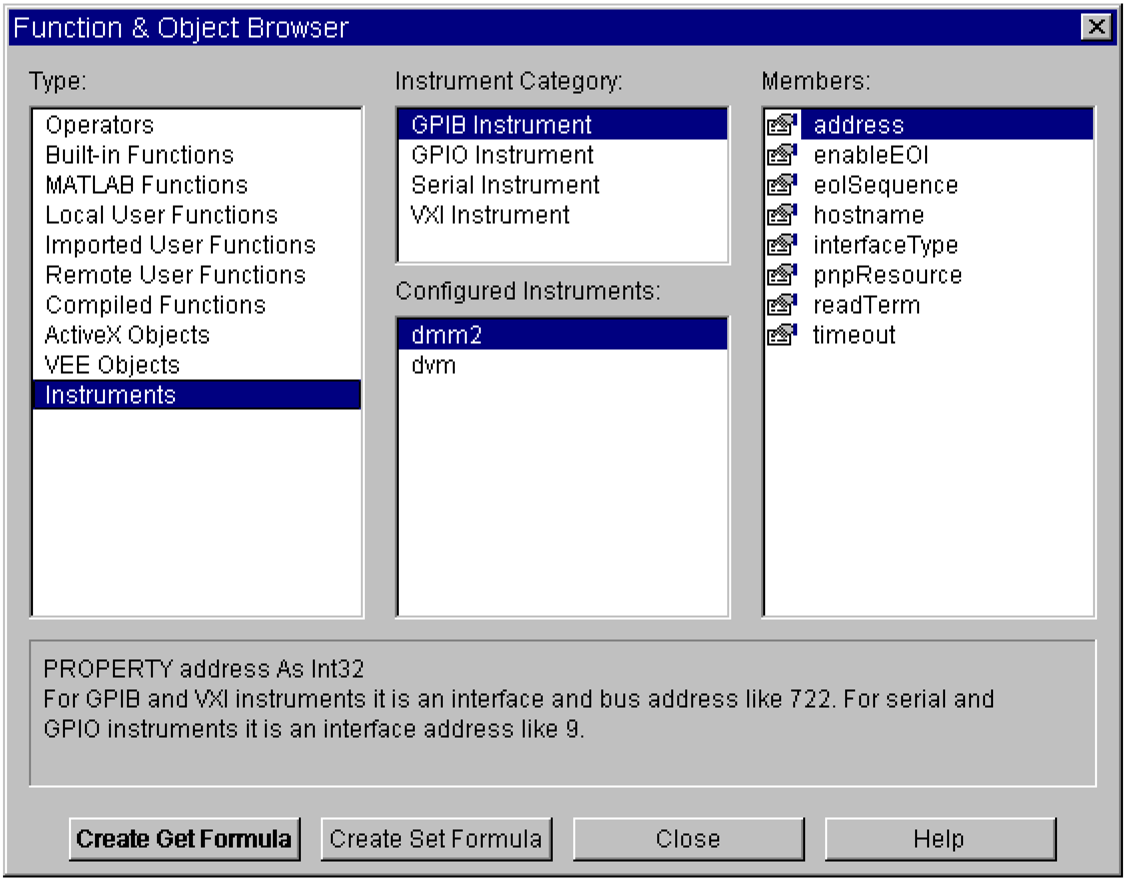


Figure 5-1. Function and Object Browser

Clicking Create Set Formula brings up the Formula Object dialog box shown in Figure 5-2.

Figure 5-2. Create Set Formula Dialog Box

Previous versions of VEE allowed programmatic configuration through control pins. These pins are obsolete but still supported. Control pins are available for the `Panel Driver`, `Component Driver`, and `Direct I/O` instrument control objects that let you input other values for device address and timeout. Control pins for setting timeout values are also available for the `Interface Operations`, `Instrument Event`, and `Interface Event` objects.

When a new timeout or address is sent to one of the control pins, the new value is changed globally for that device. This means that *all* instrument control objects communicating with a particular device begin using the new timeout or address value. The new value can be different from that entered in the Instrument Properties dialog box and placed in the VEE configuration file. However, this new value is *never* written to the VEE configuration file.

The example in Figure 5-3 shows a `Direct I/O` object with an `Address` control pin. The HP E1413B is originally configured for address 16032 as shown in the title bar. The input to the control pin is 16040 (the new address). When the control pin is sent the new address, 16040 is used for any other objects communicating with the HP E1413B.

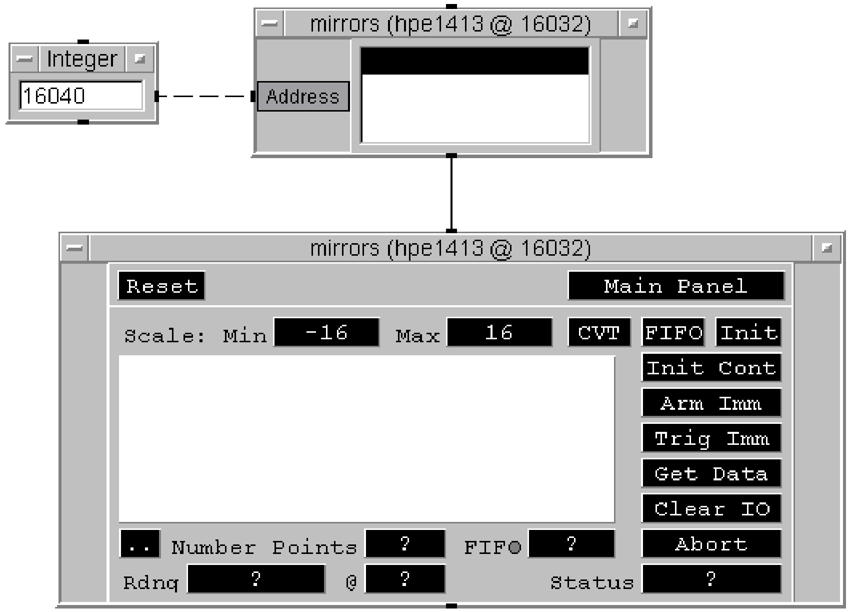


Figure 5-3. Programmatically Reconfiguring Device I/O

LAN Gateways

VEE can access LAN gateways to control instruments. A LAN gateway is a controller that allows access to its VXI, GPIB, GPIO, and Serial interfaces and the instruments on these interfaces from a remote process.

The client-server model best represents the arrangement. A VEE process acts as the client when accessing a LAN gateway on a remote computer, the server. The server has a committed process, known as a *daemon*, which is part of the SICL process running on the server. The daemon communicates with the VEE client and allows access to its interfaces and their devices.

The client process calls SICL in order to control devices on the interfaces that SICL supports. These interfaces are usually configured on the LAN gateway on which the SICL process is running. By using the LAN gateway, these interfaces can be on a remote computer.

As far as the client is concerned, the fact that the interfaces and their devices are attached physically to a remote computer is invisible.

Configuration

You must configure VEE and the LAN hardware to use the LAN gateway.

VEE Configuration. Configure VEE for gateway access during device configuration, as described in Chapter 3, "Configuring Instruments". Figure 5-4 shows the `Instrument Properties` dialog box. The `Gateway` field shows its default setting, `This host`:

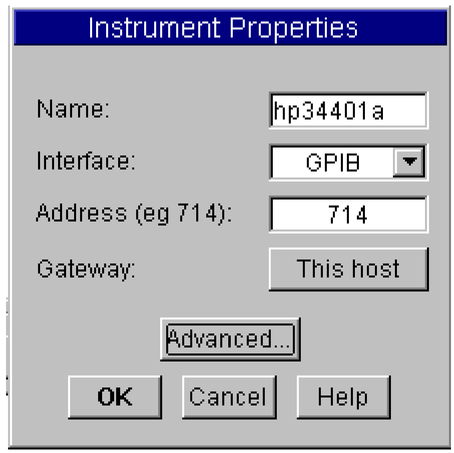


Figure 5-4. Gateway Configuration

You can select the gateway name by clicking the `Gateway` field. A list box appears showing all of the gateways that have been configured previously. `This host` always points to the computer on which VEE is running.

If there are no other choices for gateways, you can type in a gateway name. The name must be resolvable to an IP address either by a symbolic host name table or by a name-server. You can also enter an IP address in dot-format, such as `55.55.55.555`.

Beyond selecting a gateway, the configuration process remains the same. `Panel Driver` and `Direct I/O` objects are configured as before. Figure 5-5 shows various I/O devices configured for interfaces and devices on remote computers.

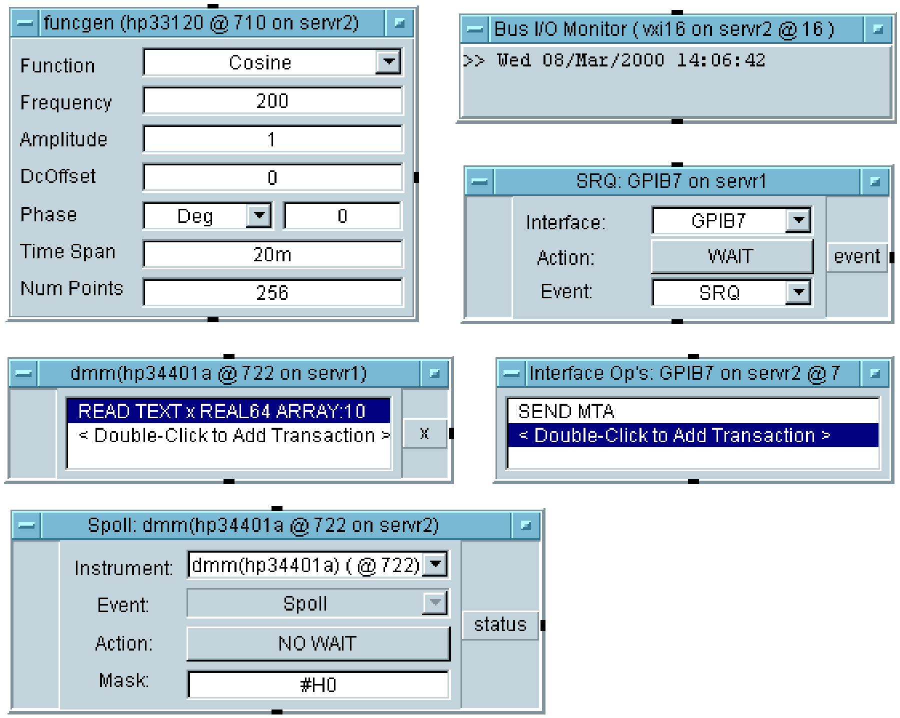


Figure 5-5. Examples of Devices Configured on Remote Machines

LAN Hardware Configuration. The SICL LAN gateway support depends on the configuration of the machine on which VEE is running, the machine on which the gateway daemon is running, and the overall configuration of the LAN. Consult with your system administrator to configure the LAN and ensure that names and IP addresses are resolvable.

For the machine running the gateway daemon it is assumed that the daemon install procedures will configure the local networking files correctly. If you are using the HP E2050A LAN/GPIB Gateway, it is self-contained and all internal configuration is done.

Execution Behavior

Ideally, I/O operations through the gateway work as if the interfaces and devices are attached directly to the client computer. However, response times can vary, depending on the LAN configuration, including the number of connected hosts, LAN-to-LAN gateways, and current load. Sometimes, a connection is terminated by disconnected cables or computer failures on the LAN. These events must be accommodated when configuring timeout periods.

When the server receives an I/O request from the client application, VEE, the server uses the timeout value that you enter in the `Instrument Properties` dialog box. This is called the SICL timeout. If the server's operation is not completed in the specified time, the server sends a reply to the client indicating that a timeout occurred and the normal VEE timeout error occurs.

When the client sends an I/O request to the server, the client starts a timer and waits for the reply from the server. If the server does not reply in time, a timeout occurs and an VEE timeout error is produced. This is called the LAN timeout.

The client timeout differs from the server timeout because the I/O transaction time for the server is usually different from the transmission time over the LAN. The server may complete an I/O transaction within five seconds (the VEE default timeout period), but the actual transmission over the LAN back to the client may take longer than five seconds.

Protecting Critical Sections

In a multi-process test system, sharing a resource among the processes requires a locking mechanism to protect critical sections. A critical section is needed when one of the processes needs exclusive access to a shared instrument resource.

To prevent another process from accessing the instrument during the critical section, the first process locks the instrument. The lock remains in effect for the time necessary to complete its task. During this time, the second process cannot execute any interaction with the instrument, including an attempt to lock the instrument for its own use.

The following EXECUTE transactions let you protect critical sections and can be used in the `Direct I/O`, `MultiInstrument Direct I/O`, and `Interface Operations` transaction objects. The transaction syntax varies depending on the interface and transaction object being used. For

GPIB, Serial, and GPIO, the entire interface is locked. For VXI, individual devices are locked.

To lock VXI devices via direct backplane access in the `Direct I/O` object, use the transactions:

```
EXECUTE LOCK DEVICE
EXECUTE UNLOCK DEVICE
```

In the `MultiInstrument Direct I/O` object, use the transactions:

```
EXECUTE vxiScope LOCK DEVICE
EXECUTE vxiScope UNLOCK DEVICE
```

where `vxiScope` is the configured name of a VXI oscilloscope such as the HP E1428B.

To lock GPIB, Serial, and GPIO Interfaces in the `Interface Operations` object, use the transactions:

```
EXECUTE LOCK INTERFACE
EXECUTE UNLOCK INTERFACE
```

Supported Platforms

Table 5-1. EXECUTE LOCK/UNLOCK Support

Platform	Supported I/O Interfaces
Windows 95 (PC, HP 6232, HP 6233, or EPC7/8)	GPIB[a] Serial VXI (PC with VXLink, or embedded)[b]
Windows NT (PC, HP 6232, HP 6233, or EPC7/8)	GPIB[a] Serial VXI (PC with VXLink, or embedded)[b]

a. The National Instruments GPIB interface does not support LOCK.
b. Register and memory access of VXI devices (READ/WRITE REGISTER/MEMORY transactions) are not lockable. Only the very first execution of a transaction that attempts a direct memory access could be locked out if the memory is mapped into the VEE process space) by a prior lock in another process. After that, there is no way to prevent multiple processes from simultaneously accessing a memory location since this is shared memory.

Execution Behavior

When a version of the EXECUTE LOCK transaction executes, it tries to acquire a lock on the device or interface. If there is no pre-existing lock owned by another process, the transaction executes completely and the lock acquisition succeeds. If a prior lock exists, the transaction blocks for the current timeout configured for that device or interface.

If the other process gives up the lock within the timeout period, the transaction completes and acquires the lock. If the timeout period lapses, an error occurs and an error message box appears. This error can be captured by an error pin on the transaction object.

After the lock is acquired, all subsequent I/O from `Direct I/O`, `MultiInstrument Direct I/O`, `Panel Driver`, `Component Driver`, and `Interface Operations` objects are protected from any other process attempting to communicate to that device or interface.

After the critical section has passed, the corresponding version of the EXECUTE UNLOCK transaction can be executed.

Locks only protect critical sections across process boundaries. A single process can create nested locks by performing two EXECUTE LOCK transactions in sequence. Both transactions will succeed as long as there are no prior locks by another process.

The process must then perform two EXECUTE UNLOCK transactions. If only one EXECUTE UNLOCK transaction is executed the device or interface remains locked. If a transaction attempts an unlock without a prior lock, a run-time error occurs.

Locks only exist while the VEE program is executing. When a VEE program finishes executing, all locks are removed from devices and interfaces. This protects the user from leaving devices or interfaces locked if the program stops executing due to normal completion, run-time errors, or a pressed `Stop` button, and no EXECUTE UNLOCK transaction has executed.

Example: EXECUTE LOCK/UNLOCK Transactions - GPIB

The example program in Figure 5-6 shows EXECUTE LOCK/UNLOCK INTERFACE transactions in an `Interface Operations` object configured for GPIB. (This example is identical for a serial interface.) The lock and unlock transactions frame the UserObjects performing I/O to the devices on the GPIB interface at logical unit 7. This program will attempt to acquire the lock three times. If the lock cannot be acquired after three attempts, a user-defined error occurs.

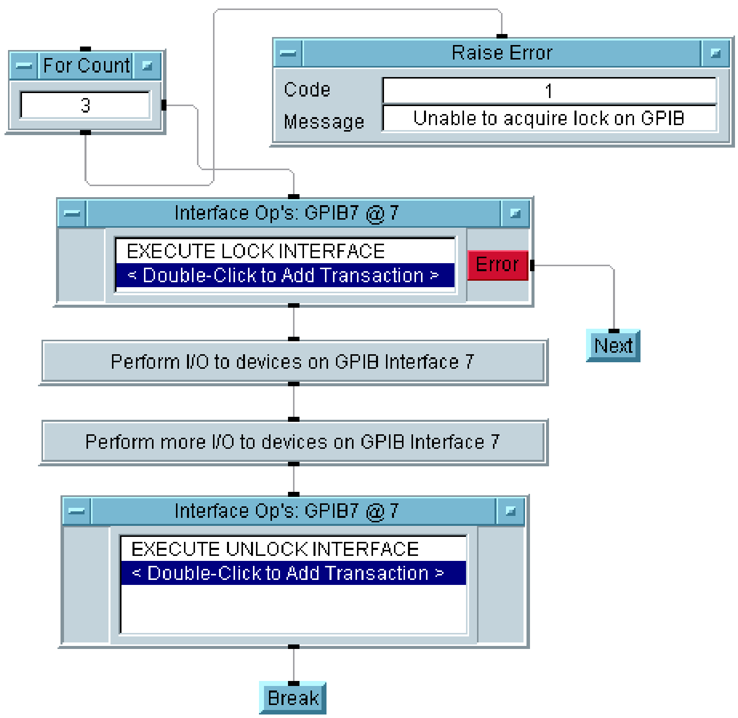


Figure 5-6. EXECUTE LOCK/UNLOCK Transactions - GPIB

For each attempt, the EXECUTE LOCK INTERFACE transaction tries to acquire the lock in the time allowed by the configured timeout period. You can set the timeout period in the `Properties` dialog box of the `Interface Operation` object. The error pin attached to the `Next` object in the first transaction object will cause the thread to be re-executed in another attempt. The break object after the last transaction object ensures that the thread does not get executed unnecessarily a second time.

Example: EXECUTE LOCK/UNLOCK Transactions - VXI

The example program in Figure 5-7 shows the EXECUTE LOCK/UNLOCK DEVICE transactions in a `MultiInstrument Direct I/O` object. You could use the `Direct I/O` object instead of the `MultiInstrument Direct I/O`, but that would mean using an object for each device instead of one object for the group of devices. This is very similar to the program in Figure 5-6. A `For Count` object drives a thread which tries to acquire locks on three different devices. After the I/O activity is done in the user objects, a series of unlocks are executed.

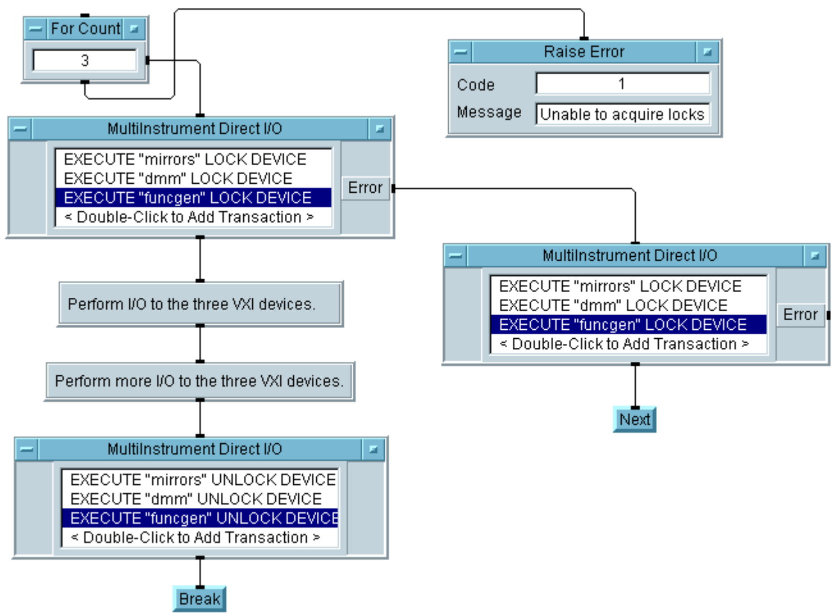


Figure 5-7. EXECUTE LOCK/UNLOCK Transactions - VXI

Each transaction tries to acquire its respective lock for the timeout period configured for each device. If any of the three transactions timeout, an error occurs that is trapped by the error pin. If a successful lock is followed by an attempt resulting in a timeout error, the error pin traps the error.

Before the program can re-execute the lock transactions, all acquired locks must be unlocked. That is why the `MultiInstrument Direct I/O` object is attached to the error pin. It is very important that this object try to unlock each device *in the same order* as the first object acquired the locks.

Since an error occurs if an unlock transaction is executed before the lock transaction, an error pin is also added to the object with the unlock transactions. If a transaction fails to acquire the lock in the first object, the same unlock transaction fails in the following object.

I/O Control Techniques

This section describes some additional techniques for instrument I/O control.

Polling

VEE supports all serial poll operations defined by IEEE 488.1. All GPIB instruments and all VXI message-based instruments support serial poll operations. VXI message-based devices are, by definition, IEEE 488.2 compliant. VXI register-based devices are IEEE 488.2 compliant if an I-SCPI driver is available. VEE does not support parallel poll operations.

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

Object	Serial Poll Behavior
`Instrument Event`	The `Instrument 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 `Instrument 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 GPIB 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. See *Chapter 4, "Using Transaction I/O"* for additional information about `Direct I/O`.

The `Instrument Event` object has special execution properties when configured for `Spoll` that are discussed in the next section, "Service Requests". This behavior allows 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 execute immediately and return the status byte of the GPIB or message-based VXI instrument. Both objects have a Timeout control input available from their object menus (Add Terminal) so you can programmatically set a timeout period. Figure 5-8 shows an example.

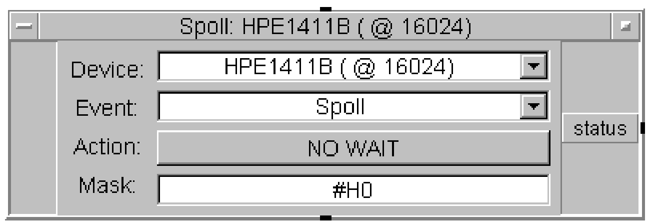


Figure 5-8. Instrument Event Configured for Serial Polling

Service Requests

To detect a service request (SRQ message) for a VXI instrument, use the Instrument Event object (I/O ⇒ Advanced I/O ⇒ Instrument Event). To detect a service request for a GPIB instrument or RS-232, use the Interface Event object (I/O ⇒ Advanced I/O ⇒ Interface Event).

The Instrument Event and Interface Event objects provide special behavior for interrupt-like execution. To view this behavior, you may wish to run your program with Debug ⇒ Show Execution Flow enabled.

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

1. Before an Interface Event object (configured for GPIB and with the WAIT option specified) operates, execution proceeds normally with each thread sharing execution with equal priority.

2. When an 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 *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.

Example: Service Request. The program in Figure 5-9 shows one way to handle service requests. In this example, it is possible that either `dvm` or `scope` is responsible for a service request. This program is saved in the file `manual16.vee` in the examples directory.

Note

The program in Figure 5-9 will run only if the specified instruments are connected, configured, and powered up. However, you can use this program as an example of programming techniques to use in your own programs or you can modify the program to communicate with your own instruments.

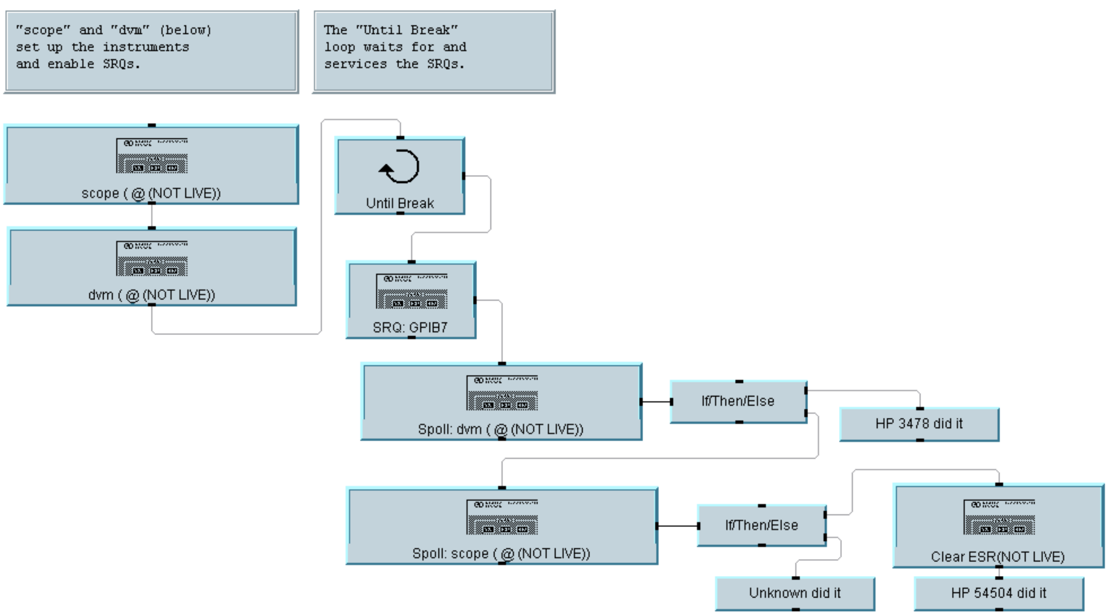


Figure 5-9. Handling Service Requests

The program determines the originator of the service request by using `Instrument 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, 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 `Instrument 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 `Instrument Event` object is configured for `ANY SET`, the `If/Then/Else` and `bit(x,n)` function need not be used.

Different instruments have different requirements for clearing and re-enabling service requests. In Figure 5-9, `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 `Instrument 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 receives a *Read STB* word serial protocol command.

The message-based instrument sends its status byte back to the controller, and writes a request false event (RF) into the VXI controller's signal register. The status byte is 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 `Instrument Event`, can be used to detect a service request from a message-based VXI device and determine why the request occurred.

Both objects have a `Timeout` control input available from their object menus (`Add Terminal`) so you can programmatically set a timeout period. For further information, see the `Instrument Event` and `Interface Event` reference sections in the *VEE Online Help*.

Monitoring Bus Activity

You can use the `Bus I/O Monitor` object (`I/O` ⇒ `Bus I/O Monitor`) to record all bus messages transmitted between VEE and any talkers and listeners. `Bus I/O Monitor` records *only* those bus messages inbound or outbound from VEE.

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

Figure 5-10 shows the bus messages sent to write `*RST` to an instrument at GPIB address 717.

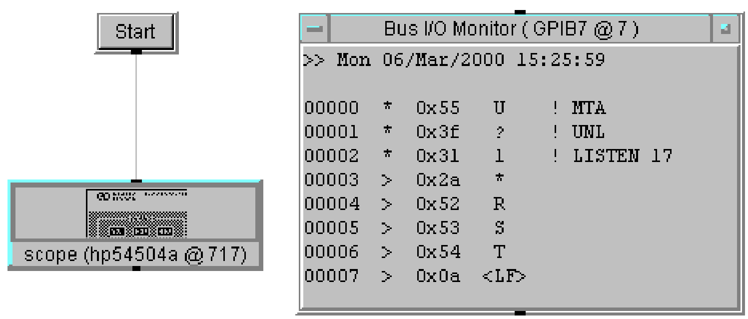


Figure 5-10. The Bus I/O Monitor

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)

The `Bus I/O Monitor` executes much faster as an icon than as an open view object.

Low-Level Bus Control

You can send low-level bus messages in two ways, as Figure 5-11 shows.

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

For further information regarding `Interface Operations` and `Direct I/O`, see Chapter 4, "Using Transaction I/O".

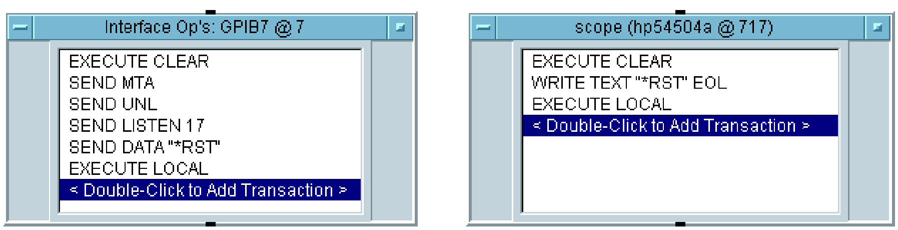


Figure 5-11. Two Methods of Low-Level GPIB Control

Instrument Downloading

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

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.

This section presents a complete example of downloading using this approach. See Chapter 4, "Using Transaction I/O" for further information regarding `Direct I/O`.

Figure 5-12 shows a program that downloads a measurement subprogram to the HP 3852A. This example downloads subprogram `BEEP2` that beeps twice and displays a message. This program is saved in the file `manual17.vee` in the examples directory.

Note The program in Figure 5-12 will run only if the specified instruments are connected, configured and powered up. However, you can use this program as an example of programming techniques to use in your own programs or you can modify the program to communicate with your own instruments. This program, `manual17.vee`, has embedded configuration.

Below are the contents of the downloaded file `manual17.dat`. The `manual17.dat` file is provided in the examples directory.

```
DISP MSG "LOADING BEEP2"
WAIT 1

SUB BEEP2
DISP "BEEP2 CALLED"
BEEP
WAIT .5
BEEP
SUBEND

DISP MSG "BEEP2 LOADED"
```

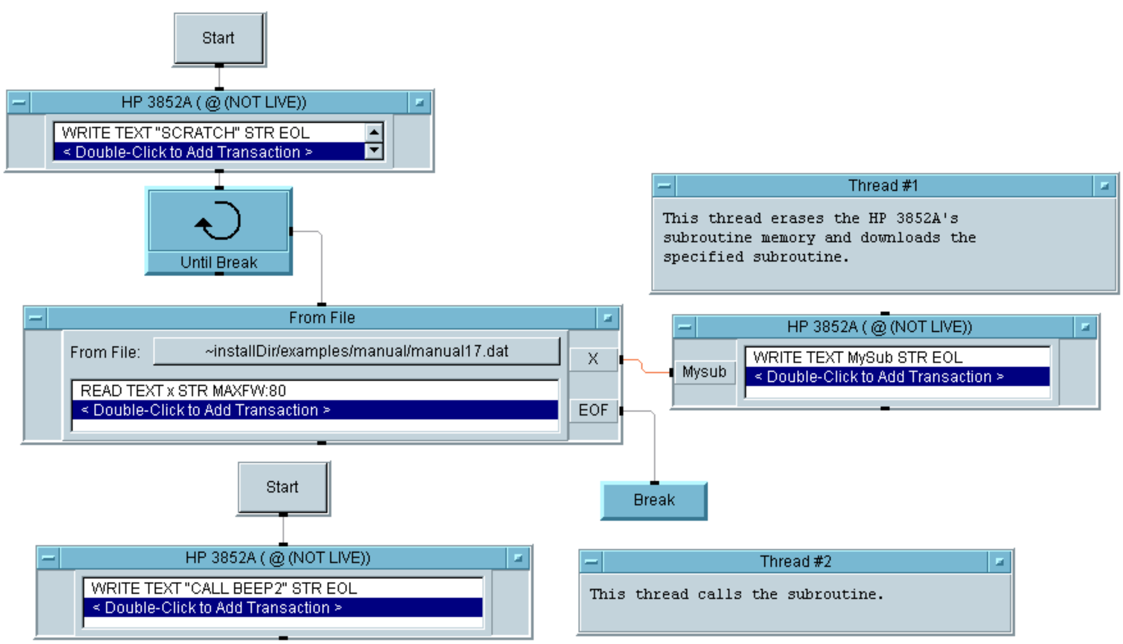


Figure 5-12. Example: Downloading to an Instrument

Logical Units and I/O Addressing

To access an I/O device, you will need to determine the correct address and enter it in the `Address` field in the `Instrument Properties` dialog box, using the `Instrument Manager` as described in Chapter 3, "Configuring Instruments".

This section covers the VEE I/O addressing scheme, including interface logical units and instrument addresses, that supports `Direct I/O`, `Panel Driver`, and `Component Driver` I/O operations. This addressing scheme is *not* used for VXI*plug&play* I/O operations. See "Configuring for a VXIplug&play Driver" on page 69 for information about VXI*plug&play* addressing in VEE.

Note VEE supports the GPIB, RS-232 serial, and GPIO interfaces. Also, you can access VXI devices by using an HP E1406 Command Module connected to one of the supported GPIB interfaces.

VEE also supports direct VXI backplane access for embedded VXI controllers, for the E1383A and E1483A VXLink interfaces for PCs, and for the HP E1489C EISA/ISA-to-MXIbus interface with HP 9000 Series 700 computers.

The VEE addressing scheme uses logical units that you can set up using the `I/O Config` utility program as part of installing and configuring the I/O libraries included with VEE. See *Installing the Agilent I/O Libraries (VEE for Windows)* for information about installing and configuring the HP I/O libraries, and setting up logical units using `I/O Config`. It is recommended that you set logical units for interfaces according to Table 5-2.

Recommended I/O Logical Units for VEE

The following interface logical units are recommended for use with VEE. See *Installing the Agilent I/O Libraries (VEE for Windows)* for information about installing and configuring the I/O libraries and setting up logical units for interfaces using the `I/O Config` utility program.

Table 5-2. Recommended I/O Logical Units

Logical Unit	PC (Windows 95, NT)
1	GPIB (82340 or 82341)
2	GPIB (82340 or 82341)
3	GPIB (82340 or 82341)
4	GPIB (82340 or 82341)
5	GPIB (82340 or 82341)
6	GPIB (82340 or 82341)
7	GPIB (82340 or 82341)
8	GPIB (82340 or 82341)
9	COM1 serial port
10	COM2 serial port
11	COM3 serial port
12	COM4 serial port
13	GPIO (HP E2075)
14	GPIB0 (National GPIB card)
15	GPIB1 (National GPIB card)
16	VXI (Embedded, or PC using VXLink)
17	GPIB2 (National GPIB card)
18	GPIB3 (National GPIB card0

Note

Logical unit 7 is the recommended default for the *first* GPIB card. Each card must have a unique logical unit.

The 82335 GPIB card is also supported for Windows 95/98 on the PC (*not* for Windows NT). However, only logical units 3 through 7 are recommended for the 82335 GPIB card and the logical unit is set by the on-card switch settings (the default setting is 7). In addition, you must exclude address space for the 82335 as described in "Excluding Address Space for the 82335 Card (Windows 95/98 Only)" on page 177.

Only logical units 14, 15, 17, and 18 are supported for National GPIB cards on the PC.

I/O Addressing

Addressing schemes for various types of devices are described in the following sections.

To Address Serial Ports

Serial ports are supported by using the logical units that you assign to them using `I/O Config`. Normally, the COM1 serial port is assigned logical unit 9 (see Table 5-2). In this case, use `9` as the address of the device connected to COM1.

To Address GPIO Devices

GPIO devices are supported by using the logical unit that you assign to the GPIO interface using `I/O Config`. Normally, the logical unit 13 is used for GPIO. In this case, use `13` as the address for the GPIO device.

To Address GPIB Interfaces and Devices

GPIB devices are addressed using the following scheme:

SPA[*SA*]

Where:

S	is the logical unit of the GPIB interface.
PA	is the primary address of a GPIB device (the valid range is 00 through 30).
SA	is the optional secondary address (the valid range is 00 through 31).

Two examples are:

- For a GPIB device at logical unit 7, primary address 01, enter `701` in the `Address` field of the `Instrument Properties` dialog box.
- For a GPIB device at logical unit 14, primary address 09, secondary address 02, enter `140902` in the `Address` field of the `Instrument Properties` dialog box.

GPIB Logical Units. GPIB interfaces are supported by using the logical units that you assign to them using `I/O Config`. The recommended logical units for GPIB interfaces are as listed in Table 5-2. If the recommended logical units (1 through 8) are configured by the I/O libraries for GPIB interfaces, VEE can theoretically access up to eight GPIB cards, which can be a mix of the supported cards:

- For an E2070, the logical unit is assigned by the software. The logical units are assigned in the order: 7, 8, 1, 2, 3, 4, 5, and 6. However, each card must be set to a unique base address. (See the owner's manual for information on setting the base address.)
- For an 82340 or 82341 (for PCs), the logical unit is assigned by the software. The logical units are assigned in the order: 7, 8, 1, 2, 3, 4, 5, and 6. However, each card must be set to a unique base address. (See the owner's manual for information on setting the base address.)

- For an 82335 (for PCs, Windows 95/98 only), the logical unit is determined by switch settings on the card (the default is 7). If you install more than one 82335 card, each card must be set for a unique logical unit in the range 3 through 7. (See the owner's manual for instructions.) Also, you must exclude address space for each card. See "Excluding Address Space for the 82335 Card (Windows 95/98 Only)" on page 177.

GPIB Logical Units (PCs Only). The National Instruments GPIB driver configures up to four GPIB cards with the designations GPIB0, GPIB1, GPIB2, and GPIB3. To access these GPIB cards, you *must* assign the logical units 14, 15, 17, and 18 to the GPIB cards (see Table 5-2) using `I/O Config`. VEE does not support any other logical units for GPIB cards. Otherwise, the addressing is the same as for any other GPIB card.

To Address VXI Devices on the GPIB

To access VXI devices through the GPIB with an HP-IB command module, you can use secondary addresses. If you are using an HP E1406 Command Module in a VXI card cage, the primary address is set by a switch on the command module (default = 09) and the secondary address is the individual VXI device's logical address divided by eight.

For example, suppose you have an HP E1406A Command Module (address = 09) in a C-Size Mainframe, with the HP E1406A connected to the GPIB interface at logical unit 7. For an HP E1326B Multimeter in a VXI slot with its logical address set to 24, the multimeter address is `70903`.

Two instrument drivers are provided to help you find the correct addresses for VXI devices connected by means of a GPIB command module:

- Use the `hpe140x.cid` driver to locate VXI devices connected by means of an HP E1405 or HP E1406 GPIB Command Module in a C-size VXI mainframe.

- Use the `hpe1300a.cid` driver to locate VXI devices connected by means of an HP E1306 GPIB Command Module in a B-size VXI mainframe. (This driver also supports the HP E1300 and HP E1301 B-Size VXI Mainframes, which include built-in command modules.)

To use either of these drivers, add an instrument panel for the driver using the `Instrument Manager` as described in Chapter 3, “Configuring Instruments”.

Note Do not enter a sub address value for VXI devices, except for modules in a VXI switch box. See the next section for details.

To Set Address/Sub Address Values

Most GPIB and VXI instruments do not use sub addresses. Do not enter a sub address value unless you are accessing a VXI switch box or one of the mainframe instruments that use sub addresses, such as the HP 3235A Switch/Test Unit or the HP 3488A Switch/Control Unit.

Note Sub address values are used only if you are using an HP Instrument Driver for a device that supports sub addresses. Do not use sub address values if you are using Direct I/O.

Two examples follow:

- To access a module in an HP 3235A Switch/Test Unit, enter the GPIB address (for example, `701`) of the HP 3235A itself in the `Address` field of the `Instrument Properties` dialog box, using the `Instrument Manager` as described in Chapter 3, “Configuring Instruments”.

  Enter the sub address of the individual module in the `Sub Address` field of the `Advanced Instrument Properties` dialog box (on the `Panel Driver` tab). For information on entries in the `Sub Address` field, see online help for the HP 3235A instrument driver (`Help` ⇒ `Instruments`).

- To access a module in a VXI switch box, enter the GPIB address of the switch box (for example, `70902`) in the `Address` field and the sub address of the individual module in the `Sub Address` field. For information on entries in the `Sub Address` field, see online help for the VXI switch box instrument driver.

To Address the VXI Backplane Directly

VEE can address the VXI backplane directly for the following systems:

- An HP 623x VXI Pentium Controller.
- An EPC-7 or EPC-8 VXI Controller, provided the EPConnect software is installed.
- A PC connected to a VXI card cage using an HP E1383A or HP E1483A VXLink (ISA-to-VXI) interface, provided the EPConnect software is installed.

Assuming recommended logical units have been set using `I/O Config` (see Table 5-2), VEE accesses the VXI backplane via logical unit 16. The address for a VXI device is the logical unit (16) with the logical address of the VXI device appended.

For example, suppose an HP EPC-7 VXI Controller and an HP 1411B Digital Multimeter are installed in a VXI mainframe. If the logical address of the HP 1411B is set to 24, the VXI address is `16024`. You do not divide the logical address by 8 as you would if you were accessing the VXI device via GPIB.

Excluding Address Space for the 82335 Card (Windows 95/98 Only)

For an 82335 card, which uses memory-mapped I/O addressing, you must exclude the address space required by the GPIB interface so memory manager programs will not try to use that space.

Note

The 82340 and 82341 cards and the National Instruments GPIB cards do not use memory-mapped I/O addressing, so this section does not apply to those cards. Also, this section does not apply to the built-in GPIB interface for an embedded controller.

The 82335 card is supported for Windows 95/98 only, not for Windows NT.

To exclude address space:

1. Install the 82335 card. The card is pre-set at the factory for logical unit 7. Normally, you should use logical unit 7. However, if there is more than one 82335 card, each card must be set for a different logical unit in the range 3 through 7.

2. Add the appropriate line for your logical unit to the `[386Enh]` section of your SYSTEM.INI file (in the C:\Windows directory):

For Logical Unit:	Add to SYSTEM.INI:
3	`EMMEXCLUDE=0CC00-0CFFF`
4	`EMMEXCLUDE=0D000-0D3FF`
5	`EMMEXCLUDE=0D400-0D7FF`
6	`EMMEXCLUDE=0D800-0DBFF`
7 (default)	`EMMEXCLUDE=0DC00-0DFFF`

3. If there is a memory manager DEVICE line (for example, `DEVICE=EMM386.EXE`) in the CONFIG.SYS file (in the root directory), you will need to modify the file. Add a parameter to exclude the address space (for example, `X=DC00-DFFF` for logical unit 7), as shown in the following table:

For Logical Unit:	Modify in CONFIG.SYS:
3	`DEVICE=EMM386.EXE X=CC00-CFFF`
4	`DEVICE=EMM386.EXE X=D000-D3FF`
5	`DEVICE=EMM386.EXE X=D400-D7FF`
6	`DEVICE=EMM386.EXE X=D800-DBFF`
7 (default)	`DEVICE=EMM386.EXE X=DC00-DFFF`

Note

If multiple 82335 cards are installed, you must exclude address space for each card. For example, for two cards installed (logical units 3 and 7), add the following lines to the `[386Enh]` section of SYSTEM.INI:

```
EMMEXCLUDE=0CC00-0CFFF
```

```
EMMEXCLUDE=0DC00-0DFFF
```

Also, if your CONFIG.SYS file contains the DEVICE line for EMM386.EXE, add parameters to it as shown:

```
DEVICE=EMM386.EXE X=CC00-CFFF X=DC00-DFFF
```

4. Reboot your computer (select `Start` ⇒ `Shut Down`) and restart Windows.

6

Using Panel Driver and Component Driver Objects

Using Panel Driver and Component Driver Objects

This chapter describes how to use `Panel Driver` and `Component Driver` objects with VEE.

Understanding Panel Driver and Component Driver Objects

This section explains some background and details that will help you use `Panel Driver` and `Component Driver` objects more effectively.

Inside Panel Drivers

The VEE `Panel Driver` and `Component Driver` objects both require that the appropriate Panel Driver ("ID") be present, and that the instrument be configured to that driver. These instrument drivers are sometimes called "VEE drivers" or "Instrument Drivers". The Panel Driver file (the `.cid` file) must be present and configured to use `Panel Driver` and `Component Driver` objects. However, these files are not used for `Direct I/O` or VXI*plug&play* operations.

Panel Driver Files

Each **Panel Driver** describes the unique personality of a particular test instrument. A driver file is required to control any instrument using a `Panel Driver` or `Component Driver` object.

Panel Driver files (`.cid` files) are optionally copied onto your system disk when VEE is installed. Each driver file contains two basic types of information:

- A description of the instrument's functions and the commands used to set and query them.
- A description of the appearance and behavior of the graphical control panel visible in the open view of a `Panel Driver` object.

Components

Internally, `Panel Driver` and `Component Driver` objects 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 instrument function settings or measured values.

Figure 6-1 shows some of the components in the HP 3478A voltmeter.

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.

Access components interactively or through a program. To access a component interactively, click a labeled button or display in the open view of a `Panel Driver`. To access components using a graphical program, add them as input or output terminals. Figure 6-1 shows an example. For detailed procedures on using components, see “Selected Techniques” on page 189 and “Using Component Driver Objects in a Program” on page 190.

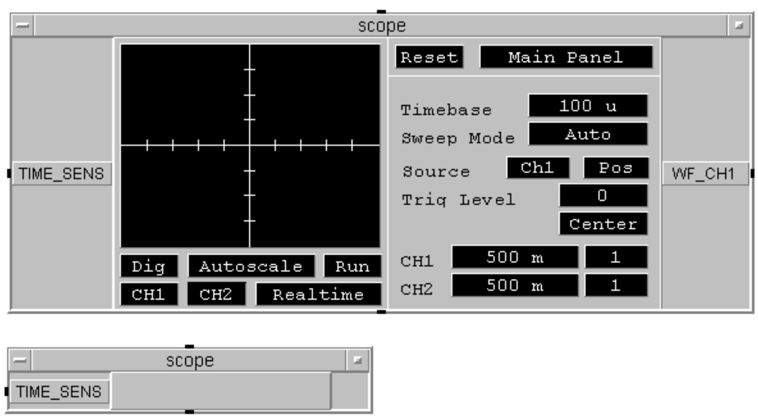


Figure 6-1. Accessing Driver Components

States

An instrument **state** is a specific set of values for all components in a particular driver. You set all the components in a voltmeter driver to particular values for AC voltage measurements. You use a different set of component values to measure DC current. These are two states for the voltmeter. Figure 6-2 shows two voltmeter states.

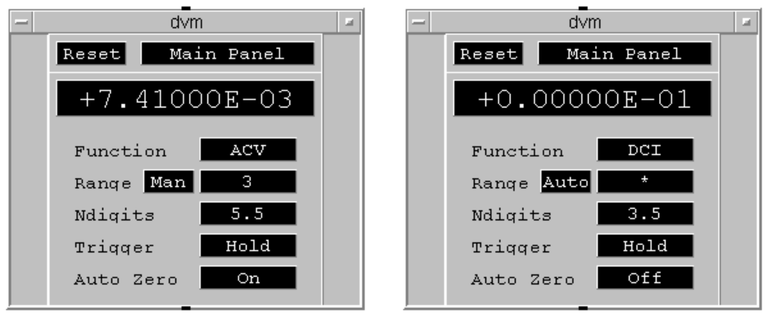


Figure 6-2. Two Voltmeter States

In VEE, each instance of a Panel Driver represents a separate measurement state. (Panel Driver objects are often called "state drivers".) It is common to have more than one Panel Driver in a program, where each Panel Driver programs the same physical instrument to a unique measurement state.

