Using HP VEE-Engine and HP VEE-Test

HP Part No. E2100-90011
Printed in USA December 1992

Edition 2

Notice

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

Hewlett-Packard Company (HP) shall not be liable for any errors contained in this document. HP MAKES NO WARRANTIES OF ANY KIND WITH REGARD TO THIS DOCUMENT, WHETHER EXPRESS OR IMPLIED. HP SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. HP shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory, in connection with the furnishing of this document or the use of the information in this document.

Warranty Information

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

Restricted Rights Legend

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DoD agencies, Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

Use of this manual and magnetic media supplied for this product are restricted. Additional copies of the software can be made for security and backup purposes only. Resale of the software in its present form or with alterations is expressly prohibited.

Printing History

Edition 1 - April 1991
Edition 2 - December 1992

Conventions Used in this Manual

This manual uses the following typographical conventions:

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

Contents

Figures

Tables

1

Introduction

About This Manual

This manual gives detailed information on using HP VEE for tasks that you may want to perform. If you are new to HP VEE, you may want to start by working through the examples in *Getting Started with HP VEE*. For reference information on specific HP VEE features, refer to the *HP VEE Reference* manual. If you haven't already installed HP VEE, refer to *Installing HP VEE*.

What is HP VEE?

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

Note

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

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

The HP VEE family includes three separate products:

- **HP VEE-Engine** is designed for the engineer and scientist. It allows you to analyze and display data stored in a file, input by the user, or generated mathematically.
- **HP VEE-Test** is designed for the test and measurement professional. It includes everything in HP VEE-Engine, plus it allows you to communicate with instruments from the visual environment.
- **HP VEE-Test/RunOnly** is a *run-only* environment that runs models developed with HP VEE-Test. (Models developed with HP VEE-Engine also will run.) No editing capabilities are provided by HP VEE-Test/RunOnly.

Some HP VEE Concepts

The rest of this chapter covers some key concepts for understanding HP VEE.

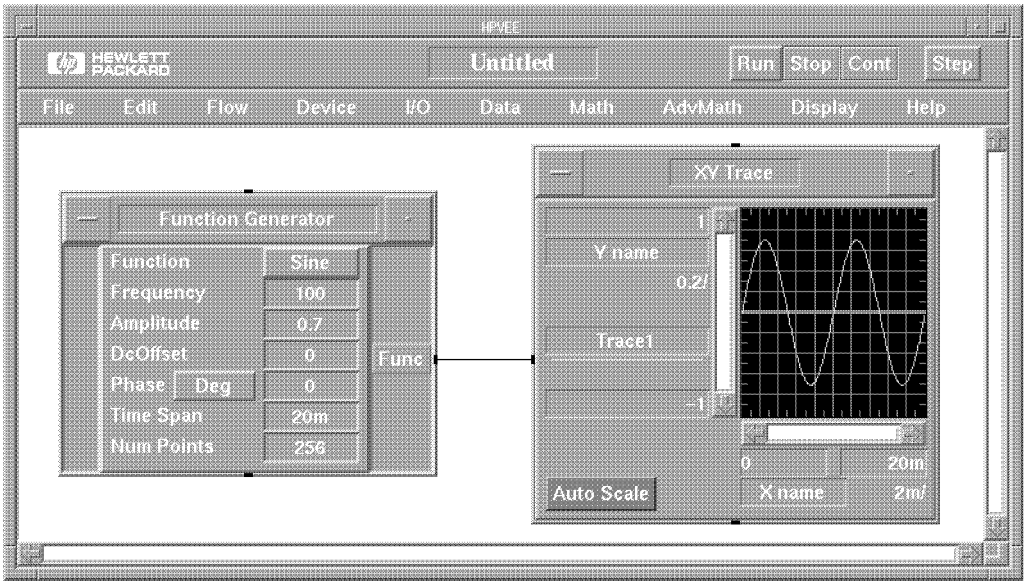
Models

To solve your engineering problem with HP VEE you create a graphical **model**, which is actually an executable block diagram. This is the key difference between HP VEE and a text-based programming language. With a text-based programming language, first you create a block diagram to define the problem, then you have to translate that definition into lines of code. *With HP VEE you simply create the block diagram in your HP VEE work area, and then execute it!* This enables you to prototype your solution quickly, and then go on to use the solution you developed without writing any code.

Objects

The "blocks" in the block diagram (your HP VEE model) are called **objects**. If you have worked through some of the examples in *Getting Started with HP VEE*, you are already familiar with creating a model from individual objects. However, you should know that developing an HP VEE model is not the same as writing an "iconic" program. Each HP VEE object may take the place of hundreds of lines of code. But seeing is believing, so let's look at an example.

The following HP VEE model consists of just two objects:



- The `Function Generator` object generates a sine wave, in this case with a frequency of 100 Hz and an amplitude of 0.7.
- The `XY Trace` object displays the sine wave output by the `Function Generator` object.

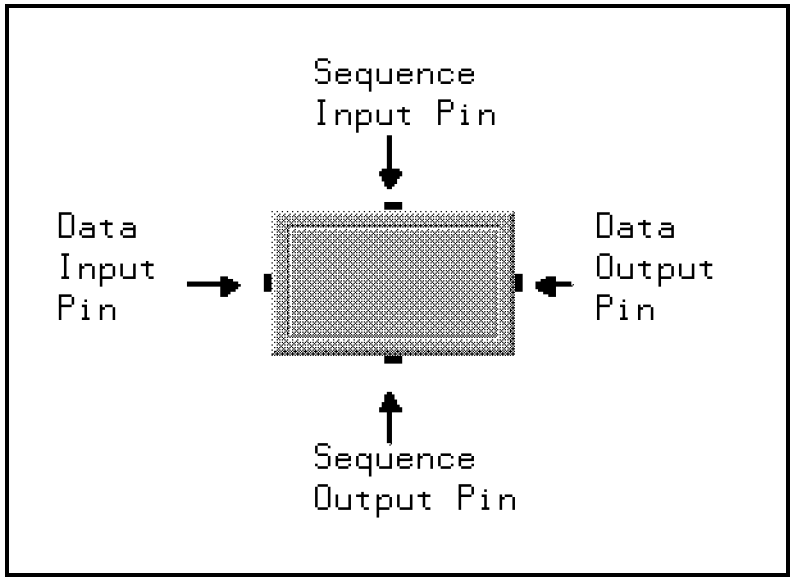
You could write a program in HP BASIC, C, or some other text-based programming language to do the same thing, but it would take hundreds of lines of code, and perhaps several hours to do. With HP VEE, this model took only a couple of minutes to create.

Objects can have either of two views. In the above example, both objects are shown in their **open view** — all of the details are showing. However, in the **icon view**, the details of the object are not shown — only a name or picture to identify it. Use iconified objects to simplify the appearance of a model.

Pins

In order to construct a model, you must connect objects with lines between their input and output **pins**. Objects can have four kinds of pins:

- Pins on the left-hand side of an object, if present, are **data input pins**.
- Pins on the right-hand side of an object, if present, are **data output pins**.
- The pin on the top of an object, if present, is the **sequence input pin**.
- The pin on the bottom of an object, if present, is the **sequence output pin**.



Some objects have all four kinds of pins, while others have only one or two kinds of pins. In an object's open view, the input and output pins can be represented as **input terminals** and **output terminals**, which display information about the pin. The terminals are visible only in the open view, and only if `Show Terminals` is active. You can display and edit the terminal information by double clicking on the terminal's information area (not the pin). Refer to "Working with Open Views" in chapter 2 for more information.

Data Flow

In a text-based programming language, the order in which program statements execute is determined by a set of sequence and selection rules. Generally, statements execute in the order they appear in the program, except where branching statements cause execution to "jump" to another statement or section of code.

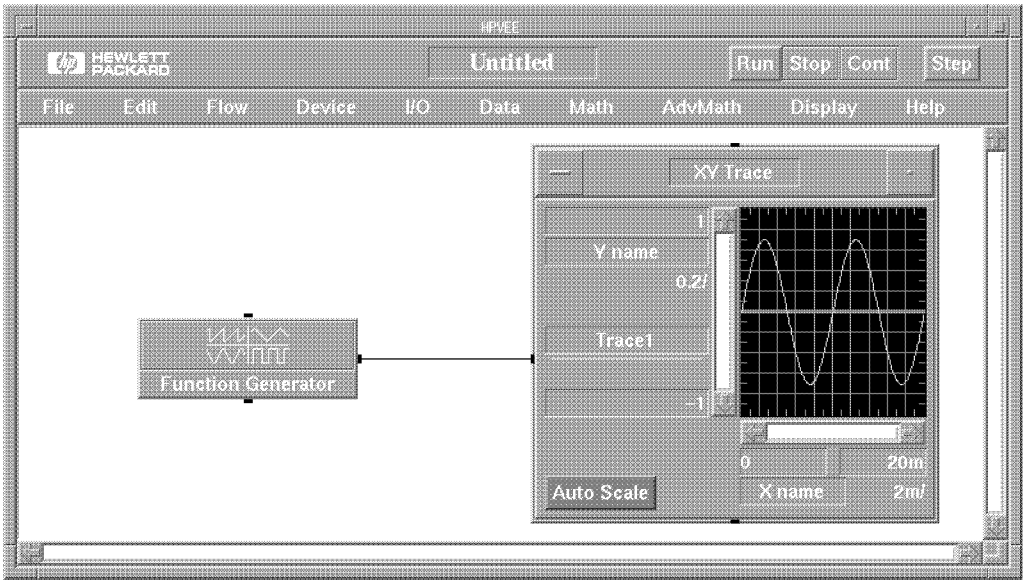
In HP VEE the general flow of execution through a model is called **propagation**. Propagation follows a set of rules that are covered in detail in the "Understanding Propagation" section in chapter 3. Propagation through a model is not determined by the geographic locations of the objects in the model, but rather by the way the objects are connected. Propagation is primarily determined by **data flow**, which is determined by how the data input and output pins of the objects are connected. Here are the rules of data flow:

- *Data flows from left to right through an object.* This means that on all objects with data pins, the left data pins are inputs and the right data pins are outputs.
- *All of an object's data input pins must be connected.* Otherwise an error will occur when the model is run.
- *An object will not execute until all of its data input pins have received data.*
- *An object finishes executing only after all appropriate data output pins have been activated.*

You can change the order of execution by using sequence input and output pins. However, it is generally best to avoid using sequence pins unless they are really needed. (Refer to chapter 3 for further information.) *If possible, let data flow control the execution of your model.*

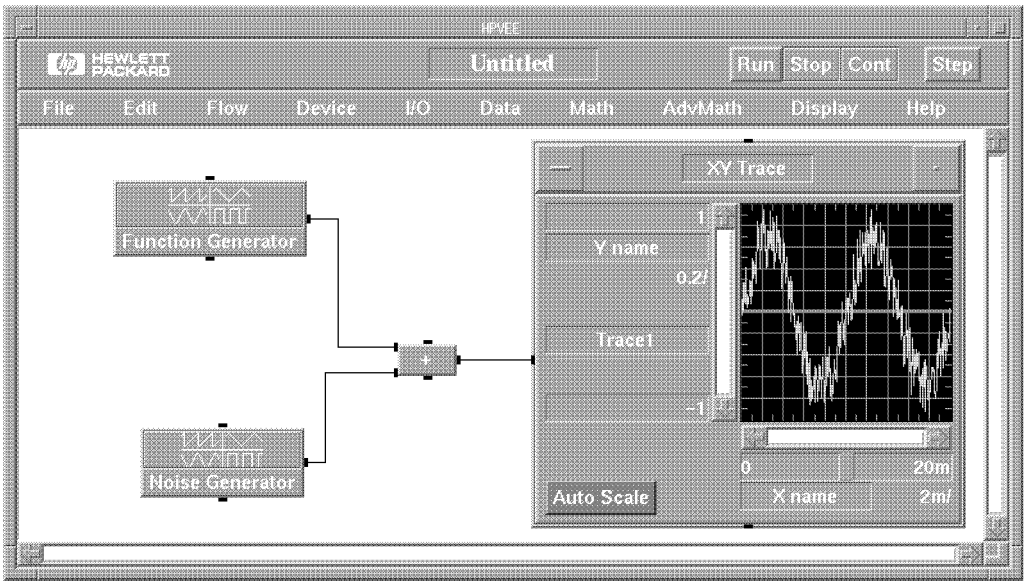
Some examples will show how models can operate with the order of execution determined solely by data flow.

We've already looked at one example — the `Function Generator` object connected to the `XY Trace` object. The following is another view of this model with the `Function Generator` object shown in its icon view.



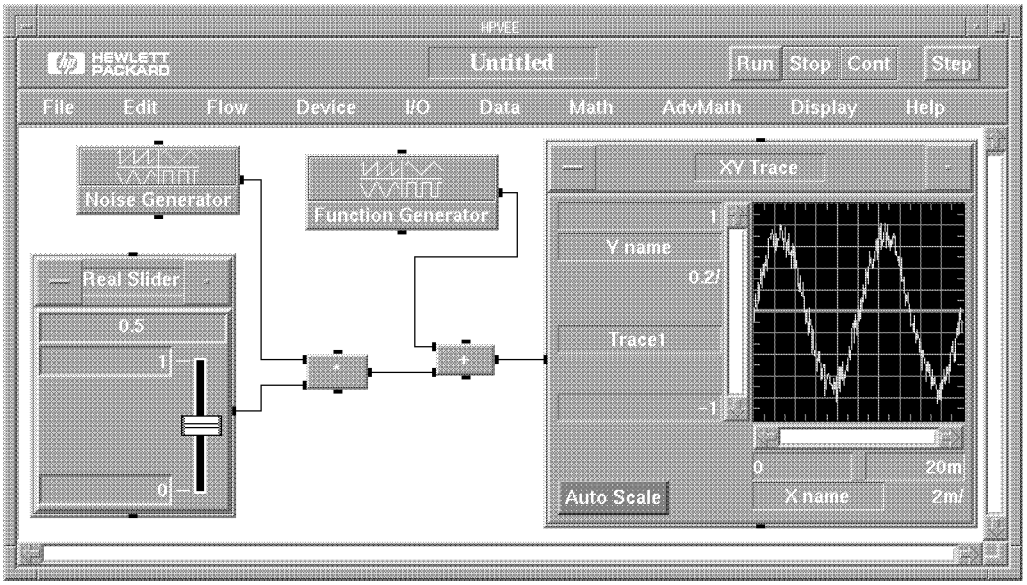
Note that the data output pin of the `Function Generator` object is connected to the data input pin of the `XY Trace` object. When the model is executed by pressing `Run`, the `XY Trace` object won't execute until it receives data from the `Function Generator` object. This is a simple example of data flow.

Now let's create a "noisy sine wave" by adding a `Noise Generator` object to our model. In the following example, the `+` object adds the outputs of the `Function Generator` and `Noise Generator` objects, and outputs the result to the `XY Trace` object.



What is the order of execution? The `XY Trace` object won't execute until it has received data from the `+` object. The `+` object won't execute until *both* of its inputs have been satisfied by the `Function Generator` and `Noise Generator` objects. This leaves the question of which executes first, the `Function Generator` object or the `Noise Generator` object? The answer is that *it doesn't matter.* In either case, the result is the same. The `+` object doesn't execute until both generator objects execute. Once both of its inputs receive data, the `+` object executes, summing the two signals and outputting the result to the `XY Trace` object. Thus, the model operates just fine, execution being determined strictly by data flow.

Now let's add a slider to our model to vary the "noise" signal:



In this example, the order in which the `Function Generator`, `Noise Generator`, and `Real Slider` objects execute doesn't matter. However, data flow determines that the `*` object must execute (once both its inputs are satisfied) before the `+` object can execute. The result is a noisy sine wave with the size of the noise component controlled by the multiplier output by the slider (0.5 in this case).

For a detailed discussion of propagation and data flow in models, refer to chapter 3, "Understanding Models."

2

Using HP VEE Elements

This chapter explains how to interact with HP VEE elements such as menus, objects, and pins. It also presents global information about HP VEE such as setting options, getting help, and recovering from errors.

This chapter discusses the following topics:

- Starting HP VEE
- Using a mouse
- Accessing menus
- Changing preferences
- Finding files
- Using online help
- Recovering from errors
- Accessing examples and library models
- Working with objects
- Connecting pins
- Manipulating interface elements
- Using short cuts

2

Starting HP VEE

To start HP VEE, type `veeengine` or `veetest` at the shell prompt. Because HP VEE's default directory paths for `Open`, `Save`, and `From File`, `To File`, and `Execute Program` objects point to the startup directory, start HP VEE from the same directory each time to find your files quicker.

You can specify how you want HP VEE to run by using the following command line options:

- *filename*

  HP VEE is started and the model in *filename* is loaded into the HP VEE work area. If you do not supply *filename*, HP VEE starts with an empty work area.

- `-d` *directory*

  The `-d` option starts HP VEE and uses the related files, such as help files and instrument drivers (HP VEE-Test only), located in *directory* instead of the default HP VEE installation directory.

- `-r` *filename*

  The `-r` option starts HP VEE and runs the model specified by *filename*. When the model has stopped, HP VEE exits. If you do not supply *filename*, HP VEE ignores the `-r` option. If you specify a file name that doesn't exist, HP VEE returns an error and exits.

- `-display` *Xservername*

  Specifies the X Windows display server instead of using the default X display. In this way, you can have HP VEE execute on one workstation, but use the keyboard and display of a different workstation.

- `-geometry` *width height xoffset yoffset*

  Specifies an initial window geometry instead of the default geometry. For example, `veeengine -geometry 800x500+0-0` starts HP VEE in a window that is 800 pixels wide and 500 pixels tall and placed in the lower left corner of your screen.

- `-help`

  Shows the command line options.

- `-nowarn`

  Blocks the display of warnings encountered at pre-run. All other error warnings are still reported. This is useful to block the "live-mode off" warnings encountered at pre-run when using instruments.

- `-iconic`

  The `-iconic` option starts HP VEE as an icon instead of a window. Double-click on the icon to open it to a window.

- `-name` *name*

  The `-name` option starts HP VEE and sets the application name of HP VEE to *name* instead of `veeengine` or `veetest`. HP VEE uses *name* to specify X11 options in addition to the X11 options specified by the default application class (`Vee`).

  An example of using `-name` is in "Using Multiple Color Sets/Fonts" in Appendix A.

- `-prname` *name*

  The `-prname` option specifies a different color palette for printing. HP VEE will use *name* to specify the X11 options to use for printing instead of the default `VeePrint` resources. Refer to "Using Multiple Color Sets/Fonts" in Appendix A for further information.

Using a Mouse

You can use HP VEE with either a two- or three-button mouse. This manual will always refer to button 1 as the left mouse button and button 3 as the right mouse button. If you are using a three-button mouse, you will not use the middle button with HP VEE.

In general, you use the left mouse button to select items and the right mouse button to access pop-up menus.

Clicking and Double-Clicking

You click to select objects and features. To click, quickly press down then release the mouse button

Use double-clicking to open icons, terminals, and dialog box selections. To double-click, quickly press then release the mouse button twice in rapid succession.

Dragging

Dragging is used to move and size objects and to select features. To drag, press the mouse button *and hold it down* while you move the mouse. At the end of the drag, release the button.

Accessing Menus

You work through menus to access objects, perform actions, and set states.

Understanding Menu Markings

Figure 2-1 shows HP VEE menu markings and their meanings.

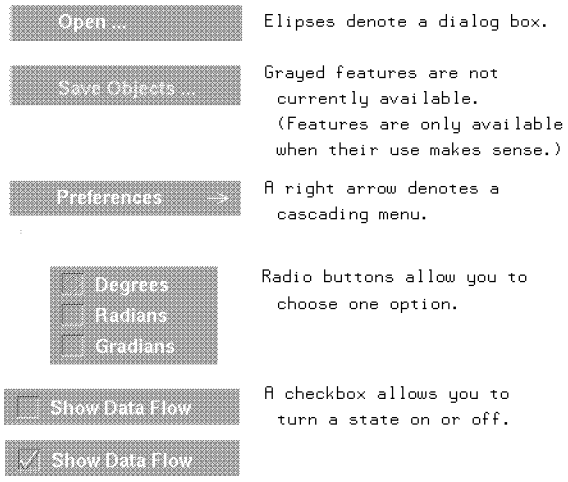


Figure 2-1. Menu Markings

Selecting Menu Features

There are two ways to select a menu feature. Use whichever feels most comfortable:

- Clicking

  To select a feature, click to open the menu and click on the feature.

  To close the menu without selecting a feature, position the pointer off the menu in an empty place on the work area and click.

- Dragging

  To select a feature, press down on the mouse button to open the menu, drag the pointer to the feature, then release the button.