Each Panel Driver object you create using the same instrument Name communicates with the same physical instrument.

How Panel Driver-Based I/O Works

When you place a `Panel Driver` or `Component Driver` object in a program, VEE establishes a state record in memory. This state record is specific to a particular instrument `Name`. `Names` are discussed in greater detail in “The Importance of Names” on page 187.

All driver-based objects that reference a particular `Name` share a single state record. The state record reflects the *current* values of each of the instrument's components. When you write to components using `Panel Driver` or `Component Driver` objects, VEE updates both the physical instrument and the state record.

If you write to the instrument using `Direct I/O`, VEE marks the state record as invalid because the state record no longer matches the true state of the physical instrument. However, subsequent use of a `Panel Driver` or `Component Driver` object causes VEE to recall the instrument's state, which resynchronizes the physical instrument state and state record.

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

Panel Driver Operation

When a `Panel Driver` operates, it sends only those commands necessary to make the state of the physical instrument match the state defined in the graphical control panel.

If necessary, a `Panel Driver` sends commands to reset and update all settings in the corresponding physical instrument. This behavior is affected by the `Incremental Mode` setting described in Chapter 3, "Configuring Instruments".

If you set `Incremental Mode` to `ON`, VEE compares the current state record to the desired state defined in the `Panel Driver` and determines which components must be changed. 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, VEE explicitly sends commands to update each component in order to guarantee synchronization between the desired state and the state of the physical instrument.

A `Panel Driver` operates when its sequence input pin is activated *or* when you click one of the control panel buttons visible in the open view.

Component Driver Operation

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. That is why `Component Driver` objects generally operate faster than `Panel Driver` objects. A `Panel Driver` potentially writes to *many* components to achieve a particular state; a `Component Driver` writes to only the components specified.

Components are read and written in the order they appear as terminals, from top to bottom. This order of operation is important in 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

Some situations that can be confusing when using multiple objects that:

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

The Importance of Names. The `Name` field in the `Instrument Properties` dialog box 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 `Show Config` in the object menu. The text in the object title is *not* necessarily the same as the `Name`.

In general, only one configured `Name` should reference a particular physical instrument. Multiple `Name` references to the same instrument address causes unpredictable results in a program using `Panel Driver` objects. VEE's internal records of instrument states are organized by `Names`. Two `Panel Driver` objects with different names will blindly write to the same address, invalidating each other's state records.

In some cases involving `Direct I/O`, you may need more than one `Name` for the same physical instrument. This may be necessary if certain settings in the `Direct I/O` tab of the `Advanced Instrument Properties` dialog box need to be varied depending on the direct I/O operation.

For example, you may want to send some commands to an oscilloscope with EOI asserted on the last character of data and some commands without EOI. In this 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`.

The configured Name appears as the default title in instrument objects when you select them from the menu. 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, VEE looks in the current I/O configuration for the Name of each instrument being loaded.

For example, if you saved a program containing a Direct I/O object with a name of My_Scope, there must be an instrument named My_Scope in the current I/O configuration. If the object under consideration is a Panel Driver or Component Driver, the ID Filename (driver file) in the current I/O configuration must match the one used in the saved program.

Names must match *exactly*, including spaces, except that Name is not case-sensitive.

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, VEE maintains a separate state record for each name; a Panel Driver for Supply1 will have no effect on anything related to Supply2 or Supply3.

Selected Techniques

This section describes some techniques for using `Panel Driver` and `Component Driver` objects interactively or in a program.

Using Panel Driver Objects Interactively

The open view of a `Panel Driver` object 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 the corresponding field in the graphical control panel and complete the resulting dialog box. To make a measurement and view the result, click the display region of a numeric or XY display. XY displays may take a few seconds to update.

Using Panel Driver Objects Programmatically

To add a `Panel Driver` object to your program:

1. Click `I/O` ⇒ `Instrument Manager`.... The `Instrument Manager` dialog box appears.

2. Click the desired instrument to highlight it and then click the `Panel Driver` button.

Note The `Panel Driver` button is inactive ("grayed out") if the instrument has not been configured with a Panel Driver file. See Chapter 3, “Configuring Instruments” for configuration procedures.

3. When the object outline appears, position the cursor and click once to place the object in the work area.

To use Panel Driver objects 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 Add Terminal ⇒ Data Input or Add Terminal ⇒ Data Output from the Panel Driver object menu. A list box appears that lists all the valid driver components not yet used as terminals. Double-click the component in the list that you wish to add as a terminal.
- Select Add Terminal by Component ⇒ Select Input Component or Add Terminal by Component ⇒ Select Output Component from the Panel Driver object menu. After making this selection, click 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 want 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 Driver Objects in a Program

To add a Component Driver object to a program:

1. Click I/O ⇒ Instrument Manager.... A list of configured instruments appears.
2. Click the desired instrument to highlight it and then click the Component Driver button.

Note

The Component Driver button will be inactive ("grayed out") if the instrument has not been configured with a Panel Driver file. See Chapter 3, “Configuring Instruments” for configuration procedures.

3. When the object outline appears, position the pointer and click once to place the object in the work area.

Component Driver objects are generally used when you need to repeatedly execute an instrument control object while changing only a few components. Component Driver objects are preferred over Panel Driver objects in these situations because Component Driver objects write and read only the components you specify and execute somewhat faster.

Figure 6-3 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 Panel Driver objects are still appropriate for the initial set up of fgen and dvm.

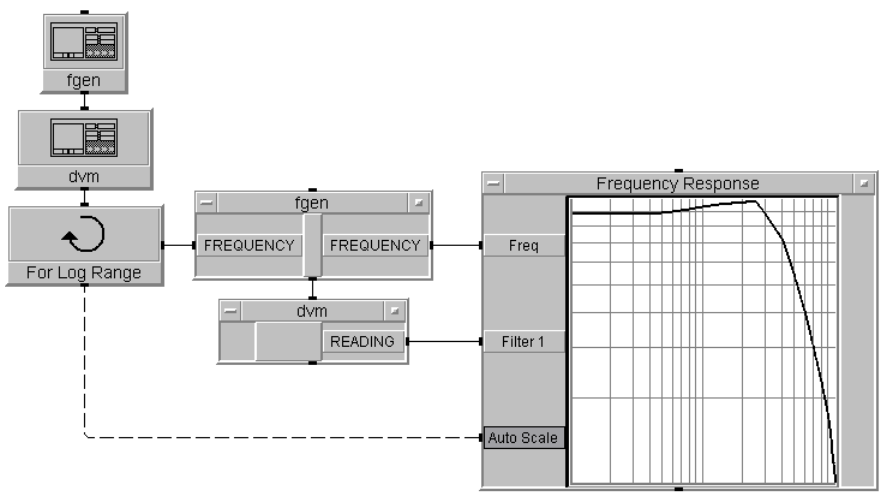


Figure 6-3. Using Panel Drivers and Component Drivers

The program shown in Figure 6-3 is stored in the file manual15.vee in the examples directory.

Getting Panel Driver Help

To obtain help about an Panel Driver, select `Help` from the object menu of a `Panel Driver` or `Component Driver` object. Then, open the appropriate help topic from the resulting dialog box.

7

Using VXI*plug&play* Drivers

Using VXI*plug&play* Drivers

To use a VXI*plug&play* driver to communicate with an instrument, you must install the appropriate VXI*plug&play* driver files and the VISA I/O library. See “Introduction to VXIplug&play” on page 38 for VISA installation information. You must also configure VEE for the instrument as described in “Configuring for a VXIplug&play Driver” on page 69.

The primary means of communicating with a VXI*plug&play* driver in VEE is the `To/From VXIplug&play` object, described in the following section. You can also call VXI*plug&play* functions from VEE `Call` objects (see “Using VXIplug&play Functions from Call Objects” on page 210.) The latter method is provided for backward compatibility with VEE Version 3.1.

Note

Program Compatibility:

Previous versions of VEE have supported VXI*plug&play* drivers. VEE Version 3.2 provided the `To/From VXIplug&play` object. VEE 3.2 programs using this object are compatible with later versions of VEE.

VEE Version 3.1 provided *only* direct `Call` access to VXI*plug&play* drivers. If you used `Call` objects to control VXI*plug&play* instruments in VEE Version 3.1, your program will work in later versions of VEE after you make certain changes to use the 32-bit version of the driver.

You must install the Windows 95/98 version of VISA and the 32-bit version of the VXI*plug&play* driver, and you may have to change the `Import` objects to use the new location of the VXI*plug&play* driver files. For more information on using `Call` objects to access VXI*plug&play* drivers, see “Using VXIplug&play Functions from Call Objects” on page 210.

Using the To/From VXIplug&play Object

After you have added VXI*plug&play* instruments to the VEE instrument configuration, you can use the VXI*plug&play* drivers in your program. Access the instruments by the functions contained in the drivers. The `To/From VXIplug&play` object provides access to the VXI*plug&play* function panels.

To get the `To/From VXIplug&play` object:

1. Select `I/O` ⇒ `Instrument Manager`. The `Instrument Manager` appears and displays all currently configured VXI*plug&play* instruments (as well as any other instruments that are configured).

2. Select the instrument with which you want to communicate, and click the `Plug&play Driver` button. The outline of the object appears.

3. Place the outline of the `To/From VXIplug&play` object where you want it in the work area and click the mouse button. The object appears as shown in Figure 7-1.

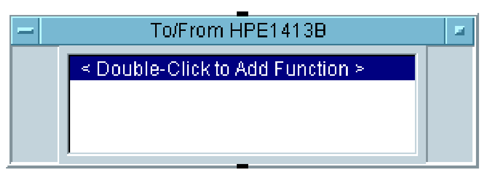


Figure 7-1. To/From VXIplug&play Object

Selecting a Function

Select the VXI*plug&play* functions from the `To/From VXIplug&play` object.

1. Double-click an empty transaction or select `Add Trans` or `Insert Trans` from the object menu. The `Select a Function Panel` dialog box appears. It displays function panels grouped into logical categories, such as `Measure` or `Configure`, as shown in Figure 7-2. Each driver has different categories.

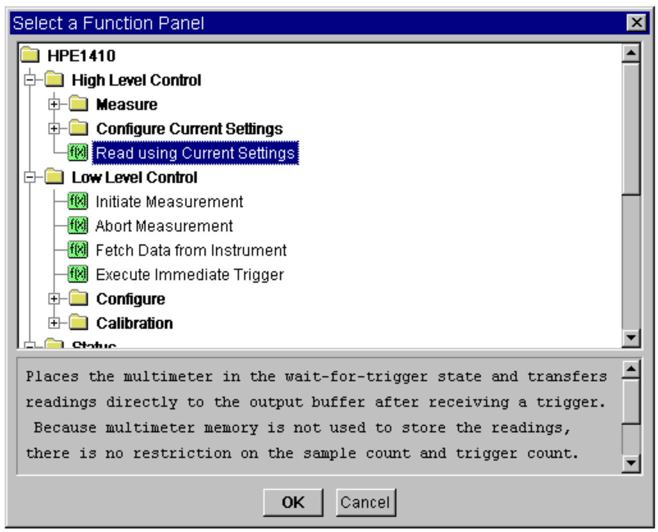


Figure 7-2. Select a Function Panel Dialog Box

- ❑ Click the `[+]` icons to view the hierarchical structure of function panels.

- ❑ Click the `[-]` icons to hide the function panels in the hierarchical structure.

- ❑ Click the `[f(x)]` icons to select the function panel. You will see a short description of the function panel in the lower part of the dialog box.

To completely expand a branch of the tree, select the item to expand and press the ***** key.

Generally, you will see only function panels that adhere to the VXI*plug&play* version 3.x specification and are allowed by VEE.

Note

VEE automatically calls `init()` at the appropriate time. However, there may be other initialization functions, such as `init_all()`, `init_next()`, or `init_first()` in the list. These functions are not defined in the VXI*plug&play* specification and are not supported by VEE.

Do not select these functions. If you must use these functions, you need to create your program differently and call the VXI*plug&play* driver from a `Call` object as described in "Using VXIplug&play Functions from Call Objects" on page 210.

There are no entries for *PREFIX*_`init()` or *PREFIX*_`close()`. These functions are performed automatically by VEE.

2. Click `OK` on the `Select a Function Panel` dialog box.
3. You see a tabbed dialog box called `Edit a Function Panel` that allows you to specify the parameters for the function panel.

Editing Function Panel Parameters

The `Edit a Function Panel` dialog box allows you to set controls and variables to pass to the selected VXI*plug&play* driver's function. There are two tabs, `Panel` and `Configuration`.

The Panel Tab. The `Panel` tab, shown in Figure 7-3, allows you to specify the constant (control) values to pass to the function.

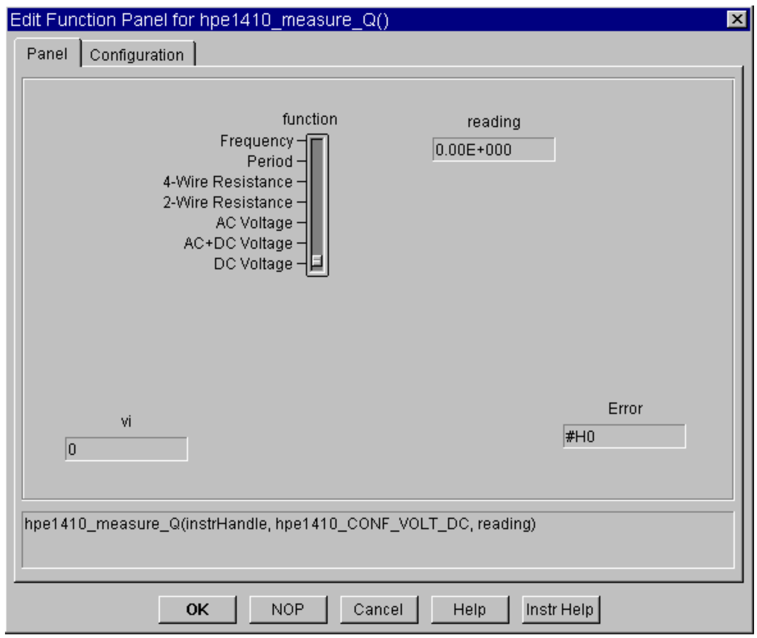


Figure 7-3. Panel Tab of Edit Function Panel Dialog Box

- Controls - The top part of this dialog box contains controls to specify constant parameters. The names of the controls are labels specified from the function panel file.
- `vi` - Displays the unique "virtual instrument" handle (also called the "session handle") of the instrument. Depending on the driver version, the name of this field may change, but the location is always in the lower-left corner of the function panel.

- `Error` - Displays a non-zero value if an error occurred when executing this function panel. Depending on the driver version, the name of this field may change, but the location is always in the lower-right corner of the function panel.
- Function call - At the bottom of the dialog box is the C function and the parameters that are sent to the driver when the object executes. This command string is also shown as a transaction on the open view of the object.

Getting Help on a VXIplug&play Function Panel. In the `Edit Function Panel` dialog box, click the right mouse button on the background of the `Panel` tab for help on the function panel. A dialog box containing a description of the function appears.

Click the right mouse button on a control (not the label) for an explanation of the parameter.

For complete help on the VXI*plug&play* driver, select `Instrument Help` from the object menu of the `To/From VXIplug&play` object.

The Configuration Tab. The `Configuration` tab, shown in Figure 7-4, allows you to specify the variables to pass to the function. This allows you to set the parameter values programmatically.

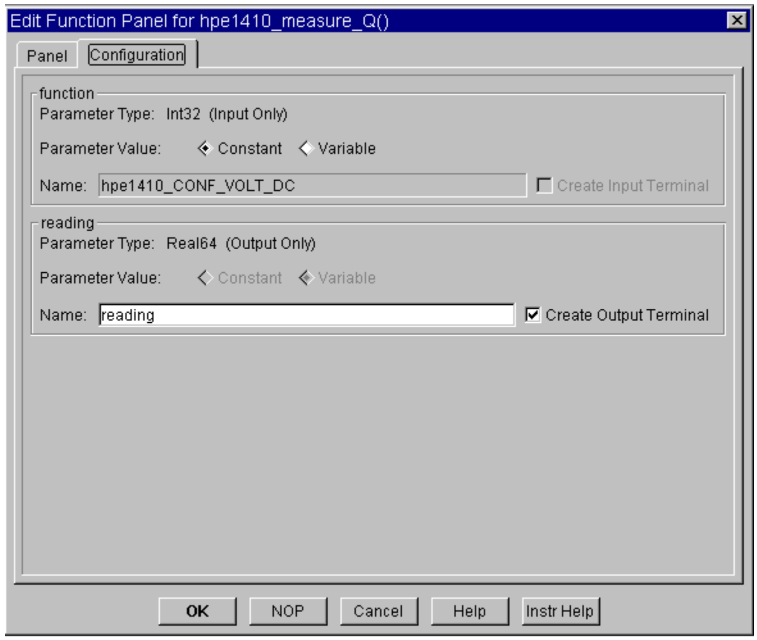


Figure 7-4. Parameter Tab of Edit Function Panel Dialog Box

Parameter values are shown in groups. The name of each group is the label name of the parameter as specified in the VXI*plug&play* function panel. In Figure 7-4, `function` and `reading` are labels. Each group may contain the following information.

- `Parameter Type` - This provides information about the parameter data type and whether the parameter is Input Only, Input/Output, or Output Only.

- `Parameter Value` - When `Constant` is selected, this parameter is passed as a constant value that is set on the `Panel` tab. When `Variable` is selected, this parameter is passed as a variable. The value of the parameter may be changed programmatically. Some fields are always variables, such as the output for a reading.

- `Name` - When the `Parameter Type` is set to `Variable`, this field is editable. By default, the name of the variable is set to its label name (or a similar name to make it a valid VEE variable name). You can change this to any valid variable name in VEE. If the variable is an input variable, you can also put an expression, function call, or global variable in this edit field.
- `Create Terminal` - When the `Parameter Type` is set to `Variable`, this field is editable. When the check box is checked and `Name` does not currently exist as a terminal name, pressing `OK` creates the terminal (with the name specified in `Name`) as an input, output, or input/output terminal, as indicated in the dialog box. To delete a terminal once it is created, you must use `Delete Terminal` from the object menu.

  If the `Name` is changed and `Create Terminal` is checked, a new terminal is added.

  If the `Name` is set to an invalid terminal name, `Create Terminal` is grayed out.
- Auto-Allocate Input - This appears on Input/Output parameters that have been set to Variable, not Constant. The next section provides more information.

Press the `NOP` button to save the latest settings shown in this dialog box and make this transaction a "no operation". This is the same as commenting out a line of code in a text-based computer program.

Press the `Help` button for help about the `To/From VXIplug&play` object. Press the `OK` button when you have finished editing.

Press the `Instr Help` button to get instrument-specific help written by the driver developer.

The Auto-Allocate Feature (Passing Arrays and Strings). Some VXI*plug&play* functions want to return data in an array or `Text` string. The VXI*plug&play* specification requires that the application (VEE) allocate the memory for the array or string since the VXI*plug&play* function cannot pass back allocated memory. VEE must allocate the memory, and the function can write to that memory.

The `Auto-Allocate` feature lets you easily tell VEE how much memory to allocate. VEE allocates the correct data type and shape, in the size required.

If a parameter to a function is a variable that requires an array or a `Text` string, the `Parameters` tab displays an additional field: `Auto-Allocate input`. For example, in the dialog box in Figure 7-5 `readings` can input an array. The `Parameters` tab shows `Auto-Allocate input` selected:

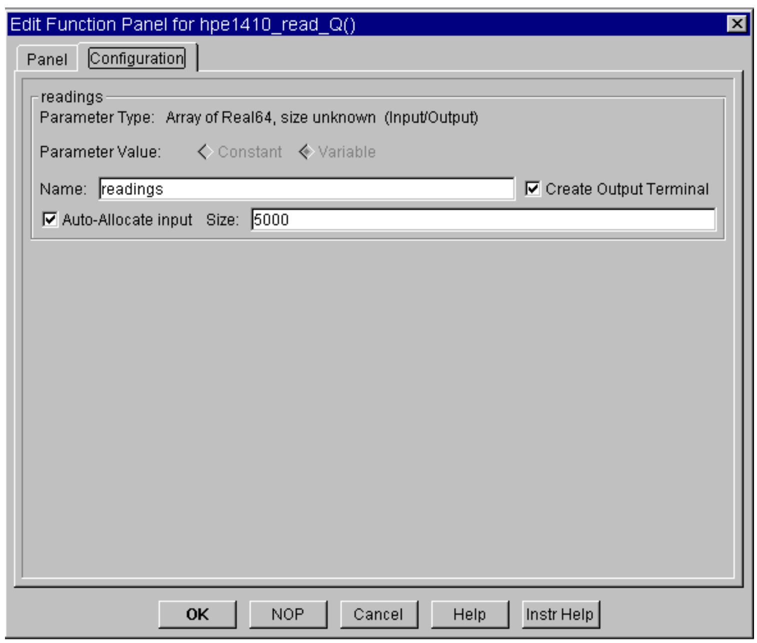


Figure 7-5. Selecting the Auto-Allocate Input Feature

When `Auto-Allocate input` is selected, the `Size` field becomes active. The default size is `5000`, but you can enter any appropriate size to allocate the input data. You must determine how large an array or string needs to be passed. An input terminal is not created for this parameter and VEE automatically allocates the memory for the parameter.

For an array, `Size` denotes the number of elements in the array. For a text string, `Size` denotes the number of characters (bytes). See `Instrument Help` or click the right mouse button on the `Panel` background or on the parameter for more information on the size of array or string the function requires.

Note If you use the `Auto-Allocate input` feature, a data input terminal is *not* created for the function. If the data input terminal already exists, you should delete it from the `To/From VXIplug&play` object.

If you do not select ("check") `Auto-Allocate input`, both input and output terminals are created for the function by default. You must create an object to allocate the correct type, shape, and amount of memory and connect it to the input terminal. See “Passing Parameters” on page 206 for information on how to manually allocate the memory needed for inputs.

Be sure to allocate enough memory for all the values the function wants to return. If insufficient memory is allocated, this action will overwrite memory and cause a General Protection Fault. Since the VXI*plug&play* DLL is linked directly into VEE, this situation can cause VEE to crash and exit.

Getting Help on a VXIplug&play Driver

From the object menu of the `To/From VXIplug&play` object, select `Instrument Help` to access the help file provided by the instrument manufacturer. This help topic contains information about using the VXI*plug&play* driver including the data types required for the parameters.

For help on each particular function, see “Getting Help on a VXIplug&play Function Panel” on page 199.

Running a VEE Program

The transactions in the `To/From VXIplug&play` object execute from top to bottom. This section explains what happens when `To/From VXIplug&play` objects execute.

Initializing and Closing Drivers

The first time you run a program after you load or create it, a delay occurs to initialize each instrument controlled with `To/From VXIplug&play` objects. This initialization sets the instrument to a known initial state. Each subsequent time you run the program, your program executes normally, without performing the initialize actions.

Each instrument controlled by the program must be initialized once in a VEE session. The VXI*plug&play* Resource Manager does an "instrument find" to verify the instrument is connected to the address and to set the instrument to a known state. This will take an indeterminate amount of time, possibly up to 10 seconds per instrument. This delay happens the first time the `To/From VXIplug&play` object for each instrument is executed.

Because the initialization is only performed once per VEE session, you should execute functions (such as clear or reset) that set an instrument to a known state every time the program runs. When you load another program or exit VEE, the VXI*plug&play* drivers are automatically closed.

Advanced Initialization Information

This section explains some of the details behind some of the VEE implementation of VXI*plug&play* initialization. Understanding these concepts is not required to successfully write a VEE program that uses VXI*plug&play* drivers.

Each VXI*plug&play* driver is required to have a *PREFIX*`_init()` and a *PREFIX*`_close()` function. These functions are called automatically by VEE.

The purpose of the `init()` function is to set your instrument to a known state and to get a "session handle". Each instrument specified by a VEE `Name`, when configured, will have a unique session handle assigned the first time it is executed in a program. That session handle is used through the program to uniquely identify that instrument.

All `To/From VXIplug&play` objects communicating with the same instrument (with the same VEE `Name`) are identified by the same session handle. The session handle is shown in the `vi` field in the lower left corner of `Panel` tab of the function panel. VEE automatically takes care of passing this session handle between the various `To/From VXIplug&play` objects.

Because the `init()` call is usually a lengthy operation, it is only called when necessary. When the first `To/From VXIplug&play` object is executed in a program, the appropriate `init()` function is called. When `init()` is called, it may also perform an `Identification Query` and/or a `Reset` depending on how you configured the driver.

The purpose of the `close()` function is to close the session handle (there are a limited number of them), take the instrument off-line, clear any data associated with the instrument, and perform instrument-specific actions, if needed. VEE calls the `close()` function at the following times:

- After `New`, `Open`, or `Exit` is selected.
- When all `To/From VXIplug&play` objects for a single VEE `Name` (such as `dvm`) are deleted.
- When the `Address` or `init()` parameter values are changed in the `VXIplug&play Instrument Properties` dialog box. In this case, `close()` is called so that `init()` will be called again with the new values.

Error and Caution Checking

After each transaction is executed, the function returns a status value to VEE, which automatically checks this value. If the value indicates that the function executed successfully, the next transaction executes.

Error Checking. If the status value returned indicates an error, VEE stops the program and reports the error. If you have an error output pin to trap the error, the error output pin propagates instead of stopping the program. Use the `errorInfo()` object to get the details of the error message.

VEE automatically calls the *PREFIX*`_error_message()` function to get as much error information from the VXI*plug&play* driver as the manufacturer includes. This information is output in the VEE error message or from `errorInfo()`.

Note

After an error occurs the instrument is left in an unknown state. Unless you call specific reset or clear functions at the beginning of your program, you will not know the state of your instruments the next time you start the program.

Caution Checking. If the status value returned is a caution, the program pauses and displays a caution dialog box. The caution dialog box contains information from the instrument manufacturer and lets you choose to continue running the program or stop.

Caution messages cannot be trapped programmatically. However, if you are aware of the common caution messages from the driver, you can handle them in the VEE program. For example, if you get a caution message that the instrument is not ready to let you read data, you can use a `Delay` object or put the `To/From VXIplug&play` object in a loop to retry reading.

If you handle a known caution condition in the VEE program, you may want to suppress the caution message dialog box. To do this, from the `To/From VXIplug&play` object's `Properties` dialog box select the check box for `Ignore Cautions Returned`.

Note

Generally, ignoring caution messages (by checking the `Ignore Cautions Returned` check box) is not necessary and, unless you are sure of how to handle the caution condition in your program, is discouraged.

Passing Parameters

According to the VXI*plug&play* specification, you must allocate memory and pass it to the driver before requesting data. Some VXI*plug&play* functions place the data read into an array. Most of these VXI*plug&play* functions also have a parameter that specifies the size of the array sent in and will error if the array is not big enough. In this case, you may allocate an array of any size and tell the function how big it is. The function will then write data into the array only to the size specified.

Caution

Other VXI*plug&play* functions assume the array passed in is big enough for the data read and write to it regardless of its size. This is especially common for `Text` strings. If insufficient memory is allocated, this action will overwrite memory and cause a General Protection Fault. Since the

VXI*plug&play* DLL is linked directly into VEE, this situation can cause VEE to crash and exit.

Note The most straightforward method to allocate memory for an array or string data input is to use the `Auto-Allocate` feature. See “Getting Help on a VXIplug&play Function Panel” on page 199. You still need to determine the size to allocate, but once you specify the size, the memory is allocated automatically.

Find out how much memory you need for your data by reading the driver's help file. Select `Instrument Help` from the `To/From VXIplug&play` object's object menu. This help file tells you how large the array must be.

If you do not use `Auto-Allocate`, you must create an object to allocate the memory and connect it to the data input terminal of the `To/From VXIplug&play` object:

- For an array input, use an `Alloc Array` object of the appropriate type, and set the size appropriately.
- For a string input, use a `Formula` object. Delete the data input terminal from the `Formula` object and enter an expression like `256*"a"`. This creates a string that is 256 characters long (plus a null byte) filled with `a`’s. Most VXI*plug&play* functions will not write more than 256 characters into a `Text` parameter. However, it is best to check the help on each function panel that requires a `Text` input to be sure.

An Example Program

Figure 7-6 shows a simple program that uses `To/From VXIplug&play` objects to communicate with the HP E1410A VXI Multimeter:

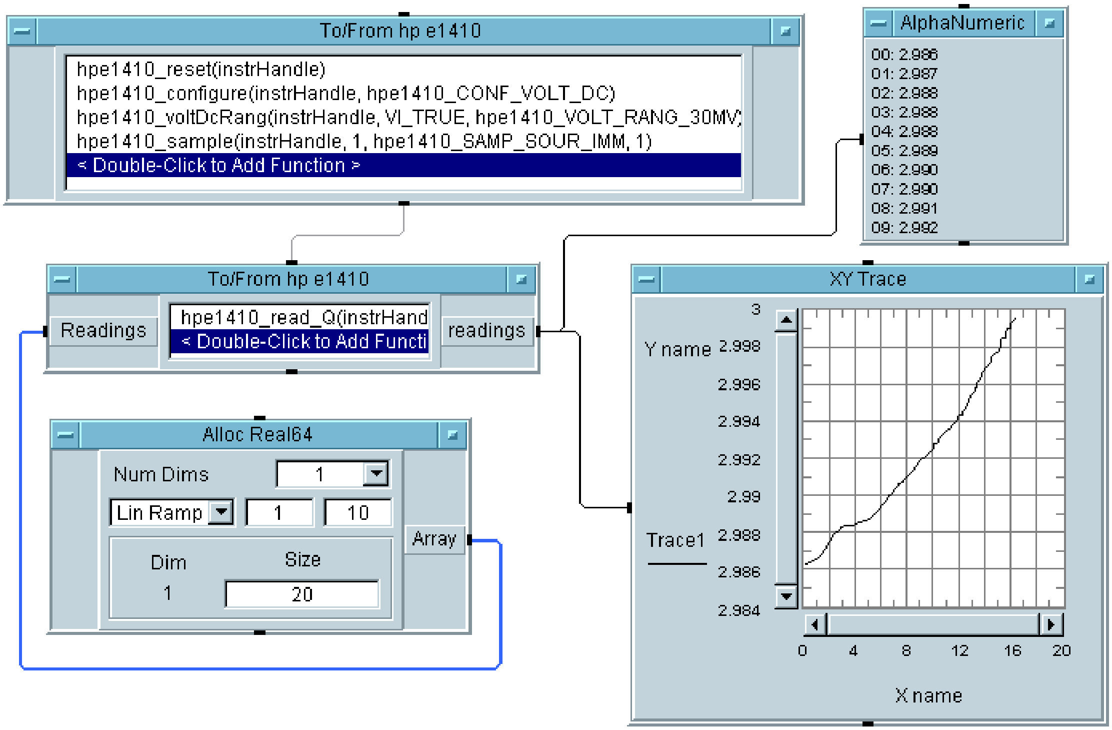


Figure 7-6. A Program Using To/From VXIplug&play Objects

Limitations to VXIplug&play

There are some limitations to using VXI*plug&play* drivers in VEE.

- Because the `Bus I/O Monitor` object only shows I/O to and from VEE itself, it does not show any I/O from VXI*plug&play* drivers. VXI*plug&play* drivers are C programs that are linked into VEE. We recommend that you use a hardware bus monitor, if needed.
- Some optional features that are not required by the VXI*plug&play* specification, such as callbacks, are not supported by VEE.
- All `I/O` ⇒ `Advanced I/O` objects (including `Interface Operations`, `Instrument Event` (`SPOLL`), and `Interface Event`) are not supported for VXI*plug&play*.
- VXI*plug&play* does not support the concept of `LIVE MODE/NOT LIVE MODE`. When you run a program, all instruments used in your program must be connected to your computer. However, you can open a program without the instruments used in the program being connected. Also, you can create a program without having the instruments connected. You can use `To/From VXIplug&play` objects and specify the function calls as long as the VXI*plug&play* driver is installed.
- You cannot use VXI*plug&play* drivers and any of the other VEE instrument control methods (`Direct I/O`, `Panel Driver`, or `Component Driver` objects) to communicate with the same instrument in the same program. However, you can use VXI*plug&play* drivers for one instrument and other instrument control methods for other instruments in the same program.

Note

The VXI*plug&play* specification is continually being updated and enhanced. New features may be voted into the specification by the VXI*plug&play* consortium between revisions of VEE. Because the VXI*plug&play* specification does not specify that revision information should be included in the driver library, VEE cannot check the driver for compatibility. Therefore, you need to check with the instrument manufacturer to make sure the driver conforms to the currently supported VXI*plug&play* specification.

Using VXIplug&play Functions from Call Objects

You may want to use VXI*plug&play* with a VEE `Call` object for the following reasons:

- Existing Program Compatibility.

  If you have existing programs using VXI*plug&play* that were created using VEE Version 3.1, you may want to continue to use them with minimal modifications. However, if you plan to maintain these programs over the long term, it would be better to rewrite them using the standard function panel access in the `To/From VXIplug&play` object as described in “Using the To/From VXIplug&play Object” on page 195.

- Access to Older Drivers.

  Some earlier versions of non-HP VXI*plug&play* drivers (1995 and earlier) were written to earlier versions of the VXI*plug&play* specification. You can still access these drivers through the `VEE Call` object.

Except for the reasons listed above, you should use VXI*plug&play* drivers with the methods described in “Using the To/From VXIplug&play Object” on page 195.

Using a Dynamic Link Library in VEE

This section will show you the steps in loading a VXI*plug&play* driver into VEE once the required files are installed.

To use a VXI*plug&play* driver in a VEE program, do the following:

1. Import the library.
2. Run the routines which use the library.
3. Delete the library when the program is done.

The three VEE objects associated with these steps are `Import Library`, `Call`, and `Delete Library`.

Importing the Library

Before you can use a `Call` object (or `Formula` object) to execute the driver, you must import the function into the VEE environment via the `Import Library` object.

In the `Import Library` object, under `Library Type`, select `Compiled Function`. Enter the path and name of *PREFIX*`.h`using the `Definition File` button. See Table 2-2, “Location of WIN95 and WINNT Framework Driver Files,” on page 39 for the location of these files.

Then, select the path and name of *PREFIX*`_32.DLL` using the `File Name` button. The `Library Name` button assigns a logical name to a set of functions. It is recommended that the name be *PREFIX*, where *PREFIX* refers to the name of the instrument, such as `HP E1410`.

Before using a driver with the `Call` object, you must configure the `Call` object. The easiest way to do this is to select `Load Lib` from the `Import Library` object menu to load the driver file into the VEE environment. Bring up a `Call` object from the `Device` menu. Then, select `Select Function` on the `Call` object menu. VEE will bring up a dialog box with a list of all the functions listed in the header file that are exported from driver file.

Calling a VXIplug&play Driver from VEE

Use a `Call` object to make the calls to a VXI*plug&play* driver.

Sequence of Calls. The sequence of calls for a VXI*plug&play* driver is very important. The sequence is:

1. Call the initialize function. (This function returns a session handle.)
2. Perform calls to the driver using the handle returned by the initialization function.
3. Call the close function.

Initialize Function. The initialize function *PREFIX*_`init` has three input pins and two output pins.

The three input parameters are:

- *Instrument Address*

  See "Configuring for a VXIplug&play Driver" on page 69 for information about VXI*plug&play* addressing.

- *Identification Verification Flag*

  If the verification flag is `1`, the initialize function checks the identity of the instrument. This is to be done by checking the manufacturer ID and model number, using the `"*IDN?"` query, or other means specified by the instrument manufacturer. Set the flag to `0` if the check should not be done.

- *Reset Flag*

  The reset flag should be `1` if the initialize function is to place the instrument in a pre-defined state. Set the flag to `0` if the reset should not be done.

The two output parameters are:

- *Return Value*

  VXI*plug&play* defines the return value from a VXI*plug&play* driver to be the status of the operation performed. The integer returned can be translated into a meaningful message by calling *PREFIX*_`error_query` from a separate `Call` object. If the return value is `0`, the `init()` call was successful.

- *Handle for VXIplug&play Functions*

  If the return value from the initialize function is `0`, the output parameter contains an instrument handle. An instrument handle is simply a number which associates a function call with this initialization. Most VXI*plug&play* functions require this handle as an input parameter.

Each initialization returns a unique handle in the output parameter `vi`. The parameter may be called by a different name, such as `session handle`, but it is always the last parameter returned from the `init()` function. When the `close()` function is called, the handle is returned to the system.

Calling VXIplug&play Functions. Other functions can be called using the `Call` object. For each function called, the handle from the *PREFIX*`_init` function must be provided to the `instrID` input pin of the `Call` object.

Using Other Common VXIplug&play Functions. Besides the *PREFIX*`_init` and *PREFIX*`_close` functions, VXI*plug&play* drivers may implement other common driver functions. These functions are *PREFIX*`_reset`, *PREFIX*`_self_test`, *PREFIX*`_revision_query`, *PREFIX*`_error_query`, and *PREFIX*`_error_message`.

Using Arrays As Parameters. The VXI*plug&play* specification states that the caller must allocate space for an array or text parameter. This means that VEE must allocate the array before passing it as a parameter to the VXI*plug&play* function, as shown in Figure 7-8.

Using the Close Function. The close function *PREFIX*`_close` has one input parameter and no output parameters. The input parameter is the handle returned from *PREFIX*`_init`. Executing *PREFIX*`_close` takes the instrument off-line and clears any data associated with the instrument handle. There may also be some other driver-specific actions related to closing the instrument. The handle cannot be used again by instrument functions. The *PREFIX*`_init` routine must be called again to obtain a new handle.

Deleting the Library

After you finish using the VXI*plug&play* driver, the `Delete Library` object needs to be invoked for each driver loaded. After the library is unloaded, the library must be loaded again using the `Import Library` object before any functions using that library can be called.

A Simple Example

Figure 7-7 is an example program using a VXI*plug&play* driver in VEE. This program imports the library, initializes the device, closes the device, and deletes the library. (Each program thread is started independently with a `Start` button.)

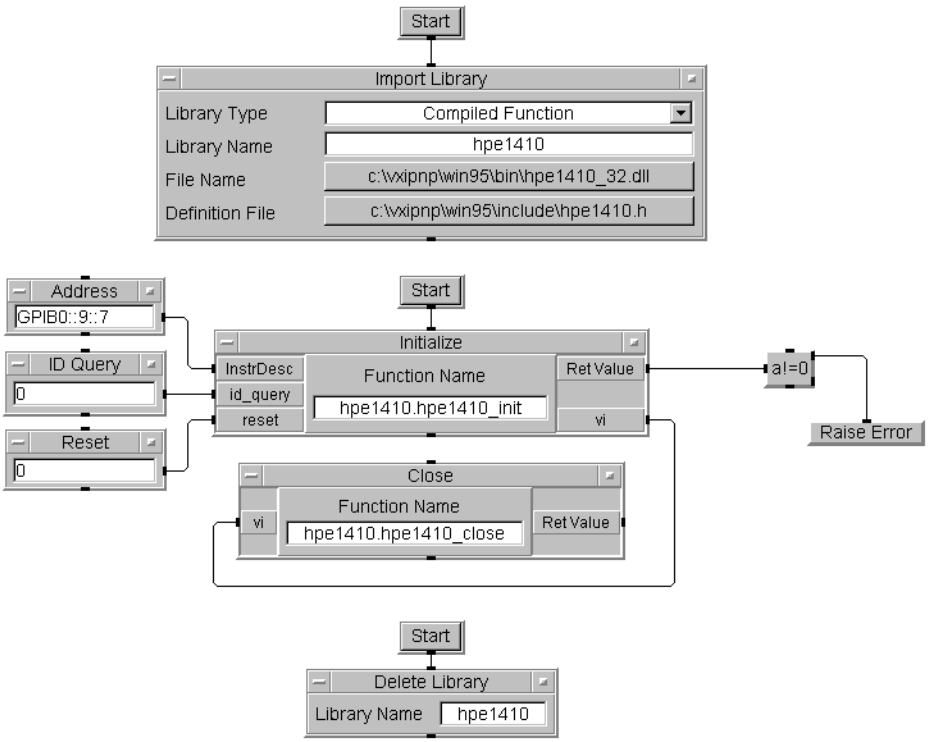


Figure 7-7. Simple Example: Using VXI*plug&play* Drivers

A More Complete Example

Figure 7-8 shows a more complete example program that uses a VXI*plug&play* driver and allocates an array to be used as an output parameter.

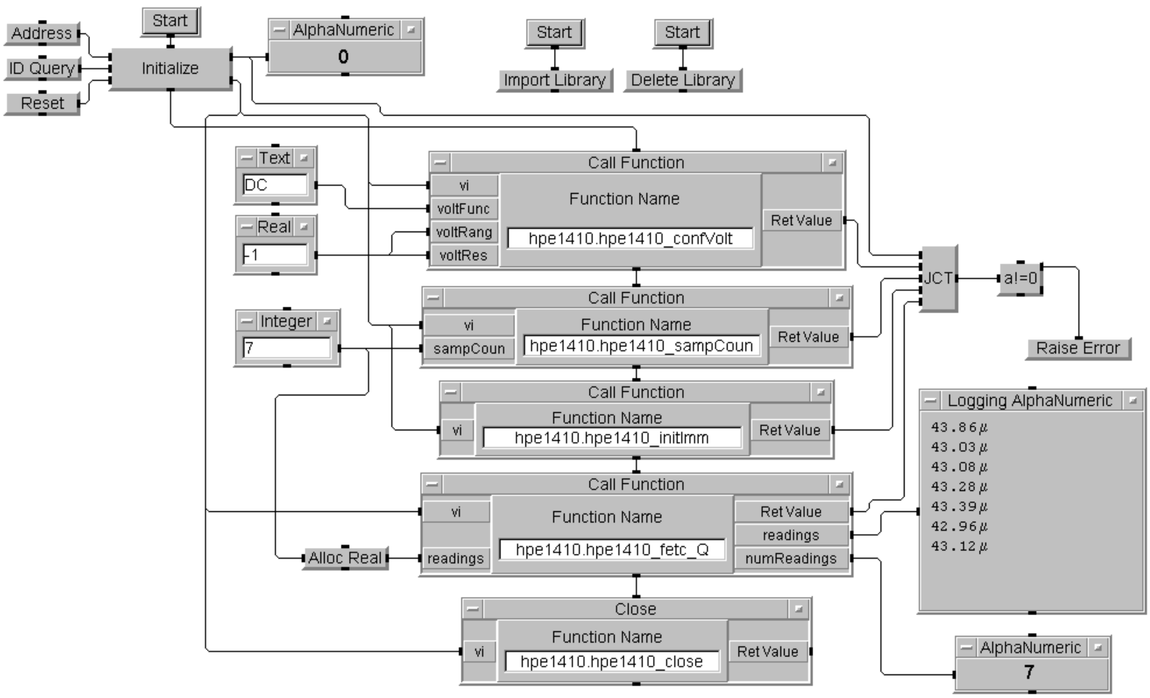


Figure 7-8. More Complete Example: Using VXI*plug&play* Drivers

Some Helpful Hints

Keeping Track of Handles. The handle returned by *PREFIX*`_init` must be used by successive driver functions. There are two ways to accomplish this:

- Connecting Pins

  The value of a handle can be passed by connecting the *PREFIX*`_init` routine data output pin to the `vi` data input pins on each function.

- Keeping Track of Handles Globally

  The handle can be kept as a global variable. The handle from *PREFIX*`_init` routine is connected to a `Set Global` object. Each function that uses this handle, takes it from a `Get Global` object.

Control Flow. The driver needs to perform actions in a certain sequence (initialization, calling functions, and closing). The VEE program must be written to ensure that the handle is valid for all functions that require its usage.

8

Data Propagation

Data Propagation

You can create VEE programs by applying textual programming language techniques, visually recreating a written program. However, you may find it more efficient to produce the program with VEE objects, thinking in terms of data propagation between the objects. This chapter explains data propagation techniques for VEE, including:

- Understanding Propagation
- Propagation in UserObjects
- Controlling Program Flow
- Handling Propagation Problems

Understanding Propagation

Propagation is the general flow of execution through a VEE program. The propagation guidelines define the order in which VEE objects operate. In general, propagation is determined by **data flow -** the flow of data from object to object within an VEE program.

How Objects Operate

A VEE object operates by accepting the data on its input pins, processing that data, and returning the resulting data on its output pins. A VEE object will not operate until all of its data input pins are activated with data on them. (There is one exception. The `JCT` object will operate when *one* of its data input pins is activated with data.)

In the program in Figure 8-1, the `a+b` object will not operate until there is data on both of its data input pins. Both of the `Real` constant objects must operate first (in no particular order).

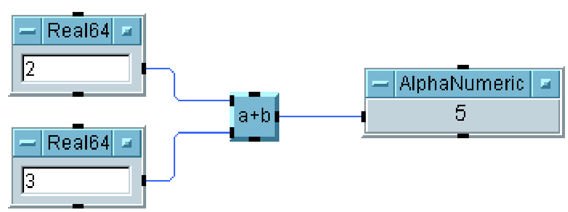


Figure 8-1. The a+b Object Propagates When Both Inputs Have Data

When the `a+b` object operates, it adds the data and activates its output pin with the resulting data. The `AlphaNumeric` object does not operate until its data input pin has data so it operates last, displaying the result.

As you can see, data flow has determined the order of operation of the objects in the above program. That is, data flow determines the propagation order.

The sequence pins also can be used as a hold-off to control when an object operates. This is useful when you want to prevent the object from operating until valid data is available. In the program in Figure 8-2, a `Confirm OK` object has been added to the previous example.

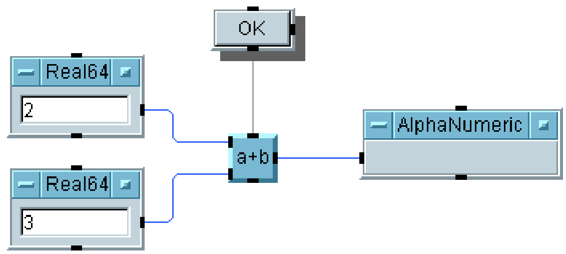


Figure 8-2. Controlling Propagation Using a Sequence Input Pin

The sequence output pin of the `Confirm OK` object is connected to the sequence input pin of the `a+b` object. Sequence input pins need not be connected. If a sequence input pin is not connected, it is ignored by the object. However, if a sequence input pin *is* connected, the object will not operate until it has been activated. In the above example the `a+b` object will not operate until you press (click) the `OK` button. Then the data input pins accept the data and the object executes.

The `XEQ` pin has the opposite effect on object operation. An object propagates immediately when the `XEQ` pin is activated using any data present on its data input pins. This is important to consider when using both the `XEQ` pin and sequence input pin on an object. The `XEQ` pin must be connected and the object will not propagate until the `XEQ` pin is activated. Figure 8-3 is an example.

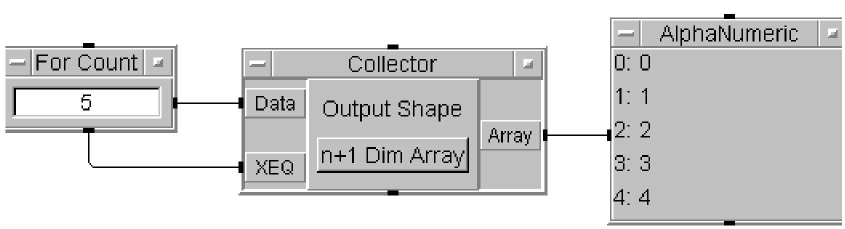


Figure 8-3. Controlling Propagation Using the XEQ Pin

The For Count object repeats five times, outputting data to the Data input terminal on the Collector. The Collector collects the five values into an array, which it propagates when the XEQ terminal is activated by the sequence output pin of the For Count object.

Note

You can use Properties from the object menu to turn on Show Terminals. With Show Terminals turned on the data input and output pins become “terminals”, showing their names.

Basic Propagation Order

Based on the propagation rules, the objects in a VEE program executing in VEE 4 mode and higher operate in the following basic order when you press Run:

1. Objects that have no data input pins and no sequence input pins connected operate first.

2. Other objects operate in the order determined by data flow. In other words, objects with data input pins operate only when data is present on all data inputs, except for JCT, XEQ and sequence pins as noted in “How Objects Operate” on page 219.

3. The order of propagation can be modified by connecting sequence pins.

The next section, Pins and Propagation, describes how various pins work.

Pins and Propagation

This topic summarizes all types of pins and their effect on propagation. In an object’s open view you can view pin labels and get terminal information when Show Terminals in the Properties dialog box is on, as Figure 8-4 shows. Objects may not contain all of the pins described here.

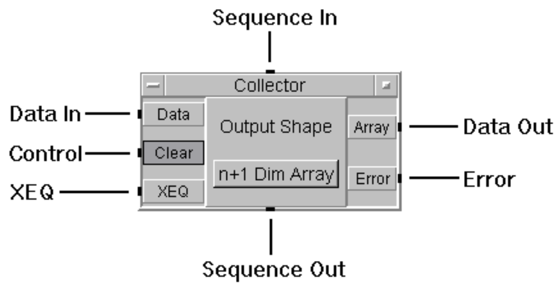


Figure 8-4. Pins Available on Objects

- Data pins input or output a data container.
  - An object will not operate until all of its data input pins are activated. (Except the `JCT` object, which operates when any data input pin is activated.)
  - After an object operates, its data output pins propagate (if no error conditions have occurred).

  Some objects may not propagate all of their data output pins, which can cause confusing behavior. Such objects include `If/Then/Else`, `DeMultiplexer`, `Comparator`, and all `Data` ⇒ `Dialog Box` objects. Please see “Handling Propagation Problems” on page 244 for more information.

- Control pins (optional) are inputs that affect the state of the object but have no effect on propagation. Common control pins include `Clear`, `Reset` and `Default Value`. Outputs from other objects to control pins are connected with dashed lines to indicate that propagation is not affected.
  Since control pins do not affect propagation, there are some conditions where your program may not run correctly. See “Handling Propagation Problems” on page 244 for more information about control pins.

- Sequence pins are used only to specify the order of execution. They are useful to resolve ambiguity in a program’s propagation. Sequence pins generally are not necessary and can be overused so you should not use them as a substitute for clear data flow.

- ❑ An object operates only after all data input pins and sequence input pins (if connected) are activated.

  A sequence input pin is activated by the presence of a data container, but the data in the container is ignored.

- ❑ A sequence output pin propagates after all the data output pins have activated and data flow has propagated as far as possible.

  A sequence output pin propagates an empty (nil) container when it activates.

- Error pin (optional). You can add an `Error` pin to trap an error condition the object generates. The `Error` pin propagates the appropriate error number if an error condition occurs.

  If an error occurs, the `Error` pin and the sequence output pin (if connected) propagate. *Data output pins stop propagating immediately when an error occurs*. You should be aware of this potentially confusing behavior since some data output pins may propagate before the error condition occurs.

- `XEQ` pin is a pin that forces an object to operate immediately (even if a data input pin has not yet been activated). Only the `Collector` and `Sample & Hold` objects use an `XEQ` pin to force the object to execute immediately and propagate its data.

  The `XEQ` pin is activated by the presence of a data container, but the data in the container is ignored.

Note

Do not leave any data input pins or the `XEQ` pin unconnected or an error will occur when you run your program.

You may leave data output pins, control pins and `Error` pins unconnected. Sequence pins *should* be left unconnected except when needed to resolve ambiguous program propagation.

See “Handling Propagation Problems” on page 244 for more information.

Propagation of Threads and Subthreads

A very simple VEE program usually contains only one thread. Programs that are more complicated contain additional threads and subthreads that affect the program’s propagation.

- Threads – Objects connected through data and sequence lines, which are solid, form a thread. Objects connected only through control lines, which are dashed, are not considered to be in a thread. A program can contain several threads. For example, the program in Figure 8-5 contains two parallel threads. The threads are independent because they are not connected by data or sequence lines.

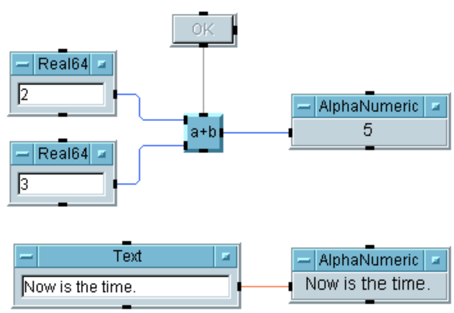


Figure 8-5. A Program with Two Parallel Threads

- Subthreads – A branch of a thread is called a subthread. When two subthreads begin at the same data output pin of the same object and there are no sequence or data lines between them, they are parallel subthreads. The program in Figure 8-6 shows two parallel subthreads branching from the data output pin of the `Real64` constant object:

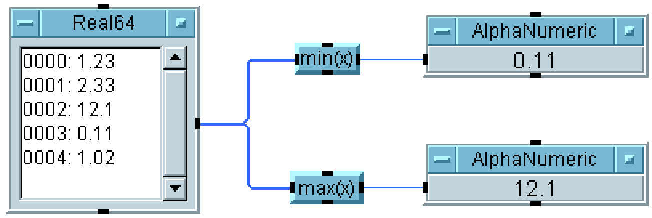


Figure 8-6. A Program with Two Parallel Subthreads

Parallel threads and subthreads operate in random order relative to each other. One or more objects (or all objects) in a thread will operate, then one or more in another thread operate. However, there are two exceptions to this:

- If a thread contains an `Interface Event` or `Instrument Event` object, it takes over execution when an event is trapped. For example, if `Interface Event` detects a GPIB `SRQ` message, the thread will continue to completion before any other thread can continue. Other threads are held off to allow the event to be serviced. For further information, see `Interface Event` and `Instrument Event` in *VEE Online Help*.

- If a thread has a `Start` object and if you start the thread by pressing the `Start` button, that thread will run to completion before you can start any other threads. The `Start` object is not recommended for `VEE 4` mode and higher.

Propagation Summary

The following is a summary of the propagation rules VEE uses when a program executes in `VEE 4` Execution Mode or higher:

- Data flows through objects from left-to-right — sequence flows from top-to-bottom.

- All data and `XEQ` input pins must be connected.

- Objects with no data input pins or sequence input pin connections operate first.

- All data input pins must be activated before an object operates (except for the `JCT` object).
- If the sequence input pin is connected it must be activated before an object can operate.
- Objects operate only once unless connected to a repeat object (for example, `For Count`) or unless forced to operate by an `XEQ` pin.
- Control pins execute immediately and do not cause the object to operate or propagate. See "Capturing Control Pin Errors" on page 245.
- When an error is generated from an object with an `Error` pin, the `Error` pin propagates instead of the data output pins. However, the sequence output pin *is* activated. (If there is no `Error` pin, an error message is displayed.)
- Parallel subthreads may operate in any order.
- Multiple threads may operate in any order.

Propagation in UserObjects

A `UserObject` provides the means for you to encapsulate a group of objects that perform a particular task into a single, custom object. This encapsulation allows you to:

- Use modular design techniques in building your VEE program. This allows you to solve a complex problem through an organized approach. `UserObjects` allow you to use top-down design techniques to create a more flexible and maintainable program.
- Build user-defined objects that you can save in a library for later re-use. Once a `UserObject` is created and saved, you can `Merge` it in other programs.

UserObject Features

When you add a UserObject to the Main Window, it appears in an icon view and remains that way in your program. When you double-click the icon, the UserObject's edit window pops up presenting the work area where you can build a specific program segment by adding objects and connecting them. The terminal areas accommodate data and control terminals so the UserObject can communicate with the rest of your program. Figure 8-7 shows the UserObject named `UserObject1` in its icon view and edit window.

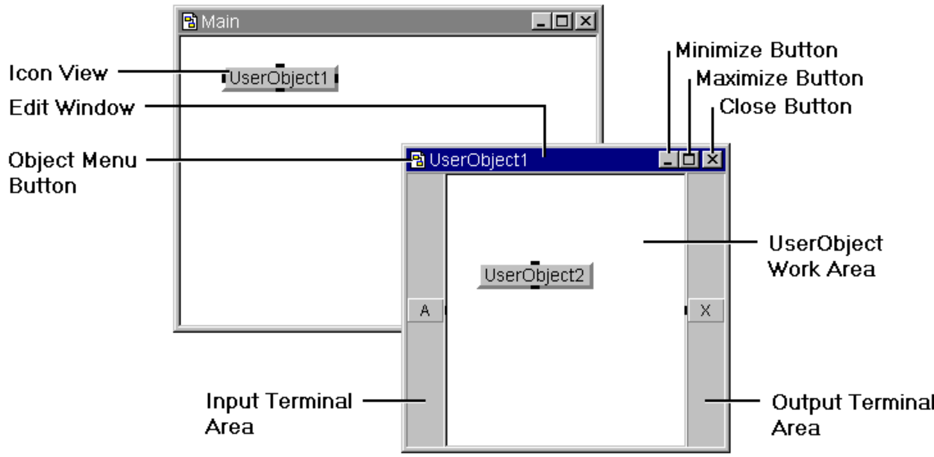


Figure 8-7. UserObject Features

Contexts and UserObjects

The Main Window and UserObjects represent separate **contexts** within a VEE program, just as subprograms represent separate contexts within a C or BASIC program. As shown in Figure 8-7, you can nest UserObjects in a VEE program, which results in additional contexts. In Figure 8-7, there are three contexts. More objects can be added to each context.

1. The Main Window is one context that contains `UserObject1`.

2. `UserObject1` is a context and contains `UserObject2`.

3. `UserObject2` is a context that can contain other objects.

Propagation and UserObjects

Propagation in a program containing UserObjects is affected by the fact that a UserObject is a separate context. The UserObject propagation rules are as follows:

Note The propagation rules for UserObjects also apply to UserFunctions. For detailed information about UserFunctions see Chapter 12, “User-Defined Functions/Libraries”.

- All data input terminals (and the sequence input terminal if connected) of the `UserObject` must be activated before any objects within the `UserObject` operate.

- When the data input terminals (and the sequence input terminal if connected) of the `UserObject` have been activated the `UserObject` operates. The objects within the `UserObject` operate following the rules of propagation.

- UserObjects in programs written before VEE Version 4.0 may contain an optional `XEQ` terminal. If it is activated, the `UserObject` immediately begins operation of the objects within it, using whatever “old” data may be on the inactivated input terminals of the `UserObject`.

  In most cases, you need not use the `XEQ` terminal for a `UserObject`. It is not available in VEE 4.0 and later versions and existing programs with `XEQ` pins on `UserObjects` will not compile if run in the `VEE 4` or higher Execution Modes.

- The `UserObject` data output terminals do not propagate until all objects within the `UserObject` finish operating (unless the `UserObject` is exited prematurely by an error or an `Exit UserObject`). Only those output terminals activated from inside the `UserObject` pass data to objects outside the `UserObject`. When activated, each data output terminal propagates only one data container.

- In programs written before VEE Version 4.0 (running in the VEE 3 Execution Mode) the objects within the `UserObject` time-share in operation with external objects on different subthreads. This is time-slicing. The `UserObject` does *not* block the operation of objects outside the `UserObject`. In programs running in the `VEE 4` or higher Execution Modes, the `UserObject` will time-slice only when invoked from separate threads.

For a review of the basic propagation rules see “Propagation Summary” on page 225.

Note

If there is a `Start` object in a `UserObject`, pressing `Start` runs only the objects connected to the same thread as the `Start` object. No data will be read from the input terminals of the `UserObject`, nor will its output terminals propagate. Therefore, no propagation outside the `UserObject` takes place.

Data Output from a UserObject

When the objects within the `UserObject` finish propagating, each data output terminal of a `UserObject` propagates only one **data container** (the last received by the terminal) to the context outside the `UserObject`. This can lead to unexpected results in your program if you neglect to account for it. The example in Figure 8-8 illustrates this situation:

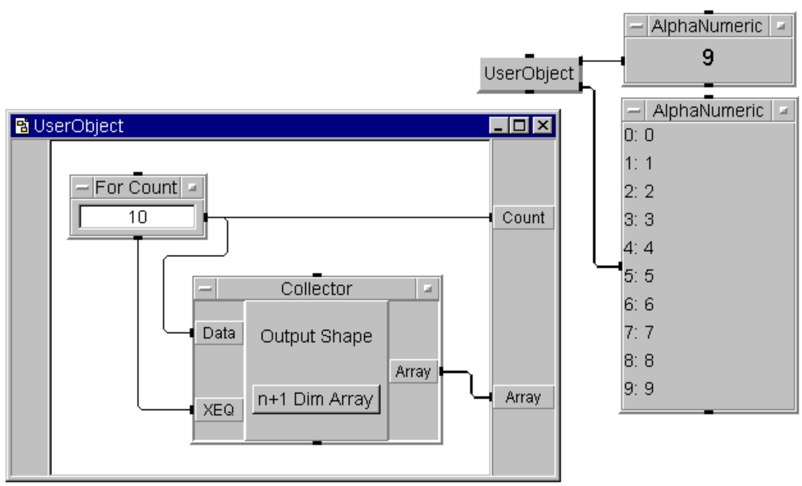


Figure 8-8. Data Propagation from a UserObject

Although the `For Count` object sends 10 data containers (the numbers 0 through 9) to the `Count` output terminal, only one data container (the last number) propagates from the `UserObject`. However, you can use a `Collector` object to collect the data from the `For Count` object into an array. The `Array` output terminal also propagates only one data container, but that container is a one-dimensional array of 10 values (0 through 9).

Controlling Program Flow

Though propagation rules in VEE are logical, it is not always obvious how a program will propagate. The examples in this section will help you understand and apply propagation concepts when you write your own programs. First, here are some rules of VEE programming style:

- Build a program using program flow that is clear and propagates in a hierarchical fashion. If you can visualize the flow easily, you normally will not have problems.
- If the execution order between objects is important but ambiguous, connect sequence input and sequence output pins. Though you should not need to use them often, there are cases when they are necessary to ensure the execution order required for your program.
- Avoid using feedback loops for iterations. Such constructs cause unpredictable results. Loops are intended for passing back containers with data to the start of a thread. If you must use feedback, `JCT` (Junction) objects are required in feedback loops.
- Avoid parallel threads fed by a looping object. It is difficult to tell which thread will be executed.
- Avoid using `Gate` and `Sample & Hold` objects. These objects are mainly used as patches for poor knowledge of propagation rules. Good programming style helps avoid the need for these objects.

Basic Program Control

It is important to understand how basic combinations of objects work together to control program flow. The program in Figure 8-9 shows how to generate a simple count useful for a loop, a common program control. The `For Count` object counts from 0 through 9 when the program runs.

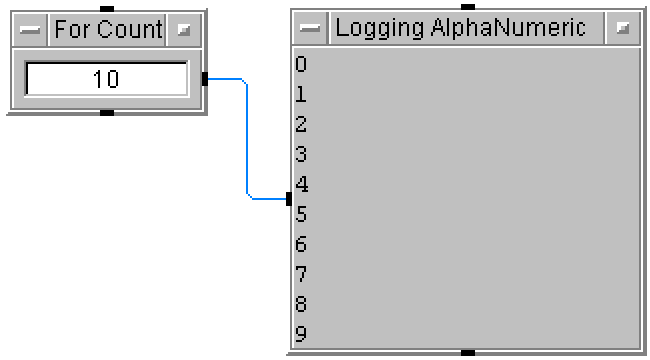


Figure 8-9. A Simple Loop Counter

You can nest `For Count` objects to create nested loops. In the program in Figure 8-10, the inner loop's `For Count` counts from 0 through 9 for each count sent to its sequence input pin by the outer loop's `For Count`. The outer `For Count` does not send its next output count until the inner `For Count` finishes its entire loop.

When the outer `For Count` sends its last count, it outputs a pulse from its sequence output pin, activating the `Beep` object. This is an important feature of such looping objects. They do not generate a sequence-out pulse until after the threads they are driving have executed.

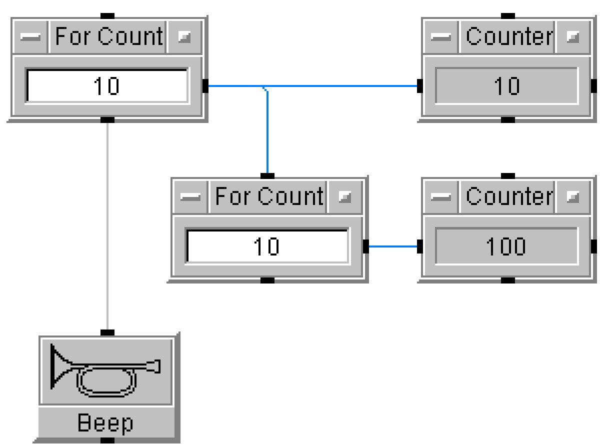


Figure 8-10. A Simple Nested Loop Counter

Continuous Loops

To generate a continuous loop, you can use an `Until Break` object as shown in the program in Figure 8-11. The `Delay` object controls the program to update once per second. A better approach is to replace the `Until Break` with an `On Cycle`, which can generate a container with any delay setting to drive the `now()` object. You can set the `AlphaNumeric` display format in its `Properties` dialog box on the `Number` tab.

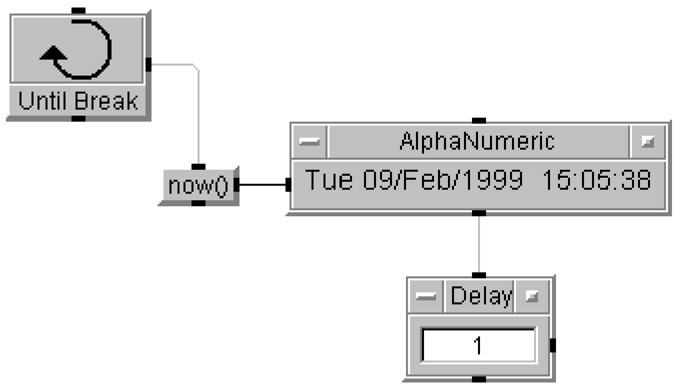


Figure 8-11. A Simple Continuous Loop

A continuous loop is useful to repeat a program's action until a certain condition is met. To end this loop at any time, add the `OK`, `Break` and `Next` objects as shown in the program in Figure 8-12.

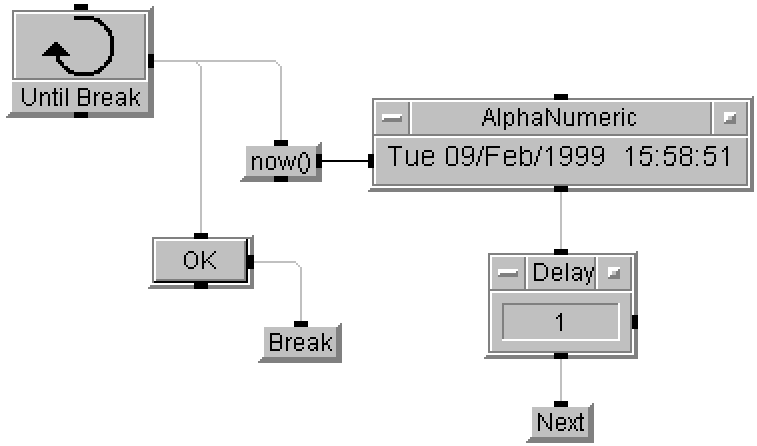


Figure 8-12. Stopping a Continuous Loop

This example illustrates another feature of control constructs in two parallel threads. The program is intended to update the time continuously, until you press the `OK` to force the `Break`. Without the `Next` object, `Until Break` would generate a container, then wait until everything downstream from it executes. The time would update and the program would wait until `OK` is pressed. The `Next` object forces `Until Break` to output containers continuously until `OK` is pressed.

In this case, a `Stop` object could be used in the place of the `Break` object without making any difference.

To provide more direct control over the continuous loop, you can use a `Toggle` object. The program in Figure 8-13 shows how to use a `Toggle` (in its `Button` format) to break a loop.

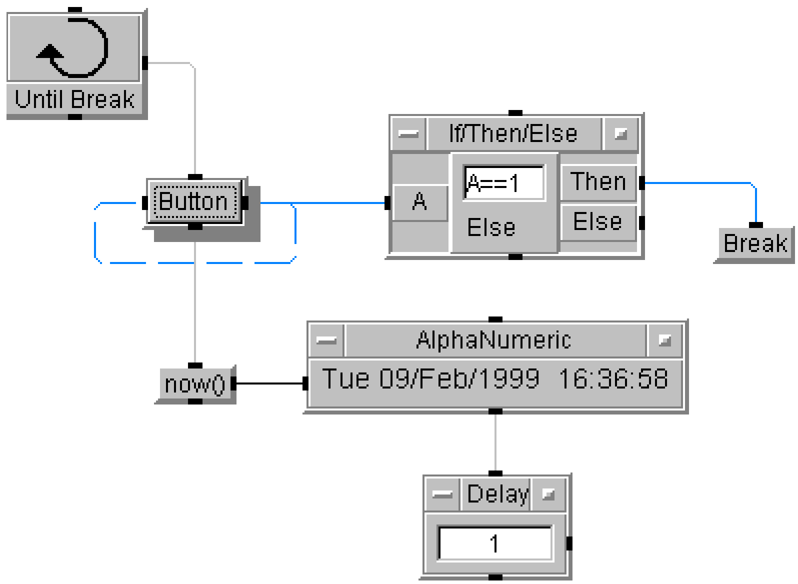


Figure 8-13. Using If/Then/Else to Stop a Continuous Loop

The `Toggle` output is connected to its `Reset` control input. The default initial value is 0, which is output to the `If/Then/Else` during each loop. While the `Toggle` value remains 0, program flow continues to the `Now` object. Clicking the `Toggle`'s `Button` toggles the value to 1, which satisfies the expression's condition in the `If/Then/Else` and activates the `Break` object.

Making Programs Interactive

Given the previous techniques for loops, the program in Figure 8-14 shows how to create a general architecture for interactive programs. Consider a simple program where the user can select one of two actions or exit the program by clicking the appropriate `Toggle` buttons:

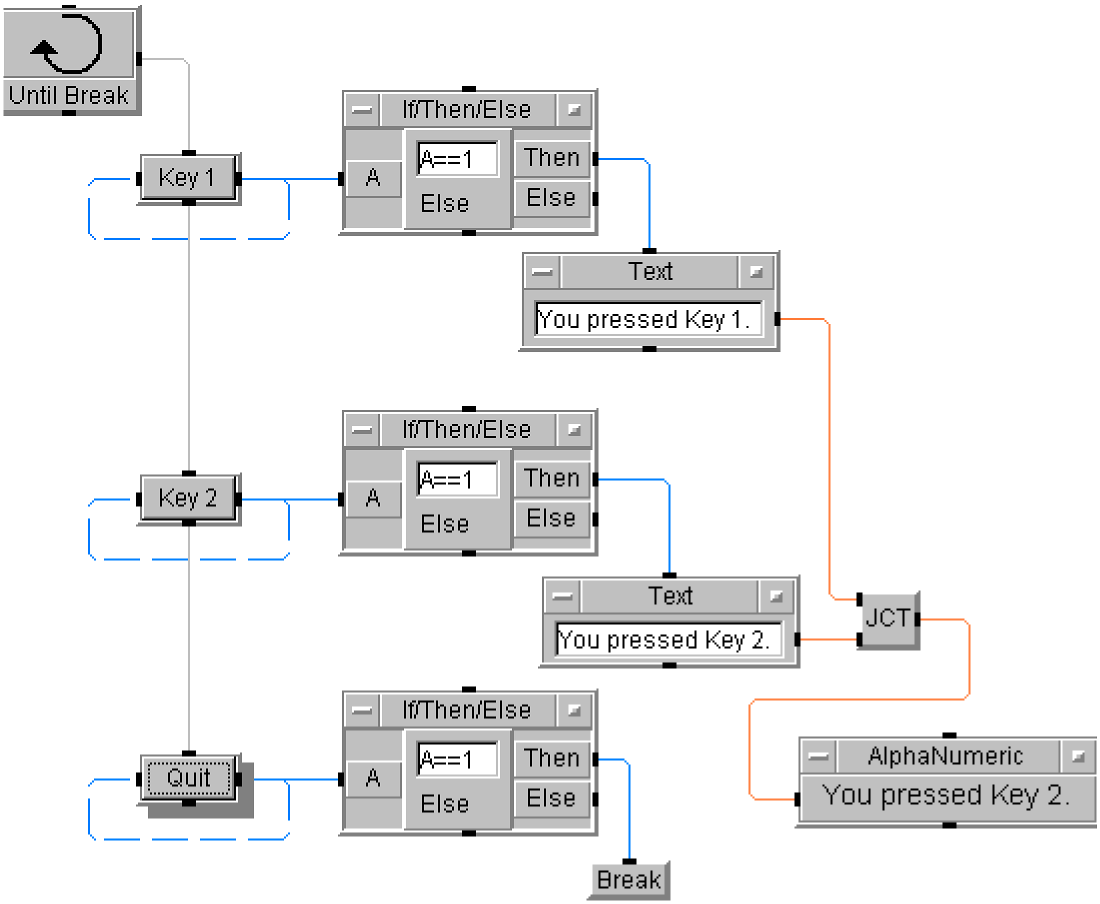


Figure 8-14. Using the Until Break Loop to Select a Program's Subthread

The concept is that each separate action has its own parallel thread. The loop continuously checks each `Toggle` object's output (initial value is 0) in each `If/Then/Else` expression. When a button is pressed, the `Toggle`'s output changes to 1, which sends the corresponding `Text` output to the `AlphaNumeric` display or ends the program. You can add as many parallel threads as you like to perform I/O and computation as needed.

The implication of this architecture is that the executing thread must finish before another thread can execute. If the executing thread takes a long time to finish, you will have to wait until this thread is finished before another thread can execute.

Advanced Program Control

Your programs can perform more complicated control tasks if you expand the previous techniques.

Example: Initiating Program Tasks

As an example, the program in Figure 8-15 lets the user select one of several tasks or lets a service request (SRQ) from an instrument initiate a task.

When the program runs, you click the `Task 1` or `Task 2` buttons to get the appropriate display output. If you press the `Clear Status` on the Panel Driver's `Status Panel`, you get the `SRQ!` message output, then return to perform another task. The program stops when you click the `Quit` button.

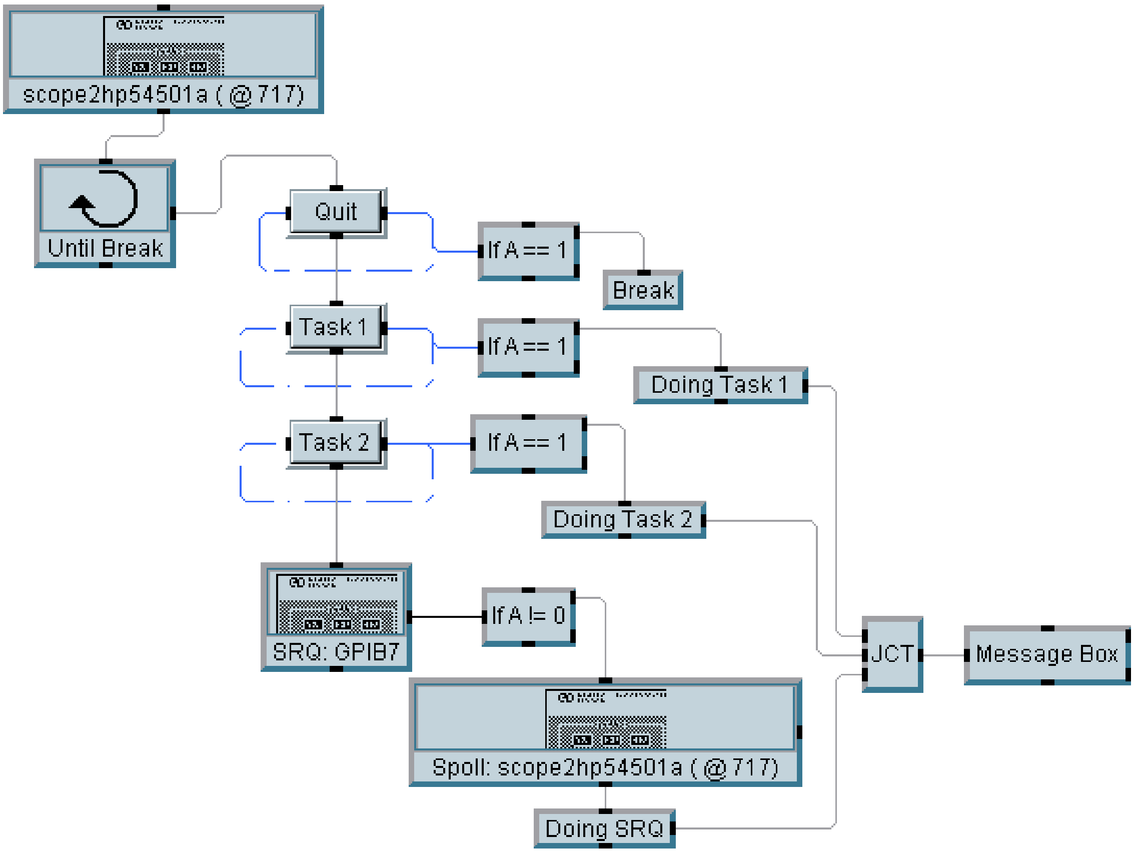


Figure 8-15. Using the Until Break Loop to Detect an Instrument's Service Request

The `Until Break` object drives the four parallel threads within the program that are controlled by three `Toggle` buttons (`Task 1`, `Task 2`, `Quit`) and the `Interface Event` object (`SRQ:GPIB7`).

The two threads defined by the `Task 1` and `Task 2` buttons display the text `Doing Task 1` and `Doing Task 2` in the `Message` dialog box. The thread defined by the `Quit` button stops the program and clears the display.

The interesting thread involves the SRQ. The HP 54501A Panel Driver, connected to the `Until Break` object, is set with its `SPoll Enable` (on the Status Panel) set to request service. A service request is sent when the

`Clear Status` button is pressed on the Status Panel. With the mask set, the thread that handles the SRQ uses the `Interface Event` object to wait for the SRQ by using the settings in Figure 8-16.

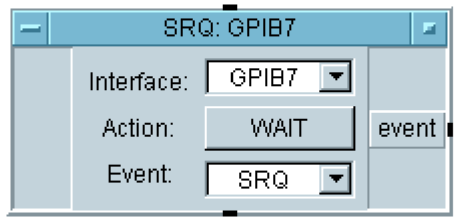


Figure 8-16. SRQ Settings

When an SRQ occurs, the `Interface Event` object pings the `Instrument Event` object to do a serial poll, which clears the SRQ on the scope, as Figure 8-17 shows.

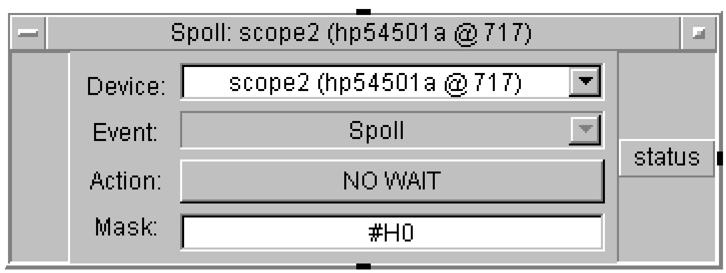


Figure 8-17. Clearing SRQ

Since `NO WAIT` is set, this object does the serial poll and then pings the `Text` object. The `SRQ!` message is sent to the `Message` dialog box announcing that an SRQ has occurred. The `Message` dialog is configured to wait for the operator to respond. The mask value is irrelevant.

Calling Functions

The program in Figure 8-18 is similar. You can call one of three user-defined functions – A, B, or C – to initiate an action, then have the program continue to execute the selected function.

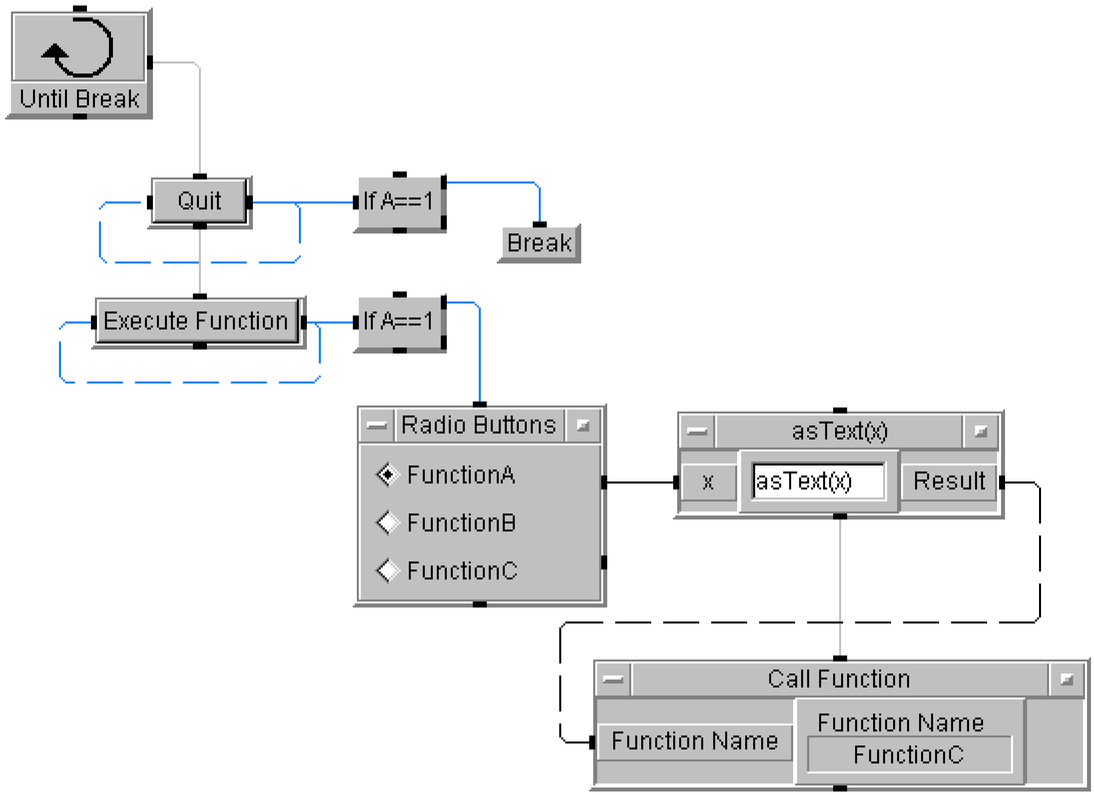


Figure 8-18. Using the Until Break Loop to Call a UserFunction

The `Radio Buttons` object lists the available functions. The `Toggle` buttons let you execute the selected function or quit the program. If you choose to execute a function, the function name is output to the `asText Formula` object. `AsText` is a built-in function that converts inputs to the Text data type.

The text function name is output to the `Call Function` object's control input, `Function Name`, so the selected function is called. When the program runs, the operator chooses a function name in the `Radio Buttons` object and presses the `Execute Function` button.

Note

Control inputs have no effect on objects' propagation. The `asText` sequence output is connected to the `Call Function` sequence input to hold off the `Call Function` propagation until after receiving the `Function Name`.

Clearing Strip Charts

In a related program flow problem, the program in Figure 8-19 generates a strip chart that is cleared after counting a certain number of points or whenever the user clicks a button:

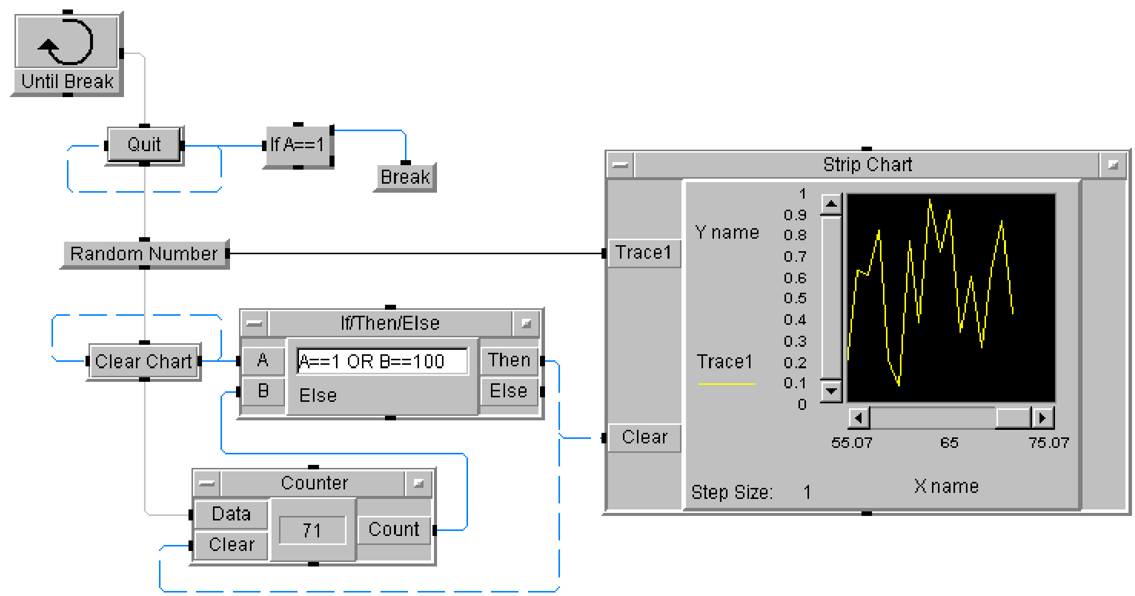


Figure 8-19. Using the Until Break Loop to Control a Strip Chart's Data Collection

The `Until Break` object drives the program. It controls the `Quit` Toggle, the `Random Number`, `Clear Chart` Toggle, and `Counter` objects through the sequence input and output pins. `Random Number` (the `random(high, low)` built-in function) object feeds values to the `Strip Chart` display. The `Counter` counts each loop iteration and outputs the count to the `If/Then/Else`. The `If/Then/Else` clears the `Strip Chart` and the `Counter` when a user presses the `Clear Chart` button or `Count` equals 100.

The `Toggle Control` object is driven continuously by `Until Break`, generating a 0 most of the time. Clicking `Toggle Control` toggles it to 1 and it is then reset by the feedback connection. The 1 is an input to the `If/Then/Else`. You can change the default appearance of the `Toggle Control` object by using `Properties` (object menu) to hide the Title Bar and then adding the `Reset` terminal.

Handling Propagation Problems

Sometimes program results are not what you might expect due to control pin usage, the way some objects propagate inside loops, or how parallel threads propagate. The following guidelines can help identify such problems.

Error Handling

Error handling is an important concept in VEE. It lets you perform an action then either repeat the action or continue after an error occurs. The program in Figure 8-20 demonstrates this with a dialog box to represent an action that can have different outcomes.

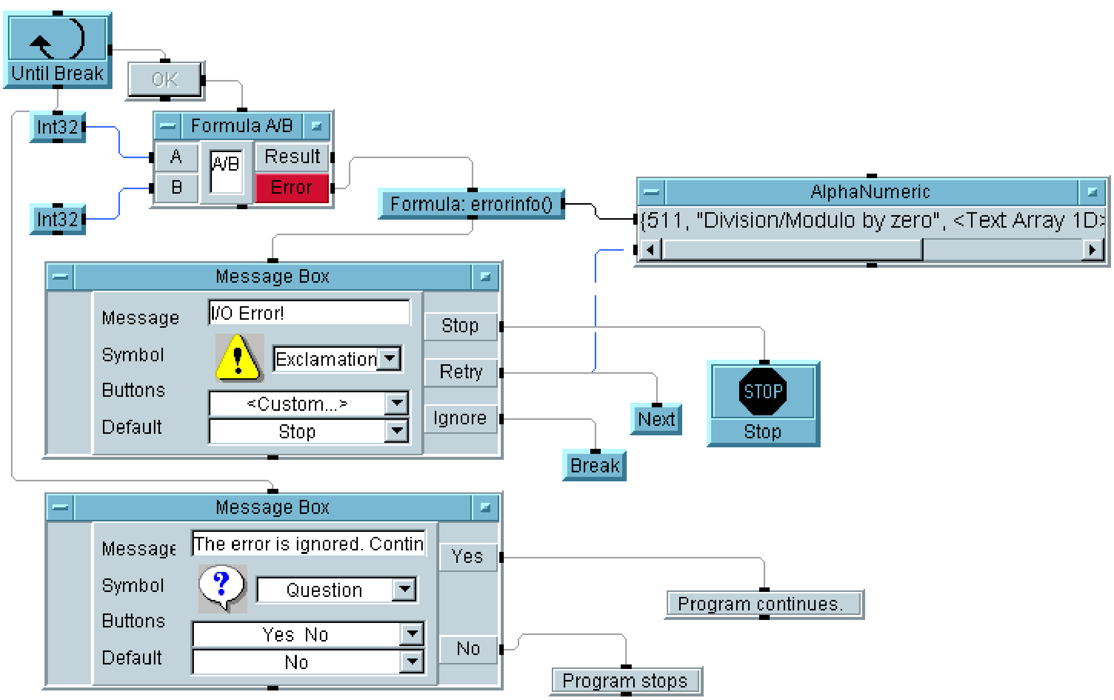


Figure 8-20. Using the Until Break Loop to Handle Error Conditions

This program pops up the `Exclamation Message Box`

asking if you want to Stop, Retry, or Ignore.

- `Stop` pings the `Stop` object to stop the program.
- `Retry` pings `Next` to reiterate the loop and redisplay the same `Message Box`.
- `Ignore` pings the `Break` object to stop the `Until Break` loop.

When `Until Break` stops, it pings the `Question Message Box` to offer more program-control options.

Notice that the element(s) to be executed sequentially after the “I/O” loop are connected to the sequence-out pin of the `Until Break`; they are not connected to any of the loop elements.

Avoid using error-handling as a standard practice, particularly with a Transaction object whose transactions contain complicated math formulas. VEE allocates memory to execute these formulas, and if an error occurs during execution that memory is not released, causing an incremental memory leak.

Capturing Control Pin Errors

Since control pins execute immediately and do not cause an object to propagate, certain conditions may cause your program to work incorrectly. If a control pin causes an error, you must use special programming techniques to capture the error programmatically.

In situations where a control pin causes an object to error, the program stops and VEE displays an error dialog. To capture and resolve the error programmatically, the typical solution is to add an `Error` output to the object. This solution works in most cases except when a control pin causes the error. Since control pins do not affect an object’s propagation, the object does not propagate the error information. That is, because the control pin does not cause the object to execute, the object cannot propagate. It is not allowed to propagate any output pins, including the Error pin, until it has executed. To capture an error caused by a control pin you must add an `Error` output to the context that contains the object.

The program in Figure 8-21 shows the wrong way to capture an error caused by a control pin. The program displays a waveform in the `XY Trace` display. The `XY Trace` has a `Scales` control input that requires a `Record` data type to change a scale on the display. `Text` and `Integer` provide values to the `Build Record` for the `Scales` control input. An `Error` output on the `XY Trace` is intended to capture any error condition, sending it to the `errorInfo()` function.

This program handles the error by displaying the error number and message. The program will capture any error generated in the `XY Trace` except when the `Scales` control input causes an error.

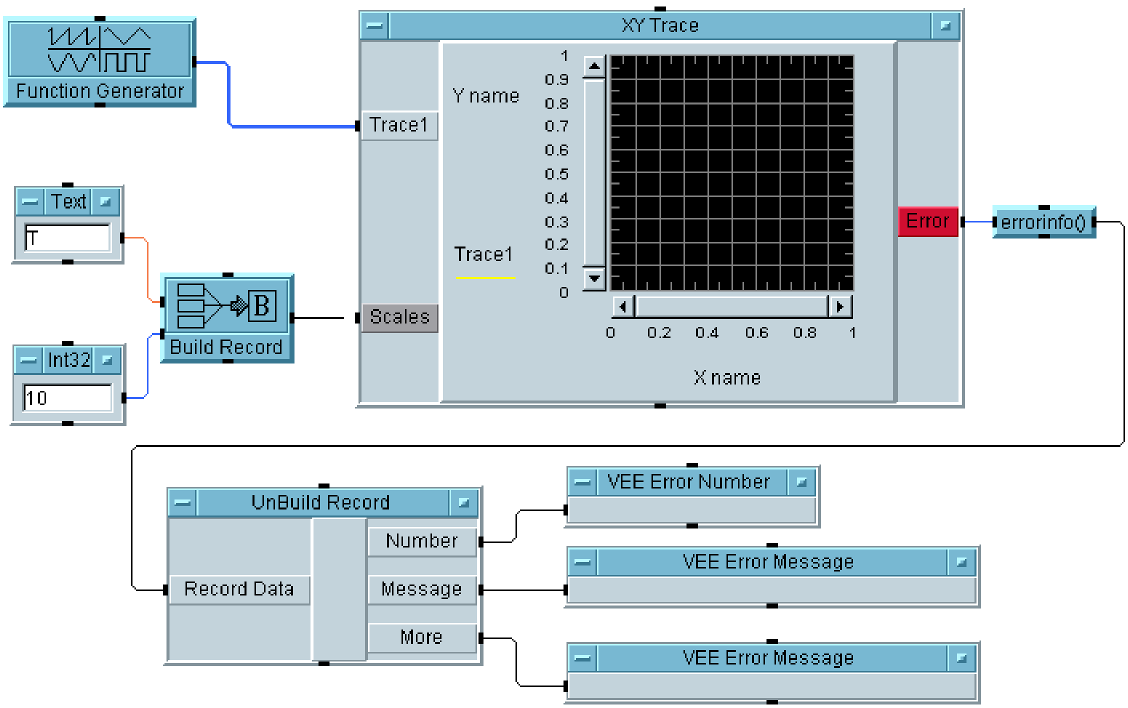


Figure 8-21. The Incorrect Way to Capture Control Pin Errors

Running the program reveals the problem of trying to programmatically capture an error caused by a control pin. The `Scales` control pin expects a Record containing, at a minimum, the value identifying the scale being changed. Allowed values are X, Y, Y1, Y2 and Y3.