  To close the menu without selecting a feature, drag the pointer off the menu in an empty place on the work area, then release the button.

Note

No matter how you access a menu, the menu you open is determined by the position of your pointer and the mouse button you press. Table 2-1 lists how to access different menus.

Table 2-1. Accessing Menus

To Get This Menu ...	Position the Pointer Here ...	Press This Mouse Button ...
Any main menu	On the menu bar	Left button
Pop-up `Edit` menu	On the main work area -or- On the `UserObject` work area	Right button Right button
Object menu	Over the object menu button -or- Over the object	Left button Right button

Using Cascading Menus

Cascading menus are submenus available from the main menus and object menus. Their location is noted by a right pointing arrow (⇒) on the menu.

Open a cascading menu by moving the pointer to the right over the arrow until the cascading menu is displayed. Make selections as you did from the parent menu.

To close the cascading menu, move the pointer off the cascading menu to the left and back to the parent menu.

Placing Objects

To place an object, select the feature from the main menu. You'll see an outline box of the object. Move the outline box to where you want the object to be on the work area and click.

You may size the object *before* it appears on the work area by moving the outline box to the desired position and sizing the outline box by *dragging* it.

Changing Model Preferences

The defaults for your environment are set with the `Preferences` features on the `File` menu. `Preferences` specifies overall HP VEE options. Some of these preferences are saved with each model and as the defaults. The defaults are used whenever you start HP VEE or select `New` from the `File` menu.

The `Preferences` options are as follows:

- `Trig Mode` specifies the units that the model uses: degrees, radians, or gradians. `Trig Mode` may also be set for each `UserObject`. This preference is saved with each model as well as in `.veerc`.
- `Number Format` specifies the default display format for real and integer numbers. The `Number Format` settings are used in most entry fields except `State Drivers` (HP VEE-Test only). This preference is saved with each model as well as in `.veerc`.
- `Waveform Defaults` specifies the time span and number of points that is the default for Waveforms (such as those created by `Function Generator`, `Pulse Generator`, and `Noise Generator`). This preference is saved with each model as well as in `.veerc`.
- `Auto Line Routing` specifies if lines are automatically routed around other objects each time you draw a line, or move or size an object.
- `Printer Config` specifies the printer options for a graphics printer (used for printing the HP VEE window with `Print Screen`, `Print Objects`, and `Print All`), and a text printer (used for outputting data with the `To Printer` objects).

- `Plotter Config` specifies the options for a graphics plotter. Used for plotting the graphical display objects such as `XY Trace`, `Strip Chart`, `Polar Plot`, and so forth.
- Default directory path where `Merge` and `Save Objects` point. When you select `Save Preferences`, the most recently used path is saved.

When you select `Save Preferences`, the preferences are saved in the `.veerc` file in your `$HOME` directory (typically your `/users` directory).

For more information about customizing your HP VEE work sessions, refer to Appendix A.

Finding Files

When you specify a file name to `Open`, `Save`, or `Save As`, or when you use the `To File`, `From File`, or `Execute Program` objects, the default directory path is listed from the HP VEE startup directory.

When you `Merge` or `Save Objects`, the default directory is `/usr/lib/veeengine/lib/` or `/usr/lib/veetest/lib/` the first time you start HP VEE. When you start HP VEE again, the default directory is the last directory you specified in `Merge` or `Save Objects` if you selected `Save Preferences` from the `File` menu.

To access a file in a directory other than the default, traverse the directory structure using the operating system file path name conventions. (For example, to move to the parent directory, double-click on `../` or type `../` in the entry field of the dialog box.)

HP VEE keeps track of the directories you accessed during each work session. Each subsequent time you access a file, the default directory is the one you previously used.

The file path listed in any dialog box that accesses files is relative to the HP VEE startup directory (`./`) unless you specify otherwise (by specifying a path from root, `/`).

Using Online Help

HP VEE has online help that allows you to access information about building a model, objects, terminology, instruments (HP VEE-Test only), short cuts, and software version.

Help Menu

Access online help by selecting the help feature you want from the `Help` menu.

- `On Features` provides reference information about each menu feature (actions and objects). This is the same information as in the *HP VEE Reference* manual.
- `On Instruments` (HP VEE-Test only) contains information about each driver file.
- `How To` summarizes commonly needed information.
- `Glossary` defines terms.
- `Short Cuts` lists keyboard accelerators.
- `On Help` contains information about using the online help facility.
- `On Version` contains release-specific information about the HP VEE software.

The `Help` features (except for `Help` ⟹ `On Version`) are modeless; once you have the help dialog box(es) open on your work area, you can leave them open while you build your model. If help dialog boxes are open and you select another help feature, the new selection replaces the previous help information.

The following buttons are used in help dialog boxes:

- `Open Topic` opens the selected help topic.
- `End Help` exits the help selection and closes all help boxes.
- `Next` displays the next topic in the sequence from the list box.
- `Prev` displays the previous topic in the sequence from the list box.
- `Done` closes the help information box. However, help list boxes (if they exist) remain so you can select another topic from the list.

Note

Because all `Math` and `AdvMath` objects can be used in a `Formula` object, help for them is listed under `Formula`.

Help from Objects

You can also get help about an object from its object menu. When you get help from the object menu, you cannot use the `Prev` and `Next` buttons to traverse other help screens. To close the help dialog box for an object, press `Done`.

Recovering from Errors

When HP VEE doesn't understand what you want to do, it displays a `Caution` (yellow titled) or an `Error` (red titled) dialog box. Note that the title colors are defaults that may be changed in your X11 resources.

You will not get an `Error` dialog box if the object that generated the error has an error output pin. In this case, the error number is output on the pin. You may look up the message, cause, and recovery action under the `Help` ⟹ `How To` topic titled `Error Codes`. For information about trapping errors with an error output pin, refer to "Trapping Errors" in Chapter 4.

If you don't get a caution or error message, but your model doesn't work as you expected, refer to Chapter 14.

Note

In very rare situations, you may get a "Serious Error" message, which will appear in an all-red "pop-up" display. In the unlikely event that you encounter such a message, exit and restart HP VEE.

Accessing Example and Library Models

HP VEE provides examples that help you quickly learn how to build models and library models that let you use HP VEE more effectively.

Each example or library directory contains a `CONTENTS` file that briefly describes the models in that directory. You can read `CONTENTS` in HP VEE by using `Open` or `Merge`.

Examples

For your convenience, many example models are provided with HP VEE. On installation, these examples are stored in subdirectories under `/usr/lib/veeengine/examples/` or `/usr/lib/veetest/examples/`. The particular subdirectories under `examples` (there are several) depend on which version of HP VEE you have.

The example files and directories are read-only. If you modify an example model, you'll have to save it in a different directory, such as in your `$HOME` directory.

Refer to Appendix B, "Example Models and Library Objects," for further information.

Libraries

HP VEE also contains library models that you can incorporate in your models. They are stored in `/usr/lib/veeengine/lib/` or `/usr/lib/veetest/lib/`. The `lib` directory is read-only. If you modify a library model, you'll have to save it in a different directory such as `contrib/`. The default path for `Merge` and `Save Objects` points to the `lib/` directory (until the path is changed and `Save Preferences` is selected).

The `lib/` directory contains the `contrib/` directory where you can store your own useful library models. This directory is writable by everyone so you can use files you and others saved in `contrib/`.

2

Working with Objects

Objects are the building blocks in your visual engineering environment.

Understanding Object Menus

Every object has an object menu that contains features that allow you to modify the object's appearance, location, and operation.

You access the object menu with the right mouse button when the pointer is over the object or with the left mouse button on the object menu button on the open view (see Figure 2-2).

All objects have the following object menu features:

- `Move` allows you to change the location of this object.
- `Size` allows you to change the size of this view of the object.
- `Clone` presents a copy of this object for immediate use and stores the copy in the `Paste` buffer.
- `Help` gives reference information about this object.
- `Show Description` shows an empty area where you can write notes about this object.
- `Breakpoint` sets an execution breakpoint on the object.
- `Show Title` is "on" by default, and the title bar is displayed at the top of the object. Turn `Show Title` off to eliminate the title bar.
- `Terminals` ⟹ allows you to view, add, or delete input and output terminals.
- `Layout` ⟹ (icon only) allows you to change the appearance of the icon.
- `Cut` deletes this object (a copy is stored in the `Paste` buffer).

The object menu for all icons (except `OK`, `Start`, and `Toggle`) is the same. The open view of an object has additional, object-specific features that allow you to change the way the object operates.

Icons vs. Open Views

There are two views of an object: the **icon** and the **open view**. An icon is the smallest representation of an object. Use the icon when you want to know the general function of the object but don't need to know all the details about it. HP VEE supplies **bitmaps** (symbolic pictures) for some icons to help you quickly identify them.

An open view allows you to access all the information about an object. The object menu for each open view contains features specific to the object.

Working with Open Views

Some objects are initially displayed as open views because you generally want to modify the open view in some way (such as typing an expression in a `Formula` object). Other objects are initially displayed as icons because they are not generally modified before they're used.

To open an icon into an open view, double-click on the icon. The open view lets you access the full functionality of the object. Figure 2-2 shows the parts of an open view.

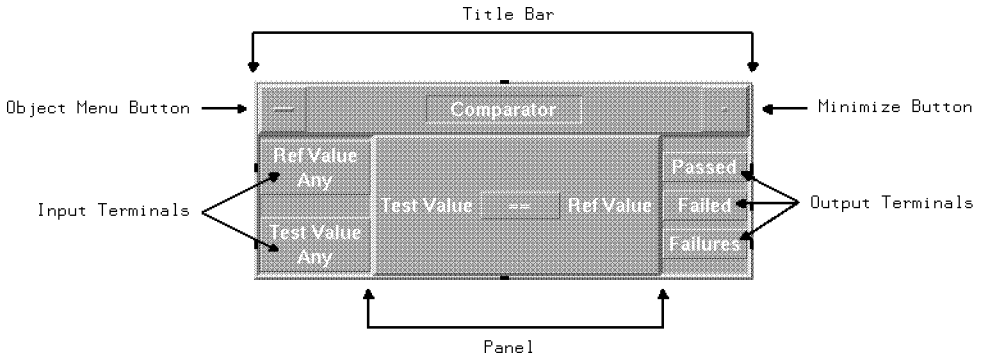


Figure 2-2. Parts of an Open View

Viewing and Modifying Terminals

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. If terminals are present but not visible, select `Terminals` ⟹ `Show Terminals` from the object menu to see them.

To view or modify the attributes of a terminal, double-click on the terminal's information area (not the pin). You'll see a dialog box similar to the following:

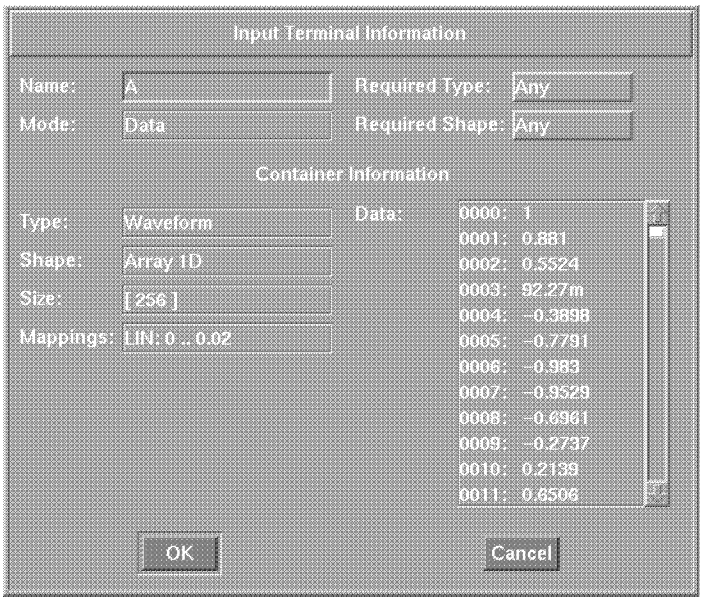


Figure 2-3. Terminal Information

If all the fields are flat rectangles, the terminal cannot be modified. However if some of them are entry fields (recessed) 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 equation.
- `Mode` displays the terminal type, such as `Data`, `Control`, `XEQ`, or `Error`. You cannot modify this field. For more information about terminal types, refer to "Understanding Pins" in Chapter 3.
- `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`.
- `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.

For more information about containers, data types, and data shapes refer to "Understanding Containers" in Chapter 3.

Adding and Deleting Terminals

To add or delete input or output terminals for an object, use the features under `Terminals` on the object menu. You can also use the short cuts to add or delete terminals. Position the cursor over the input or output terminal area and type CTRL-A to add a terminal. Or position the cursor over a terminal and type CTRL-D to delete it. You can add or delete terminals from either the open view or the icon view by using `Terminals` from the object menu, but the short cuts are available only from the open view.

Note that you cannot add or delete inputs or outputs when the model is running. These cascading menu features are grayed when objects are operating.

Closing Open Views

Close an open view to an icon by pressing the minimize button on the upper right of the object.

Understanding Object Highlights

An object may be surrounded by a highlight. Each type of highlighting has a special meaning. Note that the highlight colors are defaults that can be changed in your X11 resources.

Table 2-2. Object Highlighting

Highlight	Meaning
Black shadow	The object is selected.
Black border	There is a breakpoint set on the object.
Yellow border	`Show Exec Flow` is turned on and the object is operating. To clear the highlight after the model has stopped, press the `Stop` button.
Red border	The object generated an error when operating. To clear the highlight, press the `Stop` button.

Connecting Pins

To connect objects, draw a line between the pins. To draw the line, click near (but outside) the pin of the first object, move the pointer to the destination pin on the second object, and click again.

If you want to specify the path of lines drawn, set `Auto Line Routing` to off (by unchecking it), then click on the work area as you are drawing the line to "anchor" the line to a certain path. Usually you'll set `Auto Line Routing` on or use `Clean Up Lines` (from the `Edit` menu) periodically to route lines around objects to make your model lines easier to follow.

To connect objects that are not visible on the screen at the same time, click near the pin of the first object and then drag the work area until the other object is visible. Then connect the line to the destination pin.

You can have only one line connected to an input pin. You may have as many lines as you need coming from an output pin.

Exiting Line Drawing Mode

If you start to draw a line by mistake, double-click on an empty place in the work area to exit line drawing mode.

Deleting Lines

To delete a line after it is drawn, select `Delete Line` from the `Edit` menu, then click on the line to be deleted. You can hold down CTRL-Shift instead of selecting `Delete Line` and then click on a line to quickly delete it.

To check the endpoints of a line before deleting it, select `Delete Lines`, but then *drag* the pointer over the line; the line and its endpoints are highlighted. To delete the line, release the pointer when the line is highlighted, otherwise drag the pointer off the line then release the mouse button to exit delete line mode.

Manipulating Other Interface Elements

There are many other interface elements that HP VEE uses. Some of them may be familiar to you from previous experience using windows.

Using Scroll Bars

Scroll bars are usually on the right and/or the bottom of a window, work area, or list that contains more information than can be displayed currently. One example of scroll bars is at the right and bottom of the HP VEE window; another example is at the right of the file list when you select `Open` from the HP VEE `File` menu.

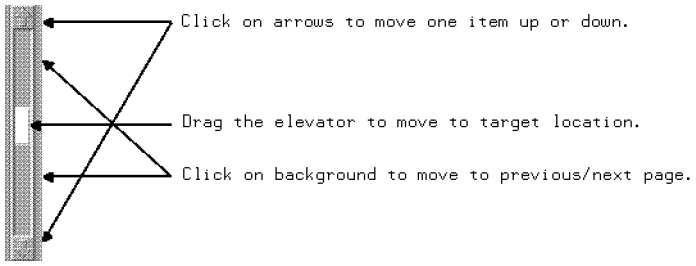


Figure 2-4. Using a Scroll Bar

Using List Boxes

List boxes give you a set of selections from which to choose. One example of a list box is on the dialog box you see when you select `Open` from the HP VEE `File` menu. Each list box has a scroll bar. If you want a selection that is not currently visible in the list box, use the scroll bar to change the current view of the list.

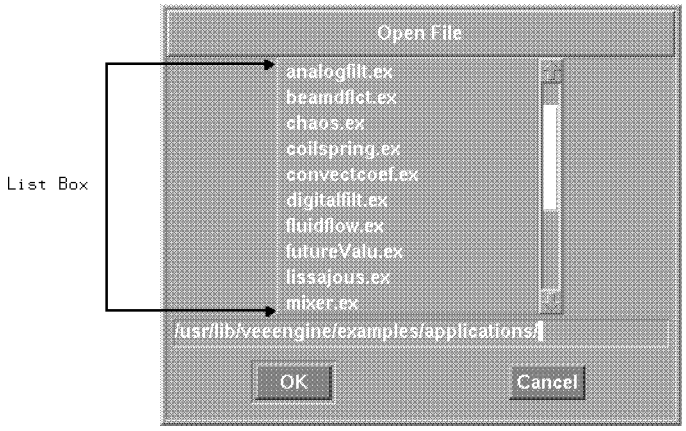


Figure 2-5. A List Box

To select an item, position the pointer on the selection, click the left mouse button, then select the `OK` button. Another way to select an item is to double-click on the selection.

Using Buttons

Click on buttons to specify an action or change a value. Examples of buttons are the `OK` and `Cancel` buttons on the `Open` dialog box from the HP VEE `File` menu.

All dialog boxes have buttons to specify actions. The default button is outlined with a recessed rectangle. If you press Return when a dialog box is displayed, it is the same as clicking the default button. If you double-click on a list box item, it is the same as selecting an item, then clicking the default button.

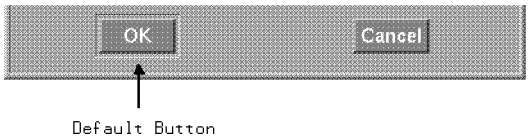


Figure 2-6. A Default Button

You will see the following buttons in many dialog boxes:

- `OK` confirms the action and enters your input. `OK` is generally the default for most dialog boxes.
- `Cancel` cancels the action and does not enter your input.

Other buttons specify a value to be selected from a finite set of choices. These buttons are used in three ways:

- Cyclic buttons display the next choice in the list of choices.
- List buttons display a list box from which you can select your choice.
- Radio buttons display a list of choices with one of the buttons selected. Selecting one button un-selects all of the others.

Using Entry Fields

An entry field looks like a recessed rectangle.

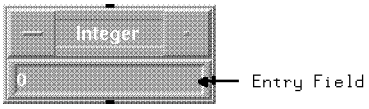


Figure 2-7. An Entry Field

You type in information to change the value of this field, then press Return or click on the empty work area when you're done. The input value is then checked to verify that it is a valid value in a valid range. For example, if you put a real number into an `Integer` constant then press Return, the value is immediately demoted to an integer. If you put a text string into an `Integer` constant and press Return, HP VEE displays an error message.

For more information about promotion and demotion, refer to "Converting Data Types on Input Terminals" in Chapter 3.

Note

When the cursor is at the beginning of an entry field, the existing value is replaced when you type. To recover the initial entry, press CTRL-C *before* pressing Return.

To insert characters at the beginning of the entry field, press an arrow key (such as ◀) before typing.

Editing

There are many places where you edit information: entry fields such as title bars or `Constant` objects (numeric entry fields), transactions on `I/O` objects such as `To File` or `From StdIn`, and general text entry areas such as `Note Pad` and `Show Description`.

To edit, use the editing keys from your keyboard (such as Delete line, Insert char, and the arrow keys). As you type, the characters are inserted into the existing entry. The entire list of editing actions is in Table 2-3 and Table 2-4. Note that there may be multiple ways to perform the same edit action.

Table 2-3. To Begin Editing

Action	Keys
Edit line	Click on field
Edit line under pointer	Select
Select transaction	Click on transaction
Edit selected transaction	Select Click on transaction

Table 2-4. Editing Keys

Action	Keys
Edit next line or entry field	Tab CTRL-N ▼
Edit previous line or entry field	Shift-Tab CTRL-P ▲
Jump to previous page in a scrolled list	Prev
Jump to next page in a scrolled list	Next
Jump to beginning	↖ (home key)
Jump to end	Shift-↖
Scroll up	Shift-▲
Scroll down	Shift-▼
Move left one character	◀ CTRL-B
Move right one character	▶ CTRL-F
Move up one line	▲ CTRL-P
Move down one line	▼ CTRL-N
Move to the beginning of the line	Shift-◀ CTRL-A
Move to the end of the line	Shift-▶ CTRL-E

Table 2-4. Editing Keys (continued)

Action	Keys
Insert line (above cursor)	Insert line CTRL-O
Delete to the end of line	Clear line CTRL-K
Delete character	Delete char CTRL-D

Editing File Paths

When you start to type a file name or path, HP VEE scrolls the list box to the entry that matches the letters you've typed, but *does not* select it until you complete the name.

If you press the spacebar when typing a file path or name, HP VEE tries to complete the name.

To clear the existing path and begin the path from root (instead of from the startup directory) , press Clear display and start the path with `/`

Editing Numeric Entry Fields

Numeric entry fields are evaluated when you press Return. Entry fields may abbreviate the value to fit. The internal precision is maintained; you can view the entire value by clicking on the field.

Numeric entry fields evaluate mathematical equations such as `2*3` or `5+pi`. You can use `+`, `-`, `*`, `/`, and `^`, and the other functions under the `Math` and `AdvMath` menus. For example, if you type `cubert(4096)` in a `Real` object's entry field and press Return, `16` is displayed.

You can use most standard ISO (International Standards Organization) abbreviations in entry fields. For example, intead of typing `1200000`, you can type `1.2M`.

Table 2-5. ISO Abbreviations

Abbreviation	Suffixes	Multiple
P	peta	10^{15}
T	tera	10^{12}
G	giga	10^{9}
M	mega	10^{6}
k or K	kilo	10^{3}
m	milli	10^{-3}
u	micro	10^{-6}
n	nano	10^{-9}
p	pico	10^{-12}
f	femto	10^{-15}

Using Keyboard Short Cuts

The following keyboard short cuts allow you to perform common HP VEE functions more quickly. These short cuts are also listed online in `Help` ⟹ `Short Cuts`.

Keys	Action
CTRL-left mouse button click on objects	Selects unselected objects or unselects selected objects
Shift-left mouse button click near a line	`Edit` ⟹ `Line Probe`
Shift-CTRL-left mouse button click	Deletes the line under the pointer
Clear display	`File` ⟹ `New`

CTRL-A	Object menu ⟹ `Terminals` ⟹ `Add Data Input` or Object menu ⟹ `Terminals` ⟹ `Add Data Output`.
CTRL-D	Object menu ⟹ `Terminals` ⟹ `Delete Input` or Object menu ⟹ `Terminals` ⟹ `Delete Output`.
CTRL-O	`File` ⟹ `Open`
CTRL-S	`File` ⟹ `Save`
CTRL-W	`File` ⟹ `Save As`
CTRL-E	`File` ⟹ `Exit`
CTRL-R	Repaints window
CTRL-D	Deletes object under pointer
CTRL-C	Pauses a running model or cancels an edit
Shift-Print	Prints screen with the current options specified in `Printer Config`
Arrow Keys (▲, ▼, ◀, ▶)	Moves the pointer
Shift-Arrow Key	Scrolls the work area in the direction of the arrow key
Next	Scrolls the work area up one screen
Prev	Scrolls the work area down one screen
Shift-Next	Scrolls the work area left one screen
Shift-Prev	Scrolls the work area right one screen
↖ (home key)	Moves the upper left corner of the model to the upper left corner of the work area
Shift-↖	Moves the lower right corner of the model to the lower right corner of the work area

There are some additional short cuts that you can use when the mouse cursor is positioned over an object containing *transactions* (for example, the `Sequencer`, `To File`, and so forth. These short cuts are as follows:

Keys	Action
CTRL-K	Object menu ⟹ `Cut Trans`
CTRL-Y	Object menu ⟹ `Paste Trans`
CTRL-O	Object menu ⟹ `Insert Trans`
CTRL-X	Object menu ⟹ `Step Trans` (`Sequencer` only)
CTRL-N	Move to next transaction.
CTRL-P	Move to previous transaction.

3

Understanding Models

This chapter describes the general rules models follow in HP VEE and some techniques to follow when you're building models. This chapter describes the following topics:

- What happens when you run and stop a model
- Execution order of objects
- Types of pins used on the objects and how they affect execution flow
- Data types, shapes, and mappings
- Model-building techniques
  - Documenting
  - Debugging
  - Sharing models with others
  - Archiving

Understanding How Models Run

With HP VEE, you build a model to solve your engineering problem. A model looks like a block diagram. When you run the model, the objects operate on the data that is input to them by you interactively or by lines that are connected from other objects.

Threads and Subthreads

Each independent set of connected objects is called a **thread**. Multiple threads are completely independent of each other; they are not connected by data or sequence lines, however they may be connected by control (dashed) lines.

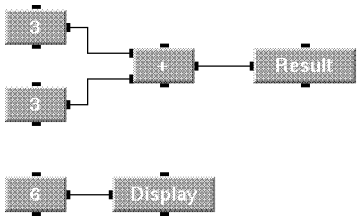


Figure 3-1. Two Parallel Threads

A branch of a thread is a **subthread**. When subthreads begin at the data output of the same object and have no sequence or data dependencies (no solid line connections) between them, they are parallel.

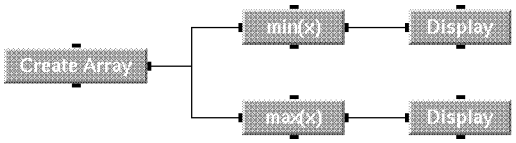


Figure 3-2. Two Subthreads

Running Models

You run a model by pressing the `Run` button in the upper right corner of the HP VEE window. After you press `Run`, all threads run. You can run a single thread in the model by adding a `Start` object to the thread and then pressing `Start`.

After you start to run your model (by pressing `Run` or `Start`), HP VEE initializes your model in a process called **PreRun**. PreRun clears the signals from the lines and resets the objects so they are ready to run. For specific information about what happens immediately before objects start to operate, refer to "Advanced Topic - Understanding PreRun".

Stopping Models

There are three reasons why a model stops.

- The model runs to completion.
- You press the `Stop` button twice (pressing `Stop` once pauses the model).
- HP VEE encounters an untrapped error.

When a model stops:

- Any files communicated with are closed.
- Any opened named pipes are closed.
- The `To Printer` object delivers data to the printer.

Note

If your model communicates with other programs, you need to monitor their execution. They do not necessarily stop when the model stops.

Advanced Topic - Understanding PreRun

After you start a model but before the objects start to operate, HP VEE PreRuns your model. The following actions occur at PreRun.

- Data input terminals are set to nil. Output terminals and `Line Probe` will show nil containers on the lines.
- Objects are reset to their initial conditions (unless you specify otherwise). For example, the default settings for `Counter` cause it to be reset to zero at PreRun but you can change the settings.
- HP VEE checks to see if all data and XEQ inputs are connected.
- Any objects with initial values have their initial values set (if you specified this to happen). For example, the default settings for `Real Slider` do not set an initial value, but you can change the settings.
- Any feedback loops are checked for resolution (a `Start` object on the loop).
- Any files specified in `From File` are rewound so that file pointers are at the beginning of the file.

- Any files specified in `To File` (when `Clear File at PreRun & Open` is set) are rewound.
- Any previous error conditions are cleared.
- (HP VEE-Test only.) You are cautioned regarding any devices that would do instrument I/O, but have `Live Mode` turned off. Such devices include HP Instrument Drivers as well as Direct I/O objects. These warnings are suppressed if you start HP VEE using the `-nowarn` option.

Note

If you have multiple threads in your model and start one of them by pressing `Start`, the PreRun activities occur only on the thread started.

Each `UserObject` completes a stage similar to PreRun, called **Activate**, before the `UserObject` operates each time. Activate clears the data lines between objects to ensure that the `UserObject` processes new values (if you specify) each time it operates.

Many objects, such as displays, counters, collectors, and constants have features on the object menu that allow you to specify what happens to the object at Activate and PreRun. These features include `Clear` and `Initialize` to reset an object to a known state when the model runs or a `UserObject` operates.

Activate vs. PreRun

The scope of Activate is only one level of `UserObject` (a context); the scope of PreRun is the entire model (when `Run` is pressed) or thread (when `Start` is pressed) including all levels of nested `UserObjects`. If you have a `UserObject` nested inside a `UserObject`, the objects in the nested `UserObject` are only Activated when the nested `UserObject` operates.

PreRun is done only once per run. Activate is done for the root context every time the model is run and is done for each `UserObject` every time it operates. For more information about contexts, refer to Chapter 7.

Understanding Propagation

Propagation is the general flow of execution through your model. The propagation guidelines define the order in which objects operate.

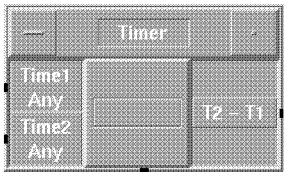
Understanding How Objects Operate

An object operates by processing the input data, completing its function, and putting active data on its output data lines. An object operates only after there is active data on all of its input data terminal(s) (the only exception is the `JCT` object, which has asynchronous data input terminals, therefore if one of its data inputs is activated, `JCT` operates).

Note

All data and XEQ input pins *must* be connected before you press `Run` or HP VEE returns an error.

For example in Figure 3-3, the `Timer` must have both its input data pins activated before it operates (subtracts the two times). Each input data pin is activated when it receives a container from another object. After the `Timer` operates, the output pin is activated, and the output container data is the time difference. After all the objects to the right of the `Timer` (on the same subthread) that can operate have done so, the sequence output pin of the `Timer` is activated.



1. Both inputs must be activated.
2. Then the Timer operates.
3. Then the output is activated.

Figure 3-3. Object Operation

Basic Propagation Order

When you press `Run` (or `Start`), the objects in your model (or thread) operate in the following order:

1. All `Start` objects operate first (if they exist).

Note

`Start` is not required *unless* your model has at least one data feedback loop. For details about feedback, refer to "Understanding Feedback"

2. Objects without data or sequence dependencies operate next.
3. Objects that have all data and sequence (if connected) input pins activated operate next. These objects in turn activate the data and/or sequence input terminals of other objects so that they operate.

In Figure 3-4, the `A` objects operate first (in no particular order) then `B`, `C`, `D`, and so on. No object can operate until its data and sequence (if connected) inputs are activated.

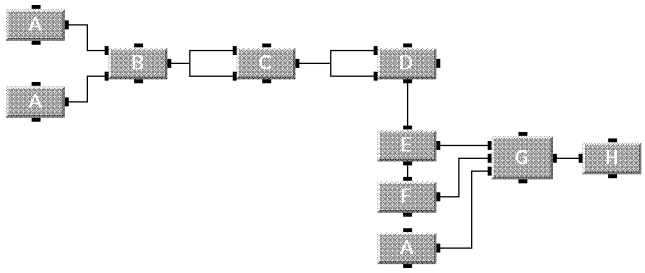


Figure 3-4. Example of Basic Propagation

The model shown in Figure 3-4 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual01.ex
 - or -
/usr/lib/veetest/examples/concepts/manual01.ex
```

Sequence output pins do not activate until propagation has progressed as far as possible for data output pins. The order of sequence output pin activation is reverse the order of data output pin activation; the sequence pins at the beginning of a thread or subthread activate last.

In Figure 3-5, the order of operation is A, B, C, D, E, F, and G.

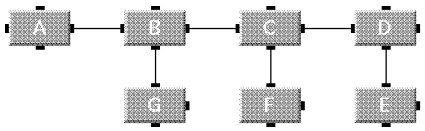


Figure 3-5. Propagation Through Data and Sequence Pins

The model shown in Figure 3-5 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual02.ex
 - or -
/usr/lib/veetest/examples/concepts/manual02.ex
```

When you have parallel subthreads (subthreads that begin at the same data output), they operate in no particular order to each other. In Figure 3-6, either B or C could operate after A. To ensure B operates before C, connect B's sequence output pin to C's sequence input pin.

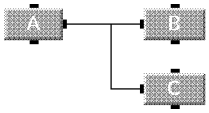


Figure 3-6. Propagation of Parallel Subthreads

Understanding Pins

The pins on objects affect the way that they operate, therefore affecting the execution flow of the model. Here is a summary of the types of pins available on objects and what they do:

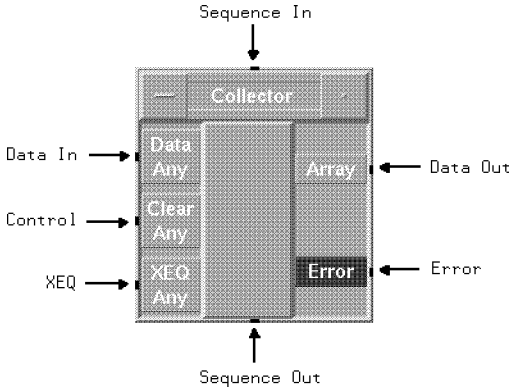


Figure 3-7. Pins on an Object

- **Data pins** input and output a container. An object will not operate until all of its synchronous data input pins are activated. (Note that the `JCT` object has asynchronous data inputs and therefore will operate when any of its data input pins is activated.)

  After the object operates, the output data pin(s) are activated.

- **Control pins** (optional) are asynchronous inputs that affect the state of the object but have no effect on the propagation. Common control pins include `Clear`, `Reset`, and `Default Value`. Lines that connect to a control pin are dashed to show that their inputs do not affect propagation.

- **XEQ pins** are asynchronous input pins that force the object to operate. An XEQ pin must be used on `Collector` and `Set Values` to tell the object when you're done inputting data.

  You can add an XEQ input pin to a `UserObject` to force it to operate before the data and sequence input pins have been activated. Adding an XEQ pin is rarely necessary to have your model run correctly.

- **Sequence pins** are used only to specify the order of execution. You generally don't have to use them. An object operates only after all the synchronous data input pins and sequence input pins (if connected) are activated. You cannot open a sequence pin; it does not have a terminal.

A sequence output pin activates after all the data output pins have activated and data flow has propagated as far as possible.

Note

Sequence input and XEQ pins are activated by the presence of a container. These pins ignore the data in the container. A sequence output pin activates with an empty (nil) container.

- **Error pins** (optional) are output pins that trap the error condition that is generated by the object and outputs the associated error number.

  If an error occurs, the error pin is activated *instead* of any data output pins. Only the error pin and the sequence output pin (if connected) are activated.

  For information about trapping errors, refer to "Trapping Errors" in Chapter 4. For information about generating your own error codes within a `UserObject`, refer to "Raise Error" in Chapter 6.

Note

You may leave data output pins, control pins, and error pins unconnected. Sequence pins, more often than not, *should* be left unconnected. However, an error will occur when you run your model if any data input pins or XEQ pins (if present) have been left unconnected.

Propagation of Threads and Subthreads

Parallel threads and subthreads operate round-robin style. Propagation through each thread proceeds according to the rules above. But it is important to know that multiple threads and subthreads operate in a parallel manner; no single thread or subthread takes over and runs to completion.

Note

(HP VEE-Test only.) The only exceptions to parallel execution are threads or subthreads hosted by `Interface Event` or `Device Event`. When either object traps an event (such as an HP-IB SRQ for `Interface Event`), no objects in the rest of the model will execute until the thread hosted by `Interface Event` or `Device Event` executes to completion. For further information about `Device Event` and `Interface Event`, refer to the *HP VEE Reference* manual.

When `Run` is pressed in Figure 3-8, the objects on both threads operate in parallel.

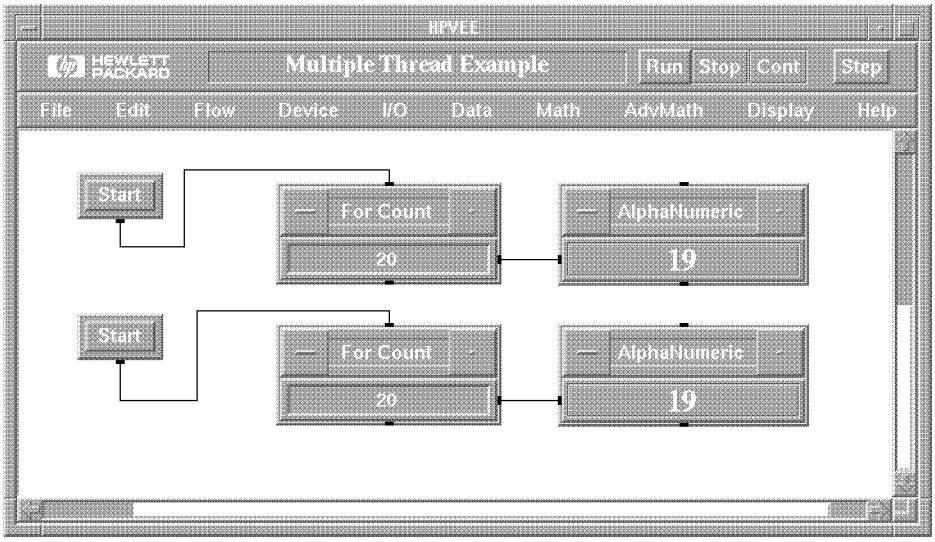


Figure 3-8. Running Multiple Threads

The model shown in Figure 3-8 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual03.ex
 - or -
/usr/lib/veetest/examples/concepts/manual03.ex
```

Note

If you run one thread by pressing `Start`, you cannot start another thread until the first has finished.

Understanding Auto Execute

You can select `Auto Execute` from the object menu on objects that allow user input. These objects include many of the objects under the `Data` menu; for example, the `Real Slider` and `Real Constant` objects.

The purpose of `Auto Execute` is to allow an object to automatically execute whenever you change its value, thus outputting the new value and initiating propagation through those objects in the thread that are "downstream" from the object that is automatically executing. However, unlike the `Start` object, an auto executing object does *not* initiate full propagation of the thread. To see how this works, let's look at an example.

Suppose you turn on `Auto Execute` for the `Real Slider` in the following thread:

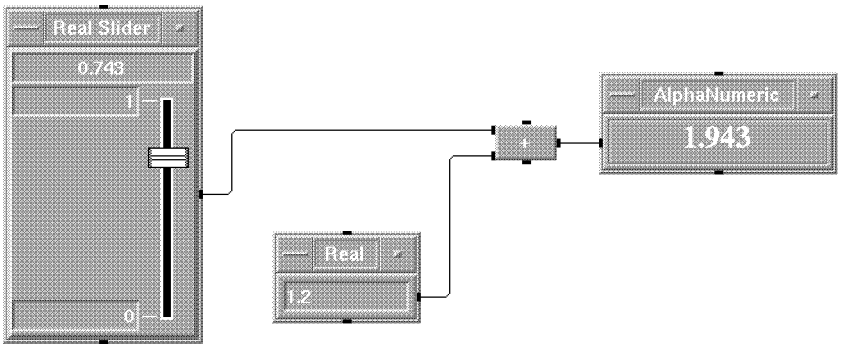


Figure 3-9. Using Auto Execute

When you press `Run`, the thread will propagate to completion. The `+` object will sum the values output by the `Real Slider` and the `Real Constant` objects, outputting `1.943` in this case. (You do have to press `Run` once to initiate propagation in the thread. Until run has been pressed the first time, `Auto Execute` has no effect.)

Once Run has been pressed, changing a value on the Real Slider causes the Real Slider to execute, outputting the new value and initiating propagation through the + and AlphaNumeric objects. However, Auto Execute does not initiate full propagation of the thread. The Real Constant object does not execute when you move the slider in our example. Rather, the + object uses its "old" data from the Real Constant each time the Real Slider auto executes. This is only a problem if you change the value of the Real Constant, which doesn't have Auto Execute turned on. There are three solutions:

1. Press Run after each change to the value of the Real Constant, initiating full propagation of the thread.
2. Turn on Auto Execute for the Real Constant, as well as the Real Slider. However, in many cases this solution is not available. If, instead of a Real Constant, the thread contains a From File or an instrument I/O object, which has no Auto Execute feature, you'll have to use the third solution.
3. Connect the sequence output pin on the Real Slider (with Auto Execute on) to the sequence input pin on the Real Constant (with Auto Execute off), as shown below. *This is the recommended solution for most cases.*

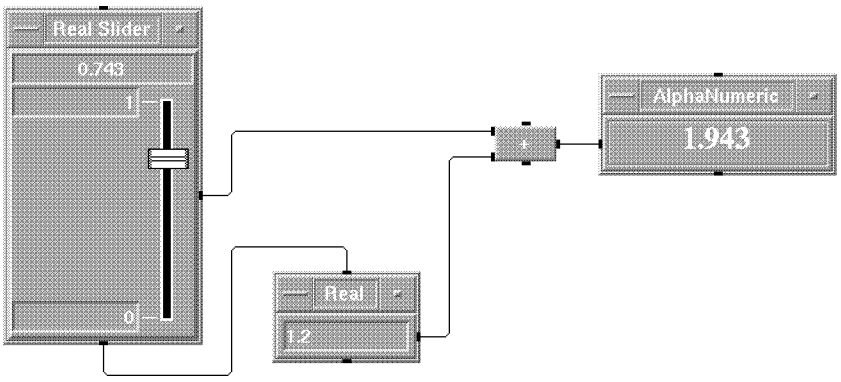


Figure 3-10. Using Sequence Pins with Auto Execute

There is one other point to be made about Auto Execute. So far we've considered the effect of Auto Execute when the thread has completed execution. But what happens if the thread is currently executing when you edit an object for which Auto Execute is turned on? Again looking at our example, if you move the slider on the Real Slider (with Auto Execute on), the object

will execute *only if the thread has finished executing.* If the thread is currently executing, the new value will be set on the `Real Slider` object, but the object won't execute and propagation won't be affected.

Understanding Feedback

Use feedback to access previous values or to allow a set of values to change together. (If one value changes, the others are forced to operate again.) You cannot use feedback to end an iteration subthread — iteration ends at the end of the thread.

HP VEE feedback can occur through either data or sequence pins. If a thread contains feedback through control pins (dashed lines), it will not operate properly.

When your model contains at least one feedback loop, you *must* have a `Start` object on each thread that has feedback so that HP VEE can resolve the initial order of operation.

Data Feedback

Iteration objects are always used when there is data feedback. Data feedback is often used to evaluate equations that use previous results in the calculation. In Figure 3-11, the feedback loop is necessary to perform the recursive calculation: $x_n=(x_{n-1})+2$.

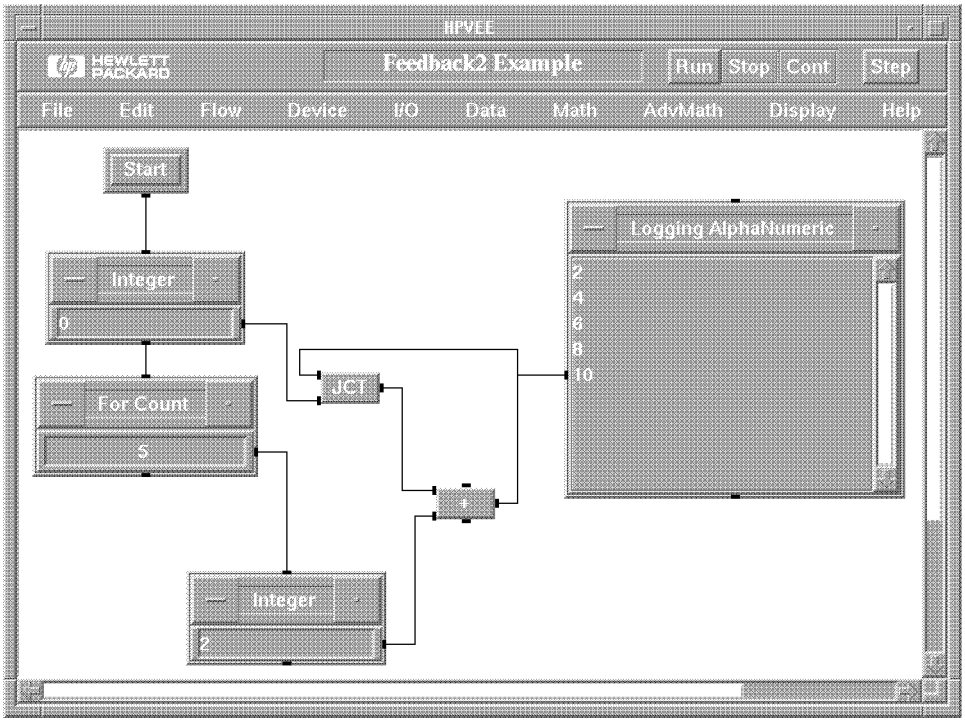


Figure 3-11. Example of Data Feedback

The model shown in Figure 3-11 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual04.ex
 - or -
/usr/lib/veetest/examples/concepts/manual04.ex
```