The control pin generates an error because it receives the incorrect value `T` from `Text`. Since it is a control pin error, there is no further propagation in `XY Trace` and the `Error` output does not receive the error information. The program stops abruptly and VEE displays the error dialog in Figure 8-22.

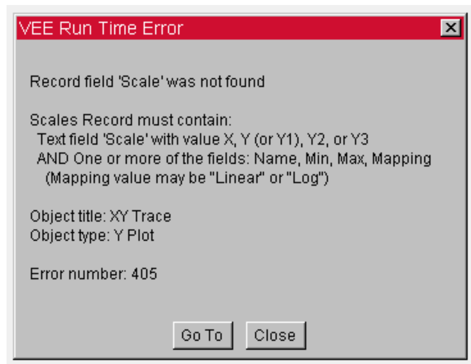


Figure 8-22. Error Dialog box

As explained previously, the correct way to capture a control pin error is to add an `Error` output to the context containing the object. The program in Figure 8-22 shows a solution where the `Build Record` and `XY Trace` are put into a UserObject. Notice that the `XY Trace` display's `Error` output has been deleted and an `Error` output is added to the UserObject. Also, the UserObject's `Error` output has been connected to the `errorInfo()`.

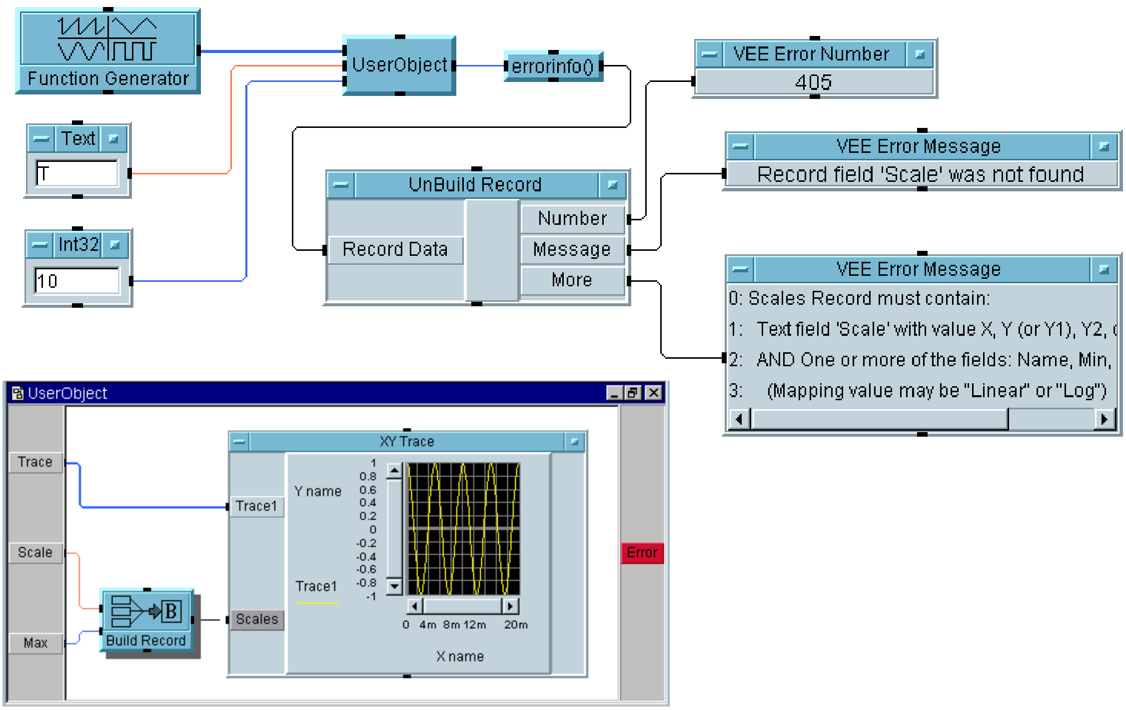


Figure 8-22. A Correct Way to Capture Control Pin Errors

Data Propagation on Control Pins

When an object's control pin receives data, such as a file name or default value, you should connect the object's sequence input pin or the program might fail. Since a control pin does not affect an object's propagation, the object will propagate when the data inputs receive data even if the control pin's value is not set.

The program in Figure 8-23 shows this sequencing problem. `Alloc Array` sends data to `To File` before the `dataFile2` file name is sent to the `File Name` control pin. When `To File` receives the data, it immediately changes the contents in `dataFile1` instead of the intended `dataFile2`. `To File` does receive the new file name on its control pin but it is too late.

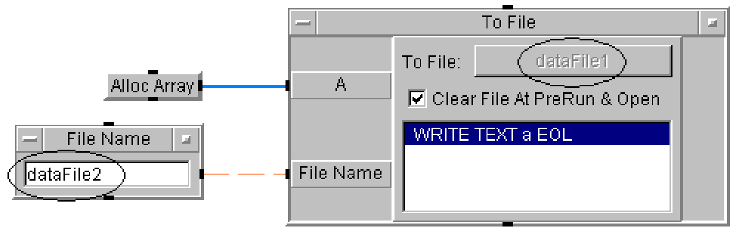


Figure 8-23. Sequencing Problems on Objects with Control Pins

To fix this problem, use `To File`'s sequence input to hold off the object's operation until after the control pin receives its data. The program in Figure 8-24 shows that connecting `File Name`'s sequence output to `To File`'s sequence input ensures data is written to the correct file.

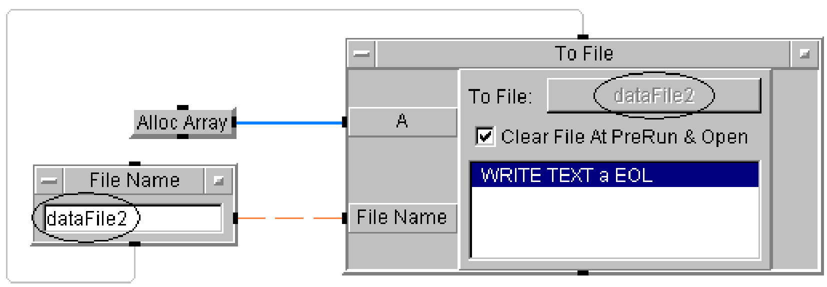


Figure 8-24. Using the Sequence Input on Objects with Control Pins

Building a Record

When trying to build a record of three waveforms as shown in the program in Figure 8-25, the `Build Record` object will never propagate. After the `Function Generator` sends its output to the `DeMultiplexer`, the `For Range` object starts its loop counting from 0 through 2. Each count sends the corresponding `Addr` out from the `DeMultiplexer` to the respective `Build Record` input.

Note

Within a loop, when an object with multiple outputs, such as the `DeMultiplexer`, sends data from one output on each loop iteration the other output values are invalidated at the beginning of each loop. This prevents propagating possibly old, incorrect data to the next object. This is also true for an object with multiple inputs, such as the `Build Record`.

When one input receives data on each loop iteration, values on all of its other inputs are invalidated at the beginning of each loop. VEE works this way to prevent a program from working with previous rather than current values, which can cause incorrect results.

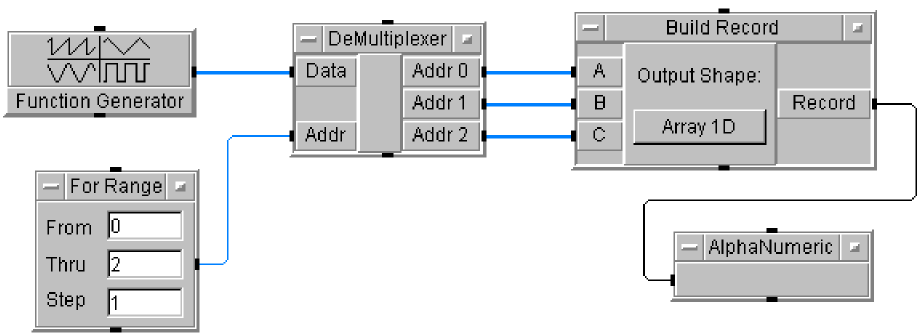


Figure 8-25. Invalid Data Inputs Stops Propagation on Build Record in a Loop

The `Build Record` object never propagates its `Record` output due to the way VEE loops work. In this program, each `For Range` iteration invalidates the data put on the `Build Record`'s inputs from the previous iteration. Since the `Build Record` object receives only one input on each loop iteration, only one input is valid at a time so there is no `Record` output.

Build this program yourself and turn on Show Data Flow to see how the DeMultiplexer only propagates one data output each time through the loop.

The program in Figure 8-26 makes this solution work as expected by using a UserObject. This solution works because a UserObject's output terminals hold the data until the iterations are done. The data are valid on the `Build Record` inputs since the UserObject sends the three outputs at the same time. When all three inputs contain valid data, the `Build Record` object outputs the expected results.

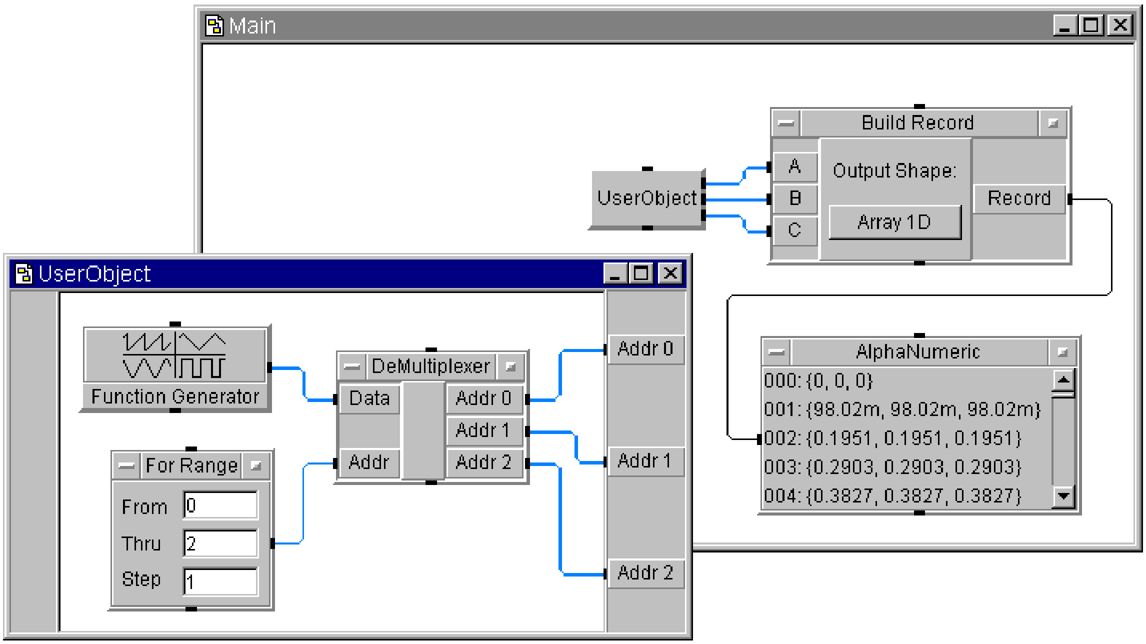


Figure 8-26. Maintaining Propagation When Data Inputs are Invalid

The program in Figure 8-27 solves the problem by replacing the `DeMultiplexer` with a `Shift Register`. Unlike the `DeMultiplexer` that has only one valid output at a time, the `Shift Register`'s three outputs are valid simultaneously since they are all sent at the same time to the `Build Record`'s three inputs. `Shift Register` outputs that contain no data propagate a nil.

This particular program clocks three waveforms into the `Shift Register` and then pings the `Build Record` to generate a record of them. If you prefer an array instead of a record output, you can use a `Collector`.

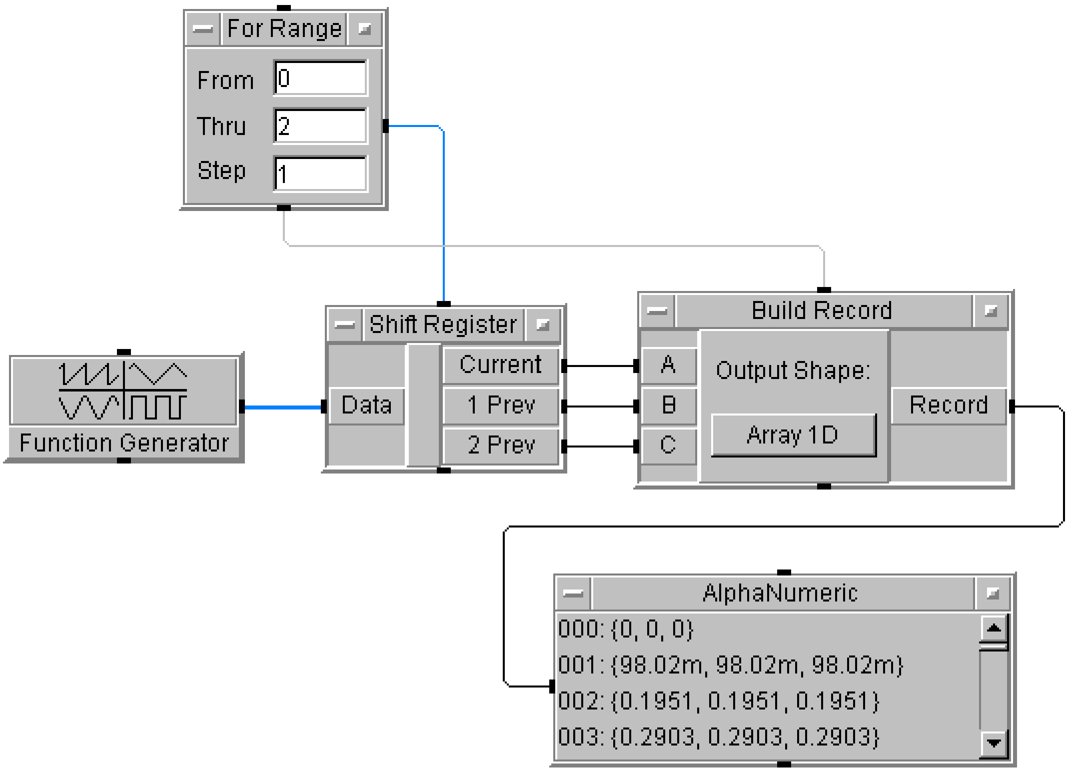


Figure 8-27. Maintaining Propagation by Preventing Invalid Data Inputs

Multiple Inputs to a Formula

Sending values to multiple inputs on a `Formula` object inside a loop can cause propagation problems if the values are sent during separate loop cycles. As shown previously with the `Build Record` object, the `Formula` will not operate if any input terminals contain invalid data. Using a `DeMultiplexer` object inside the loop adds to the confusion. The program in Figure 8-28 shows the problem.

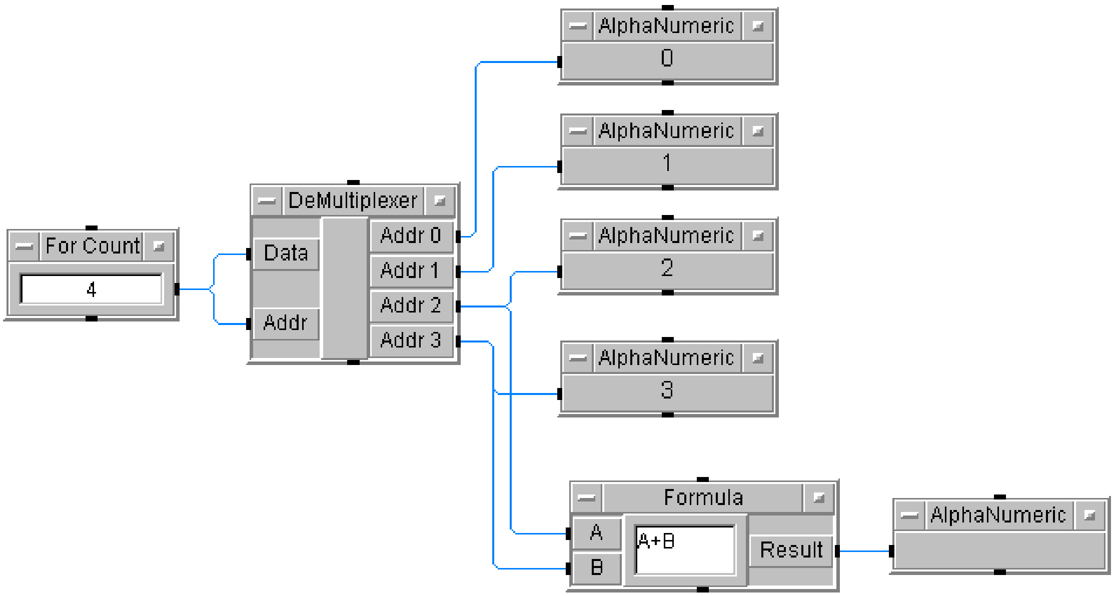


Figure 8-28. Invalid Data Inputs Stop Propagation on a Formula in a Loop

The `DeMultiplexer` is driven by the `For Count` object to output a single value (0 through 3) for each cycle of the count. Only one value is output per cycle so the inputs to the `Formula` object are made invalid after each cycle. Since the two `Formula` inputs `A` and `B` are never valid at the same time, the Formula never executes and there is no `Formula` output.

You may have decided that the value of 2 on pin A is still useful during the next loop iteration, but VEE does not have that insight. The loop might be calculating several coefficients for the same formula. There is no logical reason to solve a formula with half old coefficients and half new coefficients. As a general rule, it is safest for a programming language to assume that data from a previous iteration is "stale." That is why VEE invalidates an object's inputs at the start of each loop iteration.

Working with Loops

There are ways to work with a situation where unchanged input values are invalidated after a loop iteration. An example in the previous section showed how a `Shift Register` delivers multiple valid outputs simultaneously. A direct way to accomplish most tasks involves using variables.The program in Figure 8-29 shows how to use variables to supply valid values to a `Formula` object.

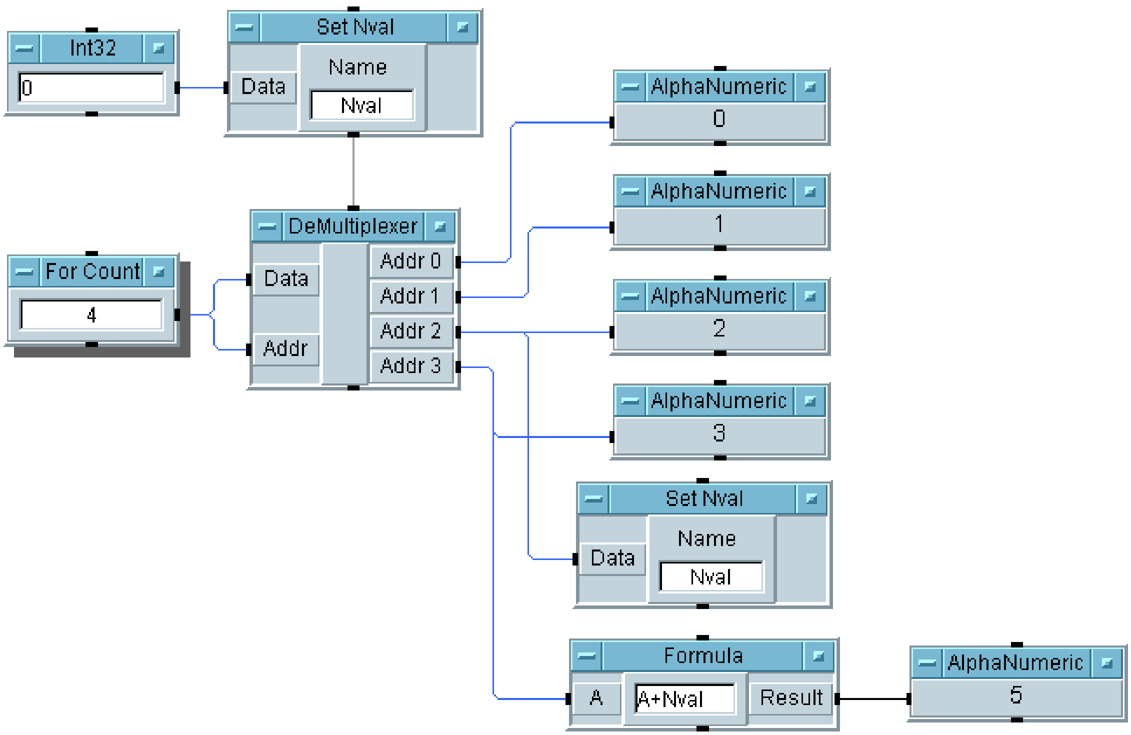


Figure 8-29. Using a Variable to Prevent Invalid Data Inputs on a Formula

There are two important points about the way the global variable `Nval` is used in this program. First, `Nval` is initialized when the program starts running. It is not necessary to do so in this particular program, since VEE initializes it before it is used, but it's a good programming practice. Second, `Nval` is always set with a new value before the `Formula` uses it.

You must be sure always to set a variable before an object must use it, or your program will have a problem similar to those with invalid data. If the variable is initialized, its value might be incorrect. If the variable is not initialized, your program will cause an error, such as `Variable was not found`.

Timing Events

The `Timer` object can display odd (possibly bad) results depending on how it is connected in a program. The program in Figure 8-30 demonstrates how VEE propagation issues should influence the way you connect objects in a program.

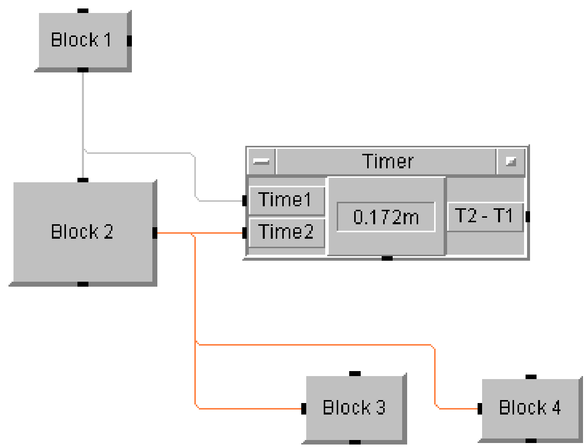


Figure 8-30. Uncontrolled Timer Inputs can Cause Timing Errors

The “Blocks” in this program are arbitrary threads containing some combination of VEE objects. The `Timer` has been added to time how long `Block 2` takes to execute. This program may run or cause an error. If it runs, the `Timer` may produce an erroneous result. There are two problems that affect when the `Timer` starts timing and when it ends.

First, the `Block 1` sequence output “pings” both `Block 2` and the `Timer`’s `Time1` input. The program does not specify which to ping first so it can choose either. If it chooses `Block 2` first, it will not ping `Time1` until `Block 2` is done. This condition can cause a bad timing result or an error if `Block 2`’s output pings `Time2` before `Time1` is pinged.

Even if you turn on the `Show Data Flow` debugging feature to help identify the problem, the data flow indicators may not fully indicate the actual propagation. In this situation, propagation depends on the order in which you connected the lines from `Block 1`'s sequence output to `Time1` and `Block 2`.

The second problem concerns the `Block 2` output connection to the `Timer`'s `Time2` input. Since it is also connected to the inputs of Block 3 and 4, there is no guarantee which input operates first. This can cause a bad timing result.

Figure 8-31 shows a way you can revise the program to ensure correct propagation and accurate timing. Insert a `Do` object between `Block 1` and `Block 2` and connect the `Block 2` sequence output to the `Time2` input. The `Do` object forces the order in which objects operate as shown by the numbers surrounding it in the program.

The output pins on `Block 2` operate in the order shown by the numbers around it. By connecting the `Block 2` sequence output pin to the `Time2` input, the `Timer` displays its result after `Block 2` and all the blocks its output pin is driving have completed. Likewise, if there are other blocks connected to `Block 2`'s sequence output pin, insert another `Do` object to ensure correct propagation.

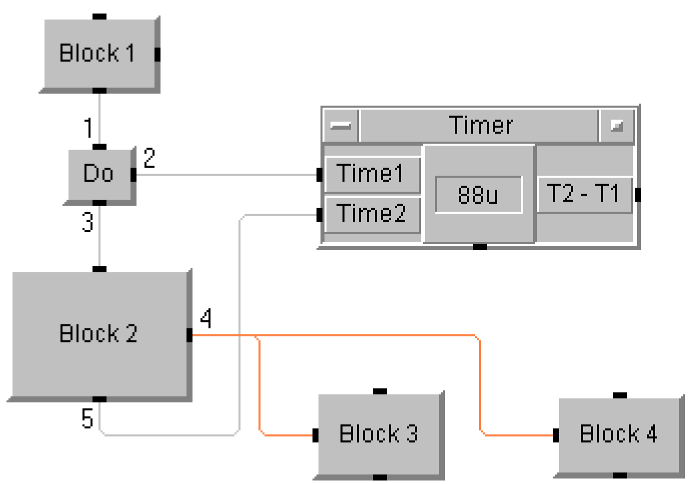


Figure 8-31. Using the Do Object with Timer for Accurate Results

9

Math Operations

Math Operations

This chapter describes math operations on scalars and arrays, including:

- Understanding Data Containers
- Data Type Conversions
- Processing Data
- Array Operations in VEE

Understanding Data Containers

Propagation of data through a VEE program consists of movement of data containers from one object to another. The data container is the VEE internal data format. Every data container has both a data type (text, real, etc.) and a data shape (scalar, one-dimensional array, etc.).

Data Container Operation

A data container may have only a single value in it, or it may have an array of several values. In either case, only one data container propagates from a particular data output pin when an object operates.

For the example program in Figure 9-1, the `Real64` constant object is configured as a one-dimensional array. The `Int32` constant object is configured as a scalar.

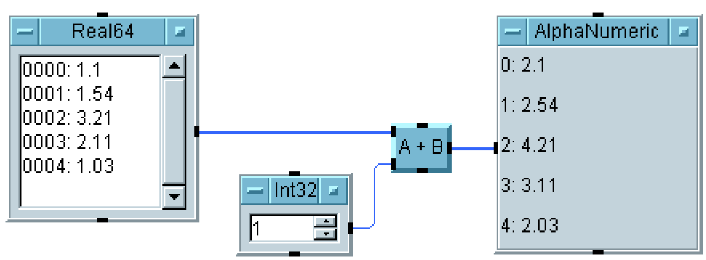


Figure 9-1. VEE Automatically Converts Data Types as Needed

When the program runs, the `Real64` constant object propagates a data container that is a one-dimensional real array. The `Int32` constant object outputs a data container that is an integer scalar (the value `1`). VEE provides automatic data type conversion to add these two containers.

VEE “promotes” the integer value `1` to become an equivalent Real value (`1.0`). The `a+b` object then adds the Real value `1.0` to every element in the one-dimensional real array and outputs the resulting one-dimensional real array, as shown above.

If you are interested in the specific container that has been passed on any VEE data line, you can use `Line Probe` to look at that information. Move the mouse pointer over the desired line so the line is highlighted and click the left mouse button. The `Line Value` box appears. For example, the container passed on the data output line from the `Real64` constant object of our example appears as shown in Figure 9-2.

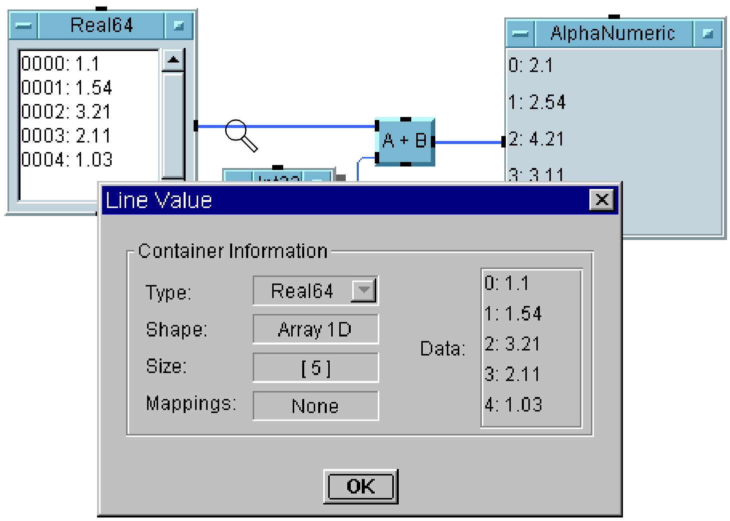


Figure 9-2. Left-Click a Line to View Its Data Container

In general, VEE converts data types automatically and resolves data shapes if possible. You normally do not need to worry about how this is done. However, for technical information about the conversion process see “Data Type Conversions” on page 262.

Terminals Information

Terminals show the object input type and shape requirements and display information about the input or output container. Pins are the connection points for terminals. You can display terminals if they are present but not visible on the object menu. Click `Properties` and, on the `General` tab, check `Show Terminals`.

To view or modify the attributes of a terminal, double-click the terminal's information area (not the pin). You'll see a dialog box showing terminal information.

If all the fields in the dialog box are grayed out, the terminal cannot be modified. However, if some of them are entry fields (white backgrounds) or buttons, you can change the values.

Terminals have the following characteristics:

- `Name` is the name of terminal. You can usually modify this field. In formula expressions the terminal name can be used in the expression.
- `Mode` displays the terminal type, such as `Data`, `Control`, `Trigger`, or `Error`. You cannot modify this field.
- `Required Type` and `Required Shape` (input terminals only) specify information about the input data that the object expects. On some objects, you can modify the `Required Type` or `Shape`, but you normally will not need to do this.
- `Container Information` contains information about the container that the object will process (according to the input requirements on an input terminal) or has processed (on an output terminal). This information includes the data type, data shape, the size (if data is an array), any mappings, and the data itself.

Data Type Conversions

Conventional programming languages typically require manual conversion between data types. VEE automatically converts most data types on the input terminals of objects and when using built-in type-conversion functions and objects.

Note Data shapes are not converted on input terminals, but data types and shapes may be converted automatically when used in math functions. See “Processing Data” on page 272. The conversion of data types for instrument I/O transactions is a special case. See “Instrument I/O Data Type Conversions” on page 270 for more information.

VEE Data Types

VEE provides 15 data types. For more information on data types and data type conversions, see “Data Type Conversions” on page 262. For more information on VEE support of ActiveX Automation and Controls, see Chapter 13, “Using ActiveX Automation Objects and Controls”.

Note If an input terminal on a VEE object specifies `Any` (the default in many cases), it will accept containers of any VEE data type. Composite data types (Waveform, Spectrum, Record, Coord and Object) are associated with particular data shapes.

Data Type Descriptions

The data types shown in Table 9-1 are used for all VEE operations. That is, every VEE data container sent between VEE objects is one of these types.

Table 9-1. VEE Data Types

Type	Description
Complex	A rectangular or Cartesian complex number. Each complex number has a real and an imaginary component in the form (real,imag). Each component is Real64. For example, the complex number 1 +2i is represented as (1,2).
Coord	A composite data type that contains at least two components in the form (x, y, ...). Each component is Real64. The data shape of a Coord must be a Scalar or an Array 1D.
Enum	A text string that has an associated integer value. The Enum data type is propagated by the objects found under Data ⇒ Selection Control (for example, the Radio Buttons object). You can access the integer value with these objects' ordinal output pin or by using the ordinal(x) function. The data shape of an Enum must be Scalar. Enum cannot be a required data input type.
UInt8	An 8-bit two's complement unsigned integer (0 to 255).
Int16	16-bit two's complement integer (-32768 to 32767).
Int32	32-bit two's complement integer (-2147483648 to 2147483647).
Object	A data type reserved for variables used for ActiveX Automation Objects and Controls when using VEE 5 or higher Execution Modes. Objects can be passed as inputs to and outputs from UserObjects and UserFunctions, but not to remote UserFunctions nor compiled functions. An Object variable contains values for the name of the Dispatch interface such as "Range" or "Application" and the pointer value of Dispatch which are exported from an Automation Object. Object data shape must be Scalar.

Table 9-1. VEE Data Types

Type	Description
PComplex	A magnitude and a phase component in the form (mag, @phase). Phase is in the currently active trigonometric units. For example, the PComplex number 4 at 30 degrees is represented as (4, @30) when Trig Mode is set to Degrees. Each component is Real64.
Real32	32-bit Real that conforms to the IEEE 754 standard (approximately 8 significant decimal digits: ±3.40282347E±38).
Real64	64-bit Real that conforms to the IEEE 754 standard (approximately 16 significant decimal digits: ±1.7976931348623157E308).
Record	a data type composed of fields. Each field has a name and a container which can be of any type (including Record) and a data shape of Scalar or 1D Array.
Spectrum	A composite data type of frequency domain values that contains the PComplex values of points and the minimum and maximum frequency values. Spectrum allows the domain data to be uniformly mapped as log or linear. The data shape of a Spectrum must be an Array 1D.
Text	A string of alphanumeric characters.
Variant	The Variant data type is not “fixed” as a specific kind of data. It can be one of the other data types as needed. Used for ActiveX automation methods that use ByRef Variant parameters.
Waveform	A composite data type of time domain values that contains the Real64 values of evenly-spaced, linearly-mapped points and the total time span of the waveform. The data shape of a Waveform must be an Array 1D (a one-dimensional array).

Line Colors for Data Types

In VEE 4 or higher Execution Modes, VEE assigns different colors to the data lines based on the type of data flowing through the line. Here are the default colors along with the names of the color properties (changeable via `File` ⇒ `Default Preferences`):

- Blue: numeric (Integer or Real type)
- Blue: complex (Complex and PComplex type)
- Orange: string (String type)
- Gray: sequence out (nil value, usually from a sequence out line)
- Black: unknown type or type that is not optimized (for example, Record types).

Note If the data type is an array, VEE displays a wider line. To increase speed, check your program for colored lines. The more non-black lines, the faster the program runs.

VEE Data Shapes

Composite data types (Waveform, Spectrum, Record, Coord, Enum, and Object) are associated with particular data shapes:

- The Waveform and Spectrum data types are always one-dimensional arrays.
- The Record and Coord data types can be either scalars or one-dimensional arrays. (They cannot be arrays of two or more dimensions.)
- The Object and Enum data types are always a scalar.

All other data types may have either a Scalar or an Array data shape:

- `Scalar` is a single number such as `10` or `(32, @10)`.
- `Array` is an array with one to ten dimensions.

Arrays may be mapped. (A mapping is a set of continuous or discrete values that express the independent variables for an array.)

In many cases, a VEE object has data pins with an input data shape requirement of `Any`, meaning that the object accepts containers of more than one of the data shapes.

Converting Data Types

This section shows how to convert data types on input terminals, how to convert data types with objects and functions, and instrument I/O data type conversions.

Converting Data Types on Input Terminals

Most objects accept any data type on their data input terminals, but a few objects require a particular data type or shape. For these objects, the data input terminal automatically attempts to convert the input container to the desired data type.

For example, a `Magnitude Spectrum` display needs Spectrum data. If the output of a `Function Generator` (a Waveform) is connected to the `Magnitude Spectrum` display, the input terminal of the `Magnitude Spectrum` automatically does an FFT to convert time-domain data to frequency-domain data (Waveform to a Spectrum).

The type conversion can be a promotion or demotion. A promotion is the conversion from a data type with less information to one with more. For example, a conversion from an `Int32` to `Real64` is a promotion. Such promotions take place automatically as needed.

A demotion is a conversion that may lose part of the data. For example, the conversion from a `Real64` to an `Int32` is a demotion because the fractional part of the Real number is lost. A demotion of data type occurs only if you force it by specifying a certain data type for an input on an object.

Once you have specified a data type, the demotion will occur automatically if it is needed and is possible.

For example, if you change the required type or input on a `Formula` object to `Int32` and another object supplies a `Real64` number to that input (such as `28.2`), the value will be demoted to an `Int32` (`28`). To change the data type on the `Formula` input from `Any` to `Int32`, double-click the input terminal's information area (not the pin) and then click the `Required Type` field. Click `Int32` in the drop-down list to change types.

VEE attempts to convert the data the next time the program runs. If the supplied data is a type that cannot be converted to the data type you select on the input, VEE returns an error.

Converting Data Types with Objects and Functions

VEE provides objects and built-in functions that convert data from one type to another. These are available to handle special type conversions that VEE cannot handle on its input terminals.

The type conversion functions built into VEE can be added to a program by using the `Function & Object Browser`, or by entering the function name into any object that accepts expressions. In the `Browser`, select `Type: Built-in Functions` and `Category: Type Conversion` for the available functions.

As an example, the function `asText(x)` converts the input `x` to the data type Text and returns the same data shape as `x`. `x` can be any shape and any type. In the expression `asText(3.4)`, the result is the Text value "`3.4`".

Note

Prior to VEE 5.0, an Integer was converted to a Real when it was entered directly into expressions in an object such as `Formula`. In VEE 5.0 and later, an Integer in an expression is no longer converted to a Real when the Execution Mode is set to `VEE 5` or higher. This change does not affect data type conversions on input terminals. For example, the Formula expression “2^4” will produce the `Real64` value 16.0 in VEE 4 and prior modes, but the `Int32` value 16 in VEE 5 and higher modes.

The Record data type has the highest priority. However, VEE does not automatically promote to or demote from the Record data type. To convert between Record and non-Record data, use the objects `Build Record` and `Unbuild Record`.

Similar results are possible in expressions using syntax described in “Using Records in Expressions” on page 276. For more information about Records, see Chapter 11, “Using Records and DataSets”.

The Coord data type has some special rules associated with it:

- Although arrays of `Int32` and Real data types can be promoted to Coord, a Coord cannot be converted to any other numeric type.
- When unmapped arrays are converted to Coord, the independent Coord values (the first Coord fields) are created from the array indexes while the dependent Coord value (the last Coord field) contains the element value. For example, if array `A` is converted to a Coord and `A` contains

[1,5,7], it is converted to a Coord array with [(0,1),(1,5),(2,7)] in it.

- When mapped arrays are converted to Coord, the independent Coord parameter ranges from the low value of the mapping to the value Xmin+(Xmax-Xmin/N)*(N-1).

The Object data type also has no automatic promotion or demotion and it cannot be converted to other data types. ActiveX automation objects use the Object data type. You can create and use automation objects by using the functions CreateObject and GetObject. For more information about VEE support for ActiveX Automation and Controls, see Chapter 13, "Using ActiveX Automation Objects and Controls".

Automatic Data Type Conversions

Table 9-2 shows the data type conversions that can occur automatically on input terminals and by using functions and which conversions cause an error. A "yes" means VEE can do the conversion, while a "no" means VEE returns an error. Demotions are indicated by the shaded areas.

The new "Variant" data type is not a distinct data type. It is used in expressions to indicate that the value it represents can be one of a number of other data types. It is the data type it holds at the moment. Any data type can promote or demote to a Variant, and a Variant can promote or demote according to the data it holds and the rules that data would follow. It does not appear in Table 9-2.

Table 9-2. Promotion and Demotion of Data Types

To → ↓ From	UInt8	Int16	Int32	Real32	Real64	Complex	PComplex	Waveform	Spectrum	Coord	Enum	Text
UInt8	n/a	yes	yes	yes	no[5]	no	no	no	no	no	no	no
Int16	yes	n/a	yes	yes	yes	yes[2]	yes[2]	no	no	yes[3]	no	yes
Int32	yes	yes	n/a[1]	yes	yes	yes[2]	yes[2]	no	no	yes[3]	no	yes
Real32	yes	yes	yes[4]	n/a	yes	yes[2]	yes[2]	no	no	yes[3]	no	yes
Real64	yes	yes	yes	yes	n/a	yes[2]	yes[2]	no	no	yes[3]	no	yes
Complex	no	no	no	no[5]	no[5]	n/a	yes	no	no	no	no	yes
PComplex	no	no	no	no[5]	no[5]	yes	n/a	no	no	no	no	yes
Waveform	no	yes[4]	yes[4]	yes[6]	no[5]	no	no	n/a	yes[7]	yes	no	yes
Spectrum	no	no	no	no	no[5]	yes[6]	yes[6]	yes[7]	n/a	no	no	yes
Coord	no	no	no	no	no[5]	no	no	no	no	n/a	no	yes
Enum	no	no[8]	no[8]	no	no[5]	no	no	no	no	no	n/a	yes
Text	no	yes[9]	yes[9]	yes[9]	yes[9]	yes[9]	yes[9]	no	no	yes[9]	no	n/a

1. n/a = Not applicable.
2. An Int32, or Real *value* promotes to Complex (*value*, 0) or to PComplex (*value*, @0).
3. The independent component(s), which are the first n-1 field(s) of an n-field Coord, are the array indexes of the value unless the array is mapped. If the array is mapped, the independent component(s) are derived from the mappings of each dimension. The dependent component, y, is the array element. If the container is a Scalar (non-array), conversion fails with an error.
4. These demotions will cause an error if the value is out of range for the destination type.
5. This demotion is not done automatically, but can be done with the re(x), im(x), mag(x) and phase(x) objects or the `Data` ⇒ `Build Data/UnBuild Data` ⇒ objects.
6. These demotions keep the Waveform and Spectrum mappings.
7. An FFT or inverse FFT is automatically done.
8. This demotion is not done automatically, but can be done with the `ordinal(x)` object.

9. This demotion causes an error if the text value is not a number (such as 34 or 42.6) or is not in an acceptable numerical format. The acceptable formats are as follows (spaces, except within each number, are ignored):
   - Text that is demoted to an Int32 or Real type may also include:
     - A preceding sign. For example, -34.
     - A suffix of e or E followed by an optional sign or space and an integer. For example, 42.6E-3.
   - Text demoted to Complex must be in the following format: (number, number).
   - Text demoted to PComplex must be in the following format: (number, @number). The phase (the second component) is considered to be radians for this conversion, regardless of the `Trig Mode` setting.
   - Text demoted to a Coord type must be in the following format: (number, number, ...).

Instrument I/O Data Type Conversions

On instrument I/O transactions involving integers, VEE performs automatic data type conversions in VEE 5 and earlier Execution Modes. See "Using VEE Execution Modes" on page 10 for an example of how this could change program behavior. VEE performs automatic data type conversions in the following ways:

- On a `READ` transaction in VEE 5 and earlier Execution Modes, `Int16` or `Byte` values from an instrument are converted to `Int32` values, preserving the sign extension. Also, `Real32` values from an instrument are converted to 64-bit `Real` numbers. Vee 6 Execution Mode produces true `Int16`, `Int32`, and `Byte` (`UInt8`) values.

- On a `WRITE` transaction in Vee 5 and earlier Execution Modes, `Int32` or `Real` values are converted to the appropriate output format for the instrument, as described in the following bullets. VEE 6 Execution Mode writes true `Int16`, `Int32`, and `Byte` (`UInt8`) values.

  - If an instrument supports the `Real32` format, VEE converts 64-bit `Real` values to `Real32` values, which are output to the instrument. If the `Real` value is outside of the range for `Real32` values, an error occurs.

  - If an instrument supports the `Int16` format, VEE truncates `Int32` values to `Int16` values, which are output to the instrument. `Real` values are first converted to `Int32` values, which are then truncated in VEE 5 Execution Mode and output. VEE 6 Execution Mode outputs the number without truncating it. `If a Real` value is outside the range for an `Int32`, an error occurs.

- If an instrument supports the `Byte` format, VEE 5 Execution Mode truncates `Int32` values to `Byte` values, which are output to the instrument. `Real` values are first converted to `Int32` values, which are then truncated in VEE 5 Execution Mode and output. VEE 6 Execution Mode outputs the number without truncating it. If a `Real` value is outside the range for an `Int32`, an error occurs.

Processing Data

To process data, you operate on it with the operators and functions available in the `Function & Object Browser`. Use the `Function & Object Browser` toolbar button to open the browser. You can combine the functions to create mathematical expressions.

The Function & Object Browser

The `Function & Object Browser` contains a set of mathematical functions to process your data in numerous ways. Each of these functions are expressions entered in a `Formula` object with the corresponding title, inputs and outputs. You can change the expressions in the open view of each `Formula` object and change their properties also.

All the functions that are listed in `Function & Object Browser` can be used in any object in other menus that allows expressions. The objects in other menus that allow expressions are:

- `Data` ⇒ `Access Array` ⇒ `Set Values`
- `Data` ⇒ `Access Array` ⇒ `Get Values`
- `Data` ⇒ `Access Record` ⇒ `Get Field`
- `Data` ⇒ `Access Record` ⇒ `Set Field`
- `Flow` ⇒ `If/Then/Else`
- `Flow` ⇒ `Conditional` (all conditional objects)
- `I/O` objects that use transactions

General Concepts

You can process data before running a program by using numeric entry fields such as those in `Constant` objects. Numeric entry fields on some objects support the use of arbitrary formulas. The formula is immediately evaluated and the resulting value is used for the field. You cannot use variables in `Constants`.

The typed-in formula must evaluate to a scalar value of the proper type or of a type that can be converted to that which the object expects. In general, you can use any of the dyadic operators, parentheses for nesting, function calls and the predefined numeric constant `PI` (`3.1416`...) in numeric entry fields.

Expressions and Functions

Expressions may contain the names of data input terminals, data output terminals (`I/O` transactions and `Formulas` only), variables (declared, of any scope, and undeclared), user-defined functions (compiled, remote and UserFunctions), and any mathematical expression from the `Function & Object Browser`. Data input terminal names are used as variables.

VEE is not case-sensitive about names of input variables within expressions for USASCII keyboards. For non-USASCII keyboards, VEE is case-insensitive for 7-bit ASCII characters only. Expressions are evaluated at run time.

If you pass an array to a function, the function operates on each element of the array, unless stated otherwise. For example, `sqrt` of a scalar returns a scalar; `sqrt(4)` returns `2`. But `sqrt` of an array returns an array of the same size; `sqrt([1,4,9,64])` returns the array `[1,2,3,8]`.

In `VEE 5` or higher Execution Modes, all numbers entered as integers in an expression field are considered to be `Int32`. In `VEE 3` and `VEE 4` Execution Modes, all such numbers are considered `Real64` values, unless you use parentheses to specify Complex or PComplex values. Therefore, `2` is considered to be a Real number or an `Int32,` depending on the Execution Mode. `(1, @2)` is a PComplex number, while `(1, 2)` is a rectangular Complex number.

Note VEE interprets any value contained within parentheses as a Complex or PComplex value. If you need to use a Coord value in an expression, use the `coord(x, y)` function. The `coord` function takes two or more parameters. `coord(1, 2)` returns a Scalar Coord container with two fields.

All functions that operate on Coord data operate only on the dependent (last) field of each Coord. For example, `abs(coord(-1, -2, -3))` returns the Coord `(-1, -2, 3)`.

An Enum container is always converted to Text before every math operation except the function `ordinal(x)`. Enum arrays are not supported. If you try to create an Enum array, a Text array is created instead.

For information on specific data type definitions, see “VEE Data Types” on page 262.

Using Strings in Expressions

Strings within expressions must be surrounded by double quotes. You may use the escape sequences in Table 9-3 within strings:

Table 9-3. Escape Sequences Characters

Escape Character	Meaning
\n	Newline
\t	Horizontal Tab
\v	Vertical Tab
\b	Backspace
\r	Carriage Return
\f	Form Feed
\”	Double Quote
\’	Single Quote
\\	Backslash
ddd	Character Value. *d* is an octal digit.