In data feedback, you must specify what value the loop will take for the initial cycle. Although generally the previous value is evaluated as zero the first time, you should use a `JCT` object to define specifically what should happen.

Sequence Feedback

Figure 3-12 shows a use of sequence feedback. The `Sliders` have `Auto Execute` set; the feedback loop ensures that whenever a value on either of them is changed, the model runs.

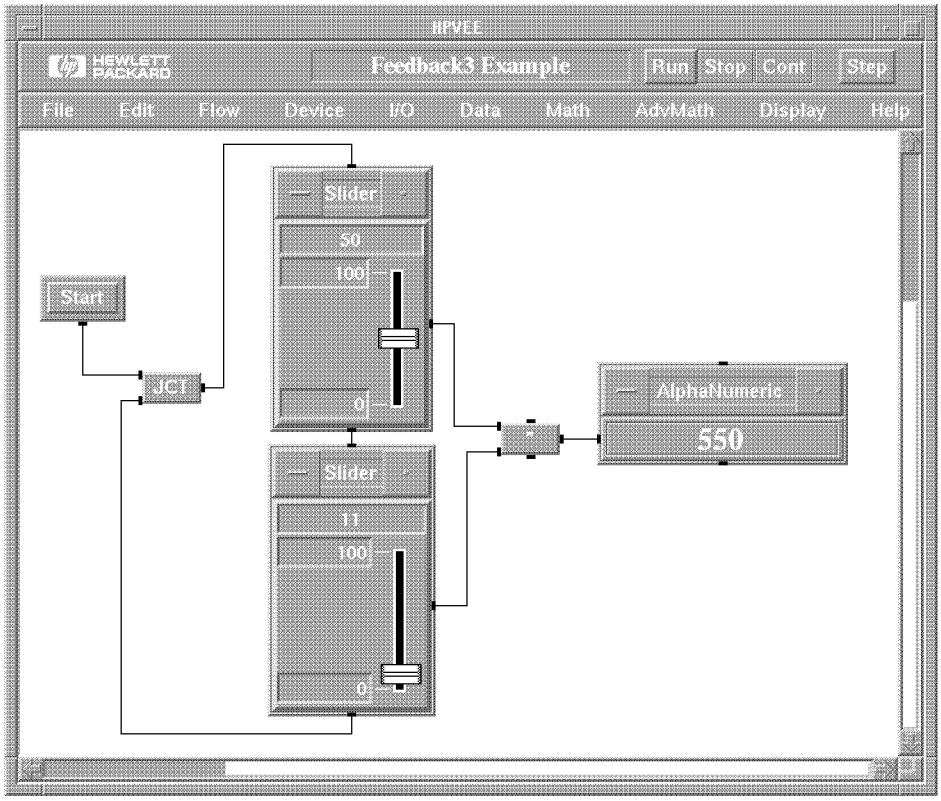


Figure 3-12. Example of Sequence Feedback

The model shown in Figure 3-12 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual05.ex
```