Using Variables in Expressions

You can create and set variables by using the `Declare Variable` and `Set Variable` objects, and you can access variables by using the `Get Variable` object. See `Declare Variable`, `Set Variable`, and `Get Variable` in *VEE Online Help* for more information.

In addition, you can access a variable by including its name in a mathematical expression. You can include a variable in a mathematical expression in a `Formula` object, or in any object with a delayed-evaluation expression field.

These objects include `If/Then/Else`, `Get Values`, `Get Field`, `Set Field` and all instruments using expressions in transactions, including `To File`, `From File`, `From DataSet`, and `Direct I/O`.

To include a variable in an expression, just use the variable name as if it were an input variable. For example, suppose a program uses a `Set Variable` object to define the variable `numFiles`. Elsewhere in the program, a `Formula` object with input `A` may use the expression `numFiles+3*A`.

Note

Variable names are case-insensitive. Either upper-case or lower-case letters may be used. Thus, `GLOBALA` is equivalent to `globalA`.

To avoid errors or unexpected results, be aware of two limitations when you include variables in an expression:

1. *Local input variables have higher precedence than global variables*. This means that in case of duplicate variable names, the local variable is chosen over the global variable. For example, if the expression `Freq*10` is included in a `Formula` object that has a `Freq` input (a local variable) and there is also a global variable named `Freq`, the expression will be evaluated with the local variable `Freq`, not the global one. No error will be reported regarding this duplication.

2. *Depending on the flow of your program, an object that evaluates an expression containing a variable may execute before the variable is defined*. For example, suppose the variable `globalA` is set with a `Set Variable` object and the expression `globalA*X^2` is included in a `Formula` object.

   Depending on the flow of your program, the `Formula` object may execute before the `Set Variable` object executes. In this case, the `Formula` object won't be able to evaluate the expression because `globalA` is undefined. An error message will appear.

   It is important that you take steps to ensure correct propagation – that `Set Variable` executes first. You can do this by connecting the sequence output pin of the `Set Variable` object to the sequence input pin of the `Formula` object, in this case, or of any other object that includes the variable in an expression to be evaluated.

   If a `Get Variable` object is used, its sequence input pin should also be connected to the sequence output pin of `Set Variable`. Also, if you declare a variable using the `Declare Variable` object, you must initialize it using `Set Variable`. For more information, see Chapter 10, “Variables”.

Note

By default, `Delete Variables at PreRun` in the `Default Preferences` dialog box is checked (enabled) so values are deleted from variables when you run a program. This prevents variables from containing “old” data and causing unexpected results.

Variables can be arrays. Just access a variable array as if it were an input variable using array syntax, for example: `GlobAry[2]`. If a variable is a `Record`, use the record access syntax, such as `globRecord.numFiles`.

Using Records in Expressions

You can use expressions to access a field or sub-field of a record. Use the `A.B` sub-field syntax to access the `B` field of a record `A`. If `A` is a record with a field `B`, which itself is a record which has a field `C`, you may use the `A.B` syntax recursively to access the `C` field. That is, use the expression `A.B.C`. If `A` does not have a `B` field, or `B` does not have a `C` field, an error will result.

There is no limit on the number of recursions of `A.B.C.D.E.F` that may be used in expressions. Field names are not case-sensitive (lowercase and uppercase letters are equivalent). Field names may be duplicated in sub-Records, so you may use the expression `A.a.A`.

Records are very useful as variables so one variable may hold several different values. A `Formula` object can be used in place of a `Get Variable`. Thus, you can accomplish the `GlobRec.numFiles` access in one object, instead of using both a `Get Variable` and a `Formula` object to unbuild the record.

The record and array syntax may be combined in expressions to access a field of a record array (for example `A[1].B`), or to access a portion of an array that is a field of a record (for example, `A.B[1]`). Note the difference between `A[1].b` and `A.b[1]` (both are supported):

- ❑ You would use the first for a record 1D with a scalar field `b`. `A[1].b` accesses the field `b` of the second record element of the record array `A`.

- You would use the second for a scalar record with a field b, which is a 1D array. A.b[1] accesses the second element of the field b of the record A.

To change a field in a record, use the assignment operator in a Formula object. For example, suppose you have a record R with a field A and you wish to change the value of R.A to be sin(R.A). Just change the expression to R.A = sin(R.A). You can continue to use the record R (with the new value for field A) later in your VEE program.

Note

For information about using Objects in expressions to manipulate ActiveX automation objects and controls, see Chapter 13, "Using ActiveX Automation Objects and Controls".

Using Assignment Operations

The Formula object allows expressions that use assignment operations to change values in parts of arrays and records and assign values to local and global variables. The Result output terminal contains that part of the array or record that changed, not the entire array or record.

For example, in a Formula with the expression A[2] = 4 the Result terminal contains array element A[2] with a value of 4. It does not contain all of array A. Formula objects preset with assignments are available in the Function & Object Browser in Type: Operators, Category: Assignment.

Note

Assignment operations are allowed only in Formula objects.

For information about using assignment operations to manipulate ActiveX automation objects and controls, see Chapter 13, "Using ActiveX Automation Objects and Controls".

Allowed Syntax. Multiple expressions, separated by semi-colons, are allowed in Formulas. The left-hand side of expressions allow syntax that change the values for array and record elements and for variables. The right-hand side must match *exactly* the part of the left-hand side that is being modified. For records, the schema (a field's type, shape, or size) cannot be changed, only its values.

The following examples show left-hand syntax that work for arrays:

```
A[2]=
A[2, 3, 4:5]=
A[2:4, 4:6, 3:*, *]=
```

The following examples show left-hand syntax that work for records:

```
RecA.B=
RecA.B.C.D=
RecA[1].B=
RecA.B[1]=
RecA.B.C.D[1]=
RecA[1].B[2].C[3].D=
RecA.B[2:3].C[3:4]=
```

The following left-hand syntax is allowed to directly set global and local variables and initialize declared local variables:

```
GLOBAL=2
TMP_LOCAL=4
```

Examples. Here are examples of assignments showing how the right-hand side must match the part of the left-hand side that is being modified. The data type of the right-hand side must be coercible to the data type of the left-hand side, such as Integer to Real or Real to String. A coercion such as Complex to Real cannot be done.

- `ArrayA[2:4] = ArrayB`
  (`ArrayB` must be a one-dimensional array with three elements.)

- `ArrayA[2, 3, 4:5, 7, 8:9] = ArrayB`
  (`ArrayB` must be a two-dimensional array, of size 2 by 2.)

- `Rec[3:4].field = ArrayB`
  (`ArrayB` must be a one-dimensional array with two elements.)

- `Rec[3:4].field[4:5] = ArrayB`
  (`ArrayB` must be a two-dimensional array, of size 2 by 2.)

Non-explicit use of arrays of records on the left-hand side of assignments is not allowed. If `Rec` is an array of records, the expression `Rec.A=2` will cause an error, prompting a request that you use the full explicit syntax, Rec[*].A=2. A similar error results with Rec.B=2 if Rec is a scalar record and B is an array. The resulting prompt will request that you use the explicit Rec.B[*]=2.

Note

In VEE 4.0 and later versions, the Set Values and Set Field objects are actually Formula objects with assignment expressions. These objects have been changed from their definitions in prior versions of VEE. Existing programs written with VEE 3.x and older that use the Set Values and Set Field objects will retain the prior definition if they continue to run in the VEE 3 Execution Mode.

Error Recovery

Since a Formula can contain a series of expressions, including assignments and other operations, errors are handled in certain ways. If an assignment or other operation is done and a following expression errors, previous expressions are not undone.

Consider the expressions, Global[2]=24; 2/0, in a Formula.

First, Global[2] is set to 24. Then, the division by zero causes an error. VEE will not set Global[2] back to its previous value.

Note

For procedures about using assignment operations, see *VEE Help* (Help ⇒ Contents). In the How Do I … section, open the Work with Data section, then look at the topics under Working with Arrays (such as To Change Values in an Array) and at topics under Working with Variables.

Using Global and Local Variables

Assignments in Formula objects can change values in global and local variables with expressions such as A=2 or GlobalA[5]=2. Since variables can be undeclared globals, declared globals or locals, or directly-set locals, the Formula object will look for the variable A using the following order of precedence:

1. A local variable which is an input terminal. This overwrites the input terminal value, including its type and shape.

2. A local variable which is an output terminal. This variable is created and placed on the output terminal.

3. A global variable. The variable must already exist. Its value is completely changed, including its type and shape.

Given these rules, an error results if `Formula` contains an assignment such as `tmp=2` where `tmp` does not meet one of these criteria.

Global and Local Variables in Assignments

Assigning values to global arrays and records requires added attention since these variables may be undeclared or declared. A global exists when it is created using an object such as `Set Variable`, or when it is declared using `Declare Variable`.

An undeclared global's type and shape can be changed by an assignment expression. However, for a declared global the right-hand side of the assignment must be coercible to the global's declared type.

Note

Very Important! When you declare a global array or record variable using the `Declare Variable` object, the entire variable must be initialized collectively before you can change part of the global. The program in Figure 9-3 shows the variable `GlobalAry` declared and initialized before an individual part is changed by the assignment in `Formula`.

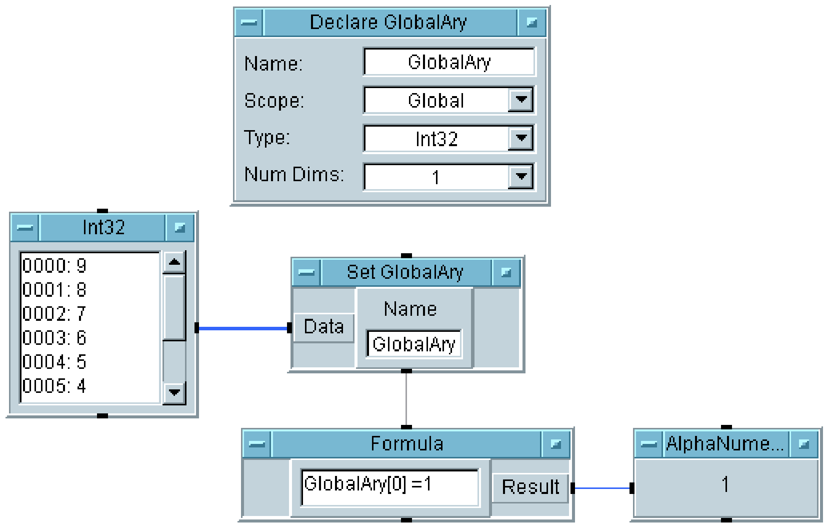


Figure 9-3. Initializing a Declared Global Variable

Data Container Contents on Terminals

`Formula` object input and output terminals affect how the values for variables are changed by assignments. The data container on the `Result` output terminal contains the modified part of an assignment expression's left-hand side, including any changes to the data type and shape.

In a `Formula` with the expression `ArrayA[2:4] = ArrayB[5:6]`, the `Result` output terminal contains `ArrayA[2:4]` with the values from `ArrayB[5:6]`. In an expression such as `ArrayA[2]=4`, where the `ArrayA` data type is Complex, the value for `ArrayA[2]` in the output terminal is converted to Complex (`4,0`).

The following output terminal names can be used:

- `Result`. This is the default output terminal on `Formula`. It is a reserved name and contains the result of an assignment expression. You can delete `Result` to use other outputs, then add it back if needed, but you cannot rename an output terminal `Result`.

- An input terminal name. This value is copied from the input. If changed in an assignment, the new value is used on the output. The variable must exist and have a value before it can be used as an input terminal name.

- A local variable name. This is a name, such as `Tmp,` that is created by an assignment expression, such as `tmp=2`.

You can use additional output pins on a `Formula` to get various parts of a modified array or record out of the object. This lets you pass the modified part of a variable out one terminal and the whole variable out another terminal. To get the whole array or record on the output terminal, use a global variable in an assignment expression or add an output terminal to the `Formula` object for the array or record. If an output terminal exists that does not get a value assigned, an error occurs.

Mappings on arrays are ignored unless the entire container is modified. For example, `ArrayA[2:4] = ArrayB[2:4]` does not modify the mappings on `ArrayA`. But `A = ArrayB[2:4]` will set the mappings on `A` since it is replaced by `ArrayB`.

Using Dyadic Operators

The set of dyadic operators have several additional conditions and guidelines. The dyadic operators are visible in the following categories of the `Function & Object Browser` in `Type: Operators.`

Dyadic Operators Categories

- Category: Arithmetic
  - `a + b`
  - `a - b`
  - `a * b`
  - `a / b`
  - `a ^ b` (exponentiation)
  - `a mod b` (modulo - returns remainder of division)
  - `a div b` (integer division - no remainder)
- Category: Comparison
  - `a ~= b`
  - `a == b`
  - `a != b`
  - `a < b`

- ❑ `a > b`
- ❑ `a <= b`
- ❑ `a >= b`

- Category: Logical
  - ❑ `a AND b`
  - ❑ `a OR b`
  - ❑ `a XOR b`
  - ❑ `NOT a` (a monadic that follows the same guidelines as dyadics)

When using dyadic operators on arrays, the array size, array shape, and array mappings (if they exist) must match. For Coords, the values of the independent variable for each Coord must match.

Precedence of Dyadic Operators

This list is the order of precedence of the dyadic operators. They are listed from highest to lowest, with operators of the same precedence listed on the same level.

1. parentheses `(` and `)` used to group expressions
2. `^`
3. unary minus `-`
4. `* / MOD DIV`
5. `+ -`
6. `~= == != < > <= >=`
7. `NOT`
8. `AND`
9. `OR XOR`

Dyadic Operators Data Type Conversion

For the dyadic operators, the input values are promoted to the highest data type, then the operation is performed. The data type of the output is the highest input data type. For example, when the complex number `(2, 3)` is added to the String `"Dog"`, `"Dog"+(2,3)`, the result is the String `"Dog(2, 3)"`.

Note

There is one exception to this rule. When you multiply a Text string by an `Int32`, the result is a repeated string. For example, `"Hello"*3` returns `HelloHelloHello`. No data type promotion occurs in this case.

The data type order (from highest to lowest) is:

1. Object
2. Record
3. Text (Enum is treated as Text)
4. Spectrum
5. PComplex
6. Complex
7. Coord (no conversion to any other numeric type possible)
8. Waveform
9. Real64
10. Real32
11. Int32
12. Int16
13. UInt8
14. Variant

The Variant data type is not "fixed" as a specific kind of data. It can be one of the other data types as needed. In the function sin(var), data in the Variant data type (var) could be an integer, a real, or a waveform, depending on the value it is assigned to. In the expression a=sin(var), a will have the data type of whatever data (var) contained.

Dyadic Operators Considerations

Object Considerations. Objects will not automatically demote to other types. No dyadic operations are supported on Objects themselves, but since most Objects have a default property which is a String or Integer value, most operations can be performed on Objects.

The difference is that you end up performing the operation on the default property. For more information about Objects and their use with ActiveX automation in VEE, see Chapter 13, "Using ActiveX Automation Objects and Controls".

Record Considerations. Records have the highest precedence of all data types, but other data types can be converted to the Record data type *only* by using special objects such as `Build Record`. Records will not automatically demote to other types, nor will other types automatically promote to the Record type. Objects and Variants cannot be Record fields.

The dyadic operators do support combining records and other data types, but they will always return a record in this case. A dyadic operation on a record and non-record will apply the operation with the non-record to every field of the record.

For example, consider a record `R` with two fields `A`, a scalar Real value (2.0) and `B`, a scalar Complex value (3,30). The expression `R+2` will produce a record `R` with two fields `A`, a scalar Real with value 4 and `B`, a scalar Complex with value (5,30). If the operation cannot be performed on every field in the record, an error occurs.

Dyadic operations on a record and any other type will return a record with the same "schema", so the resulting record will have the same fields with the same names, types, and shapes. The dyadic operation may not change the type or shape of a record field.

For example, consider a record `R` with two fields: `A`, a scalar Real and `B`, a scalar Complex. The expression `R+(2,3)` will cause an error. VEE will first try to add `(2,3)` to `R.A`, then do the same with `R.B`.

The error occurs because the `R.A` field is a Real and the result of `R.A+(2,3)` would be a Complex. The Complex cannot be demoted to a Real to be stored back into `R.A`.

Dyadic operations on records using arrays treat the record as having higher precedence than the array. For example, `[1,2,3] + [3,4,5]` produces `[4,6,8]`, so the arrays are combined piece by piece. But, records have higher precedence than arrays. This means that if `R` is a record with two fields: `A` and `B`, the expression `R + [1,2]` will try to add the array `[1,2]` to each field of R. It will *not* add `1` to `R.A` and `2` to `R.B`.

Things get even more complicated when you combine arrays with record arrays. For example, suppose `R` is a record 1D array, two long, with three fields: `A`, `B` and `C`. The expression `R + [1,2,3]`, or the expression `R + [1,2]` will add the entire array to each field `A`, `B` and `C` for every element of `R`. Even though `R` is an array, the fact that it is a record is more important.

A dyadic operation on two records will combine them field by field so the two records must have the same "schema". That is, each record must have the same number of fields and each field must have the same name, type, and shape, in the same order.

If you want to add 1 to field A, add 2 to field B, etc., the easiest way is to use multiple assignments (see Assignment in *VEE Online Help*). In a Formula object, enter the expression R.A=R.A+1,R.B=R.B+2. You can then use R.A and R.B with their new values in your program.

Spectrum Considerations. If you choose to use dB scaling, you must keep track of it yourself. Although dB-scaled data displays correctly (except on the Waveform (Time) display), many math functions such as fft(x), ifft(x), and those involving PComplex numbers do not operate correctly on dB-scaled data.

If you need to use these operations, convert the dB-scaled data to linear scaling before operating on it. VEE supplies library programs for dB conversions in its installation location, typically:

```
C:\Program Files\Agilent\VEE OneLab 6.0\lib\convert
```

When you are using particular dB units, some math functions give meaningful results, but only within the confines of those units. For example, if you add 20 to a dBW-scaled Spectrum, 20 is added to the magnitude of each element (which has the same effect as converting the Spectrum to a linear scale, multiplying each element by 100, and converting back to dBW.).

Data Shape Considerations. For dyadic operations where both operands (inputs) are arrays, the size and shape of the arrays must match. The result of the operation is an array with the same size and shape as the input arrays, except for the relational operators (==, <, etc.), which always return a scalar. If arrays have a different number of dimensions or are of different sizes, VEE returns an error. For example, [1,2] + [1,2,3] returns an error.

If you are operating on a scalar and an array, the scalar is treated as if it were a constant array of the same size and shape as the array operand. For example, 2 + [1,2,3] is treated as [2,2,2] + [1,2,3]. The result is [3,4,5].

When an *n*-dimensional array is converted to a Coord, the Coord data shape is an Array 1D with *n*+1 fields in each Coord element.

Variant Considerations. The result of dyadic (+-*/, etc.) evaluations and functions cannot be a Variant.

- In the expression, “b=a”, “b” will be the same type as “a”, even if it was a Variant.
- In the expression, “b=a+2”, “b” will never be a Variant, regardless of what “a” is.
- In the expression, “b=sin(a)”, “b” will never be a Variant, regardless of what “a” was.
- The function “func(2+a)” will always be sent a non-Variant, regardless of what “a” was.

Just a variable name, “a”, a monadic operator, “-a”, or parentheses, (a), will not change the data type of “a”.

- The function “func(a)” will be sent a Variant if “a” is a Variant.

Array Operations in VEE

VEE is optimized for array math. While you can perform array operations using traditional loop constructs, they tend to degrade program speed. This section shows ways to use the `Formula` and other objects to perform math operations on arrays. Assignment operators, discussed in "Using Assignment Operations" on page 277, also let you change values in parts of arrays.

Note

You can adapt the examples in this section and use assignment operators to avoid using time-consuming computational loops. Since these techniques are not always obvious, be careful about using them and be sure to document your programs thoroughly.

Array Operations Techniques

This section shows some array operations techniques for VEE, including comparison of array operation techniques, accessing arrays in expressions, performing array math operations, and using variables in expressions.

Comparison of Array Operation Techniques

The program segments in Figure 9-4 and Figure 9-5 compare techniques for generating an array containing all the values of sine and cosine for each degree from 0 to 360. If you try each of these techniques, you will find that the first technique takes more time.

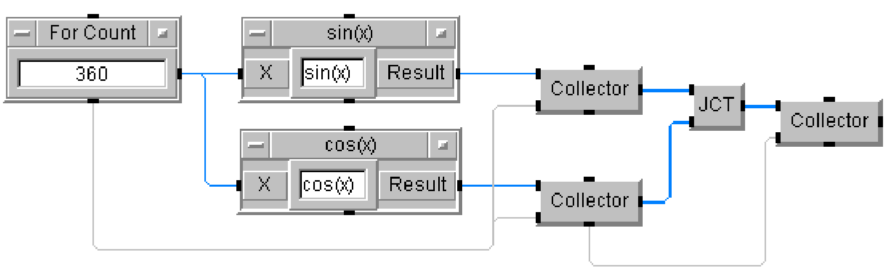


Figure 9-4. Generating an Array Using Individual Objects

Converting the logic contained in this series of objects to a mathematical operation, results in the expression shown in the `Formula` object in Figure 9-5. This technique does the same calculations in less time:

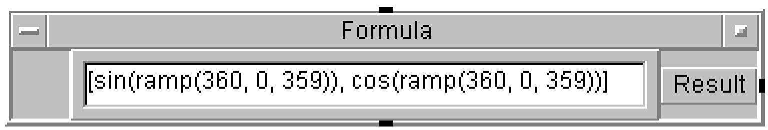


Figure 9-5. Generating an Array Using a Mathematical Expression

Though this technique is much faster, it is not obvious that it does the same calculations. Here is an explanation of the expression's operation:

1. The `ramp()` function generates an array of 360 values, increasing from 0 through 359.

2. The two ramp functions generate identical arrays, each operated on by the `sin()` and `cos()` trigonometry functions. Most VEE functions that normally accept and return a scalar value can accept and return an array as a parameter.

3. Though the `Formula` object actually contains two formulas, their outputs are converted to an array format because they are contained in the square brackets.

This technique may not be the best choice for all programs. The sine and cosine operations are done on the entire arrays produced by the `ramp()` functions, not on each value as it is generated. If your program must operate on each value as it is generated, use a loop structure instead of an expression that operates on the entire array.

Accessing Arrays in Expressions

Arrays in expressions can be used just like scalars. Refer to them by their name. Array constants can be entered directly into an expression (such as `[1,2,3]`). VEE requires that you insert commas between array elements.

Note

Array indices are 0-based. The indices start with zero and continue to *n*-1, where *n* is the number of elements in that particular dimension.

You can use expressions to access portions of an array. Once you have specified the sub-array in the expression, you can output the sub-array or use it in further expression calculations.

You can access only contiguous sub-arrays from each array. To access sub-arrays, you must specify a parameter for each dimension in the array. Use the following characters to specify array parameters:

- A comma “,” separates array dimensions. Each sub-array operation must have exactly one specification for each array dimension.
- A colon “:” specifies a range of elements from one of the array dimensions.
- An asterisk “*” is a wildcard to specify all elements from that particular array dimension.

Note

Waveform time spans, spectrum frequency spans, and array mappings are adjusted according to the number of points in the sub-array. For example, if you have a 256 point waveform (`WF`) and ask for `WF[0:127]`, you will get the first half of the waveform and a time span that is half the old span.

Examples: Values Returned from Array

The following expressions show values returned where `A` is a one-dimensional array (Array 1D) ten elements long.

- `A[1]` accesses the second element in `A` and outputs a scalar.
- `A[0:5]` returns a one-dimensional sub-array that contains the first six elements of `A`.
- `A[1:1]` returns a one-dimensional sub-array that contains one element, which is the second element of `A`. Note the difference between this and the first example, `A[1]`.
- `A[2:*]` returns a one-dimensional sub-array that contains the third through the tenth elements of `A`.
- `A` or `A[*]` returns the entire array `A`.

- A[1,2] returns an error because it specifies parameters for a two-dimensional array.

B is a 5x5 matrix (an Array 2D).

- B[*] returns an error because it specifies only one parameter and B is a two-dimensional array.
- B[1,2] returns a scalar value from the second row, third element.
- B[1,*] returns all of row one as an Array 1D.
- B[1,1:*] returns all of row one, except for the first element, as an Array 1D.
- B[4,1:4] returns an Array 1D that contains four elements: the second through fifth values from row 4.
- B[5,5] returns an error because arrays are zero-based. The array can only be accessed through B[4,4].
- B[1 1] returns an error because a comma must separate the dimension specifiers.

Building Arrays in Expressions

You can build an array from elements of other arrays or sub-arrays. Each element in the expression must specify the same number of dimensions and contain the same number of values in each dimension.

For example, the following expressions show values returned where A is a one-dimensional array (Array 1D) ten elements long and B is a 5x5 matrix.

- [1,2,3] returns a three-element Real Array 1D that contains the values 1, 2, and 3.
- [A[0], A[5:7], A[9]] causes an error because both scalar and Array 1D elements are specified.
- [A[0:4], B[0,*]] returns a ten element Array 2D (of size 2 by 5) that contains the first five elements from A as the first row and the first row from B as the second row.

- `[A[0], A[1], B[2,3], A[5]]` returns a four element Array 1D that contains the first and second element of `A`, the element from the third row and fourth column of `B` and the sixth element of `A`.

Performing Array Math Operations

Math operations on arrays uses another set of simple rules. Elementary scalar arithmetic operations on arrays simply perform their operations on each element in the array:

- A*2 multiplies each element in the array by 2.
- A-4 subtracts 4 from each element in the array.
- Arrays used in functions, like `sin([1,2,3])`, have the `sin` function applied to every element of the array.

Math operations between two arrays that have the same size and dimensions perform the operation between corresponding elements of the arrays:

- A*B multiplies each element of the array A by the corresponding element of array B. This does not perform a “matrix multiply”, which is a relatively complicated multiplication of rows times columns and summation that results in a scalar. VEE has a built-in matrix function to do that, called `matMultiply(A,B)`.

Basic Array Operations

VEE has a very flexible scheme for accessing and manipulating arrays. For a review of extracting portions of an array and performing simple array math operations see “Using Variables in Expressions” on page 274 and “Performing Array Math Operations” on page 292.

Array Functions Operations

Most elementary math functions in VEE, such as `log()`, `sin()` and `cos()` can accept an array as a parameter and return an array. Some specialized functions are handy for performing array math and manipulations. Other functions are not useful with arrays. Functions that are useful in array operations include:

- `ramp()` can be used to generate “loop” counts.
- `concat()` concatenates two arrays and returns a one-dimensional array.

- `totSize()` gives total number of elements in an array.
- `signof()` detects a value's sign (-1 if < 0, 0 if = 0, 1 if > 0).
- `abs(x)` sets the absolute value.
- `rotate()` rotates elements in array.
- `sum()` sums all elements in an array.
- `sort()` sorts an array.
- `randomize()` generates array of random numbers.
- `min()` finds the minimum value of a data set.
- `max()` finds the maximum value of a data set.
- `clipUpper()` clips below the maximum given value.
- `clipLower()` clips above the minimum given value.

A useful feature in `Formula` objects is the ability to define expressions as arrays. Notice that you must insert commas between array elements. The following expression generates an array containing the double, reciprocal, square, and natural log of the input named B:

```
[2*B, 1/B, B*B, log(B)]
```

The following examples show how to manipulate arrays using expressions in a `Formula` object. Just connect an object containing the array to the `Formula` object's input pin.

Changing Values in an Array

You can take an existing array and perform math calculations on selected elements using an assignment expression. The example in Figure 9-6 changes the values in one row of a two-dimensional array. The expression in the `Formula` object performs this task on the input array A=[1,2,3,4], where row 0 contains 1 and 2 and row 1 contains 3 and 4.

The expression multiplies the elements in the second row by 4, then assigns the results to array A. Notice that the `Result` terminal outputs only the changed values and the A terminal outputs the entire array with the new values.

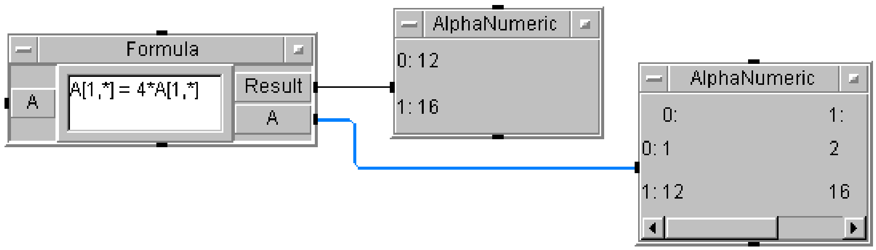


Figure 9-6. Using an Assignment Expression to Change Array Values

Splitting a Large Array

You can display the elements of a single 2048-element array as 16 sets of 128 elements each. While the problem focuses on the display of the data, rather than its generation, it may help to approach a solution that involves breaking up the array.

The program in Figure 9-7 shows how to break up the array in order to achieve the display goal.

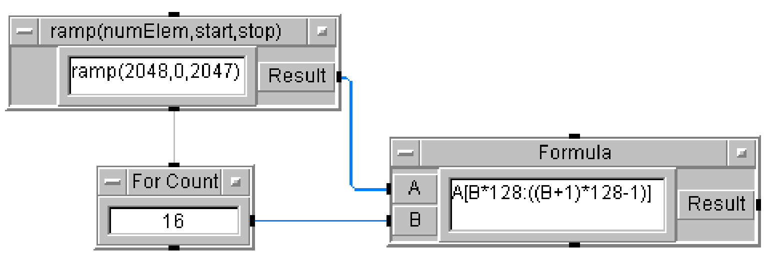


Figure 9-7. Reorganizing Values in a Large Array Using an Expression

The `ramp()` function simply generates a 2048-element array with values from 0 to 2047 for test purposes. The `For Count` object ticks off each of the 16 individual arrays to be generated, while the `Formula` box selects the appropriate sub-array using indexes generated from the count:

```
A[0:127], A[128:255], A[256:384], ..., A[1920:2047]
```

Of course, this assumes a fixed array size, number of subarrays and size of subarrays. An error occurs if any of these are mismatched.

Combining Arrays

The next example shows how to concatenate multidimensional arrays. The `concat(x,y)` function is useful for this task when used in a `Formula` object, even though it can generate only a one-dimensional output. However, it will work only if the number of rows or columns is fixed (a constraint that is usually met in practice). The `Formula` object in Figure 9-8 contains an expression that concatenates a pair of arrays that have two rows:

Figure 9-8. Combining Two Arrays Using an Expression

The `concat()` function concatenates two arrays and produces a one-dimensional array. The expression strips out the rows of each of the arrays, concatenates them and then joins them back together into a two-dimensional array with two rows containing the combined number of elements in each row.

Multiplying a Vector by a Matrix

The following example shows how to multiply a vector

```
[ X1, X2, X3, X4 ]
```

times a matrix

```
[[ y11, y12, y13, y14 ],
 [ y21, y22, y23, y24 ],
            ...
 [ y51, y52, y53, y54 ]]
```

to get the result

```
[[ X1*y11, X2*y12, X3*y13, X4*y14 ],
 [ X1*y21, X2*y22, X3*y23, X4*y24 ],
                   ...
 [ X1*y51, X2*y52, X3*y53, X4*y54 ]]
```

VEE can multiply a scalar times a vector and can perform matrix multiplication. A scalar multiplication multiplies every element in a matrix by a scalar to give a result matrix of the same size as the original. A matrix multiplication is an operation between an MxN matrix and an NxM matrix that yields a scalar. The required operation for this example does not match either case.

The operation here is effectively a scalar multiplication of each row of the matrix by each element of the vector. The implementation uses array manipulation techniques. Consider a data set consisting of a vector V of the form

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

and a matrix M of the form

```
[ [1,    2,    3,    4,    5,    6,    7,    8    ],
  [10,   20,   30,   40,   50,   60,   70,   80   ],
  [100,  200,  300,  400,  500,  600,  700,  800  ],
  [1000,2000, 3000, 4000, 5000, 6000, 7000, 8000  ]  ]
```

The desired result is

```
[ [1,    2,    3,    4,    5,    6,    7,    8    ],
  [20,   40,   60,   80,   100,  120,  140,  160  ],
  [300,  600,  900,  12K,  15K,  18K,  21K,  24K  ],
  [4000,8000, 12K,  16K,  20K,  24K,  28K,  32K   ]  ]
```

The expression in the `Formula` object in Figure 9-9 does the multiplication. The matrix array is connected to terminal `M` and the vector array is connected to terminal `V`. Tests indicate that this is only about 50% slower than a scalar multiplication of the same array.

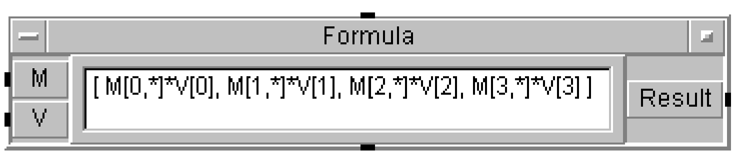


Figure 9-9. Multiplying a Vector Array by a Matrix Array

Inserting Elements into an Array

Figure 9-10 shows an expression that inserts one or more data elements into an existing array. The inputs are

`A` is the Index Value
`B` is the New Data
`C` is the Original Array

The revised array is output on `Result`.

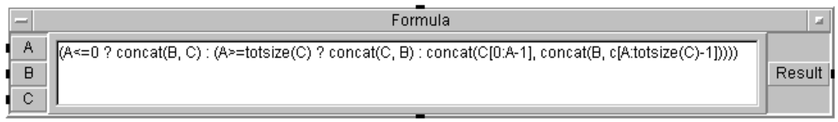


Figure 9-10. An Expression that Inserts Elements into an Existing Array

The Index Value on `A` indicates what the starting index of the new data should be. If `A` is 0 or less, the New Data on `B` is concatenated to the beginning of the Original Array on `C`.

If `A` is greater than or equal to the length of the Original Array, the New Data is concatenated to the end of the Original Array. If `A` is some value in between, the Original Array is broken into segments around the Index Value and the New Data is spliced into it.

The example in Figure 9-11 is similar to the previous one. It builds a data queue with array operations. A queue is essentially an array of fixed length, where new elements are added at one end and numbers shift down to the other end, where numbers fall off and are lost.

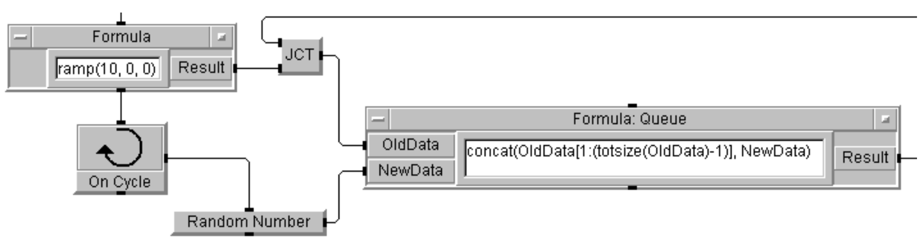


Figure 9-11. Using a Loop to Insert Elements into an Existing Array

The `ramp()` expression allocates an initial empty array of ten elements to act as a queue. The `On Cycle` loop (cycle is set to 1 second) begins and sends random numbers to the head of the queue every second. The `Formula: Queue` takes the last nine elements of the `OldData` input and concatenates it with a new random number on the `NewData` input. The array output on `Result` is fed back as the next set of `OldData` and sent to the next program segment. A new random number is sent to `NewData` on the next cycle.

Converting a Vector to a Matrix

There are occasions when you can get the results you need by using transaction objects in place of array functions. The example in Figure 9-12 uses a `From String` object to convert a vector (one-dimensional array) into a matrix (two-dimensional array). Run the vector through the `From String` and specify the array format you like as an output as shown in the next program. The conversion between Real and Text data types is automatic in VEE.

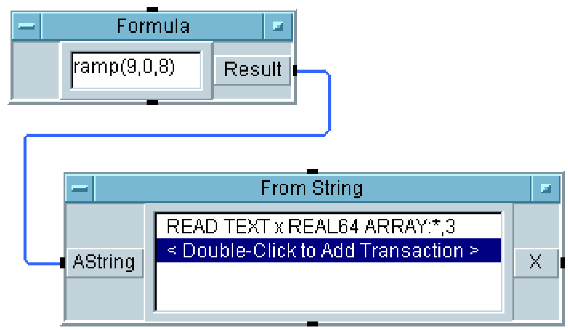


Figure 9-12. Converting a One-Dimension Array to Two Dimensions

For example, let us convert a one-dimensional 9-element array into a 3x3 two-dimensional array:

The vector output from the `ramp()` function is

```
0,1,2,3,4,5,6,7,8
```

The `From String` transaction converts it to the row-ordered matrix

```
0,1,2
3,4,5
6,7,8
```

If you prefer a column-ordered matrix, use the `transpose()` (transpose matrix) function to get the following result

```
0,3,6
1,4,7
2,5,8
```

Another way to convert a vector to a matrix is with the `Formula` build array syntax. You know that the syntax [1,2,3] generates a one-dimensional array with three elements. Similarly, if `a` is a one-dimensional array 10-long, the syntax [a] generates a 1x10 two-dimensional array. Again `transpose()` can be used if you want a 10x1 matrix.

Advanced Array Operations

This section shows some advanced array operations, including those involving comparisons on entire arrays of data.

Combining Disparate Elements into One Array

The program in Figure 9-13 shows the method required to take several data sets from a device and get a resulting data set that consists of the maximum values from all the individual data sets:

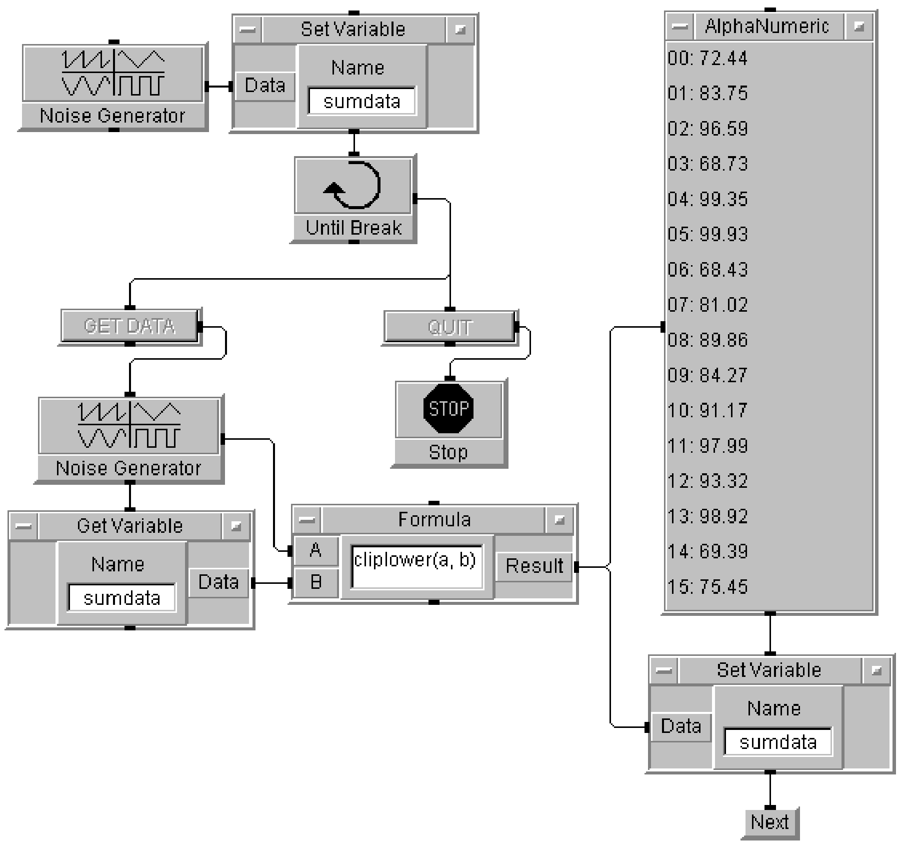


Figure 9-13. Collecting Maximum Values from Many Arrays

This program simulates input data by using a `Noise Generator`. It gets an initial data set and puts it in the global variable `sumdata`, then enters a loop to obtain new data or quit the program.

Pressing the `Get Data` button gets a new waveform from a `Noise Generator`, then recovers the data in `sumdata` with a `Get Variable` object; these two waveforms are summed into a `Formula` object, which processes them and puts the result back into `sumdata` using a `Set Variable` object.

The `Formula` object accepts the new array data on pin `A` and the array `sumdata` on pin `B`. The expression `cliplower(a,b)` outputs a result array with the value of `A` if `A` > `B` and the value of `B` otherwise. If you use `clipUpper` instead, you would obtain minimum values.

Comparing Two Arrays

The example in Figure 9-14 compares two arrays of random numbers and determines how many numbers in the first array are greater than those in the second array. Comparisons between values typically use relational operators (such as `==`, != and <=) and the triadic operator (A < B ? C : D), but these will not solve this problem. Though they accept any data shape, they only return scalar results. You can still perform relational operations by other means that yield an array result.

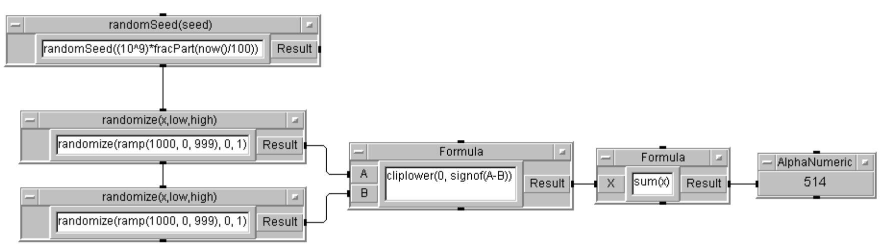


Figure 9-14. Comparing Values in Two Arrays

The `randomSeed()` function seeds the random-number generator with a seed that varies rapidly with time. This ensures that the data varies between different runs of the program. The two `randomize()` functions each generate an array of 1000 random numbers in the range 0 to 1. Finally, the expressions in the Formula objects perform the summation

```
sum(clipLower(0,signof(A-B)))
```

Here is how each part of the expression works:

1. `A-B` provides an array where values are positive if `A` > `B`, zero if `A == B`, and negative if `A` < `B`.
2. `signof(A-B)` converts positive numbers into 1, negative numbers into -1, and leaves 0 at 0.
3. `clipLower(0,signof(A-B))` strips all the -1 values out of the array, resulting in an array that is 1 if `A` > `B` and 0 otherwise.
4. `sum()` then adds up the 1s and outputs the number of comparisons where `A` > `B`.

Using Alternate Expressions

The previous section shows how conventional relational operators can be implemented for array operations using other techniques:

- `A == B:  (1-abs(signof(A-B)))`
- `A != B:  abs(signof(A-B))`
- `A >  B:  clipLower(0,signof(A-B))`
- `A <  B:  clipLower(0,-signof(A-B))`
- `A >= B:  (1-clipLower(0,-signof(A-B)))`
- `A <= B:  (1-clipLower(0,signof(A-B)))`

Notice how subtracting an array of 1s and 0s from 1 performs a `NOT` operation on the array. Similar techniques can be used for comparison with scalar values, rather than other arrays. You can also perform Boolean operations on the resulting arrays of 1s and 0s.

For example, suppose that A1 and A2 are two such arrays. The following logic operations hold:

- `NOT A1:     1 - A1`
- `A1 AND A2:  A1 * A2`
- `A1 OR  A2:  signof(A1 + A2)`
- `A1 XOR A2:  1 - abs(signof((A1 + A2) - 1))`

You can use the results of these computations to perform “masking” on arrays of the original values through multiplication. Those values that match to 0 are removed and those that match to 1 are retained.

Choosing Efficient Techniques

Applying the previous techniques could result in programs with `Formula` objects containing huge logic operations that are difficult to maintain. While the goal is to eliminate or reduce loops by replacing multiple objects with `Formula` objects, you could also use UserFunctions. A good understanding of array-manipulation techniques allows you to bypass complicated formal logic operations for more direct solutions.

The following example shows the choices you can make in array manipulation. Suppose an array of 8-bit unsigned data received from an I/O device is converted by VEE into 32-bit signed integer data and you want to get the real values back. You can do this in a single expression by adding 256 to each value of the return array:

```
(A + (clipLower(-1,clipUpper(0,A))) * (-256))
```

This expression performs the following operations:

1. `clipUpper(0,A)` converts all positive values to 0.

2. `clipLower(-1,clipUpper(0,A))` converts all negative values to -1 (recall that the inputs are integers, not reals). This creates an array that has -1 for each negative value and 0 otherwise.

3. `clipLower(-1,clipUpper(0,A)) * (-256)` multiplies that array by -256 creating an array that has 256 for each negative value and 0 otherwise. This array is then added to the original array to offset all the negative values to their “true” positive value.

Another solution could have used relational operations as shown previously, but it would have been much more complicated than this direct solution.

A very useful object for array computations is the `Comparator`, which allows extraction of array elements that meet specific criteria. Suppose you want to determine the transitions in the following data stream:

```
0 0 0 0 1 1 0 0 1 1 0
```

The solution shown in Figure 9-15 is the easiest way to find the indexes of the array elements where the value makes a transition from 0 to 1 and the reverse.

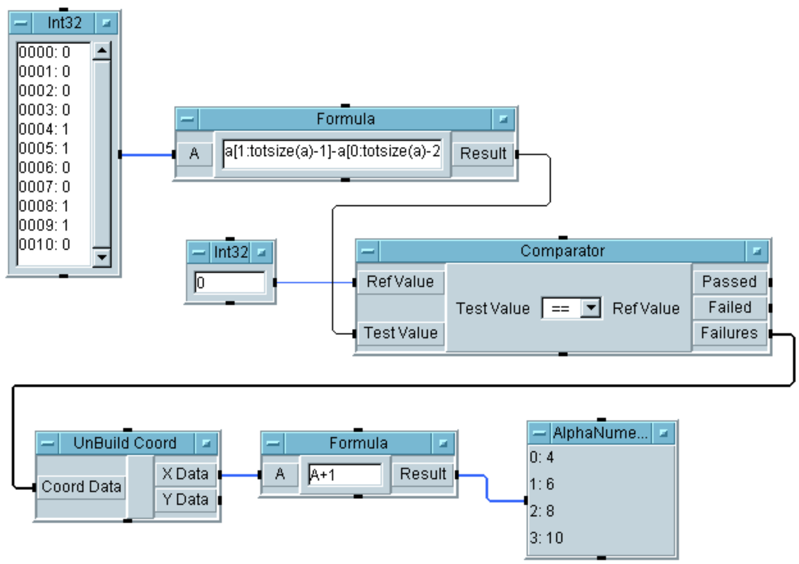


Figure 9-15. Finding Transition Points in an Array of Values

Although the program contains a `Comparator`, the key is the `Formula` object containing the expression

```
A[1:(totSize(A)-1)] - A[0:(totSize(A)-2)]
```

To see how this works, add array indexes to the data stream:

```
B0:0 1:0 2:0 3:0 4:1 5:1 6:0 7:0 8:1 9:1 10:0
```

The array indexes are marked where a transition occurs. The expression above performs a subtraction of the input array from itself, staggered by one index, to yield the following new array:

```
0:0   1:0   2:0   3:0   4:1   5:1   6:0   7:0   8:1   9:1   10:0
  -   0:0   1:0   2:0   3:0   4:1   5:1   6:0   7:0   8:1    9:1   10:0
-----------------------------------------------------------------------
      0:0   1:0   2:0   3:1   4:0 5:-1    6:0   7:1   8:0   9:-1
```

The `Comparator` checks the Result array to see which elements are equal to 0. Array elements that fail the test are the indexes and are returned on the `Failures` terminal as an array of X-Y coordinates giving the index and value of the failure. To retrieve only the index values, the `Unbuild Coord` object separates the `X` and `Y` values. Then the `X` index values are incremented by 1 to eliminate the effects of the staggered subtraction.

The data obtained in the subtraction not only indicates the index of the transaction, but its direction: 1 for a positive transition, -1 for a negative. This operation is basically a difference-equation approach to performing a differentiation.

10

Variables

Variables

This chapter describes variables in VEE, including;

- About Variables
- Using Variables

Note For information about using variables with ActiveX automation objects and controls, see Chapter 13, “Using ActiveX Automation Objects and Controls”.

About Variables

There are two types of variables in VEE: undeclared and declared. Both types of variables can contain any data type, including complex data types such as waveforms and records. They can also be any data shape, including scalars and arrays.

About Undeclared Variables

Undeclared variables are the easiest to use but execute slower and do not allow scoping (described in About Declared Variables, below). Undeclared variables include the following:

- *Global variables that can be used anywhere in the program.* They are created with the `Set Variable` object. They are deleted before the program is run if the `Delete Variables at PreRun` property is set. Global variables must be created before they can be accessed via the `Get Variable` object or used in expressions, or else your program will generate an error.

  Undeclared global variables are useful if you do not know what data type or shape your values will be or if the values may change type or shape. If you want a scoped variable (i.e., local), use declared variables (see “About Declared Variables” on page 308).

- *Temporary variables that are used only in `Formula` objects*. You can create a temporary variable, such as `tmp`, in a `Formula` by adding an output terminal. For example, to swap the values input in a `Formula` object’s terminals `a` and `b`, use the temporary variable `tmp`. The expression would look like `tmp=a, a=b, b=tmp.` For more information about temporary variables, see `Assignment` in *VEE Online Help* under `Reference` ⇒ `Math Functions and Operators`.

- Terminal names that are used as variables within objects (such as in transaction or `Formula` objects).

About Declared Variables

Declared variables are defined before they are used. They have the additional feature of scoping and allow VEE to run faster because the data type and shape are known before run time. However, if you attempt to set a declared variable with values that are different from the data type or shape of the values set in the declaration, the program will error.

To declare a variable, use the `Data` ⇒ `Variable` ⇒ `Declare Variable` object. When placed in a context, it declares the variable before any of the other objects execute. When the variable has been declared, it has no value until it is set via a `Set Variable` or a `Formula` object.

The scope of a declared variable must be specified in the `Declare Variable` object. The scopings are as follows:

- `Global` - The variable can be used anywhere in the program.
- `Local to Context` - The variable can only be used in a single UserObject or UserFunction, or in Main. This variable can be used in the context that the `Declare Variable` object is in and in UserObjects nested inside the context. The variable cannot be used in UserFunctions called from the context.
- `Local to Library` - The variable can only be used within the library of UserFunctions where the `Declare Variable` object is used. `Declare Variable` must be located in one of the UserFunctions.

You cannot define multiple variables with the same name and scope. If this happens, you will get an error.

About Variables Naming

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

To retrieve the value of the variable, you must use the name that you specified when the variable was declared or set.

When `Execution Mode` in `Default Preferences` is set to VEE 5 mode or higher, some names must be unique. See “Using VEE Execution Modes” on page 10 for information about using variable names in `VEE 5` mode. When `Execution Mode` is set to `VEE 4` or `VEE 3` mode the question of precedence arises when you have named a variable the same name as another variable. The order of precedence (from highest to lowest) is:

1. Input terminal name (such as in a `Formula` or a transaction object)
2. Temporary variable (as in a `Formula` object)
3. `Local to Context` declared variable
4. `Local to Library` declared variable
5. `Global` declared variable
6. Global undeclared variable

If two variables with the same name are in an object, there is a conflict. The variable with the highest precedence is used.

Using Variables

This section gives guidelines for using variables in VEE, including setting initial values, accessing variable values, deleting variables, and using variables in libraries.

Setting Initial Values

You must have set initial values before accessing any variables or VEE generates an error. See Figure 10-1 for a variable example.

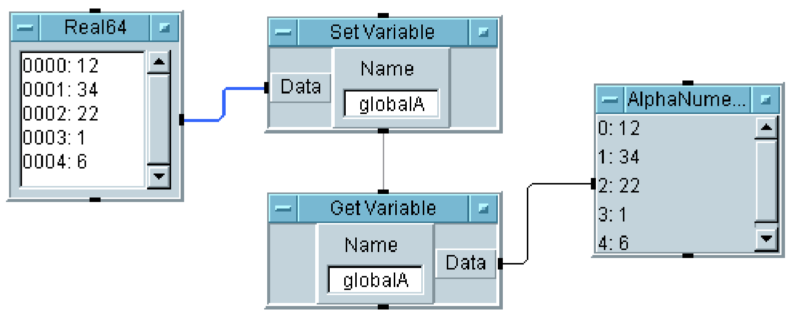


Figure 10-1. A Variable Example

The `Set Variable` must set the global variable before the `Get Variable` attempts to retrieve it. To ensure this, the sequence output pin of the `Set Variable` object is connected to the sequence input pin of the `Get Variable` object. If this is not done, the `Get Variable` may try to access a non-existent global variable and an error will occur.

If the property `Delete Variables at PreRun` is not set, you may not receive an error and may receive old data instead.

When declared variables are created, they are not initialized and must have a value set in them before they are accessed via the `Get Variable` object or used in expressions. If they do not, your program will generate an error. You set values via the `Set Variable` object or by using the `Formula` object.

If the variable is an array or a record, when using the `Formula` object you must set the values of the entire array or record before trying to access any of the elements. The example in Figure 10-2 shows two different ways to initialize values from a `Formula` object.

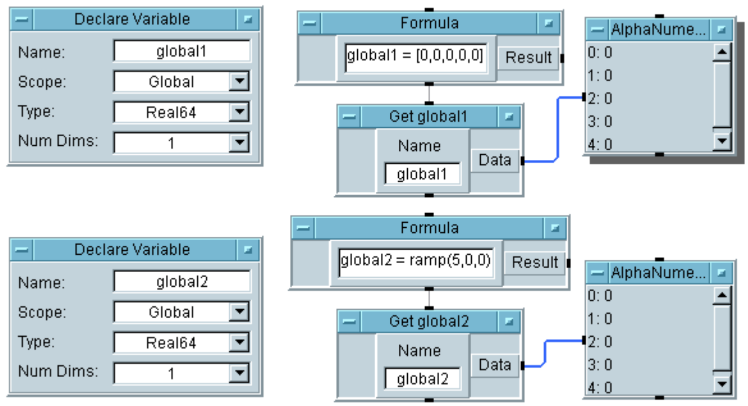


Figure 10-2. Setting Array Values

Accessing Variable Values

Once you have named a variable, you can access its value as many times as you want in your program. You can use several methods to retrieve the variable value. In the example in Figure 10-3, the value stored in the global variable `globalA` is retrieved once with a `Get Variable` 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:

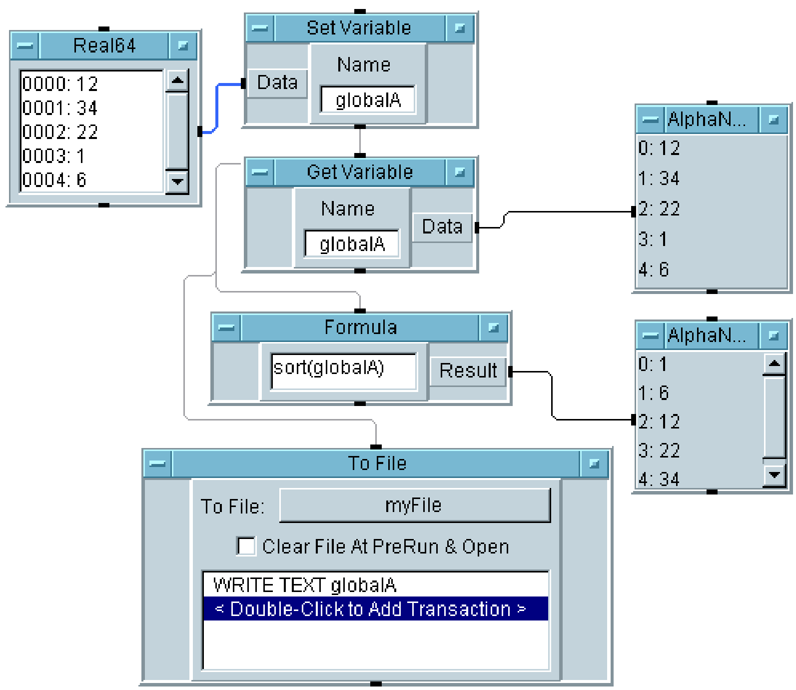


Figure 10-3. Accessing a Variable Multiple Ways

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.

Deleting Variables

To improve memory usage, use the `Delete Variable` object to free up memory space when a variable is no longer needed. When undeclared variables are deleted, their values and definitions are both deleted. When declared variables are deleted, the values are reset to uninitialized values but the definition remains.

When you set `Delete Variable` to `By Name`, the closest variable of the specified name is deleted. The closest variable is defined by the precedence.

When you set `Delete Variable` to `All`, all declared and undeclared variables in all scopings are affected, even the variables that are in imported libraries. Declared variables are uninitialized and undeclared variables are deleted (as described previously).

Deleting all Variables may not cause all memory to be freed or all ActiveX Automation pointers to be released. See “Deleting Automation Objects” on page 368 for more information.

To delete all variables before each execution of the program, select `File ⇒ Default Preferences` and click the check box `Delete Variables at PreRun`. If this check box is not selected, the values of all variables will remain and the declarations of declared variables will not reinitialize the values

11

Using Records and DataSets

Using Records and DataSets

This chapter introduces two concepts: the Record data type and the DataSet. A data set is a collection of Record containers saved into a file for later retrieval. The chapter contents are:

- Using Records
- Using DataSets

Using Records

This section gives guidelines for using objects to create and manipulate records, It includes understanding record containers, accessing records, programmatically building records, and editing record fields.

Understanding Record Containers

There are several VEE objects that allow you to create and manipulate records, including `Record`, `Build Record`, `UnBuild Record`, `Merge Record`, `SubRecord`, `Set Field`, and `Get Field`. All these objects are located in the `Data` menu.

A container of the Record data type has named fields which represent data. You can have as many named fields as you like in a record. Each field can contain another record, a scalar, or an array.

The `Record` object allows you to create records by manually entering a value for each field. Just configure the `Record` object as a scalar (array elements = 0) or as an array (array elements = non-zero) with the `Properties` dialog box, accessed from the object menu.

The `Record` object in Figure 11-1 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` object allows you to step through the record from one array element to the next by using the `First`, `Prev`, `Next`, and `Last` buttons. You can edit each field as you go.

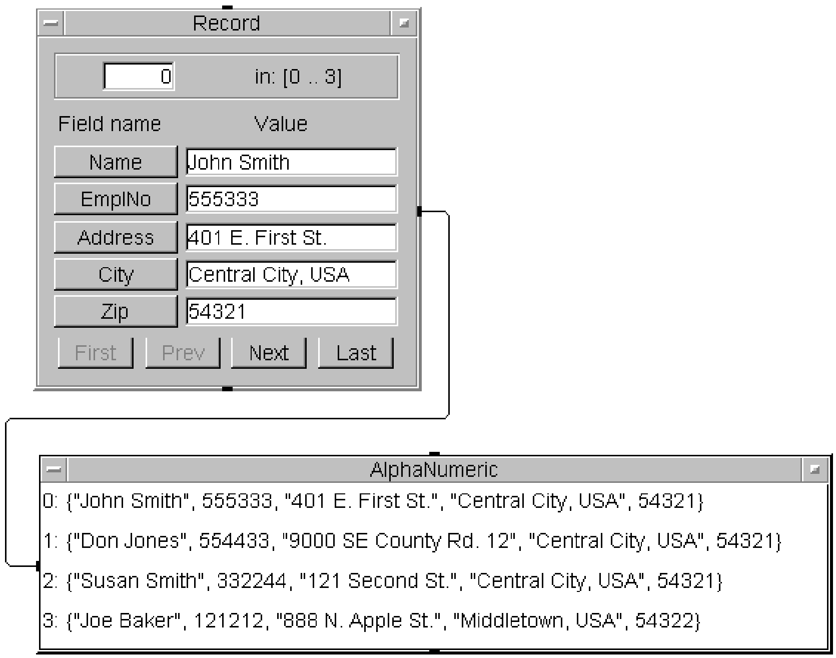


Figure 11-1. Example: A Record Container

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 (0 through 3) each consisting of five record fields enclosed in braces ("`{}`").

Accessing Records

The example programs in Figure 11-2 and Figure 11-3 show one way to access a record and extract individual fields.

Use the `Get Field` object to extract an individual field from the record. `Get Field` is located under `Data` ⇒ `Access Record`. For the example in Figure 11-2, `Get Field` objects are used to extract the entire `Name` and `EmplNo` fields: The `Get Field` object is a `Formula` object with the default expression `rec.field`.

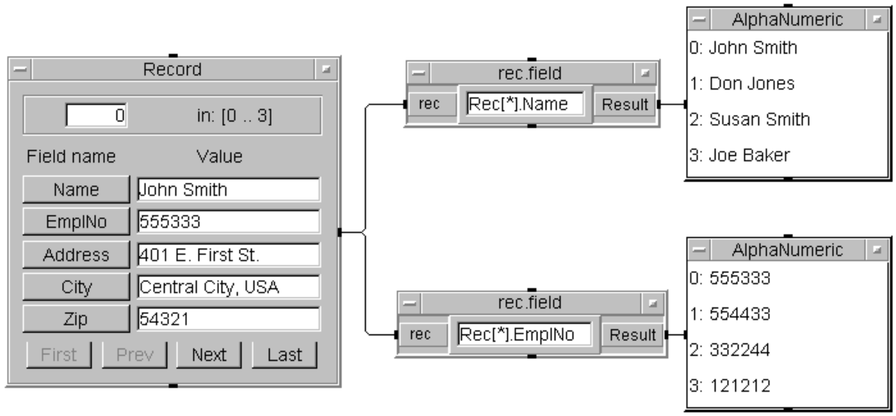


Figure 11-2. Retrieving Record Fields with Get Field

Use the "dot" syntax to access individual fields, for example: `Rec[*].Name` and `Rec[*].EmplNo`. This syntax is described in `Mathematically Processing Data` ⇒ `General Concepts` under `Tell Me About` in *VEE Help*.

`Rec[*].Name` means "get the `Name` field from all elements of the record array 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.

Use the syntax `Rec[1].Name` and `Rec[1].EmplNo` to obtain just the second element ("element `1`") of each field, as shown in Figure 11-3.

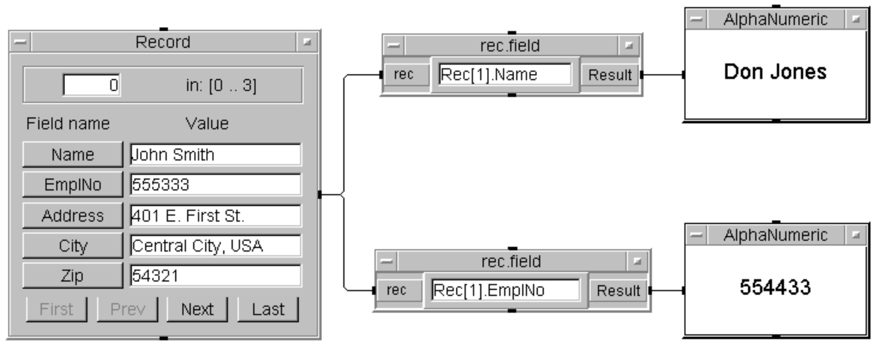


Figure 11-3. Using Array Syntax in Get Field

To retrieve several or all fields from a record use the `UnBuild Record` object, as shown in Figure 11-4.

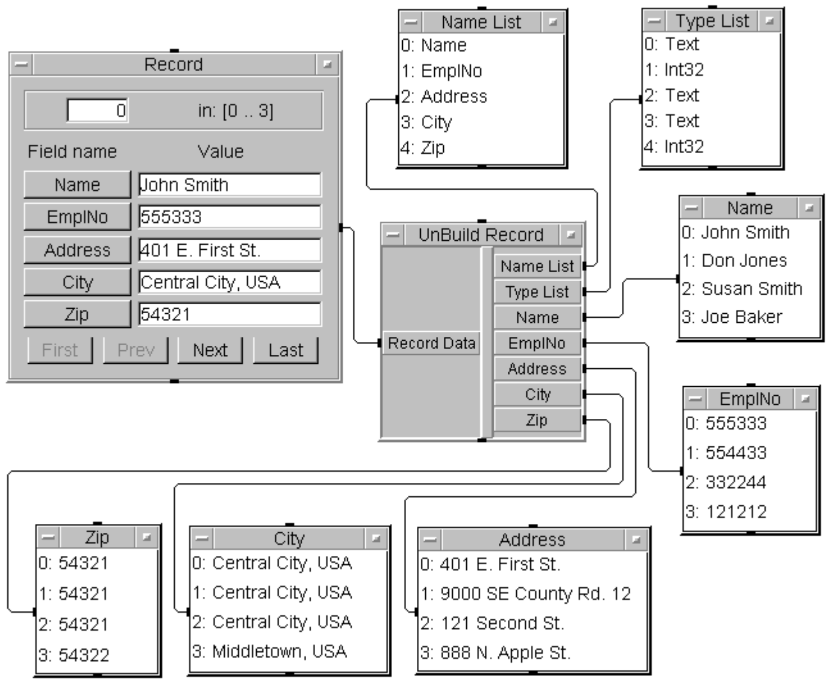


Figure 11-4. Retrieving Record Fields with UnBuild Record

The `UnBuild Record` object allows you to add outputs for every field in the record and provides `Name List` and `Type List` outputs. These outputs list the name and type of each field in the record.

The program is saved in the file `manual38.vee` in your `examples` directory.

Note

Data cannot be automatically converted to and from the Record data type. For example, to send Record data into a Real input terminal you must extract the field from the Record with the `Unbuild Record` object or use `Get Field` with the `Rec.A` syntax as described previously.

Programmatically Building Records

The `Record` object is useful for creating and editing simple records. However, it is cumbersome for creating large records. You may also want to create a record from existing data. In such cases, use `Build Record` to build a record.

When you build a record from individual data components with `Build Record`, you must 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 example in Figure 11-5 shows the difference between `Scalar` and `Array 1D` in the output record built from two input arrays:

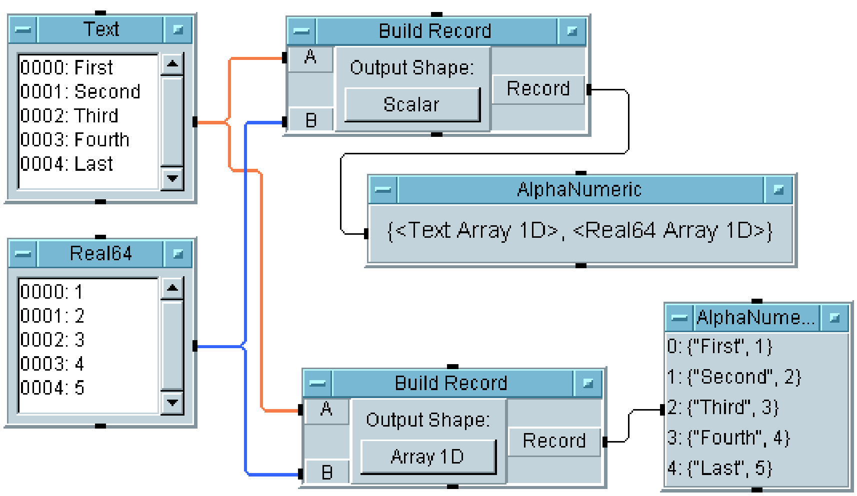


Figure 11-5. The Effect of Output Shape in Build Record

In Figure 11-5, 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 in the example in Figure 11-6.

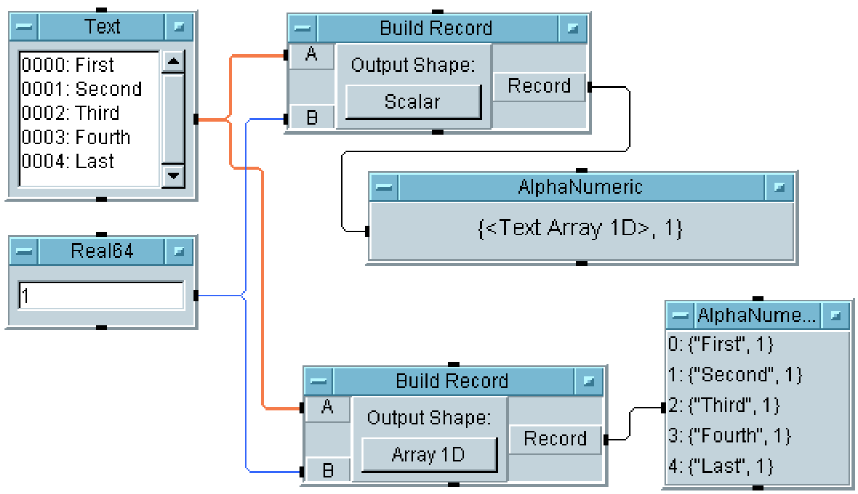


Figure 11-6. Mixing Scalar and Array Input Data

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

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 and the right-hand expression specifies the new data.

The right-hand expression is evaluated and the record field specified by the left-hand expression is assigned that value. See Figure 11-7 for an example.

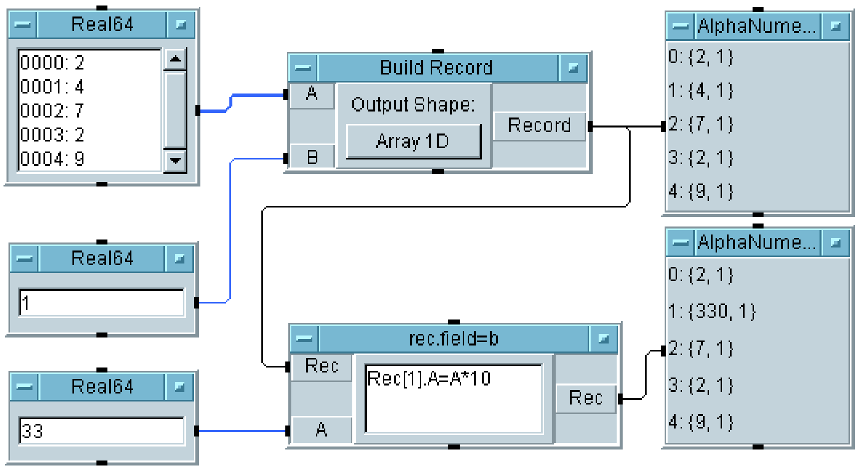


Figure 11-7. Using Set Field to Edit a Record

In this example, a five-element record array is built with `Build Record`. The `Set Field` object (titled `rec.field = b`) specifies that the field `Rec[1].A` (the `A` field of record element `1`) is to be assigned the value `A*10`.

There is a potential for confusion 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. This exemplifies the need for good names for variables and Record fields.

The variable `A` has the value 33, so `A*10` is evaluated as 330, which is assigned to `Rec[1].A`, as shown in Figure 11-7. Note that none of the other values of the record have changed.

`Set Field` is a `Formula` object. See `Assignment` in `Math Functions and Operators` under `Reference` in *VEE Online Help* for more information.

Using DataSets

VEE data (including waveforms) can be built into records and later retrieved. You can also store records into a file, called a DataSet. The `To DataSet` and `From DataSet` objects allow you to store and retrieve records to and from DataSets. They are located in the `I/O` menu.

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). See Figure 11-8 for an example.

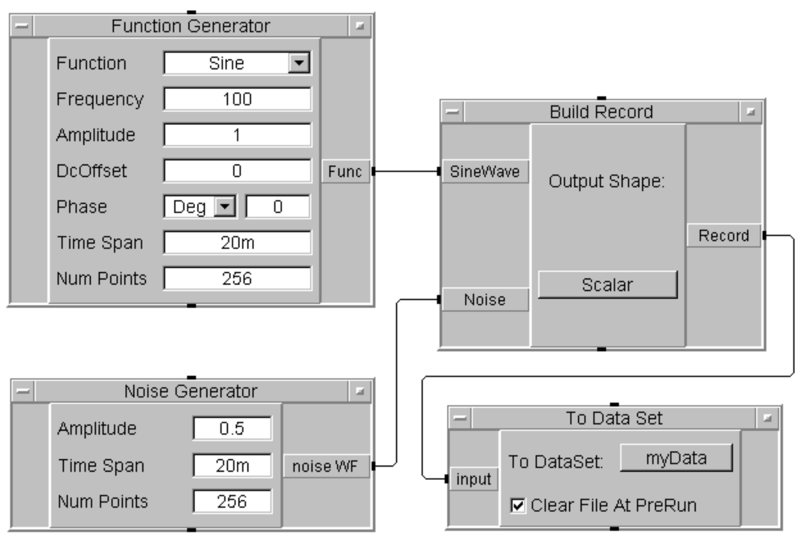


Figure 11-8. Using To DataSet to Save a Record

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`. `Clear File at PreRun` is checked so any data previously stored in `myData` is cleared.

Once the data has been saved as a DataSet, use `From DataSet` to retrieve the record, which can then be unbuilt if desired. The program in Figure 11-9 shows this technique.

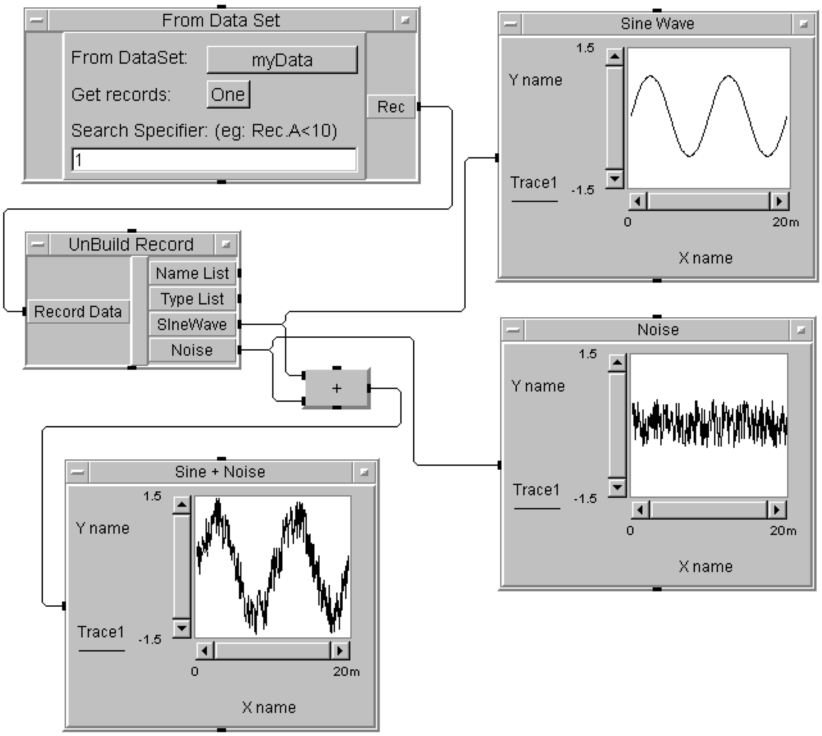


Figure 11-9. Using From DataSet to Retrieve a Record

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 `examples` directory.

12

User-Defined Functions/Libraries

User-Defined Functions/Libraries

VEE provides 19 categories of built-in functions you can use in programs. When one of these built-in functions is not exactly right for your program, you can define your own function.

This chapter describes how to create custom functions with/using UserFunctions.

VEE OneLab supports two kinds of user-defined functions:

- UserFunctions
- Compiled Functions

This chapter describes UserFunctions and Compiled Functions, in the following sections:

- About UserFunctions
- About Compiled Functions

About UserFunctions

A UserFunction is specifically designed for creating a user-defined function. You create a UserFunction by selecting it from the `Device` menu or by converting existing objects or an existing UserObject into a UserFunction. This section describes how to create a UserFunction. The next section describes how to convert a UserObject into a UserFunction.

To create a UserFunction, click `Device` ⇒ `UserFunction`. An empty UserFunction window appears in the work area. Create your function by adding terminals and objects as needed. Change the name to whatever you want (spaces not allowed). See the *VEE User's Guide* or `How Do I` in *VEE Online Help* for additional details.

When the UserFunction is complete, you can iconify it or close it to get it out of the way of the rest of your program. You can call your UserFunction using a Call object in your program (`Device` ⇒ `Call`) or other objects identified below.

The advantage of creating a UserFunction over using a UserObject is that you can call a single UserFunction several times in your program. Thus, there is only one UserFunction to edit and maintain, rather than several instances of a UserObject.

When executed in `VEE 4`, or higher Execution Mode, a UserFunction will time-slice when called from `Call`, `Formula`, or `If/Then/Else` objects.

A UserFunction will not time-slice when called from a `To File`, `To String`, or similar object or if the Formula object's formula is supplied via a control pin.

Converting Between UserObjects and UserFunctions

To convert a UserObject into a UserFunction, select `Make UserFunction` from the UserObject's object menu. The UserObject window is replaced by a UserFunction window with the same input and output terminals. The UserObject object is replaced by a UserFunction `Call` object.

To reconvert the UserFunction back into a UserObject, select `Make UserObject` from the object menu of the UserFunction window. Any calls to the UserFunction remain (you will have to manually delete them), but the UserFunction is automatically converted into a UserObject.

Calling a UserFunction from an Expression

You can call a UserFunction from an expression in a `Formula` object or from any expression evaluated at run time, such as from a `ToFile` object. The program in Figure 12-1 demonstrates several ways to call a UserFunction.

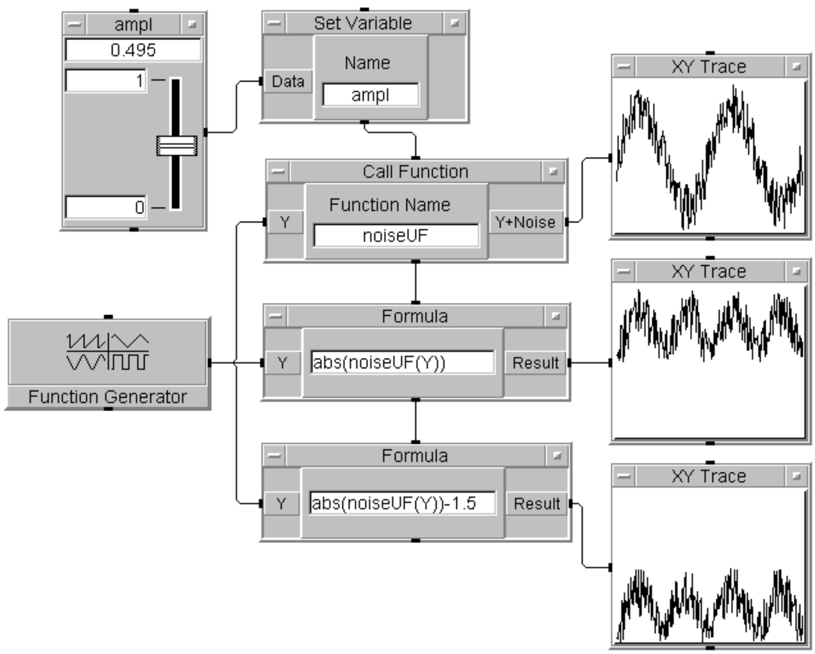


Figure 12-1. Calling a UserFunction from Expressions

In the program, the `Call` object calls the UserFunction `noiseUF` and returns a sine wave with an added noise component. The expression `abs(noiseUF(Y))` in the first `Formula` object returns the absolute value of the waveform returned by the UserFunction. Thus, the displayed noisy sine wave is rectified in the positive direction.

The expression `abs(noiseUF(Y))-1.5` in the second `Formula` object also calls the UserFunction but adds a negative dc offset to the waveform. The sequence pins are used to ensure correct propagation because the UserFunction uses the global variable.

This program is saved in the file `manual43.vee` in the `examples` directory.

About Compiled Functions

A Compiled Function is created by dynamically linking a library written in C, C++, FORTRAN, or Pascal, to the VEE process. A library of compiled functions is called a dynamic link library (DLL) in Microsoft Windows.

Creating a Compiled Function is considerably more difficult than creating a UserFunction. Once you have written a library of functions in C or another language, you will need to compile the functions into a DLL. You will also have to create a definition file that will provide VEE with information it needs to call your function.

Using a Compiled Function

To use a Compiled Function, you:

1. Write the external program.
2. Create the DLL and a definition file.
3. Import the library and call the function from VEE.
4. Delete the library from VEE memory when you are done.

The methods for importing a Compiled Function library and for calling the function are very similar to those for UserFunction libraries. The `Import Library` object attaches the DLL to the VEE process and parses the definition file declarations.

The definition file defines the type of data passed between the external library and VEE. (This file is discussed later in this section.) The Compiled Function can then be called with the `Call` object or from such objects as `Formula` and `If/Then/Else`.

Design Considerations for Compiled Functions

Using Compiled Functions, you can develop time-sensitive routines in another language and integrate them directly into your VEE program. You can also use Compiled Functions to keep proprietary routines secure.

Because Compiled Functions do not timeslice (i.e., they execute until they are done without interruption) they are only useful for specific purposes that are not otherwise available in VEE.

You can extend the capabilities of your VEE program by using Compiled Functions, but it adds complexity to the VEE process. The key design goals should be:

- Keep the purpose of the external routine highly focused on a specific task
- Use Compiled Functions only when the capability or performance you need is not available using a VEE UserFunction or an `Execute Program` escape to the operating system.

You can use any operating system facilities available in the program to be linked, including math routines, instrument I/O, etc. However, you cannot access any VEE internal functions from within the external program to be linked.

Although the use of Compiled Functions provides enhanced VEE capabilities, some problems can occur. A few key ones are:

- VEE cannot trap errors originating in the external routine. Because your external routine becomes part of the VEE process, any errors in that routine propagate back to VEE. A failure in the external routine may cause VEE to "hang" or otherwise fail. You need to be sure of what you want the external routine to do and provide for error checking in the routine. If your external routine exits so will 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 VEE does. You should configure the data input terminals of the `Call` 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. The routine also 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 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, etc.
- If your external routine performs an invalid operation, such as overwriting memory beyond the end of an array or dereferencing a null or bad pointer, this can cause VEE to exit or error with a General Protection Fault (MS Windows).
- If your external routine has arrays or char* parameters, the memory passed to these routines must be allocated in VEE. You should allocate this memory by doing the following:
  - For an array input, use an `Alloc Array` object of the appropriate type, and set the size appropriately.
  - For a string input, use a `Formula` object. Delete the data input terminal from the `Formula` object and enter an expression like `256*"a"`. This creates a string that is 256 characters long (plus a null byte) filled with `a`’s. Most VXI*plug&play* functions will not write more than 256 characters into a `Text` parameter. However, it is best to check the Help on each function panel that requires a `Text` input to be sure.

Importing and Calling a Compiled Function

You can import a DLL into your VEE program with the `Import Library` object, then call the Compiled Function with the `Call` object. The process is very much like importing a library of UserFunctions and calling the functions, as described at the beginning of this chapter.

To import a Compiled Function library, select `Compiled Function` in the `Library Type` field.

Just as for a UserFunction, 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. For a Compiled Function, there is a fourth field, which specifies the name of the `Definition File`, shown in Figure 12-2.

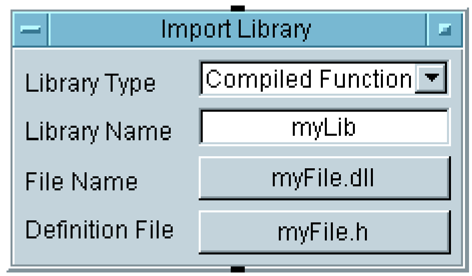


Figure 12-2. Using Import Library for Compiled Functions

The definition file defines the type of data passed between the external routine and VEE. It contains prototypes for the functions.

After importing the library with `Import Library`, you can call the Compiled Function by specifying the function name in the `Call` object. For example, the `Call` object in Figure 12-3 calls the Compiled Function named `myFunction`.

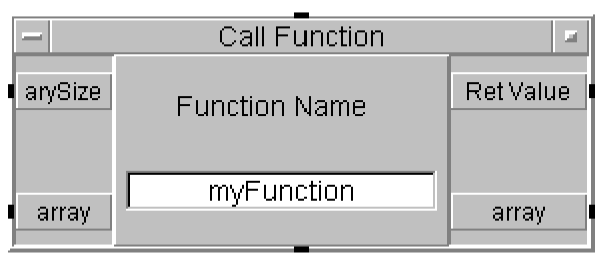


Figure 12-3. Using Call for Compiled Functions

Select the desired function using `Select Function` from the `Call` object menu or from the `Function & Object Browser` (under `Device` ⇒ `Function & Object Browser`), or type the name in the `Call` object.

If VEE recognizes the function, the input and output terminals of the `Call` object are configured automatically for the function. (The necessary information is supplied by the definition file.) You can reconfigure the `Call` input and output terminals by selecting `Configure Pinout` in the object menu.

VEE configures the `Call` object with the input terminals required by the function and with a `Ret Value` output terminal for the return value of the function. There also 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 `ToFile` transaction.

However, only the Compiled Function's return value (`Ret Value` in the `Call` object) can be obtained from within an expression. If you want to obtain other parameters from the function, you have to use the `Call` object.

The Definition File

The `Call` object or Formula expression determines the type of data it should pass to the function based on the contents of the definition file. The definition file defines the type of data the function returns, the function name, and the arguments the function accepts. The data has the following form:

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

Where:

- `<return type>` can be: `int`, `short`, `long`, `float`, `double`, `char*`, or `void`.
- `<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: `int`, `short`, `long`, `float`, `double`, `int*`, `char*`, `short*`, `long*`, `float*`, `double*`, `char**`, or `void`.
- `<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 recommended. 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 VEE Text data type)
- integers (`short`, `int`, `long`, corresponding to the VEE `Int16` and `Int32` data types)
- single and double precision floating point real numbers (`float` and `double` corresponding to the VEE `Real32` and `Real64` data types).

If you specify "pass by reference" for a parameter by preceding the parameter name with `*`, VEE will pass the address of the information to your function. If you specify "pass by value" for a parameter by leaving out the `*`, VEE will copy the value (rather than the address of the value) to your function. You will want to pass the data by reference if your external routine changes that data for propagation back to VEE. *All arrays must be passed by reference.*

Any parameter passed to a Compiled Function by reference is available as an output terminal on the `Call` object. 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.

VEE pushes 144 bytes on the stack. This allows up to 36 parameters to be passed by reference to a Compiled Function. Up to 36 long integer parameters or 18 double-precision floating-point parameters may be passed by value.

VEE allows both "enclosed" comments and "to-end-of-line" comments in the definition file.

"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

The following C function accepts a real array and adds `1` to each element in the array. The modified array is returned to VEE on the `Array` terminal, while the size of the array is returned on the `Ret Value` terminal. This function, once linked into VEE, becomes the Compiled Function called in the VEE program shown in Figure 12-4.

```
/*
  C code from manual49.c file
*/

#include <stdlib.h>

#ifdef WIN32
#  define DLLEXPORT __declspec(dllexport)
#else
#  define DLLEXPORT
#endif

/* The description will show up on the Program Explorer when you select
"Show Description" from the object menu and the Function Selection
dialog box in the small window on the bottom of the box.
*/
DLLEXPORT char myFunc_desc[] = "This function adds 1.0 to the array
passed in";

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

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

  return(arraySize);
}
```

The definition file for this function is as follows:

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

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

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

You must include any header files on which the routine depends. The library should link against any other system libraries needed to resolve the system functions it calls.

The example program uses the ANSI C function prototype. The function prototype declares the data types that VEE should pass into the function.

The array has been declared as a pointer variable. VEE will put the addresses of the information appearing on the `Call` data in terminals into this variable. The array size has been declared as a long integer. 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.

One variable in the C function (and correspondingly, one data input terminal in the `Call` 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 VEE. In languages like C, where there is no bounds checking, the result will be unpredictable, but intermittent data corruption is probable.

This example has passed a pointer to the array so it is necessary to dereference the data before the information can be used.

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

The program in Figure 12-4 calls the Compiled Function created from the example C program:

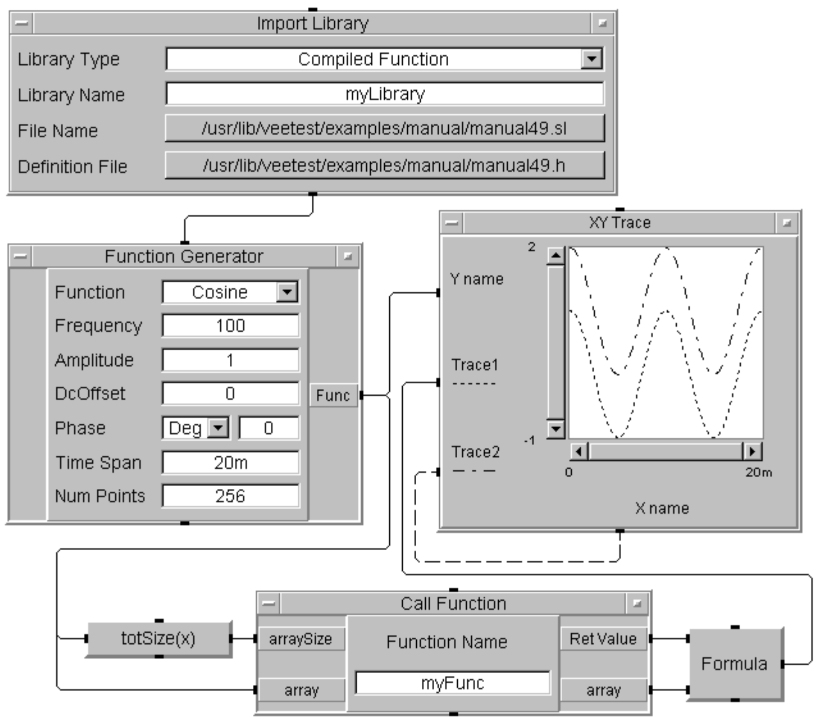


Figure 12-4. Program Calling a Compiled Function

The example in Figure 12-4 is saved in the file `manual49.vee` in the `examples` directory. The C file is saved as `manual49.c`, the definition file as `manual49.h`.

Creating a Dynamic Link Library (MS Windows)

VEE for Windows provides access to DLLs through the `Call` object and through formula objects.

Note This section describes how to call a DLL, not how to write a DLL. VEE Version 3.2 and greater only calls 32-bit DLLs, not 16-bit DLLs.

Creating the DLL

Create the DLL before writing the VEE program. Create the DLL as you would any other DLL, except that only a subset of C types are allowed. (See “Creating the Definition File” on page 341.)

Declaring DLL Functions. If you are using Microsoft Visual C++ Version 2.0 or greater, the function definition should be:

```
__declspec(dllexport) long myFunc (...);
```

This definition eliminates the need for a `.DEF` file to export the function from the DLL. Use the following command line to compile and link the DLL:

```
cl /DWIN32 $file.c /LD
```

`/LD` creates a DLL. Use `/Zi` to generate debug information.

The MS linker links to the C multi-threaded Runtime Library by default. If you use functions like `GetComputerName()`, you need to link against `Kernel32.lib`. The compile/link line would look like:

```
cl /DWIN32 file.c /LD /link Kernel32.lib
```

Declaring DLL Functions. To work with VEE, DLL functions can be declared as `__declspec(dllexport)` using Microsoft C++ version 2.0 or greater. This eliminates the need for a `.DEF` file. For example, a generic function could be created as follows:

```
__declspec(dllexport) long genericFunc(long a) {return (a*2); }
```

If you are not using Microsoft Visual C++, the `.DEF` file contains:

```
EXPORTS genericFunc
```

And the function definition looks like:

```
long genericFunc(long a);
```

Creating the Definition File. The definition file contains a list of prototypes of the imported functions. 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:

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

where:

- `<return type>` can be: `int`, `short`, `long`, `float`, `double`, `char*`, or `void`.
- `<function name>` can be a string consisting of an alpha character followed by alphanumeric characters, up to a total of 512 characters.
- `<modifier>` can be `_cdecl`, `_pascal`, or `_stdcall`.
- `<type>` can be: `int`, `short`, `long`, `float double`, `int*`, `char*`, `short*`, `long*`, `float*`, `double*`, `char**`, or `void`.
- `<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 recommended. If a parameter is to be passed by reference, the parameter name must be preceded by the indirection symbol (`*`).

For example:

Pass in four parameters, return a long:

```
long aFunc(double *,long param2,long *param3, char *);
```

No input parameters, return a double:

```
double aFunc();
```

Pass in a string, return a long:

```
long aFunc(char *aString);
```

Pass in an array of strings, return a long:

```
long aFunc(char **aString);
```

Parameter Limitations

A DLL function called from VEE pushes a maximum of 144 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 144-byte 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 36 longs, or 18 doubles, or 20 pointers and 8 doubles.

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 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 Object

Before using a DLL function with the `Call` object you must configure the `Call` 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 VEE environment. Then, select `Select Function` on the `Call` object menu.

VEE will bring up a dialog box with a list of all the functions listed in the definitions file. When you select a function, VEE automatically configures the `Call` object with the correct input and output terminals and function name.

You can also configure the `Call` object manually by modifying the function name and adding the appropriate input and output terminals:

1. Configure 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, etc.

2. Configure the output terminals so the parameters passed by reference appear as output terminals on the `Call` object. Parameters passed by value cannot be assigned as output terminals. The top output terminal is the value returned by the function. The next terminal down is the first parameter passed by reference, etc.

3. 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 does not exactly match the number of parameters in the function, 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` 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 is 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 VEE generates an error.

Note

13

Using ActiveX Automation Objects and Controls

Using ActiveX Automation Objects and Controls

VEE for Windows supports ActiveX automation and controls on PCs running Windows 95, 98, 2000, or NT 4.0. ActiveX technology is not supported on UNIX. This chapter explains how to use ActiveX automation and controls in VEE, but does not describe ActiveX technology. The chapter contents are:

- Using ActiveX Automation in VEE
- Using ActiveX Automation Objects
- Using ActiveX Automation Controls

Note Recommended Reading

Microsoft Office 97 Visual Basic Programmer's Guide.
Microsoft Press, 1997. ISBN 1572313404.

Microsoft Office 2000 Visual Basic Programmer's Guide.
Microsoft Press, 1999. ISBN 1572319526.

VEE implements ActiveX support using the standard established by Microsoft Visual Basic. If you are not familiar with ActiveX technology, review the chapters in these books that discuss Object Models and ActiveX Controls. Understanding these concepts will help you use VEE's ActiveX features.