```
 - or -
/usr/lib/veetest/examples/concepts/manual05.ex
```

Propagation Summary

- Data flows through objects from left to right; sequence flows from top to bottom.
- All data and XEQ input pins must be connected.
- `Start` objects operate first. Objects with no data or sequence input dependencies operate next.
- All synchronous data input pins must be activated before an object operates.
- If the sequence input pin is connected, it must be activated before an object can operate.
- Objects with synchronous data inputs operate only once unless connected to an iteration subthread or forced to operate by an XEQ pin.
- Control pins are asynchronous and do not affect the operation of the object.
- When an error is generated from an object with an error pin, the error pin is activated instead of the data output pins(s). The sequence output pin *is* activated.
- Parallel subthreads, hosted by a single output pin, may operate in any order.
- Multiple threads operate simultaneously.

Note

If an iteration subthread includes flow branching, a `Sample & Hold` may be required to obtain correct propagation. Refer to "Iteration with Flow Branching" in chapter 4.

Understanding Containers

A **container** carries data between objects. When a container arrives at the object's pin, the pin is activated. A non-nil container has data of a specific type and shape. Arrays may be mapped.

When `Show Data Flow` is checked, a small box indicates the movement of the containers along lines.

Data Types

HP VEE provides 13 data types, but 3 of these types are used *only* in instrument I/O transactions. The following 10 data types are used for all *internal* HP VEE operations. That is, every HP VEE data container sent between HP VEE objects is of one of these 10 types.

Note

If an input terminal on an HP VEE object specifies `Any`, it will accept containers of *any* HP VEE data type.

Composite data types (Waveform, Spectrum, and Coord) are associated with particular data shapes.

- `Int32` is a 32-bit two's complement integer (-2147483648 to 2147483647).
- `Real` (or `REAL64`) is a 64-bit real that conforms to the IEEE 754 standard (approximately 16 significant decimal digits or ± 1.7976931348623157E308).
- `PComplex` is a magnitude and a phase component in the form `(mag, @phase)`. Phase is in the trigonometric units set under `Preferences` ⟹ `Trig Mode` for the main work area or `Trig Mode` under the `UserObject` object menu for a `UserObject`. For example, the PComplex number `4 at 30 degrees` is represented as `(4,@30)` when `Trig Mode` is set to `Degrees`. Each component is Real.
- `Complex` is a rectangular or Cartesian complex number. Each complex number has a real and an imaginary component in the form `(real, imag)`. Each component is Real. For example, the complex number `1 +2i` is represented as `(1,2)`.
- `Waveform` is a composite data type of time domain values that contains the Real 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).

- `Spectrum` is 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.

- `Coord` is a composite data type that contains at least two components in the form (`x`, `y`, ...). Each component is Real. The data shape of a Coord must be a Scalar or an Array 1D.
- `Enum` is a text string that has an associated integer value. You can access the integer value with the `ordinal(x)` function.

  The data shape of an Enum must be Scalar; an array of Enum is automatically promoted to Text. Enum cannot be a required data input type.
- `Text` is a string of alphanumeric characters.
- `Record` is a data type composed of fields. Each field has a name and a container, which can be of any type (including Record) and any shape.

Special Instrument I/O Data Types (HP VEE-Test Only)

All integer values are stored and manipulated internally by HP VEE as the Int32 data type, and all real numbers are stored and manipulated as the Real (or Real64) data type. However, instruments generally support 16-bit integers or 8-bit bytes. Also, some instruments support a 32-bit real format. Therefore, HP VEE-Test supports the following three data types, which are used *only* for I/O transactions involving instruments:

- `Byte` is an 8-bit two's complement byte (-128 to 127). (Byte is used in READ BINARY, WRITE BINARY, and WRITE BYTE instrument I/O transactions. The WRITE BYTE transaction is used for specialized character output to HP-IB instruments.)
- `Int16` is a 16-bit two's complement integer (-32768 to 32767).
- `Real32` is a 32-bit real that conforms to the IEEE 754 standard ($\pm$3.40282347E$\pm$38).

Instrument I/O Data Type Conversions (HP VEE-Test Only)

On instrument I/O transactions involving integers, HP VEE-Test performs an automatic data-type conversion according to the following rules:

Note

These data-type conversions are completely automatic, so you won't normally need to be concerned with them. However, the following list shows what happens.

- On an input transaction, `Int16` or `Byte` values from an instrument are *individually* converted to `Int32` values, preserving the sign extension. On the other hand, `Real32` values from an instrument are individually converted to 64-bit `Real` numbers.
- On an output transaction, `Int32` or `Real` values are *individually* converted to the appropriate output format for the instrument:
  - If an instrument supports the `Real32` format, HP VEE-Test converts 64-bit `Real` values individually 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, HP VEE-Test 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, HP VEE-Test 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.

Data Shapes

Each non-composite data type may be in one of five data shapes:

- `Scalar` is a single number such as `10` or `(32, @10)`.
- `Array 1D` is a one-dimensional array of values.
- `Array 2D` is a two-dimensional array of values.
- `Array 3D` is a three-dimensional array of values.
- `Array` is an array with one to ten dimensions.

Note

An input data shape requirement is `Any` when the object accepts containers of more than one of the data shapes.

Mappings

A mapping is a set of continuous or discrete values that express the independent variables for an array. For example, the mappings on a Waveform are the times for each amplitude value. Waveform, Spectrum, and Coord data types are mapped. Arrays of other data types can also be mapped by using `Data` ⟹ `Access Array` ⟹ `Set Mappings`.

Mappings are either continuous (Waveform, Spectrum, or mapped arrays created with `Set Mappings`) or discrete (Coord). Continuous mappings are attached to the array and may be viewed on the terminal or by using `Line Probe`, but are not part of the array and are different than the array indices. Discrete mappings are part of the array and are displayed as the first n-1 fields in a Coord value with n fields.

The sampling interval of linear mappings is the maximum value minus the minimum value, divided by the number of points. There are the same number of points as sampling intervals; each point is at the beginning of a sampling interval. For example, if array `A` (where `A = [1, 2, 3, 4]`) is linearly mapped from `10` to `50`, the mappings and sampling intervals are as shown in Figure 3-13. Note that the mappings are from the first element to the end of the last interval.

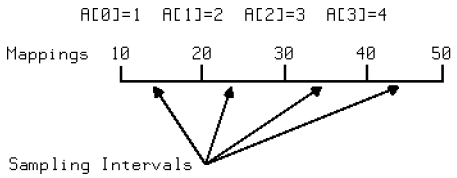


Figure 3-13. Array Mappings and Sampling Intervals

When mapped values (except Coord) are displayed, the `X` axis displays the mappings.

To get information about a continuous-mapped array, use `Data` ⟹ `Access Array` ⟹ `Get Mappings`. `Get Mappings` gives you the type of mapping (log or linear) and the minimum and maximum mapping values for each dimension.

Converting Data Types on Input Terminals

In conventional programming languages, you manually convert between data types. HP VEE automatically converts between most data types.

Note

Data shapes are not converted on input terminals, but data types and shapes may be automatically converted when used in math functions. These conversions are discussed in "Mathematically Processing Data" in Chapter 4.

Most objects accept any data type, but a few need a particular data type or shape input. For these objects, their data input terminal automatically tries to convert the container to have 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 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 Real is a promotion. Such promotions take place automatically as needed — you rarely if ever need to be concerned with them.

A **demotion** is a conversion that loses data. For example, the conversion from a Real 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 possible.

For example, if you change the input on a `Formula` object to Int32, and another object supplies a Real 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", just double-click on the input terminal's information area (not the pin), and then click on the `Required Type` field. Double click on `Int32` in the "pop-up" list to change types.

Note

The conversion of data types for instrument I/O transactions is a special case. Refer to "Data Types" for further information.

When the conversion can't be done, HP VEE returns an error. The following table shows when conversion is automatic (yes) or when HP VEE returns an error (no). Demotions are indicated by `shading`.

Note

The Record data type has the highest priority. However, HP VEE does not automatically promote to or demote from the Record data type. To convert between Record and non-Record data, use `Build Record` and `Unbuild Record`. For further information, refer to Chapter 10.

Table 3-1. Promotion and `Demotion` of Types in Input Terminals

To ▶ ▼ From	Int32	Real	Complex	PComplex	Waveform	Spectrum	Coord	Enum	Text
Int32	n/a	yes	yes(1)	yes(1)	no	no	yes(2)	no	yes
Real	yes(3)	n/a	yes(1)	yes(1)	no	no	yes(2)	no	yes
Complex	no	no(4)	n/a	yes	no	no	no	no	yes
PComplex	no	no(4)	yes	n/a	no	no	no	no	yes
Waveform	yes(3)	yes(8)	no	no	n/a	yes(5)	yes	no	yes
Spectrum	no	no	yes(8)	yes(8)	yes(5)	n/a	yes	no	yes
Coord	no	no	no	no	no	no	n/a	no	yes
Enum	no(6)	no	no	no	no	no	no	n/a	yes
Text	yes(7)	yes(7)	yes(7)	yes(7)	no	no	yes(7)	no	n/a

Notes:

n/a = Not applicable.

(1) An Int32, or Real *value* promotes to Complex (*value*, `0`) or to PComplex (*value*, `@0`).

(2) 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.

(3) These demotions will cause an error if the value is out of range for the destination type.

(4) This demotion is not done automatically, but can be done with the `re(x)`, `im(x)`, `mag(x)`, and `phase(x)` objects or the `Build/UnBuild` ⟹ objects.

(5) An FFT or inverse FFT is automatically done.

(6) This demotion is not done automatically, but can be done with the `ordinal(x)` object.

(7) 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, ...).*

(8) These demotions keep the Waveform and Spectrum mappings.

Using Modeling Techniques

The following techniques help you to document, debug, archive, and reuse your model.

Documenting Models

To make it easier to use, modify, debug, and share your model, use the following HP VEE features to document it:

- Give the model a meaningful symbolic name. You can name your model in the HP VEE title bar.
- Rename objects to names that are more meaningful to you. For example, `Repeat 100x` may be more useful than the default `For Count`.
- Rename input and output terminals to names that are more meaningful to you. Note that you cannot change the name of some input and output terminals.

- Add descriptions about key objects by using `Show Description` on the object menu. The information may include why you used that particular object, details about the inputs and the outputs, and the options that you used on the object and why.
- Add notes to yourself or others by using `Display` ⟹ `Note Pad`. The information on `Note Pad` could include:
  - Your name, phone number, and the date you created the model.
  - What this model does.
  - Who should use this model.
  - The dates that you made changes and what the changes were.
  - Any changes you're expecting to happen in the future.
- Customize icons to display bitmaps that help you recognize them quickly. Choose your own bitmap from the icon's object menu under `Layout` ⟹ `Select Bitmap`. For details about creating your own bitmaps, refer to Appendix A.

Debugging Models

When you are building a model, you may have some problems and not know how your model is running. HP VEE has many tools to help you debug your model.

Viewing Data and Propagation

The following debugging tools are under the `Edit` menu:

- `Line Probe` allows you to view the data container output from the previous object. You can use `Line Probe` when the model is running or paused. If the model is running, `Line Probe` pauses it until you press `OK`. After selecting `Line Probe`, click on the line you want information about. (You can double-click on a terminal to see similar container information.) As a short cut, you can hold down Shift instead of selecting `Line Probe` and then click on a line to quickly view the container.

  To view the endpoints of a line, select `Line Probe`, but then *drag* the pointer over the line; the line and its endpoints are highlighted. To view the

container, release the pointer when the line is highlighted, otherwise drag the pointer off the line then release the mouse button.

- `Show Data Flow` shows the position of each data container by a small box moving on the lines between objects when the model is running.
- `Show Exec Flow` shows the object currently operating by surrounding it with a yellow highlight (default) when the model is running.

Using Breakpoints

Set breakpoints to stop the model before certain objects operate. You can set, activate, and clear breakpoints. Under the `Edit` menu, the breakpoint options are under the `Breakpoints` cascading menu.

To Set Breakpoint(s)	Do This
On a single object	Select `Breakpoint` from the object menu
On multiple objects	Select the objects, then select `Edit` ⟹ `Breakpoints` ⟹ `Set Breakpoints`

Breakpoints are saved with the model.

When you run your model, execution stops before the object with a breakpoint. At this point you can check the data on lines and terminals. An arrow points to the next object to operate (the object with the breakpoint). To continue execution, press the `Cont` or `Step` button on the upper right of the title bar.

To allow your model to run without stopping at breakpoints, deselect `Activate Breakpoints` temporarily from the `Edit` menu. The breakpoints are still on each object, but not active at the moment.

To Delete Breakpoint(s)	Do This
On a single object	Deselect `Breakpoint` from the object menu
On multiple objects	Select the objects, then select `Edit` ⟹ `Breakpoints` ⟹ `Clear Breakpoints`
On all objects	Select `Edit` ⟹ `Breakpoints` ⟹ `Clear All Breakpoints`

Stepping Through Execution

To have your model run one object at a time, press the `Step` button on the upper right of the title bar. The first time you press `Step` (from a stopped model), the model PreRuns. Press `Step` to operate each object. An arrow points to the object that will operate next.

You may wish to `Run` your model until it stops at a breakpoint and `Step` it from that point on.

When you want the model to continue running, press `Cont`. To pause the model, press `Stop` once. When the model is paused, you may continue execution by pressing `Cont` or step through by pressing `Step`.

When a `UserObject` is an icon, a panel view, or `Show Panel on Exec` is set, press `Step` to operate the entire `UserObject` (all objects in the `UserObject` operate). When a `UserObject` is an open view without `Show Panel on Exec`, press `Step` to operate each object in the `UserObject`.

Note

An infinite loop can occur using `Step` in the following situation: a UserObject contains an infinite loop, and that UserObject is reduced to an icon. When you step execution into the UserObject, the UserObject will never return since the entire UserObject must be executed before control returns. To overcome this infinite loop, open the UserObject before stepping into it. Open UserObjects don't have to execute to completion before allowing parallel threads to execute.

This infinite loop situation occurs only with `Step`, not with `Run`.

Finding Line Endpoints

To highlight a continuous line and its endpoints, select `Line Probe` and *drag* the pointer over the line. If you release the mouse button over a line, you'll see the container information.

Sharing Models With Others

Often you'll be creating models for others to use. To make it easy for others to understand your models, follow these guidelines:

- Include the `.veeio` file used (HP VEE-Test only).
- Include any custom bitmaps you've created.
- Include any program used in `Execute Program` or `To/From HP BASIC/UX` (HP VEE-Test only).
- Include any library files used via `Import Library`.
- Extensively document your model using the techniques described in "Documenting Models".

If you are sharing a disk with others and want to share models also use the following guidelines:

- Specify absolute paths in `To File`, `From File`, `Execute Program`, `To/From Named Pipes`, and `HP BASIC/UX` (HP VEE-Test) objects.
- Specify absolute paths to any `.cid` files used (HP VEE-Test only).

Archiving Models

Once your model is completed, you'll want to archive it. Use the documentation techniques listed in "Documenting Models" to fully describe the functions of the model.

For an electronic archive, use the model file itself. It is an ASCII file that documents each object, its position, any options set, and connections.

For a paper archive, use the `Print All` selection under the `File` menu. If you select all options, you'll get a printout of the entire model, the contents of all `UserObjects`, both views of the object (icon and open view), and and all `Show Descriptions`. You can also use the `veedoc` utility to get a hierarchical listing of all objects, with their `Show Descriptions` and the contents of any `Note Pads`.

Note

Depending on the size of your model, a complete paper archive may take several minutes or up to an hour to print.

If you want to change the color palette for your printout, refer to Appendix A.

To complete the paper archive, print the model file to keep with the graphical printout.

Using Global Variables

A global variable is a named variable that is set globally, and which can be used by name in any context of an HP VEE model. We'll talk more about contexts and UserObects in chapter 6. But for now, let's just look at how to use global variables in your model.

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

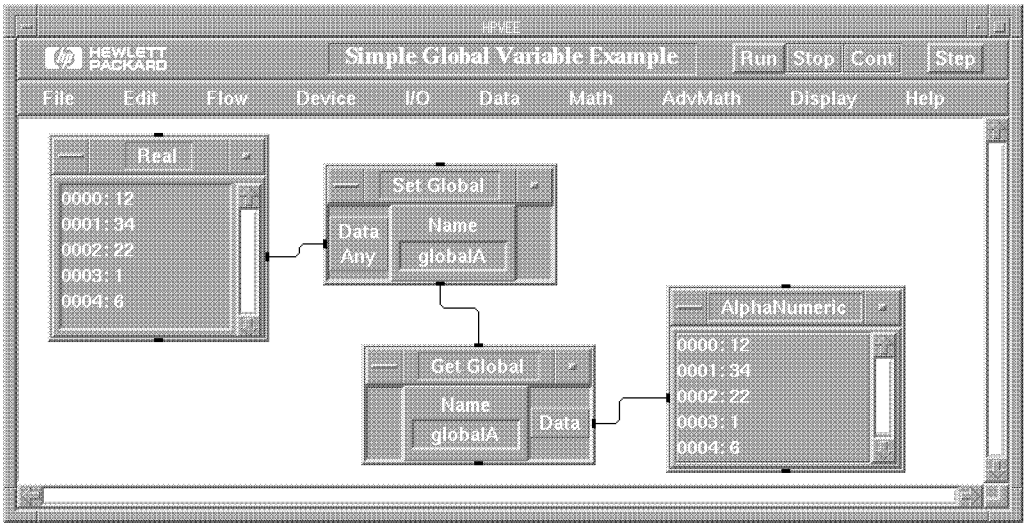


Figure 3-14. A Simple Global Variable Example

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

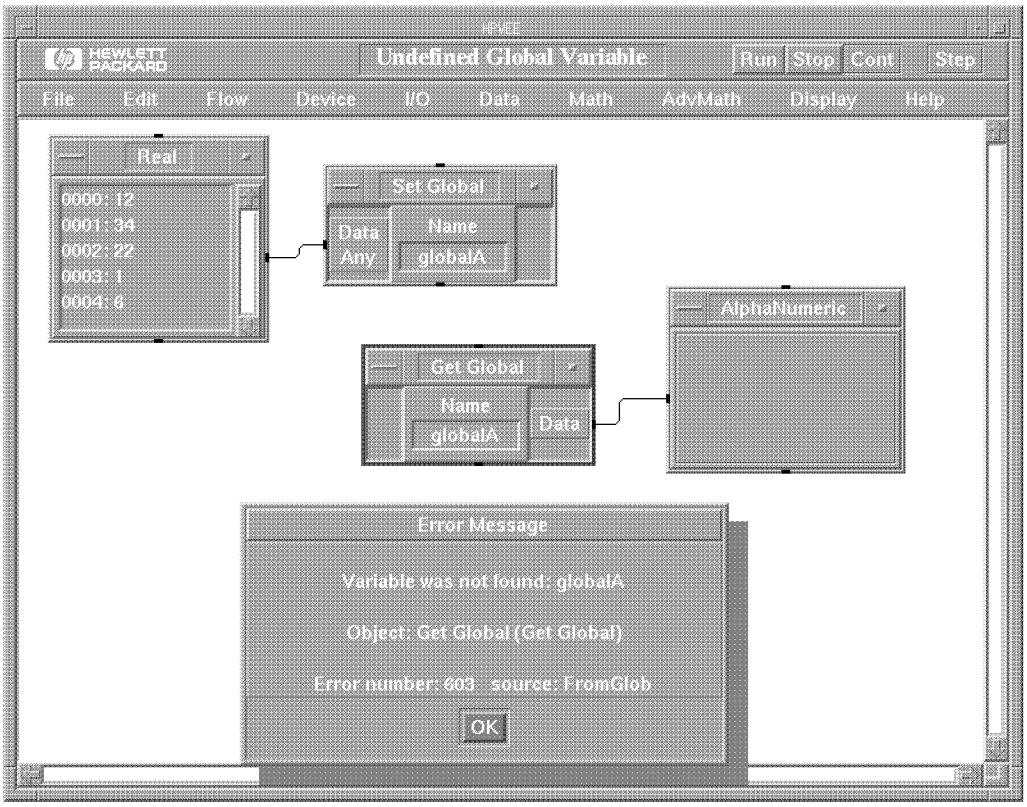


Figure 3-15. Accessing an Undefined Global Variable

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

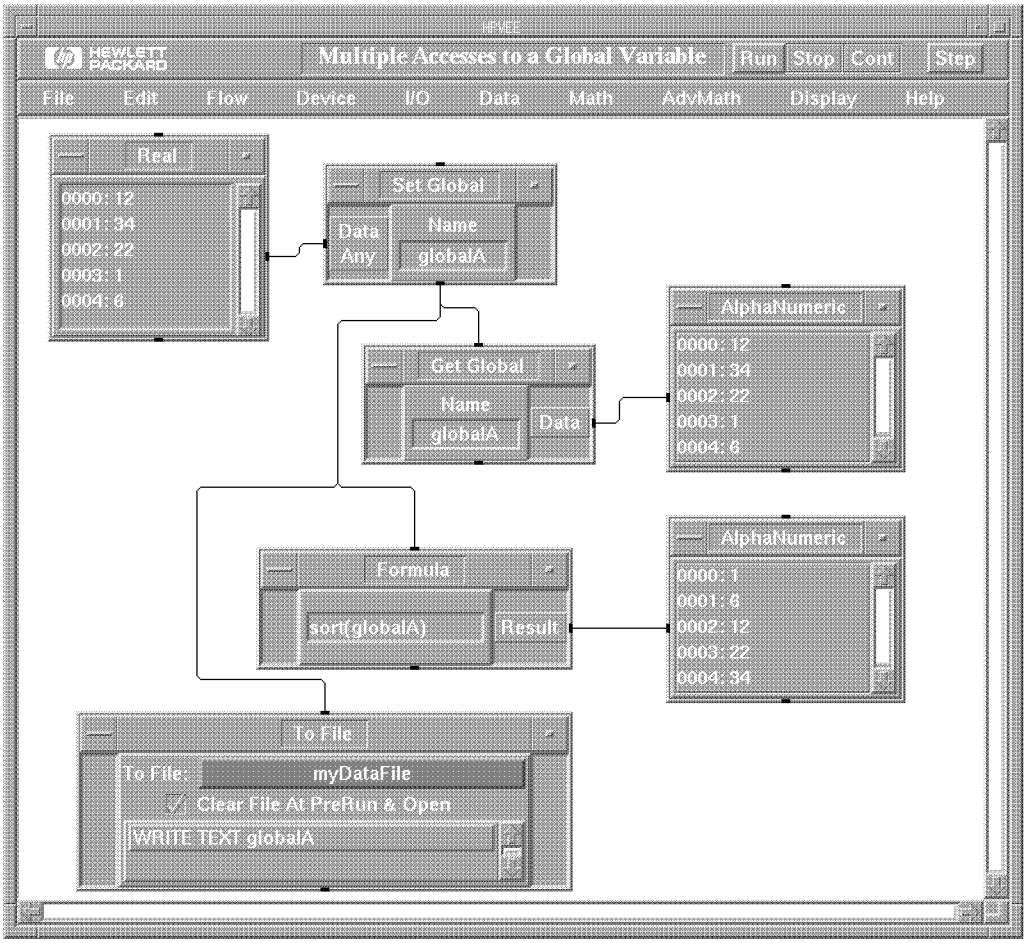


Figure 3-16. Accessing a Global Variable Multiple Times

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

Note

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

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

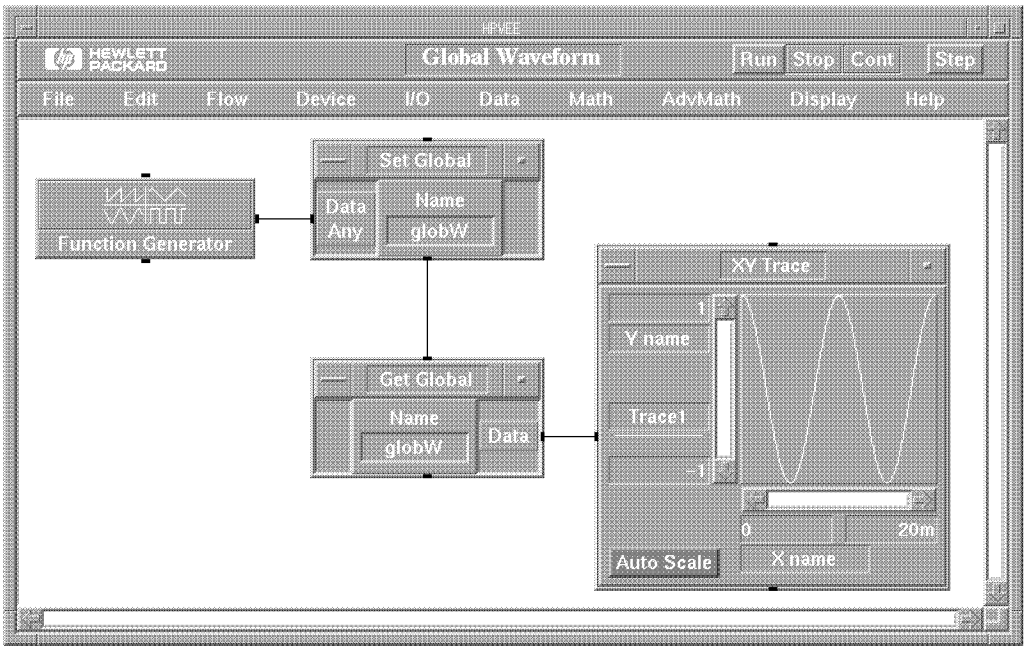


Figure 3-17. Waveform Data in a Global Variable

Note

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

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

So far we haven't talked about using global variables within different contexts of an HP VEE process. That's because we need to discuss UserObjects and contexts first. For further information about global variables, refer to "Using Global Variables in UserObjects" in Chapter 6, "Using Global Records" in Chapter 8, and the "Set Global" and "Get Global" reference sections in the *HP VEE Reference* manual.

4

Building Models

This chapter explains the general tasks you'll do when building a model. For detailed information about the specific operation of an object, refer to `Help` ⟹ `On Features` online or the *HP VEE Reference* manual.

Some common tasks involved when building a model are:

- Designing a model
- Getting data from a file
- Setting values
- Controlling flow
- Processing data
- Trapping errors
- Changing data types and shapes
- Displaying data
- Writing data to a file
- Using model information in reports

Designing Models

An important structured approach to building complex models is to modularize the many operations or functions needed. Top-down design allows you to solve complex problems by creating modules that perform particular functions. You can create a model by starting from a broad perspective and moving to the lower-level specifics when all the functions in the current level are characterized. This approach implicitly causes solutions to be modularized in terms of the functions necessary to solve a problem. HP VEE supports an environment where the creation of functional modules can be achieved quickly and, to a large part, automatically.

Although HP VEE allows you to prototype solutions quickly, you'll find that you'll create quicker, easier to understand models if you take some time to design your model in modular form instead of trying to solve the entire problem at once.

You should use `UserObjects` to create discrete modules using top-down design. Building `UserObjects` is discussed in Chapter 6.

Getting Data from Files

Often you'll want to use data created by another program or test in your model. You can read data from files by using a `From File` object from the `I/O` menu. Generally, you'll be reading numerical data from an ASCII file. The default transaction `READ TEXT x REAL` reads a single numeric value from a file.

You can change values on the transaction by clicking on the highlighted transaction, filling in fields, and clicking `OK`. Figure 4-1 reads four values (separated by spaces or on different lines) from a file.

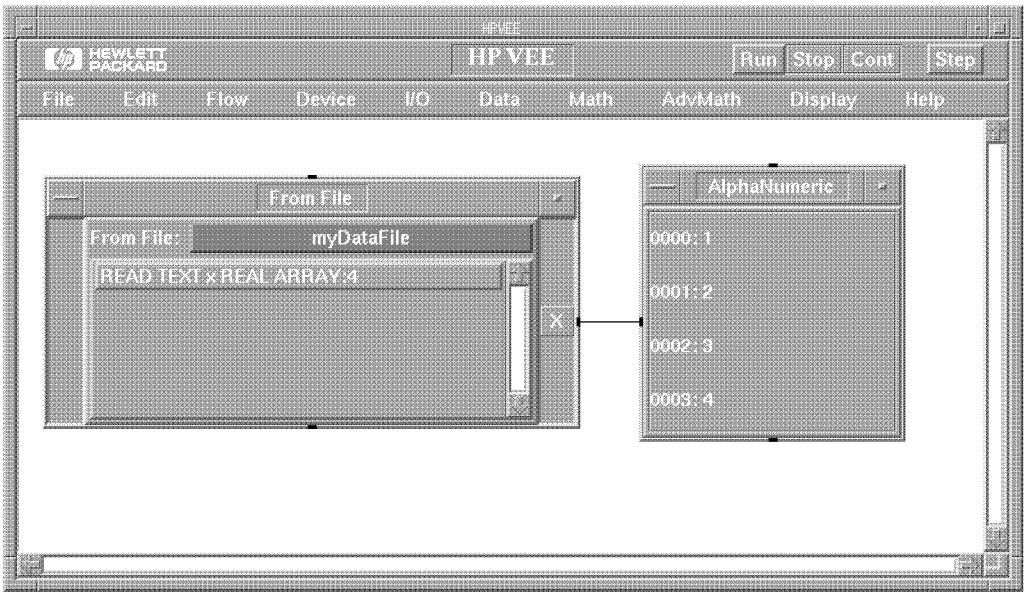


Figure 4-1. Reading Multiple Values From a File

To read multiple values from a file into an array, click on the `SCALAR` field to get the `Select Read Dimension` dialog box. Choose `ARRAY 1D` and click on `OK`. Enter the number of values to read into the field labeled `SIZE:`.

In Figure 4-2, values are read until the end of the file (`EOF`) is encountered:

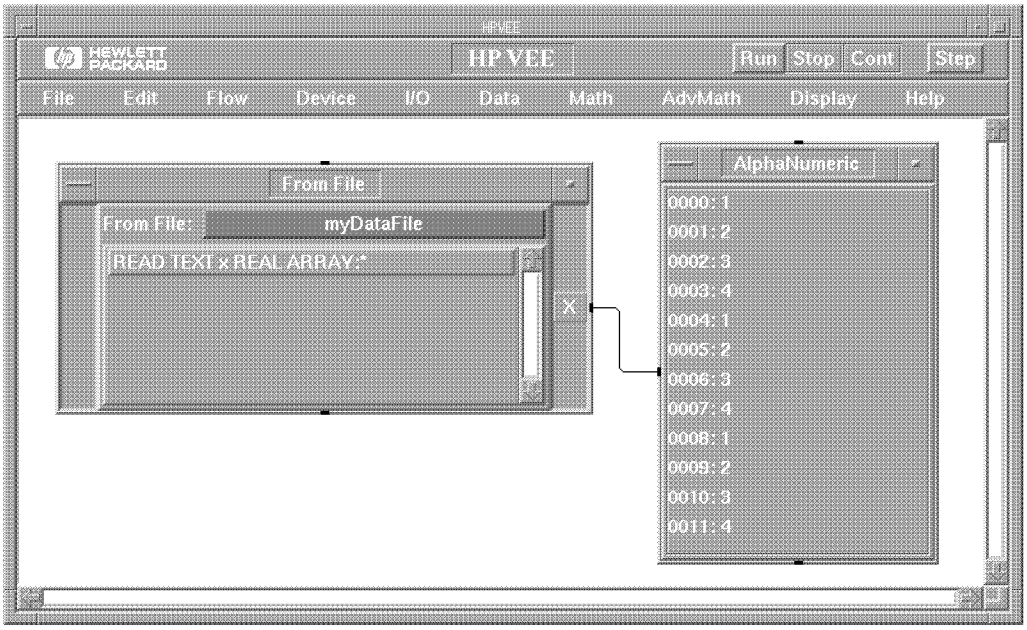


Figure 4-2. Using Read to End to Read Multiple Values

You can use the `From File` object in more complicated ways. For more information, refer to Chapter 12.

To read multiple values when you don't know the number of values, click on the `SIZE:` field to get `To End:`. In Figure 4-2, values are read until the end of file (EOF) is encountered.

Setting Initial Values

This section explains the following ways to set values for data input:

- Setting constants
- Getting user input
- Setting values at run-time
- Resetting values

Setting Constants

To set values that are constants (such as 9.8), use the `Data` ⟹ `Constant` ⟹ objects. Type the value in the entry field before you run the model. If the constant is not dependent on a sequence input, it will operate first. For more information about entry fields, refer to "Using Entry Fields" in Chapter 2.

To input a one-dimensional array of values, select `Config` from the object menu to specify the size of the array. Type the array values in the constant object.

Getting User Input

The easiest way to get user input while the model is running is to use the input objects under the `Data` menu including: `Enum`, `Slider`, and `Constant` ⟹ objects. Edit the title of the object to ask a question and let the user type in a response.

Figure 4-3 shows an example of getting user input and outputting it to a display. After you press `Run`, the `AlphaNumeric` object doesn't receive the data until the `OK` button is pressed.

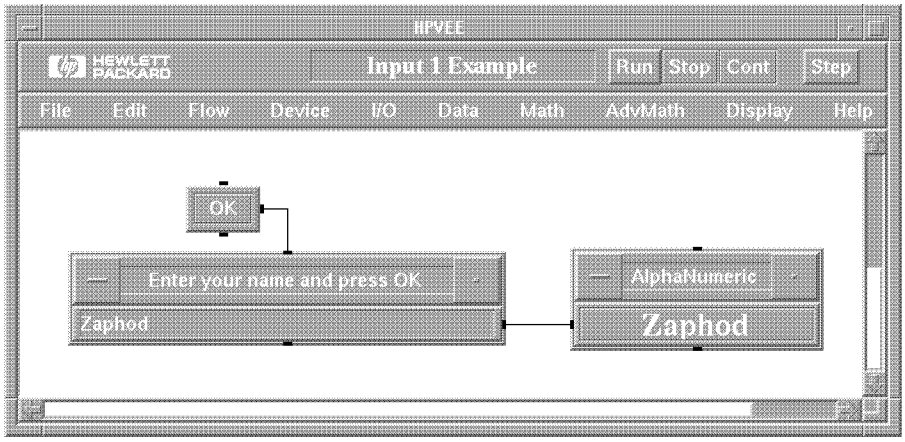


Figure 4-3. Getting User Input With a Constant Object

The model shown in Figure 4-3 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual08.ex
 - or -
/usr/lib/veetest/examples/concepts/manual08.ex
```

To have the object operate (and propagate) each time the user enters a value, set the `Auto Execute` option on the object menu. When `Auto Execute` is set, it causes the object to operate which, in turn, causes objects on the same subthread (which had their data input pins activated) to operate.

In Figure 4-4, the `Slider` is set to `Auto Execute`. A new output is displayed each time the slider is moved, . This only works because the sequence output pin of the `Slider` is connected to the sequence input pin of the `Real`. If this sequence connection was not made, the `Slider` would operate when the value was changed, but the `+` wouldn't. Because there would be no new data input to the `+` from the `Real` object, the `+` would not operate.