Using ActiveX Automation in VEE

ActiveX automation lets you use VEE as an automation controller to control other Windows applications such as Microsoft Word, Excel, Access, and Seagate Crystal Reports. You can perform such activities as sending data to the applications for report generation and reading data back from them. For automation-capable applications, this supersedes Dynamic Data Exchange (DDE).

ActiveX controls are available from various vendors. They extend VEE functionality by providing domain-specific services via ActiveX automation properties, methods, and events. Most ActiveX controls also provide a user interface that lets you manipulate a control such as a "slider" to input a value into a program, just as you would do with an VEE `Slider` object.

Note To enable ActiveX support, you must set VEE to VEE 5 or VEE 6 Execution Mode in the `Default Preferences` dialog box, under the `General` tab. VEE 6 is the default mode for new programs. The status bar at the bottom of VEE's window displays the current mode. If you are adding ActiveX functionality to a program developed in VEE Versions 3.x, 4.x, or 5.x, make sure your program runs in VEE 5 or VEE 6 Execution Mode before adding new features. See “Using VEE Execution Modes” on page 10 for more information.

Several examples are available that demonstrate the use of ActiveX automation and ActiveX controls. They are located in the VEE installation directory under `\examples\ActiveXAutomation` and `\ActiveXControls`. To open and run these examples use `Help` ⇒ `Open Example...` .

Using ActiveX Automation Objects

Set VEE to VEE 5 or VEE 6 Execution Mode (in `Default Preferences`) to enable ActiveX support.

Making Automation Objects Available in VEE

When you install Windows applications, it is very likely that ActiveX type libraries are also installed that allow the applications to act as automation servers. Type libraries describe the capabilities of an ActiveX object and are available for use if they exist on your system.You may prefer to select specific type libraries in VEE for the following reasons:

- To have VEE perform type checking on variables declared for ActiveX objects where the object type is defined (see “Declaring Automation Object Variables” on page 350).
- To catch events generated by an automation object (see “Handling Automation Object Events” on page 369).
- To view information in the ActiveX Object Browser (see “Using the ActiveX Object Browser” on page 357).

To select the type libraries you want to reference in a program, click `Device` ⇒ `ActiveX Automation References`... The `ActiveX Automation References` dialog box appears listing all type libraries registered by the Windows Registry. Figure 13-1 shows the dialog box with the Microsoft Access library selected for use.

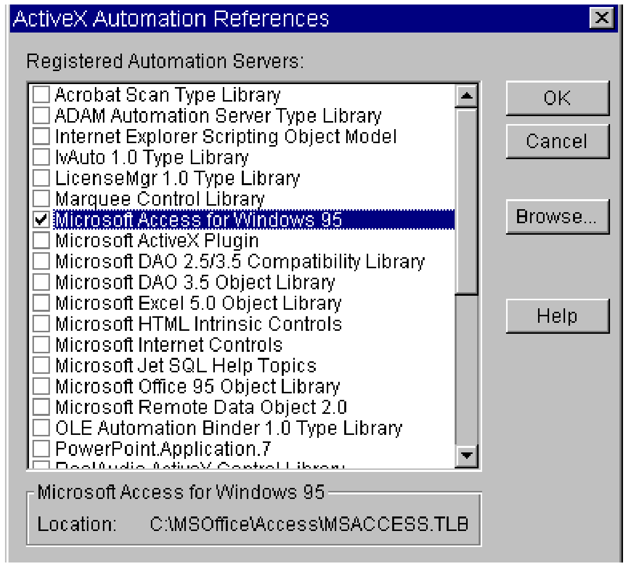


Figure 13-1. Selecting ActiveX Automation Type Libraries

Your list is probably different, depending on the applications you have installed. When you highlight a library name, its location appears in the dialog box status area. When you find the automation server you want to use, click the check box by the library name (or double-click the name itself) so a check mark appears. Then, click `OK`.

This loads the selected type library and searches it for the object classes, dispatch interfaces, and events that it exports. You can select multiple libraries, but you should select only the ones you plan to use since selected libraries use memory.

If you know a type library file exists for an automation server, but it does not appear in the list, it is possible the type library was not registered when the associated application was installed. Press the `Browse` button to find the type library missing from the list. When you locate and open the type library file, VEE will attempt to register the type library and add it to the list.

Declaring Automation Object Variables

You can declare a variable for an ActiveX automation object using the new Object data type (`Data` ⇒ `Variable` ⇒ `Declare Variable`). The declared variable is a reference to an object that lives in another process. For instance, it might point to a ComboBox in Access. As shown in Figure 13-2, when you set the variable `Type` to `Object` the dialog box expands to list the library name, class, and enabled events.

Figure 13-2. Declaring an ActiveX Automation Variable

You can specify the object variable's type further by clicking `Specify Object Type` so a check mark appears. Then, click the `Edit` button to access the `Specify Object Type` dialog box that lets you set the library and class names and enable events available for the class.

If you are using the Access Object Library, you can declare a variable `combo`, then specify the object type as `Library: Access` and `Class: ComboBox` as shown in Figure 13-3. In this example, the class `ComboBox` contains events. To use the events, click `Enable Events`. If events are not available for a class, the checkbox is grayed out.

After specifying the object type, click `OK` to dismiss the dialog box and return to `Declare Variable`, which displays the information.

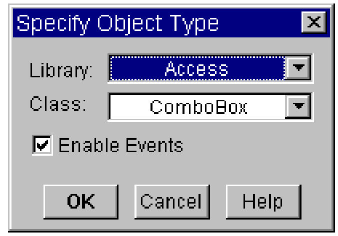


Figure 13-3. Specifying the Automation Object Type

If you enable events, you can create an event-handler UserFunction for each event that you want to catch. For information about using events, see “Handling Automation Object Events” on page 369.

As with any VEE variable, declaring a variable is optional and doing so does not create the automation object in the program. However, if you declare variables for automation objects and specify the object type details, VEE does type checking automatically to assure that only objects of the specified `Library` and `Class` type are assigned to the declared variable.

Note

If you declare a variable for an ActiveX object when developing a program in Windows and then open the program in HP-UX, the program still contains the variable declaration but ignores the object type specifications. The `Declare Variable` object maintains the object type specifications and does not let you change them.

Creating an Automation Object in a Program

To control a server application from VEE, you need to create an automation object in your program. The *`CreateObject`* function lets you do that. To put the function in your program, click the ***fx*** toolbar button to get the `Function & Object Browser` and then select:

```
Type: Built-in Functions
Category: ActiveX Automation
Member: CreateObject
```

Click `Create Formula` and place the `Formula` object in your program. The `Formula` contains the expression

```
CreateObject(ProgID)
```

which you need to modify to perform the desired action.

Note *ProgID* is a human-readable string that identifies the Automation object that you want to create. To determine the `ProgID` for an automation object, refer to the vendor's documentation.

Most of the time you want a new instance of an automation object created in a new instance of the server application. For example, the following VEE expression starts a new instance of Excel (even if Excel is already running) and returns a reference to a new "Workbook" object tied to the `excel` variable.

```
SET excel = CreateObject("Excel.Sheet")
```

Using Distributed Component Object Model (DCOM)

DCOM allows the Automation client (VEE) to control the Automation server (Excel, Access, etc.) remotely. You can run VEE on one computer and control Excel, for example, running on another computer. The second computer does not need VEE installed, just the application VEE is controlling.

To do this, the `CreateObject` function takes an optional second parameter that specifies the name of a remote host computer (server) on which to instantiate the object. (This functionality requires that DCOM be installed on both the client and server computers and that the proper security settings have been configured using `dcomcnfg.exe`.) With this additional parameter, the definition of `CreateObject()` looks like the following:

```
CreateObject ("ProgID", ["hostName"])
```

where *hostName* has to be of type Text. *hostName* can be specified as either a valid UNC or DNS domain name. Valid *hostName* specifications are shown below:

```
Set obj = CreateObject ("ProgID", "server"
Set obj = CreateObject ("ProgID", "\\\\server")
```

```
Set obj = CreateObject ("ProgID", "server.domain.com")
Set obj = CreateObject ("ProgID", "135.5.33.19")
```

VEE does not provide any programmatic control over the security settings used to instantiate objects on remote computers. This can be accomplished statically via the *dcomcnfg.exe* program.

Getting an Existing Automation Object

If you already created an automation object, you can get an active object or load an existing object from a file by using the `GetObject` function. To put the function in your program, click the ***fx*** toolbar button to get the `Function & Object Browser`, then select:

```
Type: Built-in Functions
Category: ActiveX Automation
Member: GetObject
```

Click `Create Formula` and place the `Formula` object in your program. The `Formula` contains the expression

```
GetObject("fileName", "ProgID")
```

which you need to modify to perform the desired action.

The following expression gets an active object and returns a reference to a currently running Excel application's `Application` object. This call will fail if Excel is not running.

```
SET excel = GetObject("","Excel.Application")
```

The following expressions load an existing object from file. The `ProgID` parameter is optional:

```
SET excel = GetObject("d:/tmp/TestData.xls","Excel.Sheet")
```

or

```
SET excel = GetObject("d:/tmp/TestData.xls")
```

They return a reference to the sheet object associated with `d:/tmp/TestData.xls` in the currently running Excel application. If Excel is not already running, it will be started before loading the object. If `ProgID` is omitted, VEE uses the Component Object Model (COM) library to determine what application the file is associated with.

Manipulating Automation Objects

After creating an automation object, you can manipulate the object to control server applications. Manipulating automation objects involves three basic operations: getting properties, setting properties, and calling methods. This section demonstrates these using previously initialized object variables named `cell`, `sheet`, and `excel`. The VEE keywords `SET` and `ByRef` are introduced.

Getting and Setting Properties

The expressions in this section are examples of getting and setting a property of an object. The following expression gets a property, where the `value` property returns the contents of the `cell`:

```
contents = cell.value
```

In the next expression, the `value` property returns the contents of the cell:

```
contents = sheet.cells(1,1).value
```

The next expression does the same property-getting action as the previous expression by implying the `.value` property because of default properties (explained below):

```
contents = sheet.cells(1,1)
```

Sometimes you want the contents, value and default property of the right-hand side (which happens by default) and sometimes you want a pointer to the object on the right-hand side, not its value. To get the object pointer you need to use SET to tell VEE *not* to get the default value. The next expression sets an object reference, where the `cell` variable is set to reference one cell out of the "collection" of cells:

```
SET cell = sheet.cells(1,1)
```

The difference between this example and the second example is that `SET` specifies that the left-hand-side wants the right-hand-side object itself, not its default property.

The following expressions are examples of *setting* a property of an object. The following three expressions are identical because of default properties:

```
cell.value = "Test Data2:"
sheet.cells(1,1).value = "Test Data2"
sheet.cells(1,1) = "Test Data2"
```

About Default Properties. Most automation objects support the concept of a default property or method. You can use this concept when manipulating automation objects as shown in the previous examples. In the case of `cell`, its default property is `value`. So, the first example above in getting a property could use this concept to imply the `.value` property and be entered as

```
contents = cell
```

This means that the expression

```
contents = sheet.cells(1,1)
```

would not only return a cell from the collection of cells, but it would also evaluate the default property (`.value`) on that cell as in the expression

```
contents = sheet.cells(1,1).value
```

To get a cell itself from the collection of cells, you must use the keyword `SET` in the expression such as

```
SET cell = sheet.cells(1,1)
```

This sets `cell` to be a pointer to that cell in Excel. Compare this to the expression

```
contents = sheet.cells(1,1)
```

(mentioned above) where `contents` gets the contents of that cell in Excel. Also, the `.value` property is implied on `Set Property`, such that the following two expressions perform the identical function:

```
cell.value = "Test Data”
cell = "Test Data"
```

Calling Methods

Calling a method is similar to getting a property, but methods have parenthesis-like () functions and can take parameters. Properties are generally used to get or set the value of an attribute of the object. Methods are generally used to perform an action.

The following expression is an example of calling a method on an object:

```
result = excel.CheckSpelling("aardvark")
```

By default, parameters are passed by value. For example, `cells(1,1)` actually calls a method and passes two parameters (`1` and `1`). Passing by value simply sends the parameter values to Excel and a return value comes back. The parameter values are unchanged.

Some automation methods have parameters that are passed by reference. The parameter's value is changed by the automation server and a new value for the parameter is passed back to VEE. For example, an ActiveX instrument control might contain an automation method called by this expression

```
passed = Scanner.GetReading (ByRef Reading)
```

where the method's return value for `passed` is true if the getReading worked or false if it did not, and any other values are returned in the `ByRef` parameter `Reading`. You should initialize the variable `Reading` before passing it to the function and have an output terminal on the `Formula` object containing the expression so you can use any returned values.

The `ByRef` keyword is supported in VEE, and the `Function & Object Browser` displays in its information area the parameters passed using `ByRef`. `ByRef` does not support all data types. See Table 13-4, "Converting from VEE Data Types to Automation Scalar Data Types in VEE 5 Execution Mode," on page 365.

Using Enumerations

Type libraries can provide enumerations that appear in the `Class` area of VEE's `Function & Object Browser`. Enumerations make using object methods and properties easier. For instance, the `Window` object in Excel has a `WindowState` property. The `WindowState` property is of type `XlWindowState` enumeration. There are three values for this enumeration:

```
xlMaximized (-4137)
xlMinimized (-4140)
xlNormal (-4143)
```

VEE supports enumerations, which allows you to use the following expression when using object methods and properties:

```
Window.WindowState = xlMinimzed
```

Using the ActiveX Object Browser

The ActiveX Object Browser is part of the `Function & Object Browser` that opens when you press ***fx*** on the toolbar. The browser configuration changes when you select `Type: ActiveX Objects`. The browser lets you explore the properties, methods, and events that an ActiveX object provides. ActiveX information appears here only if you selected automation or control type libraries (`Device` ⇒ `ActiveX Automation References` or `ActiveX Control References`). Figure 13-4 shows the `Function & Object Browser` with ActiveX information for the Access type library.

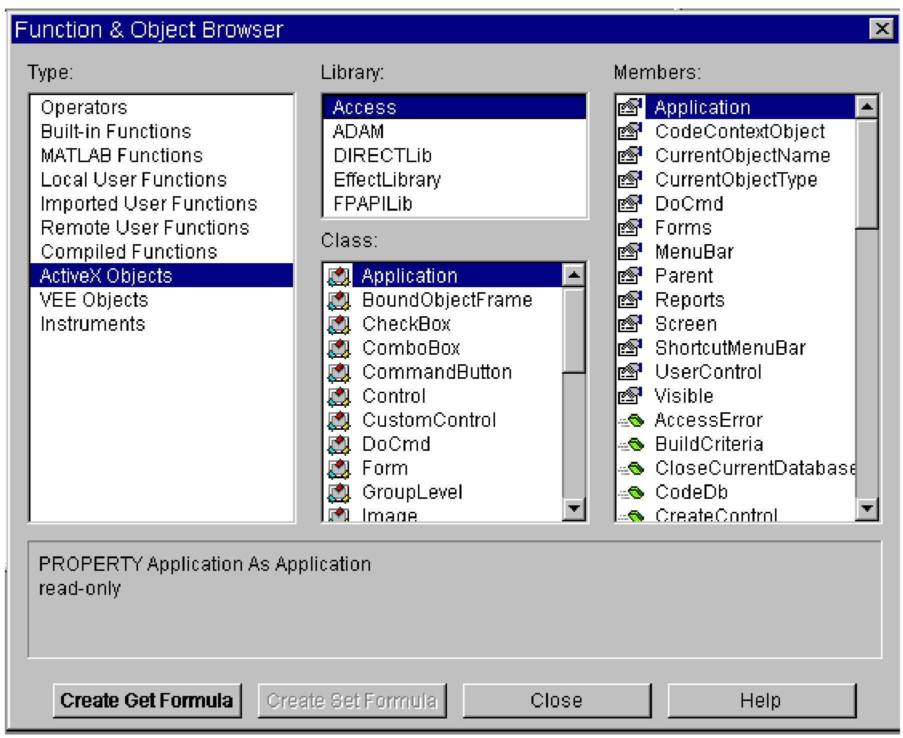


Figure 13-4. Using the ActiveX Object Browser

When a `Library` name is selected, the `Class` area displays dispatch interfaces (dispinterfaces) and enumerations that are available. For a selected dispinterface, the available properties, methods, and events appear in the `Member` area. For enumerations, the constants are listed. Figure 13-4 displays some of the functionality available for the Access library. The selected `ComboBox` dispinterface contains properties, a method, and many events that are listed in the `Member` area. Figure 13-5 shows the relationship between entries in the browser's `Classes` and `Members` areas, including their identifying icons:

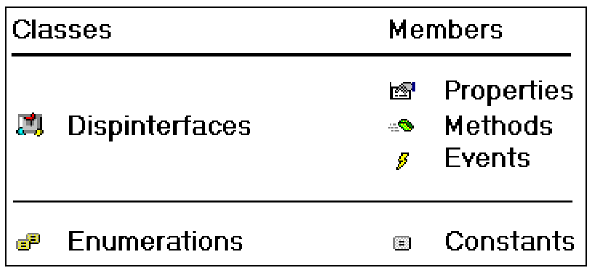


Figure 13-5. Elements Displayed in the Function & Object Browser

The browser's information area (just above the buttons) displays a help string associated with the property, method, event, or constant if this information is provided by the automation object. This syntax contains the object's type information in the parameter list. Parameters surrounded by square brackets [ ] are designated as optional. Some applications may not provide these short help strings.

Type information is provided for an ActiveX object's properties, method parameters, and return type. If no parameter type is specified, the default type is VT_VARIANT. In most cases, VEE automatically handles type conversion for VT_VARIANT. See Table 13-1 and following for more information about Automation data type and VEE data Type conversion.

For a property, the browser displays type information about the property, such as whether it is a read-only or write-only property and whether it is the default property. You can create a `Formula` object to perform a get or set of that property. The following is an example of what the browser displays in the information area for a property:

```
DEFAULT PROPERTY Name as Text
```

For a method, the browser displays type information about each parameter in the parameter list and the return value. Methods can also be the default member, so the browser also indicates this. You can create a `Formula` object for a method that is configured to call that method. The following is an example of what the browser displays in the information area for a method:

```
METHOD Void SetData(vValue, vFormat)
```

For events, the browser displays the same type information as for a method. However, the event handler associated with an event is usually called by the client application. In the case of controlling Access by automation, Access calls the event handler UserFunction. In the case of using an ActiveX control, the ActiveX control calls the event handler UserFunction.

Since your program or VEE does not call these callback event handlers, the `Create Formula` button is grayed out. You can only view information about an event. The `Function & Object Browser` does not let you create event-handler UserFunctions because events must be tied to a particular ActiveX automation variable or an ActiveX control.

To create an event handler, go to the object menu of the appropriate `Declare Variable` or ActiveX control. The following is an example of what the browser displays in the information area for an event:

```
EVENT Void Click()
```

For constants in an enumeration, the browser displays the value of the constant. The following is an example of what the browser displays in the information area for a constant:

```
CONSTANT tvwRootLines = 1
```

For constant values less than 0 and greater than 1024, VEE also displays the hexadecimal value of the constant. This information appears as:

```
CONSTANT xlNormal = -4143 (#HFFFFEFD1)
```

Clicking the `Help` button opens the help file and topic associated with the selected ActiveX object member if that information is provided by the object. If no information is available, a dialog box appears, indicating that no help is available for the selected member. This help information is provided by the application vendor and is not part of *VEE Online Help*.

Data Type Compatibility

ActiveX automation provides support for certain data types. This section describes the type conversion that takes place between VEE data types and ActiveX automation data types. Type conversion occurs automatically.

Table 13-1 indicates the automation data types that are supported and the corresponding VEE 6 Execution Mode data type.

Table 13-1. Converting from Automation Scalar Data Types to VEE Data Types in VEE 6 Execution Mode

Convert from Automation Data Type	Convert to VEE Data Type	Notes
VT_EMPTY	Text	Text with empty string ("") is returned. Use `isVariantEmpty()` to determine if variant was of type VT_EMPTY
VT_NULL	Text	Text with empty string ("") is returned. Use `isVariantNull()` to determine if variant was of type VT_NULL
VT_UI1	UInt8	
VT_I2	Int16	
VT_I4	Int32	
VT_R4	Real32	
VT_R8	Real64	
VT_CY	Real64	8-byte fixed point integer with 4 digits to right of decimal is converted to VEE Real64. Use `isVariantCurrency()` to determine if variant was of type VT_CY.

Table 13-1. Converting from Automation Scalar Data Types to VEE Data Types in VEE 6 Execution Mode

Convert from Automation Data Type	Convert to VEE Data Type	Notes
VT_DATE	Real64	Days since 12/30/1899 converted to VEE's representation of date/time in seconds since Jan 1, 0001.
VT_BSTR	Text	
VT_DISPATCH	Object	
VT_ERROR	Int32	An Int32 with value of the scode is returned. Use `isVariantError()` to determine if variant was of type VT_ERROR.
VT_BOOL	Int16	Use `isvariantBool()` to determine if variant was of type VT_BOOL.
VT_VARIANT	*	Only valid in ByRef case (VT_VARIANT | BYREF) In this case, the variant points to another variant. VEE creates a container based on the embedded variant's type.
VT_UNKNOWN	Object	

Table 13-2 shows the automation data types that are supported and the corresponding VEE 5 Execution Mode data type.

Table 13-2. Converting from Automation Scalar Data Types to VEE Data Types in VEE 5 Execution Mode

Convert from Automation Data Type	Convert to VEE Data Type	Notes
VT_EMPTY	Text (empty string)	
VT_NULL	Text (empty string)	
VT_UI1	<generates error>	unsigned char
VT_I2	Int32	
VT_I4	Int32	
VT_R4	Real64	
VT_R8	Real64	
VT_CY	Real64	8-byte fixed point integer with 4 digits to right of decimal is converted to VEE Real64.
VT_DATE	Real64	Days since 12/30/1899 converted to VEE's representation of date/ time in seconds since Jan. 1, 0001.
VT_BSTR	Text	
VT_DISPATCH	Object	
VT_ERROR	<generates error>	
VT_BOOL	Int32	
VT_VARIANT	*	Only valid in ByRef case (VT_VARIANT | BYREF) In this case, the variant points to another variant. VEE creates a container based on the embedded variant's type.

Table 13-2. Converting from Automation Scalar Data Types to VEE Data Types in VEE 5 Execution Mode

Convert from Automation Data Type	Convert to VEE Data Type	Notes
VT_UNKNOWN	Object	

Table 13-3 indicates the VEE 6 Execution Mode data types supported and the corresponding automation data types. Unlike the inverse mappings shown in the previous table, these are not fixed one-to-one mappings. Most automation server objects are capable of coercing data to the required data type.

For example, if the target property is a long integer, such as the X coordinate of a point, you can pass not only an `Int32` which is the exact match, but also a `real` or even a text string, as long as it is a string of digits. However, in case of an array, which is always passed as a VARIANT, acceptable data type and array shape depends on the implementation of the target COM object.

Table 13-3. Converting from VEE Data Types to Automation Scalar Data Types in VEE 6 Execution Mode

Convert from VEE Data Type	Convert to Automation Data Type	Notes
Uint8	VT_UI1	
Int16	VT_I2	
Int32	VT_I4	
Real32	VT_R4	
Real64	VT_R8	
Text	VT_BSTR	
<scalar of type *>	VT_BOOL	Use `asVariantBool()` on any scalar VEE data type that can be cast to an Int16, (UInt8, Int16, Int32, Real32, Real64, Text).

Table 13-3. Converting from VEE Data Types to Automation Scalar Data Types in VEE 6 Execution Mode

Convert from VEE Data Type	Convert to Automation Data Type	Notes
<scalar of type *>	VT_CY	Use `asVariantCurrency()` on any scalar VEE data type that can be cast to a Real64 (UInt8, Int16, Int32, Real32, Real64, Text).
<scalar of type *>	VT_DATE	Use `asVariantDate()` on any scalar VEE data type that can be cast to a Real64 (UInt8, Int16, Int32, Real32, Real64, Text).
<scalar of type *>	VT_ERROR	Use asVariantError() on any VEE data type that can be cast to an Int32 (UInt8, Int16, Int32, Real32, Real64, Text). Value of Int32 is assigned as the scode (error number) for the error.
Variant	<scalar of variant type *>	The current type of the VEE container is converted to the appropriate variant type. For instance, if the Variant container holds a VEE Int32, VEE will create a variant of type VT_I4.
Object	VT_DISPATCH	If VEE holds an IDispatch pointer to the object.
Object	VT_UNKNOWN	If VEE has an IUnknown pointer but not an IDispatch pointer.

Table 13-4 indicates the VEE 5 Execution Mode data types supported and the corresponding automation data types.

Table 13-4. Converting from VEE Data Types to Automation Scalar Data Types in VEE 5 Execution Mode

Convert from VEE Data Type	Convert to Automation Data Type	Notes
Int32	VT_I4	
Real64	VT_R8	
Text	VT_BSTR	
Object	VT_DISPATCH	If VEE holds an IDispatch pointer to the object.
Object	VT_UNKNOWN	If VEE has an IUnknown pointer but not an IDispatch pointer.

Table 13-5 shows array conversions from Automation data type to VEE 6 Execution Mode data type.

Table 13-5. Converting from Automation Array Data Types to VEE Data Types in VEE 6 Execution Mode

Convert from Automation Data Type	Convert to VEE Data Type	Notes
VT_UI1	UInt8 Array	
VT_I2	Int16 Array	
VT_I4	Int32 Array	
VT_R4	Real32 Array	
VT_R8	Real64 Array	
VT_BSTR	Text Array	
VT_BOOL	Int16 Array	Use `isVariantBool()` to determine if variant array was of type VT_BOOL.

Table 13-5. Converting from Automation Array Data Types to VEE Data Types in VEE 6 Execution Mode

Convert from Automation Data Type	Convert to VEE Data Type	Notes
VT_CY	Real64 Array	8-byte fixed point integer with 4 digits to right of decimal is converted to VEE Real64. Use `isVariantCurrency()` to determine if variant array was of type VT_CY.
VT_DATE	Real64 Array	Days since 12/30/1899 converted to VEE's representation of date/time in seconds since Jan. 1, 0001. Use isVariantDate() to determine if variant is of type VT_DATE.
VT_ERROR	Int32 Array	An Int32 with value of the scode is returned. Use isVariantError() to determine if variant array was of type VT_ERROR.
VT_VARIANT	<array of homogeneous type OR a record>	If the array elements are all of the same fundamental data type, VEE creates an array of the type indicated by the scalar mapping in Table 13-1. A VEE record is created for a variant containing a mixed data type array.
VT_DISPATCH	N/A	Arrays of type Object not supported.
VT_UNKNOWN	N/A	Arrays of type Object not supported.

Table 13-6 shows array conversions from Automation data type to VEE 5 Execution Mode data type.

Table 13-6. Converting from Automation Array Data Types to VEE Data Types in VEE 5 Execution Mode

Convert from Automation Data Type	Convert to VEE Data Type	Notes
VT_UI1	Int32 Array	
VT_I2	Int32 Array	
VT_I4	Int32 Array	
VT_R4	Real64 Array	
VT_R8	Real64 Array	
VT_BSTR	Text Array	
VT_BOOL	Int32 Array	
VT_CY	Real64 Array	8-byte fixed point integer with 4 digits to right of decimal is converted to VEE Real64.
VT_DATE	Real64 Array	Days since 12/30/1899 converted to VEE's representation of date/ time in seconds since Jan. 1, 0001.
VT_ERROR	<generates error>	
VT_VARIANT	Record	A VEE record is created for a variant containing a mixed data type array.
VT_DISPATCH	N/A	Arrays of type Object not supported.
VT_UNKNOWN	N/A	Arrays of type Object not supported.

Table 13-7 shows Automation Data Type modifiers.

Table 13-7. Automation Data Type Modifiers

Automation Type Modifiers	VEE Type	Notes:
VT_BYREF	Either scalar or array of the type indicated by the scalar mapping table above.	For instance, VT_BYREF | VT_BSTR would generate a VEE Text container. In the case of a scalar VT_VARIANT, the variant points to another variant. VEE creates a container based on the data type of the embedded variant.

Deleting Automation Objects

Automation objects are responsible for deleting themselves when VEE releases its reference to them. When VEE no longer holds a reference to an automation object, it tells the object that the reference has been released. The object then deletes itself unless other ActiveX automation controller applications are still using it. VEE releases references to automation objects in the following cases:

- The `Delete Variable` object is executed on the automation object's variable name. However, VEE may also have other pointers to these Automation objects. See "Delete Variable" in *VEE Online Help* for more information.
- `Delete Variables at Prerun` is enabled in `Default Preferences` and you restart the program.
- VEE exits, or the commands `File` ⇒ `New` or `File` ⇒ `Open` are used.

Handling Automation Object Events

Automation objects can generate events. VEE, as an automation controller, lets you catch events via UserFunctions. You can create event-handler UserFunctions for an automation object that generates events if you have declared a variable of the specific type and have enabled its events. You can create an event-handler UserFunction for each event an object can generate.

You can create an event-handler UserFunction when you declare a variable for the object and enable its events (if they are available).

1. After declaring the variable and specifying its type, including enabling events, open the `Declare Variable` object menu.

2. In the object menu, click `Create Event Handler`... The `Create Event Handler UserFunction` browser appears. See Figure 13-6

   The `Member` area lists all of the events available for the dispatch interface listed in the `Class` area.

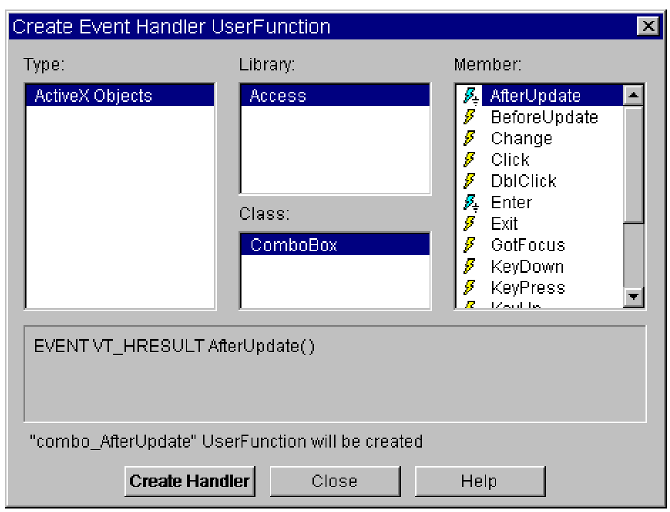


Figure 13-6. `Create Event Handler UserFunction` browser

3. Click an event name to select it.

   When you select an event, the browser information area presents event details and the status area shows the UserFunction title VEE will create. Press the `Help` button to get information about using the event. Not all events have online help as the library vendor is responsible for providing it. Online help for events is not part of *VEE Online Help*.

4. Click `Create Handler`. The new UserFunction window appears. If you open this dialog box again to create another event handler, you will notice the icons change color next to events with existing handlers.

   Each new event-handler UserFunction is empty except for any required inputs or outputs. You must program it to handle the event appropriately. To edit an existing event, in the `Declare Variable` object menu, click `Edit Event Handler`...

Events are tied to the declared variable's name. The UserFunction title combines the variable name with the event name. For example, if you declared a variable named `combo` and specified its type as `Access.ComboBox` you could create event-handler UserFunctions with names such as:

```
combo_AfterUpdate
combo_Change
combo_DblClick
combo_KeyDown
```

Events are callback functions. You must program the generated UserFunctions (the callback functions) to handle each event appropriately. If the automation object generates an event, it calls the related UserFunction to handle the event. Automation objects sometimes expect a return value from VEE when they fire an event. If so, you must program the UserFunction to return a value.

When the object expects a return value, it waits until VEE provides this return value. You should write an event-handler UserFunction to work quickly, since both VEE and the automation server, such as Access, wait until the event-handler UserFunction returns.

Since the automation server waits until the event-handler UserFunction returns, the UserFunction is executed in non-timeslicing mode. That is, the UserFunction runs to completion without timeslicing with the rest of the VEE program. Because it is not timeslicing, breakpoints do not work in an event-handler UserFunction. Also, errors do not stop VEE. Errors are turned into Cautions and execution continues.

Using ActiveX Automation Controls

Make sure VEE is set to VEE 5 or VEE 6 Execution Mode (in `Default Preferences`) to enable ActiveX support. See "Using ActiveX Automation in VEE" on page 347 for more information about ActiveX support.

Note VEE does not support all ActiveX controls. If a control is incompatible with VEE, an error may occur when you add the control to your program or while you are using the control. Controls that are known to not work properly are listed in *VEE Online Help*.

Selecting ActiveX Controls

Before you can use ActiveX controls in VEE, you need to inform VEE which ActiveX controls you want to use. Click `Device` ⇒ `ActiveX Control References`... The resulting `ActiveX Control References` dialog box lists the available control type libraries registered by the Windows Registry. Figure 13-7 shows the dialog box with several selected controls.

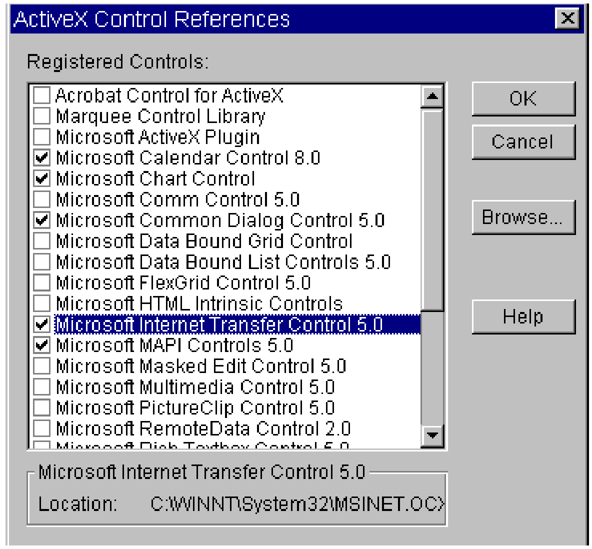


Figure 13-7. Selecting ActiveX Controls

Your list is probably different depending on the applications or controls you have installed. Controls can be installed individually or as part of other application installations. When you highlight a control name, its location appears in the dialog box status area.

When you find the control you want to use, click the check box by the control name (or double-click the name itself) so a check mark appears. Then, click `OK` to load them into memory for use in VEE and to search for their object classes, dispatch interfaces, and exported events. You can select multiple controls, but you should select only the ones you plan to use since selected libraries use memory.

If you know a control type library exists for a control, but it does not appear in the list, it is possible the library did not get registered during its installation. Press the `Browse` button to find the type library missing from the list. When you locate and open the type library file, VEE will attempt to register the type library and add it to the list.

Adding a Control to VEE

Adding a control to an VEE program is similar to adding any other object. After you select the ActiveX control(s), as explained previously, you can add them to your program. Click `Device` ⇒ `ActiveX Controls` to view a cascading menu listing the selected controls. See Figure 13-8 for an example.

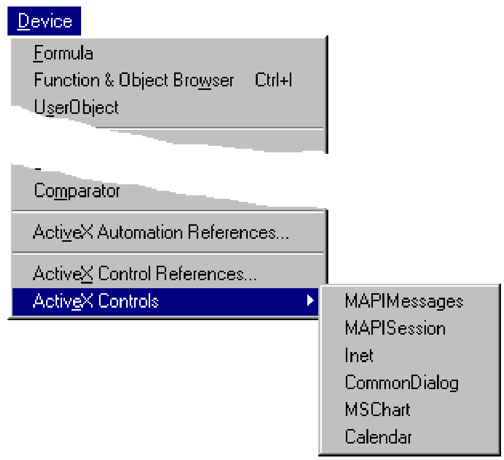


Figure 13-8. Adding ActiveX Controls from the Device Menu

Note

In Figure 13-7 and Figure 13-8, five controls are selected in the `ActiveX Control References` dialog box, but six appear in the `Device` ⇒ `ActiveX Controls` cascading menu. It is normal for some selections to result in more than one ActiveX control being added to the resulting menu.

Select a control and place the resulting object in a detail view in the VEE work area. You can place controls in any context – Main, UserObject, or UserFunction. You can delete controls by selecting `Cut` from their object menu or double-clicking the object's context menu button. See Figure 13-9.

Differences in the ActiveX Control Host

ActiveX controls are different from any other VEE object. Unlike all other VEE objects, ActiveX controls have no input or output pins nor do they have any sequence input or output pins. Controls are not data flow oriented.

To give you access to a control that is similar to the access available to other objects, VEE creates a special container in the program that is the host for the control. The container also gives you access to the control's specific properties built into it by the control's developers. Regardless of the combined features, we refer to these as ActiveX controls.

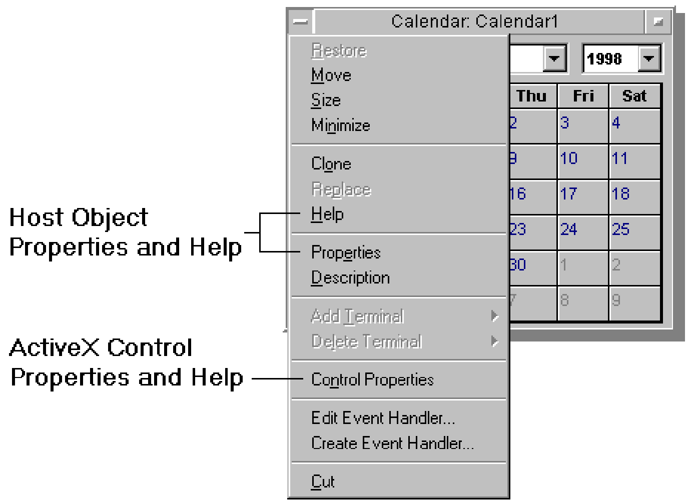


Figure 13-9. Accessing Properties and Help in an ActiveX Control

Some differences in the object menu follow. The `Properties` and `Control Properties` menu items provide access to two different sets of properties. The host container's properties are separate from the control's properties. To see the typical properties associated with VEE objects, in this case the host container, click `Properties`. To view and change the ActiveX control's properties that are provided by the control's developer, click `Control Properties`.

The `Help` button on the control's `Properties` dialog box opens the online help for that control if the developer provided one. The object menu's `Help` item opens the *VEE Online Help* topic for the host container. `Create Event Handler...` and `Edit Event Handler...` provide the same functionality as described for ActiveX automation objects in "Handling Automation Object Events" on page 369.

Using an ActiveX Control in VEE

When you add a control to the VEE work area, the control appears with an assigned local variable name in its title bar. You can change the assigned variable by double-clicking the control's title bar to get the `ActiveX Control Properties` dialog box. On the `General` tab, change the value beside `Name:`.

Since the control has no pins to connect with lines to other objects in your program, you must manipulate the control using expressions in `Formula` objects that refer to the control by its variable name. These expressions must appear in the same context as the control, since the control's variable name is scoped "local to context".

Using the Assigned Local Variable

If you add a Calendar control to your program, it is assigned the local variable name `Calendar`. The title bar contains `Calendar`. To interact with the control, add a `Formula` object that is in the same context as the Calendar control. The following examples demonstrate setting a property, getting a property and calling a method on the ActiveX control referenced by the VEE local variable called `Calendar`:

```
Calendar.Day = 24;
Month = Calendar.Month;
Calendar.AboutBox()
```

Declaring a Global Variable for a Control

If you want the variable name to be global, declare a new variable name using `Declare Variable` (`Data` ⇒ `Variable` ⇒ `Declare Variable`). This is similar to the variable declaration described in "Declaring Automation Object Variables" on page 350. Since the control's variable name already exists, such as `Calendar`, you cannot declare it as global as VEE does not allow such conflicts. A common naming convention is to adapt the local variable name (as in `g_localName`), resulting in `g_calendar`.

In `Declare Variable`, enter the new variable name, set `Scope` to `Global` and set `Type:` to `Object`. You do not need to check `Specify Object Type` to specify the particular `Library` and `Class`. However, if you do so, VEE will do type checking automatically to assure that the `Library` and `Class` are assigned only to the declared variable.

After declaring the global variable, use a `Formula` expression to set the control's local variable name (such as `Calendar`) equal to the declared variable name (such as `g_calendar`). It is important to use the `SET` keyword, as shown in this expression:

```
SET g_calendar = Calendar
```

Manipulating ActiveX Controls

Setting and getting properties, calling methods and handling events for an ActiveX control uses the same mechanisms described for ActiveX automation objects in "Manipulating Automation Objects" on page 354 and in "Handling Automation Object Events" on page 369.

Although VEE contains ActiveX controls in host objects so they are accessible, the control's behavior is slightly different when a program runs. Basically, controls are viewable in only one place at a time – either the detail view or panel view.

As an example, suppose a control is added to a program's detail view and the program dynamically displays a panel on which the control appears using `Show Panel on Execute` or `showPanel()`. The control is blanked out in the detail view until the panel closes. When the panel closes, the control reappears in the detail view.

A

I/O Transaction Reference

I/O Transaction Reference

This appendix describes VEE I/O transaction actions, encodings, and formats. The contents are:

- I/O Transactions Summary
- WRITE Transactions
- READ Transactions
- Other Transactions

I/O Transactions Summary

Table A-1 summarizes I/O transaction types for VEE, and Table A-2 summarizes the I/O Transactions Objects for VEE.

Table A-1. Summary of I/O 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, instrument, 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 instruments and hardware interfaces.
WAIT	Waits for the specified number of seconds before executing the next transaction. For Direct I/O objects, WAIT can also wait for a specific serial poll response, or for specific values in accessible VXI instrument registers.
SEND	Sends IEEE 488-defined bus messages (bus commands and data) to a GPIB interface.

Table A-2. Summary of I/O Transaction Objects

Objects	Supported Transactions				
	EXECUTE	WAIT	READ	WRITE	SEND
To File	X	X		X	
From File	X	X	X		
To Printer		X		X	
To String		X		X	
From String		X	X		
Direct I/O	X	X	X	X	
MultiInstrument Direct I/O	X	X	X	X	
Interface Operations	X				X

WRITE Transactions

This section describes the `WRITE` transaction in Table A-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 these terms:

- **MS-DOS path** is any destination other than an instrument, such as an MS-DOS file, a string, or the printer.
- **Direct I/O path** is any instrument accessed using `Direct I/O`.

Behaviors for all Paths

The behaviors described in the following sections apply to all paths except as specifically noted in Table A-3

Table A-3. `WRITE` **Encodings and Formats**

Encodings	Formats

`TEXT`	`DEFAULT`
	`STRING`
	`QUOTED STRING`
	`INT16, INT32`
	`OCTAL`
	`HEX`
	`REAL32, REAL64`
	`COMPLEX`
	`PCOMPLEX`
	`COORD`
	`TIME STAMP`

`BYTE`	Not Applicable

Table A-3. WRITE Encodings and Formats

Encodings	Formats
CASE	Not Applicable
BINARY	STRING BYTE INT16 INT32 REAL32 REAL64 COMPLEX PCOMPLEX COORD
BINBLOCK	BYTE INT16 COMPLEX INT32 PCOMPLEX REAL32 REAL64 COORD
CONTAINER	Not Applicable
STATE[a]	Not Applicable
REGISTER[b]	BYTE WORD16 WORD32 REAL32
MEMORY[b]	BYTE WORD16 WORD32 REAL32 WORD32*2 REAL64
IOCONTROL[c]	Not Applicable

a. Direct I/O to GPIB only.
b. Direct I/O to VXI only.
c. Direct I/O to GPIO 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 A-4.

Table A-4. Formats for `WRITE TEXT` Transactions

Format	Description
`DEFAULT`	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 same format as `STRING`, except the data is surrounded by double quotes (ASCII 34 decimal).
`INT16`	Writes data as a 16-bit two's complement integer in decimal form.
`INT32`	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.
`REAL32`	Writes data as a 32-bit floating point number in a variety of notations including fixed decimal and scientific notation.
`REAL64`	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.

Table A-4. Formats for `WRITE TEXT` Transactions

Format	Description
`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` as shown in Figure A-1:

```
WRITE TEXT X
```

Figure A-1. A `WRITE TEXT` Transaction

If `X` in Figure A-1 contains text, such as:

```
bird cat dog
```

no conversion is performed and the transaction writes exactly 12 characters.

If `X` in Figure A-1 contains a scalar Integer, such as:

`8923` *the value of X (decimal notation)*

the numeric value is converted to text and VEE writes exactly four characters.

If `X` in Figure A-1 contains a scalar real value, such as:

`1.2345678901234567` *the value of X (17-digit scalar real value)*

each significant digit up to 16 significant digits is written. The least significant digit is approximate because of the conversion between VEE's internal binary form and decimal notation. If you use this scalar real value using the transaction:

```
WRITE TEXT a EOL
```

then VEE writes this:

`1.234567890123457` *16-digit value*

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`*, the converted Text is written right- or left-justified within a space *`F`* characters wide.

The transactions in Figure A-2 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 A-2. Two `WRITE TEXT STRING` **Transactions**

If `X` and `Y` in Figure A-2 have these values:

`bird cat dog`	*the Text value of X*
`12345678901234567`	*the Real value of Y*

then VEE writes:

```
bird cat dog
12345678901234567
^                   ^
```

The caret characters (`^`) *are not* actually written by VEE, but 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`, the transactions appear as shown in Figure A-3.

```
WRITE TEXT X STR FW:20 RJ EOL
WRITE TEXT Y STR FW:20 RJ EOL
```

Figure A-3. Two `WRITE TEXT STRING` **Transactions**

If `X` and `Y` in Figure A-3 have these values:

`bird cat dog`	*the Text value of X*
`12345678901234567`	*the Real value of Y*

VEE writes:

```
        bird cat dog
   12345678901234567
^                   ^
```

The caret characters (`^`) *are not* actually written by VEE, but 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 as only `MAX NUM CHARS` can truncate characters.

The transaction in Figure A-4 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 A-4. A `WRITE TEXT STRING` **Transaction**

If `X` in Figure A-4 has this value:

`bird cat dog` *the Text value of X, 12 characters*

VEE writes:

`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`, all the characters generated by the conversion of each item in *`ExpressionList`* are written. If you specify `MAX NUM CHARS`: *`M`*, only the first *`M`* characters of each item in *`ExpressionList`* are written.

The transactions in Figure A-5 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 A-5. Two `WRITE TEXT STRING` **Transactions**

If `X` and `Y` in Figure A-5 have these values:

```
bird cat dog          the Text value of X
12345678901234567     the Real value of Y
```

VEE writes:

```
bird ca
1234567
^                  ^
```

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 VEE, but 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. 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 A-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`*, the converted Text and the surrounding quotes are written right or left justified within a space *`F`* characters wide.