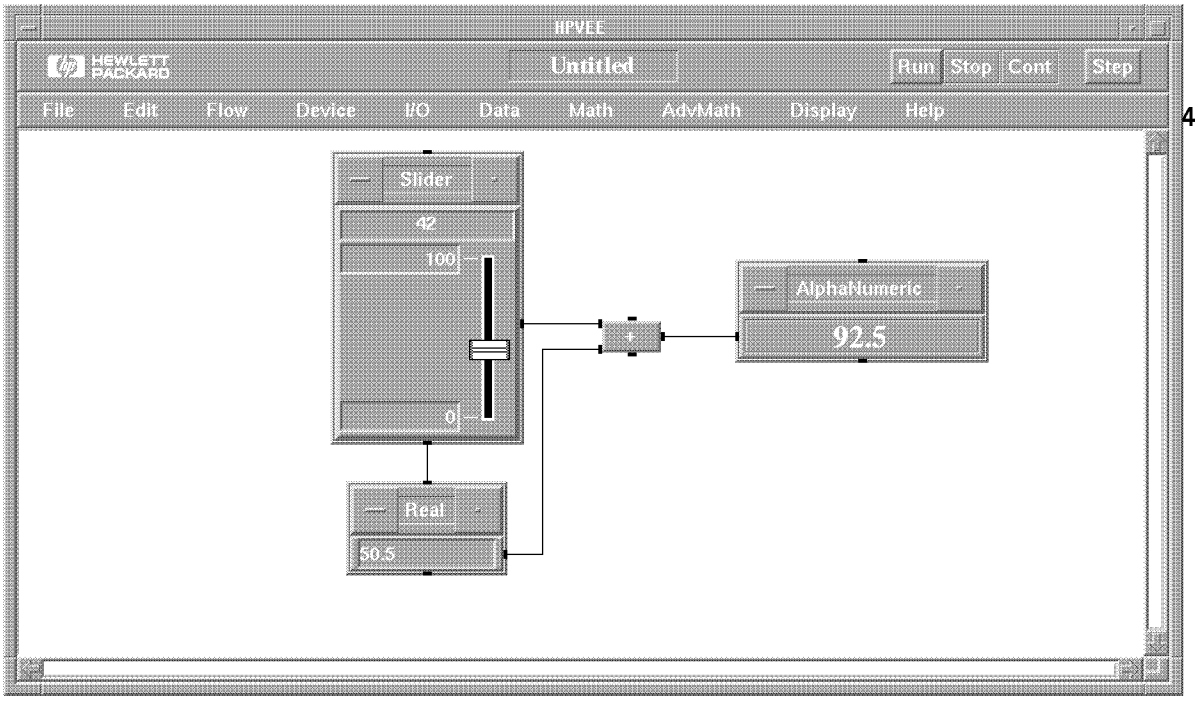


Figure 4-4. Example of Auto Execute

Refer to "Understanding Auto Execute" in Chapter 3 for information about how `Auto Execute` affects propagation. Refer to "Understanding Feedback" in Chapter 3 for information about using `Auto Execute` with feedback.

A more sophisticated method to get user input is through a dialog box. You can create your own dialog box by using a `UserObject`. For detailed information about creating your own dialog boxes, refer to Chapter 7.

Setting Values at Run-Time

Many objects allow you to set or clear values each time the model is run (at PreRun or Activate). For information about PreRun and Activate, refer to "Advanced Topic - Understanding PreRun" in Chapter 3.

The object menu features for setting values on user input objects under the `Data` menu (`Enum`, `Toggle`, `Integer Slider`, `Real Slider`, and `Constant` ⟹ objects) are:

- `Initial Value`
- `Initialize At PreRun`
- `Initialize At Activate`

The default action is not to initialize at PreRun or Activate.

Figure 4-5 shows an example where an initial value is set. If you select `Initialize` ⟹ `Initial Value` on the object menu for "Enter your name and press OK", the `Initial Value Configuration` will appear, as shown in the figure. In this case, the initial value is set to `Marvin` and is initialized at PreRun. Once the model is started, you can press `OK` to send the default value to the display, or you can type another name in the edit field and press `OK`.

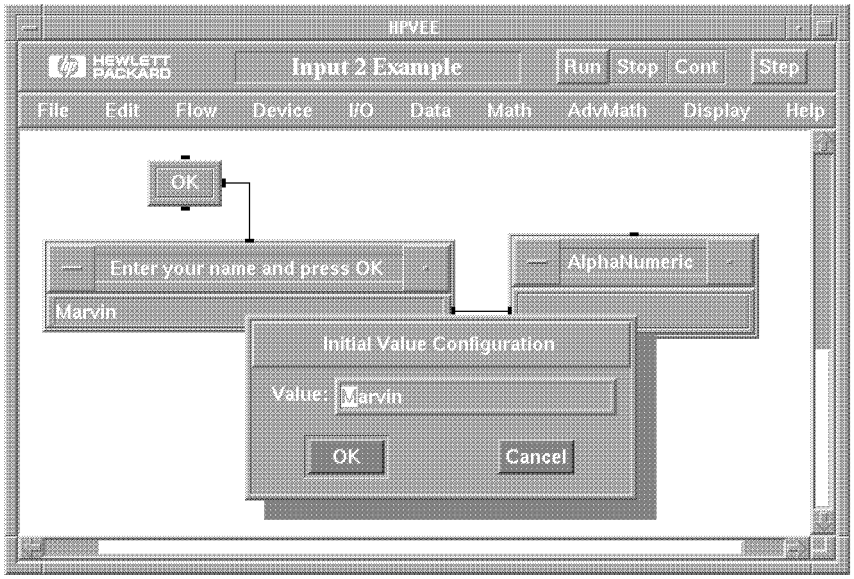


Figure 4-5. Initialize At PreRun

The model shown in Figure 4-5 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual09.ex
 - or -
/usr/lib/veetest/examples/concepts/manual09.ex
```

The object menu options for clearing values and graphs on all objects under the `Display` menu (except `Note Pad`) are:

- `Clear` (not available on `AlphaNumeric`)
- `Clear At PreRun`
- `Clear At Activate`

The default is to `Clear` at PreRun and Activate.

Resetting Values

The user-input objects under the `Data` menu (`Enum`, `Toggle`, `Integer Slider`, `Real Slider`, and `Constant` ⟹ objects), allow you to add control terminals to set a `Default Value` and to `Reset` to a cleared state. These pins set and reset the values asynchronously while the model is running (not at a specific time such as PreRun or Activate).

The `Default Value` object (a `Text` constant) in Figure 4-6 is connected to the other `Text` object (titled `Enter your Name`) by means of a control line. The `OK` button ensures (via a sequence line) that the `Default Value` object will execute before `Enter Your Name` executes.

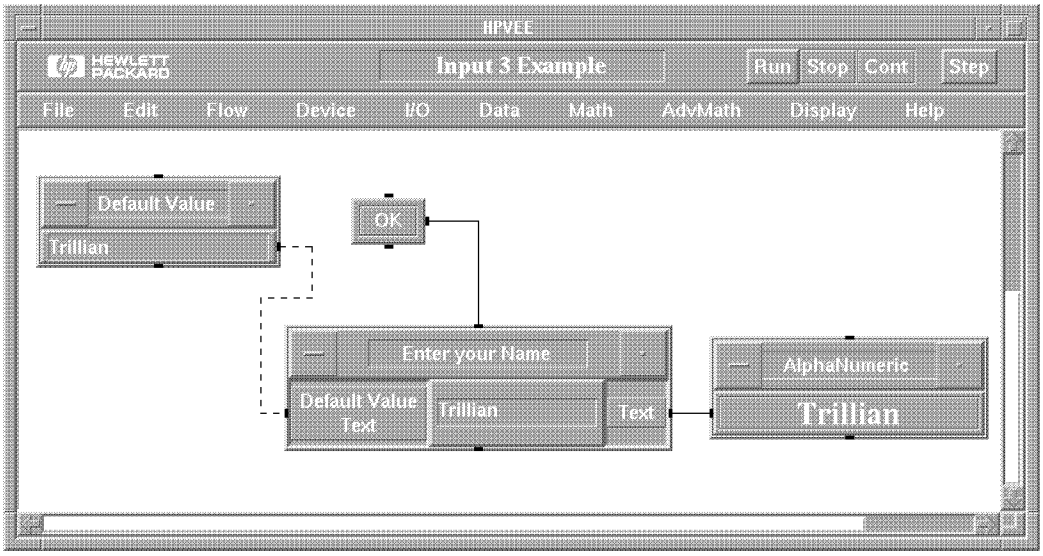


Figure 4-6. Resetting to a Default Value

The model shown in Figure 4-6 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual10.ex
 - or -
/usr/lib/veetest/examples/concepts/manual10.ex
```

Controlling Flow

Models use programming-like constructs to control the flow of operation. These controls allow you to:

- Start the model running
- Repeat the operation of a set of objects
- Branch to a subthread
- Stop running

Starting

Normally you'll start your model by simply pressing `Run`. However, if you put a `Start` object on a thread, you can start just one thread by pressing `Start`. When you are creating and debugging your model, you can use `Start` objects to test individual threads. `Run` starts all threads in the model, regardless of whether they contain `Start` objects. If you have multiple `Starts` on a thread, all of them operate when one of them (or `Run`) is pressed.

Normally, you won't need to include any `Start` objects in your model. However, there is one exception. Any thread in your model that contains feedback *must* have a `Start` object so that HP VEE knows where to start the thread. Any thread that contains feedback, but has no `Start` object, will result in an error.

Iterating

To repeat a set of operations, use the iteration objects under `Flow` ⟹ `Repeat` ⟹. Each iteration object hosts a subthread from its data output pin. The entire subthread repeats until the termination condition is met, then subthread execution stops.

Object	Terminating Condition
`For Count`	The subthread has executed *entry value* number of times
`For Range`	The output value is > the `Thru` value (if `Step` is positive) or the output value is < the `Thru` value (if `Step` is negative)
`For Log Range`	The output value is > the `Thru` value (if `/Dec` is positive) or the output value is < the `Thru` value (if `/Dec` is negative)
`Until Break`	A `Break` operates on the subthread
`On Cycle`	A `Break` operates on the subthread

When the subthread has completed, the sequence output pin of the iteration object is activated.

In Figure 4-7, the subthread that begins with the `For Count` object completes before the `For Count` sequence output pin is activated. Notice that the end of the subthread (`Times in the Loop`) marks the end of the iteration subthread.

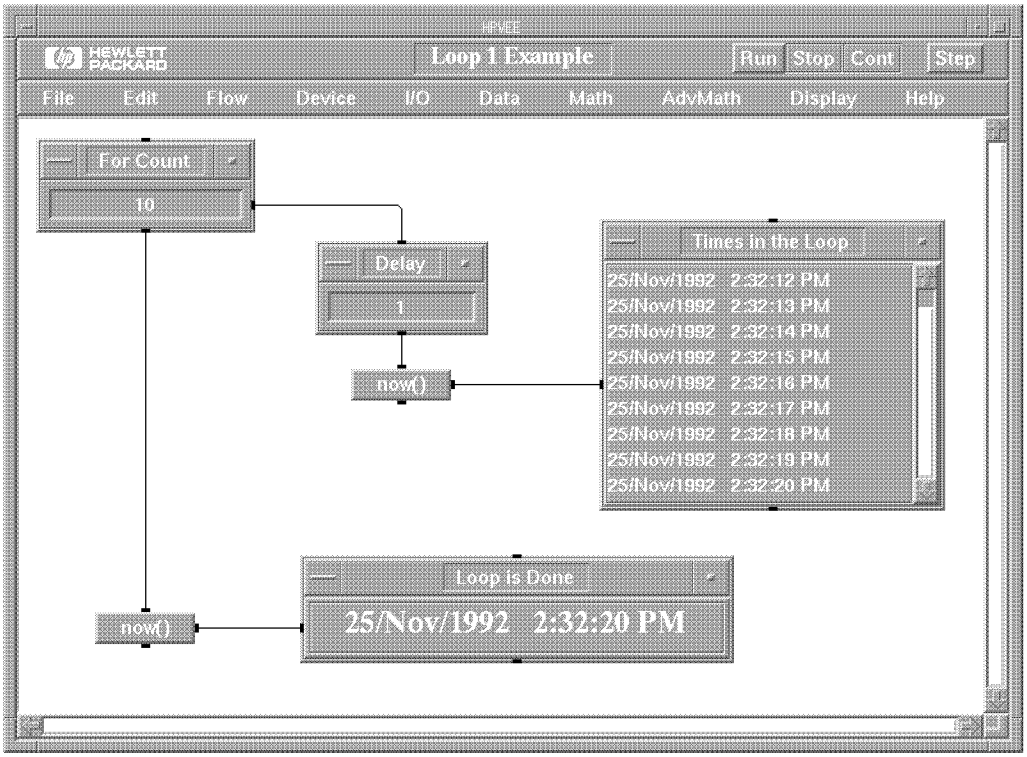


Figure 4-7. Iteration Example

The model shown in Figure 4-7 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual11.ex
 - or -
/usr/lib/veetest/examples/concepts/manual11.ex
```

To skip a set of operations in the current iteration, use the `Next` object. To stop the iteration subthread and continue propagation through the sequence output pin of the iterator, use the `Break` object.

Branching

To conditionally branch the flow of execution, use the `If/Then/Else` and `Conditional` ⟹ objects under the `Flow` menu.

The `If/Then/Else` object allows you the most flexibility. The `If/Then/Else` entry field accepts expressions and allows you to create complicated conditions. To select one of several conditions and activate the data output pin associated with that condition, add `Else/If` conditions from the object menu (note that each condition adds a data output pin to the object).

`Conditional` ⟹ objects are pre-defined `If/Then/Else` objects for your convenience. You cannot change the condition or the number of inputs on them.

Stopping

To permanently stop the model after it has run as long as you want, use the `Exit Thread` or `Stop` objects. When the `Exit Thread` object operates, it stops only the thread to which it's attached. When the `Stop` object operates, it is the same as pressing the `Stop` button on the upper right of the HP VEE title bar twice; the entire model stops.

If you want to pause the model momentarily, press the `Stop` button once. Use the `Cont` or `Step` button to continue execution. If you want your model to pause at a certain point during each execution (usually to wait for user input), use the `OK` object (`Flow` ⟹ `Confirm (OK)`) or use `Set Breakpoints`.

Iteration with Flow Branching

If your model uses *both* iteration and flow branching within a thread, there are some special considerations.

When a subthread hosted by an iterator (`For Count`, `For Range`, `For Log Range`, `Until Break`, or `On Cycle`) finishes an iteration, all data containers sent during the previous iteration are invalidated before the next iteration. This prevents "old" data from a previous iteration from being reused in the current iteration. However, if flow branching is present within the iterative subthread, some objects may not execute on every iteration. Thus, data containers sent by those objects may be invalidated before other objects can execute on the data.

An Example

In the following example, the iterative subthread hosted by `For Count` includes an `If/Then/Else` object, which causes flow branching.

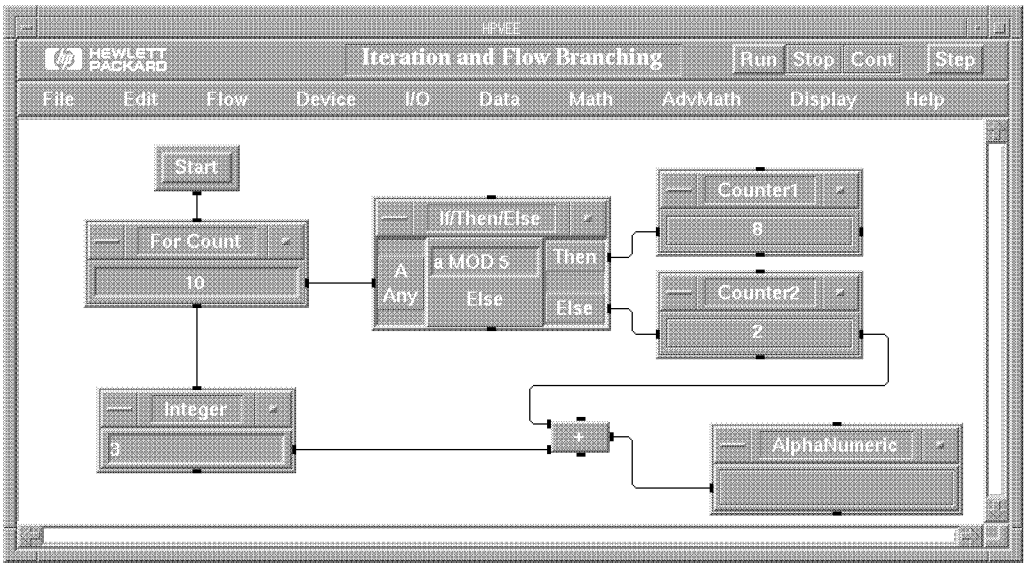


Figure 4-8. Iteration and Flow Branching

`Counter2` executes and sends the count to the `+` object. However, `Counter2` does not execute on the last iteration. Thus, the data previously sent to the `+` object is invalidated before the itertations are completed. Thus, the `+` object cannot execute to add 3 to the count because valid data is not present at one of its inputs. But this is no problem. All you need to do is add a `Sample & Hold` object.

The Sample & Hold

In the following example a `Sample & Hold` has been added to the thread of the previous example:

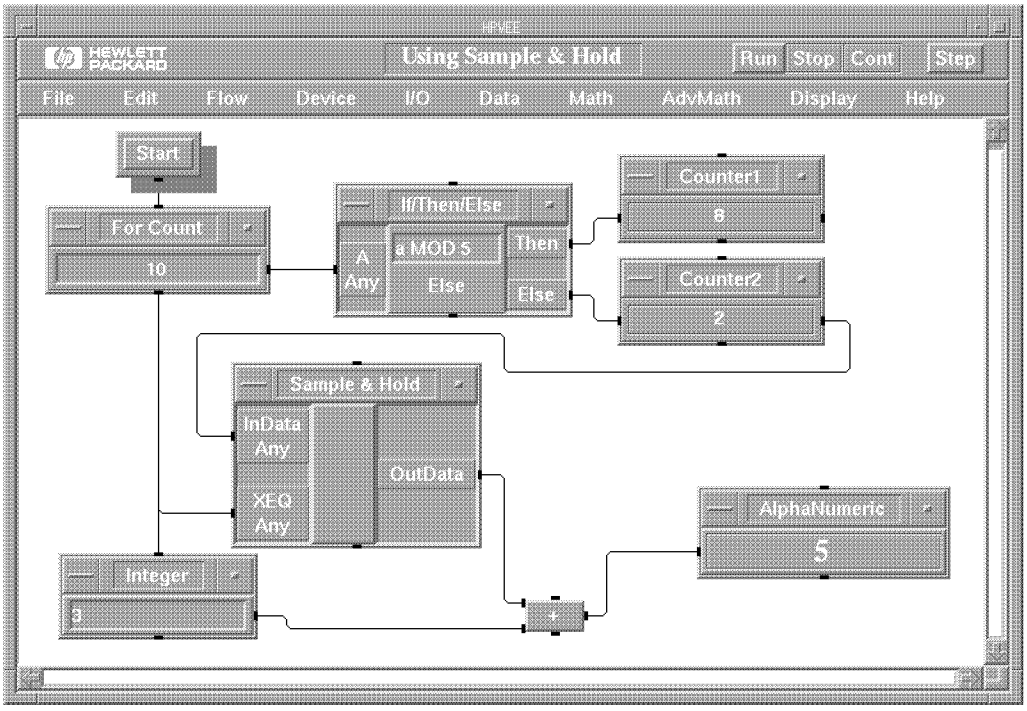


Figure 4-9. Using Sample & Hold

Each time `Counter2` executes, it sends a data container to the `Sample & Hold`, where it is stored internally. When the iterations are finished, the `XEQ` input terminal on the `Sample & Hold` is fired, and the `Sample & Hold` outputs the last data container it received to the `+` object, which adds 3 to the count.

The models shown in Figure 4-8 and Figure 4-9 are included as examples "`manual12.ex`" and "`manual13.ex`", respectively. These examples are located as follows:

```
/usr/lib/veeengine/examples/concepts/manual12.ex
/usr/lib/veeengine/examples/concepts/manual13.ex
 - or -
/usr/lib/veetest/examples/concepts/manual12.ex
/usr/lib/veetest/examples/concepts/manual13.ex
```

To see how propagation occurs in each case, load the appropriate example and run it with `Show Data Flow` and `Show Exec Flow` active.

Mathematically Processing Data

To process data, you operate on it with functions from the `Math` and `AdvMath` menus or combine the functions to create mathematical expressions.

Note

You can also process data before running a model 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; the resulting Scalar is used as the value for the field. You cannot use input variable names in the formula. You also cannot use global variables in `Constant`s. Also, 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.

The `Math` and `AdvMath` menus contain a set of mathematical functions to process your data in numerous ways. All the features that are listed under the `Math` and `AdvMath` menus (except `Regression`) can be used in any object that allows expressions. The objects that allow expressions are:

- `Math` ⟹ `Formula`
- `Data` ⟹ `Access Array` ⟹ `Get Values`
- `Data` ⟹ `Access Array` ⟹ `Get Field`
- `Data` ⟹ `Access Array` ⟹ `Set Field`
- `Device` ⟹ `Sequencer`
- `Flow` ⟹ `If/Then/Else`
- `I/O` objects that use transactions

Expressions may contain the names of data input terminals, data output terminals (`I/O` transactions only), and any mathematical expression from the `Math` menu and `AdvMath` menu. Data input terminal names are used as variables. HP VEE is not case sensitive about names of input variables within expressions for USASCII keyboards. For non-USASCII keyboards, HP VEE is case insensitive for 7-bit ASCII characters only. Expressions are evaluated at run-time.

General Concepts

Functions that are input an array operand perform the function 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]`.

All numbers in an expression field are considered Real values, unless you use parentheses to specify Complex or PComplex values. Therefore, `2` is considered to be a Real number, not an Int32. `(1, @2)` is a PComplex number, while `(1, 2)` is a rectangular Complex number.

Note

HP 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 2 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, please refer to the section titled “Understanding Containers” in Chapter 3.

Using Strings in Expressions

Strings within expressions must be surrounded by double quotes.

You may use the following escape sequences within strings:

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 Arrays in Expressions

Arrays in expressions can be used just like scalars, just refer to them by the terminal name. Array constants can be entered directly into an expression (such as `[1 2 3]`). Arrays used in functions, like `sin(x)`, have the `sin` function applied on every element of the array.

Please note, however, that negative constants in array constants are evaluated as expressions. For example, `[5 4 -3 2]` is evaluated as `[5 1 2]`. Therefore, you must specify `[5 4 (-3) 2]` instead.

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've 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:

- The comma, ",", separates array dimensions. Each sub-array operation *must* have exactly one specification for each array dimension.
- The colon, ":", specifies a range of elements from one of the array dimensions.
- The 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 you ask for `WF[0:127]`, you'll get the first half of the Waveform and a time span that is half of the old one.

Examples. `A` is an Array 1D, 10 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 6 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.

Examples. A is an Array 1D with ten elements. 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 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.

Using Global Variables in Expressions

You can create and set global variables by using the `Set Global` object, and you can access global variables by using the `Get Global` object. Refer to "Set Global" and "Get Global" in the *HP VEE Reference* manual for further information.

In addition, you can access a global variable by including its name in a mathematical expression. You can include a global 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 devices using expressions in transactions, including `To File`, `From File`, `From DataSet`, `Direct I/O`, `From Stdin`, `To/From Named Pipes`, and `Sequencer`.

To include a global variable in an expression, just use the global variable name as if it were an input variable. For example, suppose a model uses a `Set Global` device to define the global variable `numFiles`. Elsewhere in the model, a `Formula` object with input `A` may use the expression `numFiles+3*A`.

Note

Global 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, you should be aware of two limitations when you include global 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 model, an object that evaluates an expression containing a global variable may execute before the global variable is defined.* For example, suppose the global variable `globalA` is set with a `Set Global` object, and the expression `globalA*X^2` is included in a `Formula` object. Depending on the flow of your model, the `Formula` object may execute

before the Set Global 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 Global executes first. You can do this by connecting the sequence output pin of the Set Global object to the sequence input pin of the Formula object in this case, or of any other object that includes the global variable in an expression to be evaluated. If a Get Global object is used, its sequence input pin should also be connected to the sequence output pin of Set Global. For further information, refer to "Using Global Variables" in Chapter 3.

Global variables can be arrays. Just access a global variable array as if it were an input variable using array syntax, for example: GlobAry[2]. If a global 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. Note that 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 global variables, so that one global variable may hold several different values. Note that a Formula object can be used in place of a Get Global. Thus, you can accomplish the GlobRec.numFiles access in one object, instead of using both a Get Global 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 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 `Set Field` 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 use `R.A` as the left-hand expression (specifying the field to change) and `sin(R.A)` as the right-hand expression (specifying the new value for the field) in a `Set Field` object. You can continue to use the record `R` (with the new value for field `A`) later in your HP VEE model.

Using Dyadic Operators

The set of dyadic operators have several additional conditions and guidelines. The dyadic operators are under the `Math` menu and are as follows:

- `+ - * / ⟹`
  - `+`
  - `-`
  - `*`
  - `/`
  - `^` (exponentiation)
  - `mod` (modulo - returns remainder of division)
  - `div` (integer division - no remainder)
- `Relational ⟹`
  - `==`
  - `!=`
  - `<`
  - `>`
  - `<=`
  - `>=`
- `Logical ⟹`
  - `AND`
  - `OR`
  - `XOR`
  - `NOT` (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

Data Type Conversion

For the dyadic operators, the input values are promoted to the highest data type and 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. Record
2. Text (Enum is treated as Text)
3. Spectrum
4. PComplex
5. Complex
6. Coord (no conversion to any other numeric type possible)
7. Waveform
8. Real
9. Int32

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.

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 field of a record. 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. HP 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`, and so forth, there are two ways to do this. The first is to use multiple `Set Field` objects, one for each field, to change a field of an existing record. (See `Set Field` for more information.) The other way is to create a record of the same shape and "schema" as the original; put `1` in its `A` field, `2` in its `B` field, and so forth; and then add the two records.

Coord Considerations. 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; 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)`.

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 don't 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. HP VEE supplies library models for dB conversions in `/usr/lib/veeengine/lib/conversions/` or `/usr/lib/veetest/lib/conversions/`.

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 (`==`, `<`, and so on, which always return a Scalar.) If arrays have a different number of dimensions or are of different sizes, HP 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.

Trapping Errors

If you get an error while running your model, HP VEE normally stops running your model and displays an error dialog box that presents the error number and error message.

To trap the error so that the model doesn't stop and display the error dialog box, add an error output pin to the object that generates the error or to the `UserObject` that contains that object. You can put an error handling routine on the subthread that is hosted by the error pin.

When an object with an an output error pin generates an error, the error pin is activated and the container data is the error number. Double-click on the error output terminal to view the error number.

To find out the message associated with the error number, refer to the `Help` ⟹ `How To` topic `Error Codes`.

Note

An Error pin cannot trap the error you get when an object cannot convert the input container to a type or shape that the object needs. To trap this error, you must add both the object sending the container and the object receiving the container to a `UserObject` with an error pin.

An error pin on a `UserObject` traps errors generated by any object inside the `UserObject`. For more information about `UserObjects`, refer to Chapter 6.

Figure 4-10 shows how a "Divide by Zero" error is trapped and the iteration continues.

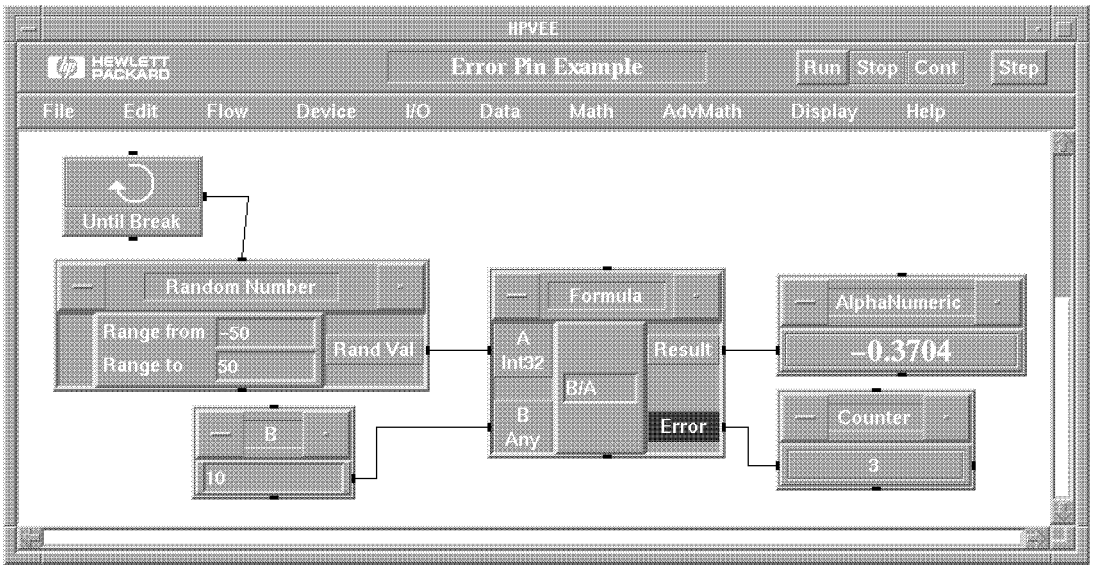


Figure 4-10. Trapping an Error

The model shown in Figure 4-10 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual14.ex
 - or -
/usr/lib/veetest/examples/concepts/manual14.ex
```

To generate an error to exit a context (a level of `UserObject`), use the `Raise Error` object. `Raise Error` generates an error and jumps the propagation to the nearest enclosing context that has an error pin. If no context has an error pin, then an error dialog box is displayed with the error number and message that you specified on the `Raise Error` object. Refer to "Raise Error" in Chapter 6 for more information.

Changing Data Types or Shapes

You may want to change the type or shape of a container if you're combining data after a user inputs it, processing data in parallel, or graphically displaying information.

To change the data type, use the `Build Data` ⟹ and `UnBuild Data` ⟹ objects from the `Data` menu.

Note

You can also change the type by changing the `Required Type` field on input terminals (where permitted); the object then converts the data type as explained in "Converting Data Types on Input Terminals" in Chapter 3.

To change the data shape, use the `Data` ⟹ objects listed in the following table:

Table 4-1. Objects That Change Data Shape

From ...	To ...	Use Object ...
no data	Array *n*-dimensions	`Alloc Array` ⟹ objects
Scalar	Array 1-dimension	`Collector`
Scalar	Array 1-dimension	`Sliding Collector`
Array	Sub-array of different size or number of dimensions	`Access Array` ⟹ `Get Values`
Array	Array of different mappings	`Access Array` ⟹ `Set Mappings`
Array (n dimensions)	Array (n+1 dimensions)	`Collector`

Note

If you build an array of Enum values, it is converted to Text. The Record and Coord data types allow only the Scalar and Array 1D data shapes.

Displaying Data

After you've processed data, you'll want to display it. HP VEE allows you to display alphanumeric values or graph numeric data.

Displaying Values

There are three objects that display values:

- `Meter` (Scalar numeric data only)
- `AlphaNumeric`
- `Logging AlphaNumeric` (Scalar and Array 1D data only)

You can change the way that numbers are displayed by using the `Number Formats` feature on the object menu.

If a Scalar value or Array 1D sent to `AlphaNumeric` or `Logging AlphaNumeric` won't fit within the object size, you'll get `***` displayed. Resize the object to see the entire value.

If an Array 2D sent to `AlphaNumeric` won't fit within the object size, scroll bars are added to the display so that you can see all the data. You may have to resize the `AlphaNumeric` to see the vertical scroll bar.

If an array with 3 or more dimensions is send to `AlphaNumeric` the string *n*`D Array` is displayed.

Graphing Data

The objects that allow you to graph your data are:

- `XY Trace`
- `Strip Chart`
- `X vs Y Plot`
- `Complex Plane`
- `Polar Plot`
- `Waveform (Time)`
- `Magnitude Spectrum`
- `Phase Spectrum`
- `Magnitude vs Phase (Polar and Smith)`

The data graphing objects require their input data shapes to be Scalar or Array 1D. If an input array is mapped, the array mappings are used for the `X` values when the data is displayed.

The axis lines or reference circle (on Polar and Smith charts) are bold on each display.

There are many features on the object menu that allow you to control and interact with the display. For example, you can change the appearance of these graphs with the `Panel Layout` ⟹ and `Grid Type` ⟹ features on the object menu.

Displaying Multiple Traces

To display more than one trace simultaneously, add a data input for each trace; the maximum number of traces is 12. Each trace is a different color. You can change the trace color, line type, and point type from the `Traces and Scales` dialog box accessed from the object menu.

To set different `Y` scales for different traces, select `Add Right Scale` from the object menu. From the `Traces and Scales` dialog box, specify the scale you want each trace to use. You can add up to two scales.

If you don't want to see the additional scales, un-check the scale choices from the `SCALES` section of the `Traces and Scales` dialog box. If you don't want to use the additional scales, select the original scale from the `TRACES` section of the `Traces and Scales` dialog box.

To change `Trace` or `Scale` attributes programmatically, use the `Traces` or `Scales` control inputs. These control inputs require data of the Record type to alter the `Trace` or `Scale`. You can merge in the file `/usr/lib/veeengine/lib/xy_cntrl` or `/usr/lib/veetest/lib/xy_cntrl` to get pre-built Records for these inputs. Or you can build your own Records. Refer to `/usr/lib/veeengine/examples/lib/xy_cntrl.ex` or `/usr/lib/veetest/examples/lib/xy_cntrl.ex` for an example of their use.

To display a family of curves (from one data input) while the model is running, activate the `Next Curve` control pin before sending each trace to the display.

To display a family of curves (from one data input) when each curve is generated each time you run your model, select the `Next Curve` object menu

feature before you run the model each time. `Clear At Activate` and `Clear At PreRun` must not be set.

Using Markers

To examine successive points on a trace, use markers to mark the points and display the values at these points. Add markers from the `Markers` ⟹ feature on the object menu. You can place markers only at actual data points unless you choose `Interpolate` from the `Marker` ⟹ features.

To move markers between traces, click on the trace color button near the marker name until it displays the destination trace's color. Then click on the destination trace to place the marker.

When you have displayed a family of curves (by using `Next Curve`), you can drag the marker along any of those curves.

Writing Data to Files

Often you'll want to save data created by your model in a file. Write data to a file by using a `To File` object from the `I/O` menu. Generally, you'll be writing to an ASCII (text) file. The default transaction `WRITE TEXT a EOL` writes the data from a single container to a file.

Once the information is in a file, it may be read by other programs or merged into reports.

You can use `To File` in very sophisticated ways. For more information, refer to Chapter 12.

Exporting Model Graphics to a Report

You can use graphical model information such as the model itself, parts of the model, or a display, in a report. You can include an image as it appears on the screen, or for displays, an HPGL representation of the display.

To get a graphical image from HP VEE, use `Printer Config` to specify a graphics directory (not a printer). To do this, click on `Graphics Printer` to toggle to `Graphics Directory` and edit the text field next to it.

Use one of the following techniques to get the desired output:

- To get an image of the open view of an object when the model is running, activate the `Print` control pin on that object. This technique is especially useful for display objects.
- To get an image at a certain point when the model is running, operate the `Print Screen` object.
- To get an image when the model is not running, select `Print Screen`, `Print Objects`, or `Print All` from the `Edit` menu

Output Formats

The screen information is stored in either the "xwd" (X Window Dump) format or the Postscript format.

Postscript files may be sent directly to a Postscript printer using the UNIX `lp` command.

The xwd format may be converted into many common graphics formats such as TIFF and PCL by graphics packages or utilities. For example, to transform an `xwd` file into PCL, use the `xwd2sb` and `pcltrans` utilities. These utilities are provided with HP VEE in case they are not already installed on your UNIX system. For example:

`cat` *xwdfile* `| /usr/lib/veeengine/xwd2sb | /usr/lib/veeengine/pcltrans -r300 -e3 >` *pclfile*

-or-

`cat` *xwdfile* `| /usr/lib/veetest/xwd2sb | /usr/lib/veetest/pcltrans -r300 -e3 >` *pclfile*

You can print a pcl file, once it has been converted, by executing the UNIX `lp -oraw` command. For example:

`lp -oraw` *pclfile*

For more information about `xwd2sb` and `pcltrans`, refer to their UNIX `man` pages.

To get an HPGL or HPGL/2 plot, set `Plotter Config` to specify a file (not a plotter). Use one of the following techniques to obtain the desired output:

- To get a plot when the model is running, activate the `Plot` control pin on the display.
- To get a plot when the model is not running, select `Plot` from the display `Object Menu`.

Exporting to Document Publishing Packages

Often, you may want to capture images of models, objects or displays in HP VEE and put them in your report. In order to do so, you need to save your image or plot into a file as described above. Choose a file format that your documentation package can import. The steps for exporting graphics to FrameMaker and to programs of the Island Productivity Series are described here. The general techniques are applicable to other documentation publishing packages as well.

FrameMaker

FrameMaker can import files in `xwd` format. It cannot import regular Postscript files. A filter for HPGL files is currently available from FrameMaker. The steps described here are for importing an `xwd` file.

1. Save your image to a file in `xwd` format. Use `Print Objects` or the object's `Print` control pin if you only want one object. Use `Print All` if you want an image of your entire model. Use `Print Screen` if you want an image of the screen.
2. In FrameMaker, use the `File` ⟹ `Import` command. Choose a scaling option (100 dpi is a good place to start).
3. Once the image is in your document, you can crop, scale or rotate the image. You can also add titles and labels.

Note

The colors (or shades of gray) may appear lighter in the resulting hardcopy than on the display. This is because the colors were adjusted to compensate for darkening that occurs during the gray-scale conversion prior to printing. You may want to use the `Dark` setting in the HP VEE `Printer Config`.

The Island Productivity Series

The Island Productivity Series consists of three programs, IslandPaint, IslandDraw, and IslandWrite. To get your image into IslandWrite, you'll have to use either IslandPaint or IslandDraw, depending on the format of your file. IslandPaint can import xwd files, and IslandDraw can import Postscript and HPGL files.

The steps to import a xwd file are:

1. Save your image to a file in xwd format. Use `Print Objects` or the object's `Print` control pin if you only want one object. Use `Print All` if you want an image of your entire model. Use `Print Screen` if you want an image of the screen.
2. In IslandPaint, use the `File` ⟹ `Convert` command. Choose the "X11 Window Dump" (xwd) format to `Open and Convert From`.
3. Once the image is imported, you can edit it in IslandPaint.
4. Next you need to move the image to IslandWrite. There are two ways to do so:
   - First, save the image to a file in IslandPaint format using the `File` ⟹ `Save As` command. Then, in IslandWrite, use the `File` ⟹ `Import` command to import the file using the TIFF/IslandPaint format. Choose an appropriate scaling factor (100 dpi is a good place to start). You must have an appropriate container in your document to put the image in.
   - Use the IslandPaint clipboard to transfer the image. In IslandPaint, select the image you want to transfer, then use Cut or Copy to put the image on the clipboard. Then go to IslandWrite and use `Paste IslandPaint` to bring the image into a container. You will probably have to scale it once it is in the document.

The steps to import a Postscript (HPGL) file are:

1. Save your image to a file in Postscript format. Use `Print Objects` or the object's `Print` control pin if you only want one object. Use `Print All` if you want an image of your entire model. Use `Print Screen` if you want an image of the screen. (Save your plot to a file in HPGL format. Use `Plot` on the display's `Object Menu` or the display's `Plot` control pin.)
2. In IslandDraw, use the `File` ⟹ `Convert` command. Choose the Postscript (HPGL) format to `Open and Convert From`.
3. Once the image is imported, you can edit it in IslandDraw.
4. Next you need to move the image to IslandWrite. There are two ways to do so:
   - First, save the image to a file in IslandDraw format using the `File` ⟹ `Save As` command. Then, in IslandWrite, use the `File` ⟹ `Import` command to import the file using the IslandDraw format. Your document must have an appropriate container in which to put the image.
   - Use the IslandDraw clipboard to transfer the image. In IslandDraw, select the image you want to transfer, then use `Cut` or `Copy` to put the image on the clipboard. Then go to IslandWrite and use `Paste IslandDraw` to bring the image into a container.

Note

The colors (or shades of gray) may appear lighter in the resulting hardcopy than on the display. This is because the colors were adjusted to compensate for darkening that occurs during the gray-scale conversion prior to printing. You may want to use the `Dark` setting in the HP VEE `Printer Config`.

5

Using Instruments

If you are using HP VEE-Engine, please skip this chapter.