The transactions in Figure A-6 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 A-6. Two `WRITE TEXT QUOTED STRING` Transactions

If `X` and `Y` in Figure A-6 have these values:

`bird cat dog`	*the Text value of X*
`12345678901234567`	*the Real value of Y*

VEE writes:

```
"bird cat dog"
"12345678901234567"
^                   ^
```

The caret characters (`^`) *are not* actually written by VEE, but 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`, the transactions appear as shown in Figure A-7.

```
WRITE TEXT X QSTR FW:20 RJ EOL
WRITE TEXT Y QSTR FW:20 RJ EOL
```

Figure A-7. Two `WRITE TEXT QUOTED STRING` Transactions

If `X` and `Y` in Figure A-7 have these values:

`bird cat dog`	*the Text value of X*
`12345678901234567`	*the Real value of Y*

VEE writes:

```
     "bird cat dog"
"12345678901234567"
^                 ^
```

The caret characters (`^`) *are not* actually written by VEE, but 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 as only `MAX NUM CHARS` can truncate characters.

The transaction in Figure A-8 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 A-8. A `WRITE TEXT QUOTED STRING` **Transaction**

If `X` in Figure A-8 has this value:

`bird cat dog` *the Text value of X, 12 characters*

VEE writes:

`"bird cat dog"` *all 12 characters*

Number of Characters. If you specify `ALL CHARS`, all 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`*, 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 A-9 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 A-9. Two `WRITE TEXT QUOTED STRING` **Transactions**

If `X` and `Y` in Figure A-9 have these values:

`bird cat dog` *the Text value of X*
`12345678901234567` *the Real value of Y*

VEE writes:

```
"bird ca"
"1234567"
^                    ^
```

The caret characters (^) *are not* actually written by VEE, but 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 VEE. VEE uses escape characters to represent control characters within quoted strings.

See Table A-5 for Escape Characters.

Table A-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 A-10.

```
WRITE TEXT X QSTR EOL
```

Figure A-10. A `WRITE TEXT QUOTED STRING` **Transaction**

If `X` in Figure A-10 has this value:

```
bird\ncat dog
```

VEE writes this to UNIX paths:

```
"bird\ncat dog"
```

For the same transaction and data, VEE writes this to direct I/O paths:

```
"bird<LF>cat dog"
```

where `<LF>` means the single character, linefeed (ASCII 10 decimal).

If `X` in Figure A-10 has this value:

```
bird \"cat\" dog
```

VEE writes this to UNIX paths and Direct I/O paths for serial interfaces:

```
"bird \"cat\" dog"
```

For the same transaction and data, VEE writes this to direct I/O paths for GPIB interfaces:

```
"bird ""cat"" dog"
```

This unique behavior for GPIB interfaces is provided to support the requirements of IEEE 488.2.

INTEGER Formats

`WRITE TEXT INTEGER` transactions are of this form:

```
WRITE TEXT ExpressionList INT16
WRITE TEXT ExpressionList INT32
```

`ExpressionList` is a single expression or a comma-separated list of expressions.

The type of integer generated by this transaction is a 16-bit or 32-bit two's complement integer. The range of 16-bit integers is -32766 to +32767. The range of 32-bit integers is `-2 147 483 648 to +2 147 483 647`. The only characters written to represent these numbers are `+–0123456789`.

VEE attempts to convert each item in *`ExpressionList`* to the `Int32` or `Int16` data type before converting it to Text for final formatting. VEE follows the usual conversion rules. See the `Data Type Conversion` topics under `Tell Me About`... in *VEE Online Help* for more details.

If a `Real32` is written using `INT16` or `INT32` format:

- Real values outside the valid range of `Int16` generate an error.
- Real values within the valid range of `Int32` or `Int16` are converted by truncating the fractional portion of the Real.

If a `Real64` is written using `INT16` or `INT32` format:

- Real values outside the valid range of `Int32` or `Int16` generate an error.
- Real values within the valid range of `Int32` or `Int16` 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 and 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 `Int16` or `Int32` transactions in Figure A-11, which differ only in their specification for the number of output digits.

```
WRITE TEXT X INT16 EOL     default number of digits
WRITE TEXT X INT16:6 EOL   six digits
                 or
WRITE TEXT X INT32 EOL     default number of digits
WRITE TEXT X INT32:6 EOL   six digits
```

Figure A-11. Two `WRITE TEXT INTEGER` Transactions

If `X` in Figure A-11 has this value:

```
4567
```

VEE writes:

```
4567
004567
```

`MIN NUM DIGITS` never causes truncation of the output string. The transaction in Figure A-12 specifies the minimum number of digits to be 1.

```
WRITE TEXT X INT16:1 EOL
          or
WRITE TEXT X INT32:1 EOL
```

Figure A-12. A `WRITE TEXT INTEGER` Transaction

If `X` in Figure A-12 has a value of:

`12345678`

VEE writes:

`12345678`   *all eight digits*

Sign Prefixes. You may optionally specify one of the sign prefixes listed in Table A-6 as part of a `WRITE TEXT INT` transaction.

Table A-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 A-13 specifies explicit leading signs for positive and negative numbers.

```
WRITE TEXT X INT16:6 SIGN:"+/-" EOL
WRITE TEXT Y INT32:6 SIGN:"+/-" EOL
```

Figure A-13. Two `WRITE TEXT INTEGER` Transactions

If `X` and `Y` in Figure A-13 have values of:

`123` *the Integer value of X*
`-123` *the Integer value of Y*

VEE writes:

`+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 `-2 147 483 648` to `+2 147 483 647`. The only characters written to represent these octal numbers are `01234567`. An optional prefix may be specified which may include other characters.

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 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 "Number of Digits" on page 398 for `WRITE TEXT INTEGER` transactions.

Octal Prefixes. You may specify one of the prefixes listed in Table A-7 as part of a `WRITE TEXT OCTAL` transaction.

Table A-7. Octal Prefixes

Prefix	Description
`NO PREFIX`	VEE writes each octal number without any prefix. Only the digits `01234567` appear in the output.
`DEFAULT PREFIX`	For direct I/O paths, VEE prefixes each octal number with `#Q`. This supports the octal Non-Decimal Numeric data format defined by IEEE 488.2. For UNIX paths, 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`*	VEE prefixes each octal number with the characters specified in *`string`*.

The transaction in Figure A-14 specifies the default prefix and six digits:

```
WRITE TEXT X OCT:6 PREFIX EOL
```

Figure A-14. A `WRITE TEXT OCTAL` Transaction

If `X` in Figure A-14 has this value:

`15` *the value 15 decimal*

VEE writes this to direct I/O paths:

`#Q000017` *exactly six digits plus prefix*

Using the same transaction and data, VEE writes this to UNIX paths:

`000017` *exactly six digits*

The transaction in Figure A-15 specifies a custom prefix and ten digits:

```
WRITE TEXT X OCT:10 PREFIX:"oct>" EOL
```

Figure A-15. A `WRITE TEXT OCTAL` **Transaction**

If `X` in Figure A-15 has this value:

`15` *the Integer value 15 decimal*

VEE writes this to UNIX paths and direct I/O paths:

```
oct>000017
```

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 `-2 147 483 648 to +2 147 483 647`. 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 A-8 as part of a `WRITE TEXT HEX` transaction.

Table A-8. Hexadecimal Prefixes

Prefix	Description
`NO PREFIX`	VEE writes each hexadecimal number without any prefix. Only the digits `0123456789abcdef` appear in the output.
`DEFAULT PREFIX`	For direct I/O paths, VEE prefixes each hexadecimal number with `#H`. This supports the hexadecimal Non-Decimal Numeric data format defined by IEEE 488.2. For UNIX paths, VEE prefixes each hexadecimal number with `0x`.
`PREFIX:`*`string`*	VEE prefixes each hexadecimal number with the characters specified in *`string`*.

The transaction in Figure A-16 specifies the default prefix and six digits:

```
WRITE TEXT X HEX:6 PREFIX EOL
```

Figure A-16. A `WRITE TEXT HEX` Transaction

If `X` in Figure A-16 has this value:

`15` *the Integer value 15 decimal*

VEE writes this to direct I/O paths:

`#H00000f` *exactly six digits plus prefix*

Using the same transaction and data, VEE this to UNIX paths:

`0x00000f` *exactly six digits plus prefix*

The transaction in Figure A-17 specifies a custom prefix and three digits:

```
WRITE TEXT X HEX:3 PREFIX:"hex>" EOL
```

Figure A-17. A `WRITE TEXT HEX` Transaction

If `X` in Figure A-17 has this value:

`15` *the Integer value 15 decimal*

VEE writes this to UNIX paths and direct I/O paths:

`hex>00f` *exactly three digits plus prefix*

The prefix written by `DEFAULT PREFIX` depends on the destination, but the prefix written by `PREFIX:` *`string`* is independent of the destination.

REAL32 and REAL64 Format

`WRITE TEXT REAL32` transactions are of the form shown below:

`WRITE TEXT` *`ExpressionList`* `REAL32`

The type of Real number generated by this transaction is a 32-bit IEEE 754 floating-point number. The range of these numbers is:

`-3.40282347E38`

to

`3.40282347E38`

`WRITE TEXT REAL64` transactions are of the form shown below:

`WRITE TEXT` *`ExpressionList`* `REAL64`

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`

to

`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 A-9 as part of a `WRITE TEXT REAL` transaction.

Table A-9. `REAL` Notations

Notation	Description
`STANDARD`	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`	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`	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 A-18 specify `STANDARD` notation and four significant digits.

```
WRITE TEXT X REAL32 STD:4 EOL
WRITE TEXT Y REAL64 STD:4 EOL
WRITE TEXT Z REAL32 STD:4 EOL
```

Figure A-18. Three `WRITE TEXT REAL` Transactions

If `X`, `Y` and `Z` in Figure A-18 have these values:

`1.23456E2`	*the Real32 value of X*
`1.23456E09`	*the Real64 value of Y*
`1.23`	*the Real32 value of Z*

VEE writes:

`123.5`	*mantissa rounded as required*
`1.235E+09`	*large numbers in exponential notation*
`1.23`	*never any trailing zeros*

The transactions in Figure A-19 specify `FIXED` notation and four fractional digits.

```
WRITE TEXT X REAL64 FIX:4 EOL
WRITE TEXT Y REAL32 FIX:4 EOL
WRITE TEXT Z REAL64 FIX:4 EOL
```

Figure A-19. Three `WRITE TEXT REAL` Transactions

If `X`, `Y` and `Z` in Figure A-19 have these values:

`1.2345678E2`	*the Real64 value of X*
`1.2345678E-09`	*the Real32 value of Y*
`1.23`	*the Real64 value of Z*

VEE writes:

`123.4568`	*mantissa rounded as required*
`0.0000`	*small numbers round to zero*
`1.2300`	*trailing zeros added as required*

The transactions in Figure A-20 specify `SCIENTIFIC` notation and four fractional digits.

```
WRITE TEXT X REAL32 SCI:4 EOL
WRITE TEXT Y REAL64 SCI:4 EOL
WRITE TEXT Z REAL32 SCI:4 EOL
```

Figure A-20. Three `WRITE TEXT REAL` Transactions

If X, Y and Z in Figure A-20 have these values:

`1.2345678E2`	*the Real32 value of X*
`-1.2345678E-09`	*the Real64 value of Y*
`0`	*the Real32 value of Z*

VEE writes:

`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 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 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 `REAL64` format. Each constituent Real value of the multi-field data types is written with the same output rules that apply to an individual `REAL64` 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` ⇒ `Instrument Manager` 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 A-21 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 A-21. A `WRITE TEXT COMPLEX` Transaction

If the `Multi-Field Format` is set to `(...) Syntax` and `X` in Figure A-21 has this value:

`( -1.23456 , 9.8 )`   *the Complex value of X*

VEE writes:

```
(   -1.235 ,      +9.800)
 ^        ^   ^        ^
```

If the `Multi-Field Format` is set to `Data Only` and `X` in Figure A-21 has the same value, VEE writes:

```
    -1.235,    +9.800
^         ^ ^         ^
```

The caret characters (^) *are not* actually written by VEE, but are shown to help you visualize the field width. The characters to the left of + are spaces (ASCII 32 decimal).

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. Phase units are independent of the units set by `Trig Mode` in `Properties`. See Table A-10.

Table A-10. `PCOMPLEX` **Phase Units**

Unit	Description
`DEG`	Degrees
`RAD`	Radians
`GRAD`	Gradians

The first transaction in Figure A-22 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 A-22. Two `WRITE TEXT PCOMPLEX` **Transactions**

If the `Multi-Field Format` is set to `Data Only` and `X` in Figure A-22 has this value:

`(-1.23456, @90)` *the PComplex value of X, phase in degrees*

VEE writes:

```
1.23456,-90
1.23456,-1.570796326794897
```

The transaction in Figure A-23 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 A-23. A `WRITE TEXT PCOMPLEX` **Transaction**

If the `Multi-Field Format` is set to `(...) Syntax` and `X` in Figure A-23 has this value:

`(-1.23456 , @9.8)`   *the PComplex value of X, angle in radians*

VEE writes:

```
(    +1.235 , @    +0.375)
 ^         ^   ^         ^
```

VEE normalizes all PComplex numbers to yield a positive magnitude and a phase between +π and -π.

If the `Multi-Field Format` is set to `Data Only` and `X` in Figure 12-23 has the same value, VEE writes:

```
    +1.235,    +0.375
^         ^ ^         ^
```

The caret characters (^) *are not* actually written by VEE, but 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
```

`TIME STAMP` can write it using any of the `Time` and `Date` notations in Table A-11.

.

Table A-11. `Time` and `Date` **Notations**

Notation	Result
Date with Weekday DD/Month/YYYY	Thu 04/Apr/1999
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/1999 15:44:29
Time&Date with HH:MM:SS, 24 HOUR and Weekday DD/Month/YYYY	15:44:29 Thu 04/Apr/1999
Delta Time with HH:MM:SS	17446270:44:29
Date with Weekday DD/Month/YYYY	Thu 04/Apr/1999
Date with DD/Month/YYYY	04/Apr/1999
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.

VEE 5 or earlier Execution Mode converts each item in *`ExpressionList`* to an `Int16` (16-bit two's complement integer) and writes the least-significant 8-bits. VEE 6 Execution Mode converts each item in *`ExpressionList`* to a `UInt8` (8-bit two's complement integer) and writes it. This is a transaction for writing single characters to a instrument. Each expression in *ExpressionList* must be a scalar.

In VEE 6 Execution Mode, a value greater than 256 causes an error. For example, in VEE 5 and lower Execution Modes, the transactions in Figure A-24 produce the following character data output:

```
ABCAA
```

```
WRITE BYTE 65,66,67
WRITE BYTE 65+1024,65+2048
```

Figure A-24. Two `WRITE BYTE` **Transactions**

In VEE 6 Execution Mode, the second transaction in Figure A-24 will cause an error.

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.

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).

Indexing of items in *ExpressionList2* is zero-based.

The transactions in Figure A-25 illustrate the behavior of `CASE` format.

```
WRITE CASE 2,1 OF "Str0","Str1","Str2"
WRITE CASE X OF 1,1+A,3+A
```

Figure A-25. Two `WRITE CASE` **Transactions**

If the variables in Figure A-25 have these values:

`2` *the Real32 value of X*
`0.1` *the Real64 value of A*

VEE writes:

```
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 VEE data types:

- `BYTE` - 8-bit unsigned 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 `REAL64s`
- `PCOMPLEX` -equivalent to two `REAL64s`
- `COORD` - equivalent to two or more `REAL64s`

Note

VEE 5 and lower Execution Modes store and manipulate 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. VEE 5 and lower Execution Modes perform 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.

`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.

In VEE 6 Execution Mode, the data is converted to the appropriate type, and an error is generated if the data is out of range.

`BINARY` encoded transactions convert each of the values specified in *`ExpressionList`* to the 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.

`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 VEE data types:

- BYTE - 8-bit unsigned 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 REAL64s
- PCOMPLEX -equivalent to two REAL64s
- COORD - equivalent to two or more REAL64s

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-GPIB BINBLOCK

If the object is *not* Direct I/O to GPIB, 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 GPIB 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.

GPIB BINBLOCK

If the object is `Direct I/O` to GPIB, 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` ⇒ `Instrument Manager` menu selection.

If `Conformance` is set to `IEEE 488.2`, `WRITE BINBLOCK` *always* writes an IEEE 488.2 Definite Length Arbitrary Block Response Data block.

If `Conformance` is set to `IEEE 488`, 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>`. (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 VEE text representation.

This representation retains all the 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:

```
(Real
(data 1.2345)
)
```

STATE Encoding

`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 `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` ⇒ `Instrument Manager` 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. See *Controlling Instruments with VEE* for information about `WRITE STATE` transactions.

REGISTER Encoding

`WRITE REGISTER` is used to write values into a VXI instrument'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 instrument. The name refers to a specific address within a instrument's register space. Specific data types for `WRITE REGISTER` transactions are:

- `BYTE` - 8 bit unsigned byte
- `WORD16` - 16-bit two's complement integer
- `WORD32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating point number
- `WORD32*2` - two 32-bit two's complement integers in adjacent elements of an Int32 array.
- `REAL64` - 64-bit

These data types are also specified during configuration of a VXI instrument 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, etc. 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

`WRITE MEMORY` is used to write values into a VXI instrument'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 instrument. The name refers to a specific address within a instrument's extended memory. Specific data types for `WRITE MEMORY` transactions are:

- `BYTE` - 8 bit unsigned byte
- `WORD16` - 16-bit two's complement integer
- `WORD32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating point number
- `WORD32*2` - two 32-bit two's complement integers in adjacent elements of an Int32 array.
- `REAL64` - 64-bit IEEE 754 floating point number.

These data types are also specified during configuration of a VXI instrument 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, etc. 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

`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
```

VEE 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`. See Table A-12.

Table A-12. HP 98622A GPIO Control Lines

Value Written	CTL1	CTL0
0	0	0
1	0	1
2	1	0
3	1	1

In Table A-12, `1` indicates a control line is asserted and `0` indicates 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.

READ Transactions

See Table A-13 for Read Encodings and Formats.

Table A-13. READ Encodings and Formats

Encodings	Formats
TEXT	CHAR TOKEN STRING QUOTED STRING INT16 INT32 OCTAL HEX REAL32 REAL64 COMPLEX PCOMPLEX COORD TIME STAMP
BINARY	STR BYTE INT16 INT32 REAL32 REAL64 COMPLEX PCOMPLEX COORD
BINBLOCK	BYTE INT16 INT32 REAL32 REAL64 COMPLEX PCOMPLEX COORD
CONTAINER	Not Applicable

Table A-13. READ Encodings and Formats

Encodings	Formats
IOSTATUS	Not Applicable
REGISTER[a]	BYTE WORD16 WORD32 REAL32
MEMORY[a]	BYTE WORD16 WORD32 WORD32*2 REAL32 REAL64

a. Direct I/O to VXI only.

TEXT Encoding

READ TEXT transactions can 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 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 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 A-14 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 A-14. Formats for READ TEXT Transactions

Format	Description
CHAR	Reads *any* 8-bit character.
TOKEN	Reads a contiguous list of characters as a unit 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.
INTEGER16	Reads a list of characters and interprets them as a decimal or non-decimal representation of a 16-bit integer. The only characters considered to be part of a decimal INTEGER are 0123456789-+. VEE recognizes the prefix 0x (hex) and all the Non-Decimal Numeric formats specified by IEEE 488.2: #H (hex), #Q (octal), #B (binary).
INTEGER32	Reads a list of characters and interprets them as a decimal or non-decimal representation of a 32-bit integer. The only characters considered to be part of a decimal INTEGER are 0123456789-+. 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. VEE also recognizes the IEEE 488.2 Non-Decimal Numeric prefix #Q for octal numbers.

Table A-14. Formats for READ TEXT Transactions

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. VEE also recognizes 0x and the IEEE 488.2 Non-Decimal Numeric prefix #H for hexadecimal numbers.
REAL32	Reads a list of characters and interprets them as the decimal representation of a Real 32-bit (floating-point) number. All common notations are recognized including leading signs, signed exponents and decimal points. The characters recognized to be part of a REAL32 are 0123456789-+.Ee. VEE also recognizes certain characters as suffix multipliers for Real numbers (see Table A-15).
REAL64	Reads a list of characters and interprets them as the decimal representation of a Real 64-bit (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. VEE also recognizes certain characters as suffix multipliers for Real numbers (see Table A-15).
COMPLEX	Reads the equivalent of two REAL64s 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 REAL64s 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.

Table A-14. Formats for `READ TEXT` Transactions

Format	Description
`COORD`	Reads the equivalent of two or more `REAL64`s and interprets them as rectangular coordinates.
`TIME STAMP`	Reads one of the specified 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`

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
INT16
INT32
OCTAL
HEX
REAL32
REAL64
```

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 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. VEE stops reading characters as soon as the `READ` is satisfied. All characters are read and 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`, 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 `INT32` format. Assume 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 INT32` 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 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) and 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 A-26.

```
"This is in quotes."  This is not.
```

Figure A-26. Quoted and Non-Quoted Data

Assume that you read the file shown in Figure A-26 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.
```

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 A-26 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 A-26 using `From File` with these transactions:

```
READ TEXT x TOKEN
READ TEXT y QSTR
```

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.

`ARRAY:1` is a one-dimensional array with one element. 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
```

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). 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. `ARRAY:1` is a one-dimensional array with one

element. 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 the contents shown in Figure A-27:

```
A phrase.
"A phrase."
Tab follows.
XOXXOOXXXOOOXXXX
XAXXBCXXXDEF
```

Figure A-27. `READ TOKEN` **Data**

The file contains only the letter `O`, not the digit zero.

There is an invisible linefeed character at the end of each of the first four lines of the file in Figure A-27 that shows the file as it would appear in a text editor such as `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 text and separate out individual words.

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 in Figure A-27 using `SPACE DELIM` with this transaction:

```
READ TEXT a TOKEN ARRAY:8
```

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 VEE reads the data shown in Figure A-27 using `INCLUDE CHARS` with this transaction:

```
READ TEXT a TOKEN INCLUDE:"X" ARRAY:7
```

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 VEE reads the data shown in Figure A-27 using `INCLUDE CHARS` with this transaction:

```
READ TEXT a TOKEN INCLUDE:"OXZ" ARRAY:4
```

the variable `a` contains these values:

```
a[0] = XOXXOOXXXOOOXXXX
a[1] = X
a[2] = XX
a[3] = XXX
```

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 A-28.

```
111 222 333 444 555
```

Figure A-28. `READ TOKEN` **Data**

If you try to read the file in Figure A-28 using this transaction:

```
READ TEXT x,y,z TOKEN INCLUDE:"1234567890"
```

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 A-28 using this transaction:

```
READ TEXT x TOKEN INCLUDE:"1234567890" ARRAY:3
```

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 A-27 using `EXCLUDE` with this transaction:

```
READ TEXT a TOKEN EXCLUDE:"X" ARRAY:8
```

the variable `a` contains these values:

```
a[0] = A phrase.<LF>"A phrase."<LF>Tab follows .<LF>
a[1] = O
a[2] = OO
a[3] = OOO
a[4] = <LF>
a[5] = A
a[6] = BC
a[7] = DEF<LF>
```

Assume the data shown in Figure A-29 is sent to VEE from an instrument.

```
++1.23++4.98++0.45++2.34++0.01++23.45++12.2++
```

Figure A-29. `READ TOKEN` **Data**

If VEE reads the data in Figure A-29 with this transaction:

```
READ TEXT x TOKEN EXCLUDE:"+" ARRAY:7
```

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
```

Even though seven "numbers" were available, only six were read. At the end of this transaction, VEE has read seven tokens terminated by the `+`, including the first character which was terminated before it was filled with any data.

Note

The behavior of `EXCLUDE CHARS` is different between VEE 5 Execution Mode and later and VEE 4 Execution Mode and earlier. See "READ TEXT Transactions" on page 26 for a description of this difference.

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. `ARRAY:1` is a one-dimensional array with one element. VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

This transaction reads all incoming characters and returns strings. Leading spaces are deleted. The following discussion pertains to instrument I/O paths only, such as GPIB or VXI. All other I/O paths, such as files or name12-pipes, will not treat Quoted Strings specially. See "Effects of Quoted Strings" on page 428 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 VEE. 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 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 want to read the following string data using `READ TEXT STRING` transactions:

```
Simple string.
Random \n % $ * ‘A'
"In quotes."
"In quotes
with control."
"In quotes\nwith escape."
```

If you read the string data using this transaction:

```
READ TEXT x STR ARRAY:5
```

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 string data using this transaction:

```
READ TEXT x STR MAXFW:16 ARRAY:5
```

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
```

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. `ARRAY:1` is a one-dimensional array with one element. 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. See “Effects of Quoted Strings” on page 428 for details about the effects of double quoted strings on `READ TEXT STRING`.

Also see “Effects of Control and Escape Characters” on page 436.

INT16 and INT32 Formats

`READ TEXT INT16` and `READ TEXT INT32` transactions are of this form:

```
READ TEXT VarList INT16(or INT32) ARRAY:NumElements
-or-
READ TEXT VarList INT16(or INT32) 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. `ARRAY:1` is a one-dimensional array with one element. VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

`READ TEXT INT16` transactions interpret incoming characters as 16-bit, two's complement integers. The valid range for these integers is 32767 to -32768. Any numbers outside this range wrap around so there is never an overflow condition. For example, 32768 is interpreted as -32768.

`READ TEXT INT32` 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, VEE discards any leading characters that are not recognized as part of a number. Once VEE starts building a number, any character that is not recognized as part of a number terminates the `READ` for that number. Table A-15 shows the only combinations of characters that are recognized as part of an `INT16` or `INT32`.

Table A-15. Characters Recognized as Part of an INT16 or INT32:

Notation	Characters Recognized
Decimal	Valid characters are `±0123456789`. Leading zeros are *not* interpreted as an octal prefix as they are in VEE data entry fields.
VEE hexadecimal	VEE interprets `0x` as a prefix for a hexadecimal number. Valid characters following the prefix are `0123456789aAbBcCdDeEfF`.
IEEE 488.2 binary	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	VEE interprets `#q` or `#Q` as a prefix for an octal number. Valid characters following the prefix are `01234567`.
IEEE 488.2 hexadecimal	VEE interprets `#h` or `#H` as a prefix for a hexadecimal number. Valid characters following the prefix are `0123456789aAbBcCdDeEfF`.

All 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
```

where:

`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. `ARRAY:1` is a one-dimensional array with one element. 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 VEE reads the following octal data:

```
37723745621456724377 7
```

using this transaction:

```
READ TEXT x OCT MAXFW:21
```

VEE reads all the digits in octal data, 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 VEE Integer, which contains 32 bits. Eleven octal digits yield 33 bits and the most significant bit is dropped to fit the value in an 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 the following octal data 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>
```

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. `ARRAY:1` is a one-dimensional array with one element. 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 VEE reads the following hexadecimal data:

`ad2469Ff725BCdef37964` *hexadecimal data*

using this transaction:

```
READ TEXT x HEX MAXFW:21
```

VEE reads all the digits in the hexadecimal data, 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 VEE Integer, which contains 32 bits. Eight hexadecimal digits yields exactly 32 bits. There is no possibility of overflow.

Assume VEE reads the same hexadecimal data, 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 comprising 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 VEE reads the same hexadecimal data, 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 comprising eight hexadecimal digits. The last five digits (`37946`) are not read.

REAL32 and REAL64 Format

`READ TEXT REAL32` and `READ TEXT REAL64` transactions are of this form:

```
READ TEXT VarList REAL32(or REAL64) ARRAY:NumElements
-or-
READ TEXT VarList REAL32(or REAL64) 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. `ARRAY:1` is a one-dimensional array with one element. VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

The decimal number read by this transaction is interpreted as a VEE `Real32`, which is a 32-bit IEEE 754 floating-point number. The range of these numbers is:

```
-1.175 494 35E-38
-3.402 823 47E+38
to
3.402 823 47E+38
1.175 494 35E-38
```

The decimal number read by this transaction is interpreted as a VEE `Real64`, 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
 to
 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 VEE reads the following real data:

`1.2345678901234567890` *real number data*

using this transaction:

```
READ TEXT x REAL64 MAXFW:19
```

VEE reads all the digits in the real data, 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 inadvisable.

Assume VEE reads the same real number data, but with a different `MAX NUM CHARS`, as in this transaction:

```
READ TEXT x REAL64 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 REAL64` and `REAL32` 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 A-16 are also recognized. The suffix character must immediately follow the last digit of the number with no intervening white space.

Table A-16. 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 following Text data represents six real numbers:

```
1001
+1001.
1001.0
1.001E3
+1.001E+03
1.001K
```

If VEE reads the real text data with this transaction:

```
READ TEXT x REAL64 ARRAY:6
```

then each element of the Real variable `x` contains the value `1001`.

If VEE reads the same data with this transaction:

```
READ TEXT x REAL64 MAXFW:20 ARRAY:6
```

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 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 VEE data types Complex, PComplex and Coord are composed of multiple Real64 numbers, the COMPLEX, PCOMPLEX and COORD formats are compound forms of the REAL64 format. Each constituent Real value of the multi-field data types is read using the same rules that apply to an individual REAL64 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 want to read a file containing the following data containing parentheses:

```
(1.23 , 3.45  (6.78 , 9.01) (1.23 , 4.56)
```

If VEE reads the data with this transaction:

```
READ TEXT x,y CPX
```

the variables x and y contain these Complex values:

```
x = (1.23 , 3.45)
y = (1.23 , 4.56)
```

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. `ARRAY:1` is a one-dimensional array with one element. 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 VEE data type with the same name:

- `BYTE` - 8-bit unsigned 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 `REAL64`s

- `PCOMPLEX` -equivalent to two `REAL64`s
- `COORD` - equivalent to two or more `REAL64`s

Note

VEE 5 and lower Execution Modes store and manipulate 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. VEE 5 and lower Execution Modes perform 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 VEE 5 and lower Execution Modes. This conversion assumes that the `INT16` data was *signed* data. If you need the resulting `INT32` data in *unsigned* form, pass the data through a formula object with the formula

```
BITAND(a, 0xFFFF)
```

`REAL32` values from an instrument are *individually* converted to `REAL64` values by VEE 5 and lower.

VEE 6 Execution Mode retains the data type.

`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`

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. `ARRAY:1` is a one-dimensional array with one element. 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-VEE applications.

BINBLOCK Encoding

`READ BINBLOCK` transactions are of this form:

`READ BINBLOCK` *`VarList DataType`* `ARRAY:`*`NumElements`*

`VarList` is a single variable or a comma-separated list of variables.

`DataType` is one of these pre-defined VEE data types:

- `BYTE` - 8-bit unsigned 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

`NumElements` is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *`VarList`*. The number of columns is equal to the number of channels contained by the binblock. The number of rows is equal to the number of readings per channel. Only the first dimension can have an asterisk rather than a number.

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`

If the transaction is configured to read a one-dimension array, for a single channel, the single dimension represents rows and can have an asterisk.

For example, the following transaction, reading from a file:

```
READ BINBLOCK a REAL64 ARRAY:*,10
```

will read until EOF is encountered, resulting in a two-dimensional array with 10 columns. Each column represents an instrument channel. The number of rows is dependent on the amount of data in each channel. The total number of data elements contained by the binblock must be evenly divisible by the number of columns, in this example: 10.

You do not need to specify any additional information about the format of incoming data as the block header contains sufficient information. `READ BINBLOCK` can read any of the block formats described previously with `WRITE BINBLOCK` transactions.

The following 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 REAL64
```

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 text representation written by `WRITE CONTAINER` transactions. No additional specifications, such as format, needs to be specified with `READ CONTAINER` since that information is part of the container.

REGISTER Encoding

READ REGISTER is used to read values from a VXI instrument'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 instrument. The name refers to a specific address within a instrument's register space. Specific data types for READ REGISTER transactions are:

- BYTE - 8 bit unsigned 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 instrument 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, etc. 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. ARRAY:1 is a one-dimensional array with one element. VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

MEMORY Encoding

`READ MEMORY` is used to read values from a VXI instrument'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 instrument. The name refers to a specific address within a instrument's extended memory. Specific data types for `READ MEMORY` transactions are:

- `BYTE` - 8 bit unsigned byte
- `WORD16` - 16-bit two's complement integer
- `WORD32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating point number
- `WORD32*2` - two 32-bit two's complement integers in adjacent elements of an Int32 array

`REAL64` - 64-bit IEEE 754 floating point number.

These data types are also specified during configuration of a VXI instrument 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, etc. 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. `ARRAY:1` is a one-dimensional array with one

element. 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,.

`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 A-17 illustrates how to interpret the value of `x` in this transaction:

```
READ IOSTATUS STS Bits a
```

Table A-17. `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 return 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 program execution until data is available.

Other Transactions

This section describes other VEE I/O transactions, including EXECUTE transactions, WAIT transactions, and SEND transactions.

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 A-18.

Table A-18. 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.

Table A-18. Summary of EXECUTE Commands

Commands	Description
Interface Operations	
CLEAR	For GPIB, clears all instruments by sending DCL (Device Clear). For VXI, resets the interface and runs the resource manager
TRIGGER	For GPIB, triggers all instruments 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 GPIB, releases the REN (Remote Enable) line and puts instrument into local mode.
REMOTE	For GPIB, asserts the REN (Remote Enable) line.
LOCAL LOCKOUT	For GPIB, sends the LLO (Local Lockout) message. Any instrument in remote at the time LLO is sent will lock out front panel operation.
ABORT	Clears the GPIB interface by asserting the IFC (Interface Clear) line.
LOCK INTERFACE	In a multiprocess system with shared resources, lets one process lock the resources for its own use during a critical section to prevent another process from trying to use them.
UNLOCK INTERFACE	In a multiprocess system where a process has locked shared resources for its own use, unlocks the resources to allow other processes access to them.

Table A-18. Summary of EXECUTE Commands

Commands	Description
Direct I/O to GPIB	
`CLEAR`	Clears instrument at the address of a `Direct I/O` object by sending the SDC (Selected Device Clear).
`TRIGGER`	Triggers the instrument at the address of a `Direct I/O` object by addressing it to listen and sending GET (Group Execute Trigger).
`LOCAL`	Places the instrument at the address of the `Direct I/O` object in the local state.
`REMOTE`	Places the instrument at the address of the `Direct I/O` object in the remote state.
Direct I/O to GPIO	
`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 VX*I	
`CLEAR`	Clears the VXI instrument associated with the `Direct I/O` object by sending the word-serial command Clear (0xffff).
`TRIGGER`	Triggers the VXI instrument associated with the `Direct I/O` object by sending the word-serial command Trigger (0xedff).
`LOCAL`	Places the VXI instrument associated with the `Direct I/O` object into local state by sending the word-serial command Clear Lock (0xefff).
`REMOTE`	Places the VXI instrument associated with the `Direct I/O` object into local state by sending the word-serial command Set Lock (0xeeff). in the remote state.

Table A-18. Summary of EXECUTE Commands

Commands	Description
Direct I/O to Serial Interfaces	
`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: A logical High for 400 milliseconds A logical Low for 60 milliseconds
`CLOSE`	Close the connection with the serial interface associated with the Direct I/O Object. A connection is reestablished at the next connection, if any.
Direct I/O, MultiInstrument Direct I/O, Interface Operations to GPIB, VXI, Serial, and GPIO	
`LOCK`	In a multiprocess system with shared resources, lets one process lock the resources for its own use during a critical section to prevent another process from trying to use them.
`UNLOCK`	In a multiprocess system where a process has locked shared resources for its own use, unlocks the resources to allow other processes access to them.

Details About GPIB

The `EXECUTE` commands used by `Direct I/O` to GPIB instruments 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.

Table A-19 through Table A-24 indicate the exact bus actions conducted by `Direct I/O` and `Interface Operations` `EXECUTE` transactions.

Table A-19. `EXECUTE ABORT` **GPIB Actions**

`Direct I/O`	`Interface Operations`
Not applicable.	IFC (≥ 100 µsec)
	REN
	ATN

Table A-20. `EXECUTE CLEAR` **GPIB Actions**

`Direct I/O`	`Interface Operations`
ATN	ATN
MTA	DCL
UNL	
LAG	
SDC	

Table A-21. `EXECUTE TRIGGER` **GPIB Actions**

Direct I/O	Interface Operations
ATN	ATN
MTA	GET
UNL	
LAG	
GET	

Table A-22. `EXECUTE LOCAL` **GPIB Actions**

Direct I/O	Interface Operations
ATN	REN
MTA	ATN
UNL	
LAG	
GTL	

Table A-23. `EXECUTE REMOTE` **GPIB Actions**

Direct I/O	Interface Operations
REN	REN
ATN	ATN
MTA	
UNL	
LAG	

Table A-24. `EXECUTE LOCAL LOCKOUT` **GPIB Actions**

Direct I/O	Interface Operations
Not applicable.	ATN LLO

Details About VXI

The `EXECUTE` commands used by `Direct I/O` to VXI instruments and `Interface Operations` are similar, but different. References to message-based VXI instruments apply to register-based instruments that are supported by I-SCPI.

- `Direct I/O` `EXECUTE` commands address a message-based VXI instrument to receive a word-serial command.
- `Interface Operations` `EXECUTE` commands affect the VXI interface directly and may affect VXI instruments 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.

Table A-25 through Table A-28 indicate the bus actions conducted by `Direct I/O` and `Interface Operations` `EXECUTE` transactions.

Table A-25. `EXECUTE CLEAR` **VXI Actions**

`Direct I/O`	`Interface Operations`
Word-serial command Clear(0xffff)	Pulse SYSRESET line, rerun Resource Manager

Table A-26. `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 A-27. `EXECUTE LOCAL` **VXI Actions**

`Direct I/O`	`Interface Operations`
Word-serial command Set Lock(0xeeff)	Not applicable.

Table A-28. 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 GPIB and message-based VXI instruments only)
- WAIT REGISTER (Direct I/O to VXI instruments only)
- WAIT MEMORY (Direct I/O to VXI instruments only)

WAIT INTERVAL transactions 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 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 one way 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 instrument. The name refers to a specific address within a instrument's A16 or extended memory.

`MASK:`*`Expression`* is an expression that evaluates to an integer. The integer will be used 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 instrument 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 of this form:

`SEND` *`BusCmd`*

`BusCmd` is one of the bus commands listed in Table A-29.

`SEND` transactions are used within `Interface Operations` objects to transmit low-level bus messages via a GPIB interface. These messages are defined in detail in IEEE 488.1.

Table A-29. 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 instrument-dependent information.
TALK	Addresses a instrument at the specified primary bus address (0-31) to talk.
LISTEN	Addresses a instrument 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 instruments to stop listening and sends UNL.
UNTALK	Forces all instruments to stop talking; sends UNT.
MY LISTEN ADDR	Addresses the computer running VEE to listen and sends MLA.
MY TALK ADDR	Addresses the computer running VEE to talk and sends MTA.

Table A-29. `SEND` Bus Commands

Command	Description
`MESSAGE`	Sends a multi-line bus message. Consult IEEE 488.1 for details. The multi-line messages 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

B

Troubleshooting Techniques

Troubleshooting Techniques

This appendix describes instrument control troubleshooting andcommon situations and possible recovery actions. Table B-1 addresses instrument control troubleshooting.

Table B-1. Instrument Control Troubleshooting

Problem	Remedy/Cause
Instruments do not respond at all.	The following conditions must be met: • Instruments must be powered up and connected to the interface by a functioning cable. The appropriate I/O libraries must be installed. • For `To/From VXIplug&play` objects: You must have installed and configured the appropriate VXI*plug&play* driver files for your instrument. Also, the correct VXI*plug&play* address string must be specified in the `Advanced Instrument Properties` dialog box for each instrument. The address for each instrument must be unique. • For `Direct I/O`, `Panel Driver`, and `Component Driver` objects: The interface logical unit and instrument addresses must match settings in the `Address` field of the `Instrument Properties` dialog box. The address for each instrument must be unique. Also, the `Live Mode` field in the `Advanced Instrument Properties` dialog box must be set to `ON`. • You or your system administrator must properly configure VEE to work with instruments. Normally this is done during VEE installation. See the installation guide.

Table B-1. Instrument Control Troubleshooting

Problem	Remedy/Cause
You cannot determine the instrument address.	For GPIO and serial interfaces, the instrument address is the same as the interface logical unit. GPIB instrument addresses are set by hardware switches or front panel commands. See the instrument's programming manual for details. VXI devices have logical addresses set by switches on the outside of the cards (usually the cards must be removed from the card cage to access the switches). See Chapter 3, "Configuring Instruments," for further information about configuring addresses.
You cannot determine the interface logical unit.	The interface logical units must be configured with the `I/O Config` utility supplied with the HP I/O libraries. See *Installing the Agilent I/O Libraries (VEE for Windows)* for further information. Table 5-2, "Recommended I/O Logical Units," on page 172 for recommended logical unit settings.

Table B-2 addresses general VEE troubleshooting.

Table B-2. VEE Troubleshooting

Problem	Possible Cause	Suggested Solutions
When running a program created in versions prior to VEE 6.0 in VEE 6 Execution Mode, the program does not operate as expected		See "Using VEE Execution Modes" on page 10 for possible solutions.
Your UserObject does not operate as expected.	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: Enable Show Execution 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 which includes features that let you add a control input terminal and edit properties.
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 does not 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.

Table B-2. VEE Troubleshooting

Problem	Possible Cause	Suggested Solutions
For Range or For Log Range does not 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.
VEE appears to hang -- the pointer is an hourglass.	Possible Cause 1:VEE is rerouting lines because you have Auto Line Routing set on and you moved an object. Possible Cause 2: VEE is printing the screen or the program. Possible Cause 3: You just Cut a large object or a large number of objects. VEE is saving the objects to the Paste buffer.	Wait. If the pointer does not change back to the crosshairs within a few minutes, type **CTRL+C** (or what your intr setting is in the terminal window from which you started VEE 6.0), close the VEE window, or kill the VEE process.
You cannot Open a program, Cut objects, or delete a line (the feature is grayed).	The program is still running.	Press Stop to stop the program, then try the action again.
You cannot Paste (the feature is grayed).	The Paste buffer is empty.	Cut, Copy, or Clone the object(s) again.
You cannot Cut, Create UserObject, or Add to Panel (the feature is grayed).	No objects are selected.	Select the 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. The buffer 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.

Table B-2. VEE Troubleshooting

Problem	Possible Cause	Suggested Solutions
You cannot get out of line drawing mode.		Double-click or press **Esc** to end line drawing mode.
You get a `Parse Error` object when you `Open` a program.		Replace the `Parse Error` object with a new object.
Your characters are not appearing correctly.	You have a non-USASCII keyboard.	See “Configuring VEE” on page 5 for recovery information.
Your colors outside of VEE are changing (although when you are in VEE , the VEE colors look normal).	Your color map planes are all used.	See “Configuring VEE” on page 5 for recovery information.

C

Instrument I/O Data Type Conversions

Instrument I/O Data Type Conversions

For instrument I/O transactions involving numeric data, VEE performs an automatic data-type conversion according to the rules listed below. (These data-type conversions are completely automatic. Normally, you will not need to be concerned with them.) These conversions only occur when running in VEE 5 and prior Execution Modes.

- On an input transaction (read), `Int16` or `Byte` values from an instrument are converted to `Int32` values, preserving the sign extension. Also, `Real32` values from an instrument are converted to 64-bit `Real` numbers.
- On an output transaction (write), `Int32` or `Real` values are converted to the appropriate output format for the instrument:
    - If an instrument supports the `Real32` format, VEE converts 64-bit `Real` values to `Real32` values, which are output to the instrument. If the `Real` value is outside of the range for `Real32` values, an error will occur.
    - If an instrument supports the `Int16` format, VEE truncates `Int32` values to `Int16` values, which are output to the instrument. `Real` values are first converted to `Int32` values, which are then truncated and output. However, if a `Real` value is outside the range for an `Int32`, an error will occur.
    - If an instrument supports the `Byte` format, VEE truncates `Int32` values to `Byte` values, which are output to the instrument. `Real` values are first converted to `Int32` values, which are then truncated and output. However, if a `Real` value is outside the range for an `Int32`, an error will occur.

D

Keys to Faster Programming

Keys To Faster Programs

This appendix gives guidelines to help improve VEE program performance. For general tips to increase the performance of your program, see `Improving the Performance of a VEE Program` under `How Do I` in *VEE Online Help.*

Note

If you developed programs on a version of VEE prior to VEE 6.0, see "Using VEE Execution Modes" on page 10 for information on converting your program to use the compiler.

The following constructs will help you get the most speed benefit from the compiler (when the `Execution Mode` is set to `VEE 4, VEE 5` or `VEE 6` in `File` ⇒ `Default Preferences`):

- **Look at line colors**

Lines are colored when VEE can determine the data type before execution. The more colored (non-black) lines, the faster the program will run.

- **Add Terminal Constraints**

Because UserFunctions can be called from multiple places, VEE cannot determine the input data types before the program runs. To speed up UserFunctions, whenever possible add terminal constraints on their data input terminals.

- **Use Declared Variables**

If you use global variables, use `Declare Variable` (located on the `Data` menu) when possible to declare the type and shape of your variables so VEE can infer types for them prior to execution. This technique also allows you to set the scope of your variables.

- **Eliminate the `Autoscale` control input**

A common programming practice is executing the `Autoscale` control input on graphical displays more often than necessary. If you can wait to execute `Autoscale` until after the display has finished updating, instead of after each point is plotted, your program will execute faster. You can eliminate the `Autoscale` control input by using the `Automatic Scaling` property (see the `Scales` tab) which can further improve execution speed.

- **Send a complete set of data**

On graphical displays, when the `Automatic Scaling` property is turned on (see the `Scales` tab), the program executes faster if a complete set of data is sent to the display. Then the display automatically rescales once. If a program sends one data point at a time to the display, the display may automatically rescale after each data point which will slow down program execution. In this case, use a `Collector` object to create an array and then send the array to the display.

- **Execute the display only once**

If a display is showing the final output of a loop, but not tracking data generated for each iteration of the loop (for example, an `AlphaNumeric` object not a `Logging AlphaNumeric`), do not have it execute every time in the loop. Connect the iterator's sequence output pin to the display's sequence input pin so the display only executes the last time.

- **Turn debugging features off**

Once you know the program is running correctly, run the program with debugging features off. Use `File` ⇒ `Default Preferences` and select `Disable Debug Features` in the `Debug` group.

You can also use the `-r` option. Because no
debug instructions are generated in those modes, your program will run a little faster. However, you will not be able to perform any debugging actions such as, pausing, stepping, `Breakpoints`, `Line Probe`, `Show Data Flow` and `Show Execution Flow`.

E

ASCII Table

ASCII Table

This appendix contains reference tables of ASCII 7-bit codes.

Table E-1. ASCII 7-bit Codes

	Binary	Oct	Hex	Dec	GPIB 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	

Table E-1. ASCII 7-bit Codes

	Binary	Oct	Hex	Dec	GPIB Msg
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	
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

Table E-1. ASCII 7-bit Codes

	Binary	Oct	Hex	Dec	GPIB Msg
*	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
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

Table E-1. ASCII 7-bit Codes

	Binary	Oct	Hex	Dec	GPIB Msg
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
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

Table E-1. ASCII 7-bit Codes

	Binary	Oct	Hex	Dec	GPIB Msg
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
`	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

Table E-1. ASCII 7-bit Codes

	Binary	Oct	Hex	Dec	GPIB Msg
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
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	

Index

Symbols

Numerics

A

B

C

D

E

F

G

Q

R

W

X