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

Note

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

Instrument Control Fundamentals

HP VEE-Test supports three types of objects for controlling instruments:

- State Drivers
- Component Drivers
- Direct I/O

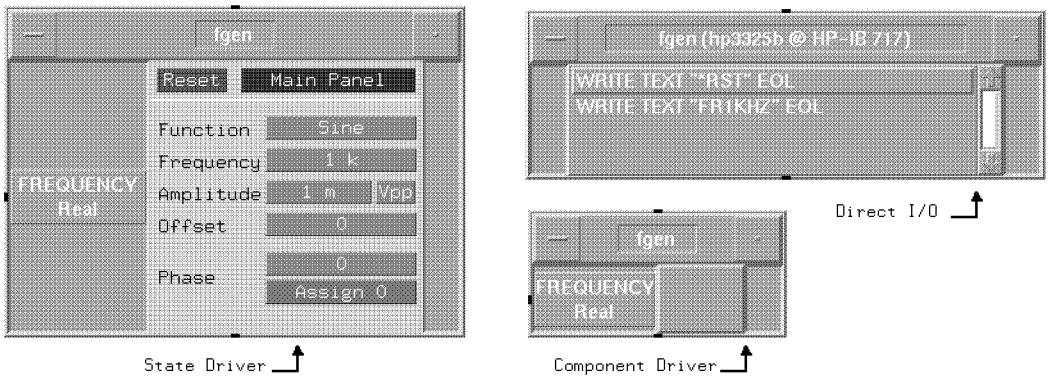


Figure 5-1. Instrument Control Objects

Each of the open-view objects shown in Figure 5-1 controls an HP 3325B function generator. Notice that each type of instrument control object has a different appearance. This appearance directly relates to the differences in how the objects operate and how you use them.

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

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

Driver-Based Objects

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

State Drivers

`State Drivers` serve two purposes in HP VEE-Test:

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

As shown in Figure 5-1, the open-view of a `State 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 on the fields in the graphical control panel. You can also make measurements and display the results by clicking on numeric and XY displays.

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

To clarify these concepts, consider a simple example. Suppose you want to program the HP 3325B function generator to provide two different output signals:

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

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

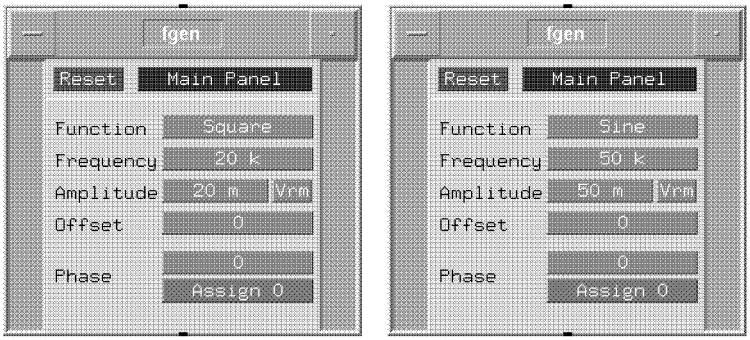


Figure 5-2. Two `State Drivers`

Component Drivers

5

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

`Component Drivers` are provided to help you improve the execution speed of your model; speed is the only advantage they provide over `State Drivers`. The execution speed of a model 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.

To clarify these points, consider a simple example. Suppose you want to program the HP 3325B Function Generator to do the following:

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

In this case, it makes sense to use a `State Driver` to perform the initial setup and a `Component Driver` to repeatedly set the output frequency. Figure 5-3 shows a model that does this.

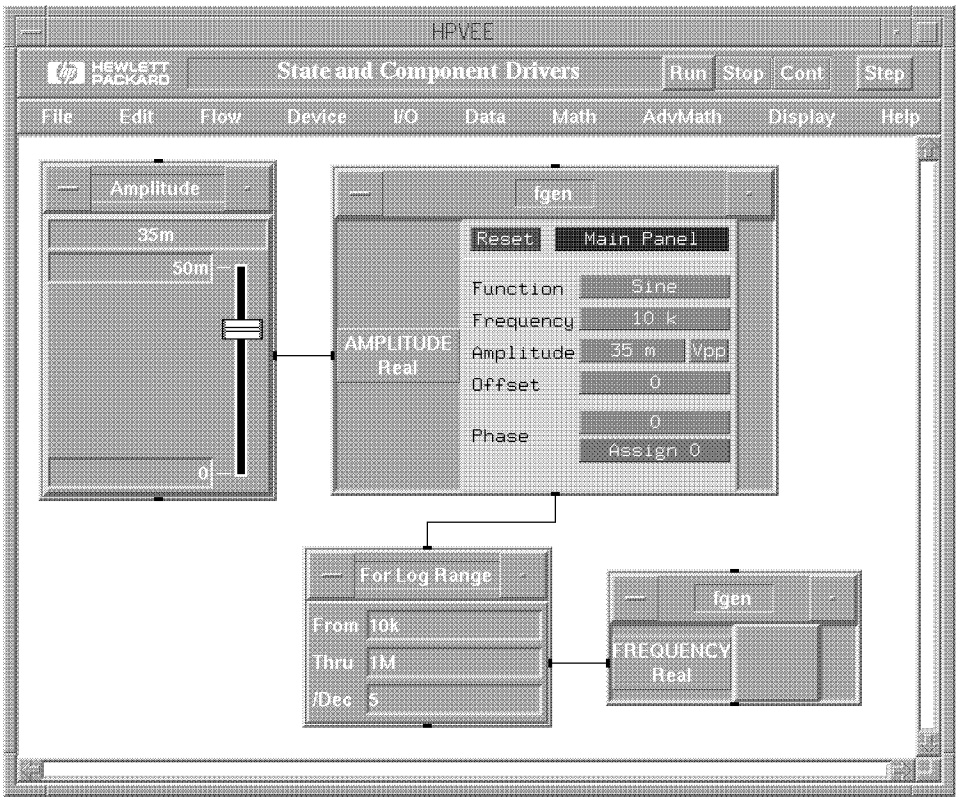


Figure 5-3. Combining `State Drivers` and `Component Drivers`

Direct I/O

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

`Direct I/O` objects also provide convenient support for learn strings. A learn string is a special feature supported by some instruments that allows you to set up measurement states from the front panel of the physical instrument. Once the instrument is configured, you simply select `Upload` from the `Direct I/O` object menu to upload the entire measurement state of the instrument. You can recall the measurement state from within your model by using the `Direct I/O` object.

5

This chapter describes how to *configure* HP VEE-Test to use `Direct I/O`. For details about how to *use* `Direct I/O` objects, please refer to "Communicating with Instruments" in Chapter 12.

To complete the comparisons of different types of I/O objects, consider the previous model in Figure 5-3. You could replace the `Component Driver` in Figure 5-3 with `Direct I/O` as shown in Figure 5-4.

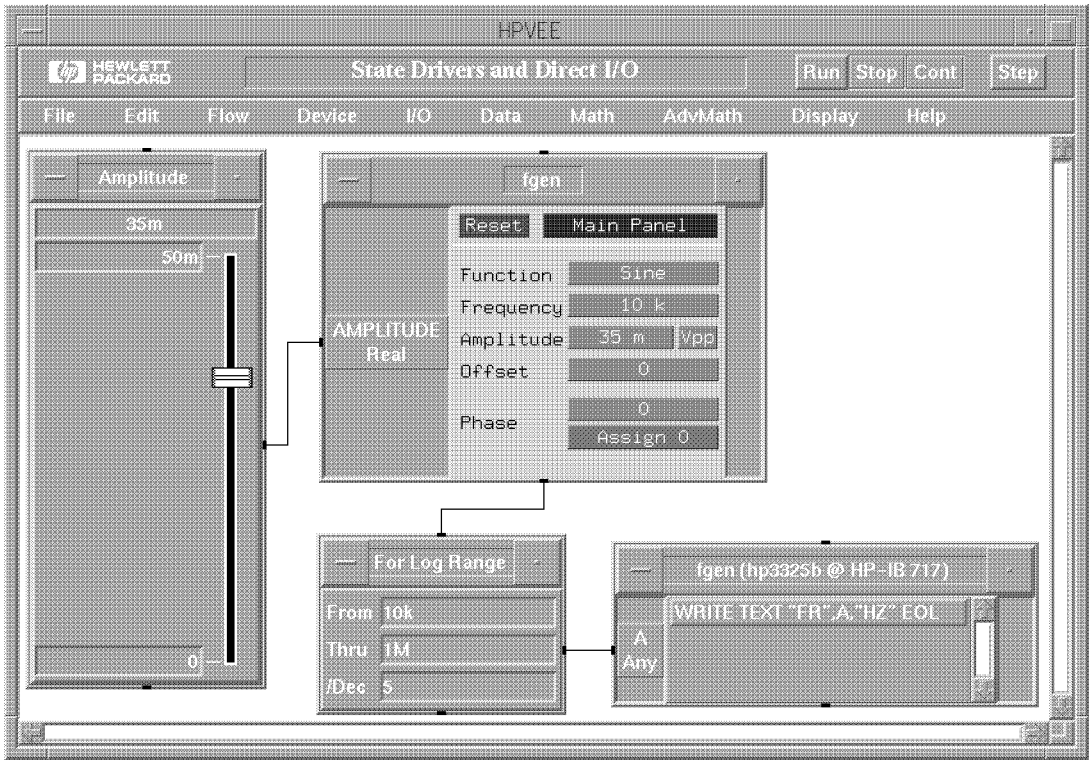


Figure 5-4. Combining `State Drivers` and `Direct I/O`

Summary of Instrument Control Objects

Table 5-1. Instrument Control Objects

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

5

Terminating I/O Operations

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

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

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

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

1. Use the mouse to position the pointer in a terminal window.
2. Determine the UNIX process identification number (PID) for HP VEE-Test using the UNIX `ps` command.

3. Type `kill -2` *vee_pid*, where *vee_pid* is the PID number determined in the previous step.

Using Instrument Control Examples

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

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

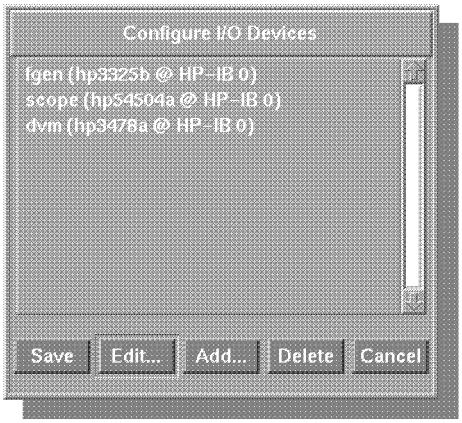


Figure 5-5. Default I/O Configuration

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

Understanding State and Component Drivers

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

Inside HP Drivers

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

Driver Files

Key Idea

Each HP VEE-Test instrument **driver file** describes the unique personality of a particular test and measurement instrument. A driver file is required to control any instrument using a `State Driver` or `Component Driver` object.

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

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

Components

Key Idea

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

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

Typical Voltmeter Driver Components

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

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

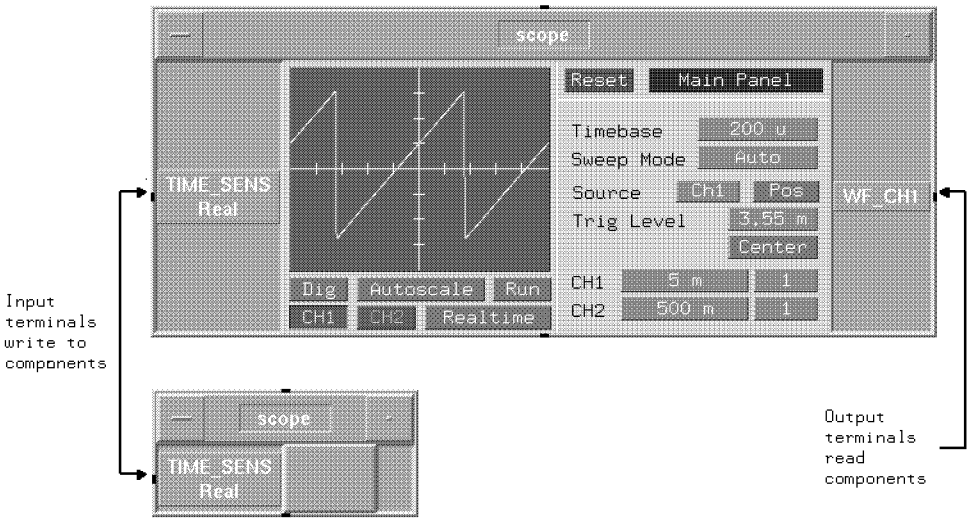


Figure 5-6. Accessing Driver Components

States

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

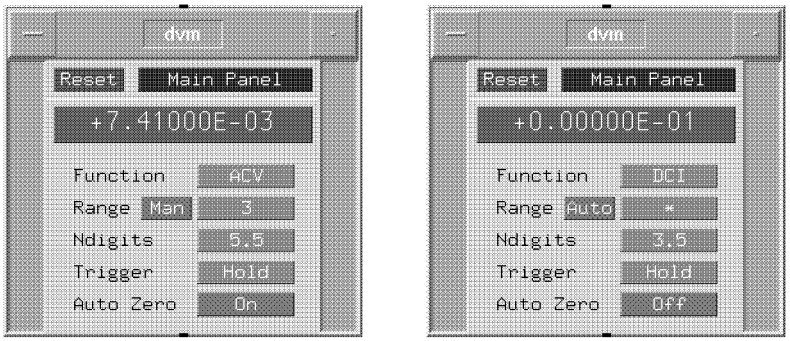


Figure 5-7. Two Voltmeter States

5

Key Idea

In HP VEE-Test, each instance of a `State Driver` represents a separate measurement state. It is common to have more than one `State Driver` in a model, where each `State Driver` programs the *same* physical instrument to a unique measurement state.

Each `State Driver` object you create using the same instrument `Name` will communicate with the same physical instrument.

How Driver-Based I/O Works

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

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

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

State Driver Operation

Key Idea

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

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

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

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

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

Component Driver Operation

Key Idea

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

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

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

5

Multiple Driver Objects

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

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

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

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

Key Idea

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

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

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

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

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

`Names` are also important for saving and opening models containing instruments. When you save a model, the `Name` of each instrument object in the model is saved. When you open a model, HP VEE-Test looks in the current I/O configuration for the `Name` of each instrument being loaded. For example, if you saved a model containing an `Direct I/O` object with a name of `My Scope`, there must be an instrument named `My Scope` in the current I/O configuration. `Name`s must match *exactly*, including any spaces and upper/lower-case letters. Furthermore, if the object under consideration is a `State Driver` or `Component Driver`, the `ID Filename` (driver file) in the current I/O configuration must match the one used in the saved model.

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

Choosing the Correct Instrument Object

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

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

Configuring Instruments

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

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

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

5

Basic Instrument Configuration

Follow this procedure to configure *any* instrument you plan to use with HP VEE-Test.

1. Click on `I/O` ⟹ `Configure I/O`.
2. Note the device (instrument) entries listed in the `Configure I/O Devices` dialog box.
   a. If you need to use instruments that do not appear in the list, go to step 3 in this procedure.
   b. If the instruments you need are listed but the addresses or other settings are incorrect, click on the instrument you wish to modify, then click on the `Edit` button. The `Device Configuration` dialog box appears. Go to step 4 in this procedure.
   c. If the list properly specifies all your instruments, click on the `Cancel` button and exit this procedure.
3. Click on the `Add` button. The `Device Configuration` dialog box appears.
4. HP VEE-Test enters default values in the fields inside the `Device Configuration` dialog box. If you have difficulty interpreting any of the fields in this dialog box, refer to the section, "Device Configuration Dialog Box" later in this chapter. Complete or modify each entry field as follows:

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

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

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

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

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

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

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

- `Timeout:` Enter the timeout in seconds. The default value of five seconds works for most applications. It is unadvisable to specify `0` in this field; if you do, HP VEE-Test 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:` Set this field to `ON` if the corresponding instrument is connected to your computer. Set this field to `OFF` if the corresponding instrument is not connected to your computer so that objects will not attempt to read or write to a non-existent instrument.

5. At this point, you have completed all the configuration steps that are common to drivers and direct I/O. You must still complete the steps for driver configuration, direct I/O configuration, or both.

   - To configure your instrument for use with a driver, go to the following section,"Driver Configuration".
   - To configure your instrument for use with direct I/O, go to the following section, "Direct I/O Configuration".
   - To configure a VXI device for register access with direct I/O, go to the "A16 Space Configuration (VXI only)" section, later in this chapter.
   - To configure a VXI device for extended memory access with direct I/O, go to the "A24/A32 Space Configuration (VXI only)" section, later in this chapter. *We will use the term "extended memory" to indicate either A24 or A32 memory in a VXI device.* (A VXI device can implement either A24 or A32 memory, but not both.)

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

Driver Configuration

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

1. Click on the `Instrument Driver Config` button. The `Instrument Driver Configuration` dialog box appears.
2. Click on the `ID Filename` field. A list of driver files appears.
3. Click on the driver file corresponding to your instrument, then click on the `OK` button.
4. HP VEE-Test enters default values in the fields inside the `Instrument Driver Configuration`. In general, you do not need to change these values. Consult the section, "Instrument Driver Configuration Dialog Box" later in this chapter for details about this dialog box.

5. Click on the `OK` button in the `Instrument Driver Configuration` dialog box.
6. At this point you have completed all the steps necessary to configure this instrument for use with a driver.
   - To configure this instrument for use with direct I/O, go to the following section, "Direct I/O Configuration".
   - To configure additional instruments, click on the `OK` button in the `Device Configuration` dialog box. Go to step 3 of the section, "Basic Instrument Configuration" earlier in this chapter.
   - To stop configuring instruments and save the current configuration, click on the `OK` button in the `Device Configuration` dialog box, then click on the `Save` button in the `Configure I/O Devices` dialog box.

Direct I/O Configuration

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

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

b. To stop configuring instruments and save the current configuration, click on the OK button in the Device Configuration dialog box, then click on the Save button in the Configure I/O Devices dialog box.

To learn how to use the Direct I/O instruments you have just configured, refer to "Communicating with Instruments" in Chapter 12.

A16 Space Configuration (VXI only)

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

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

A24/A32 Space Configuration (VXI only)

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

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

Details of Configure I/O Dialog Boxes

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

Device Configuration Dialog Box

Name

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

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

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

Note

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

Interface

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

Address

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

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

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

Note

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

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

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

- *xx* is the one- or two-digit select code of the VXI backplane interface of an embedded or external controller. The factory default select code is 16 for either the HP 75000 Series C Model V/382 VXI Controller or the *first* E1489I MXI Interface in an HP 9000 Series 700 system.

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

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

Note

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

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

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

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

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

Assume you wish to control an instrument using an HP 98642 Four-Channel Multiplexer. The instrument is connected to port 3, the highest-numbered port available on the interface. The default interface select code is 17. Assuming that the select code has not been changed, the proper `Address` field setting for this instrument is `1703`. Note that the HP 98642 interface supports separate addresses for each port: `1700`, `1701`, `1702`, and `1703`.

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

Device Type

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

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

Timeout

The `Timeout` field specifies how many seconds HP VEE-Test will wait for an instrument to respond to a request for communication before generating an error. The default value of five seconds works well for most applications. In general, you should not set this field to `0`; if you do, HP VEE-Test 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

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

Note that if `Live Mode` is `OFF` for instrument X, you can run models containing `State Drivers`, `Component Drivers`, or `Direct I/O` objects that would otherwise read and write to instrument X. However, no instrument communication actually takes place. The latter behavior can be useful if you wish to develop or debug portions of a model while instruments are not available.

Config Buttons

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

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

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

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

Instrument Driver Configuration Dialog Box

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

ID Filename

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

If you are unsure which driver to use, please refer to the online information in `Help` ⟹ `On Instruments`.

Sub Address

The `Sub Address` field specifies the subaddress used by certain drivers to identify plug-in modules in cardcage-type instruments, such as data acquisition systems and switches. If you are *not* configuring a driver for one of these plug-ins, set this field to `-1`.

Note

Since *very* few drivers use subaddresses, the default setting of `-1` is the proper setting in 99% of all situations.

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

Note

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

Incremental Mode

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

Note

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

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

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

Note that the `Incremental Mode` setting affects only `State Drivers`.

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

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

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

5

Error Checking

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

Direct I/O Configuration Dialog Box

This section explains each field in the `Direct I/O Configuration` dialog box. Interfaces which support transactions configured with this dialog box include HP-IB, VXI (message-based devices only), GPIO, and serial.

Note

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

Read Terminator

The `Read Terminator` field specifies the character that terminates `READ` transactions. The entry in this field must be a a single character surrounded by double quotes. "Double quote" means ASCII 34 decimal. HP VEE-Test recognizes any ASCII character as a `Read Terminator` as well as the escape characters shown in Table 5-2.

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

Table 5-2. Escape Characters

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

EOL Sequence

The `EOL Sequence` field specifies the characters that are sent at the end of `WRITE` transactions that use `EOL ON`. The entry in this field must be zero or more characters surrounded by double quotes. "Double quote" means ASCII 34 decimal. HP VEE-Test recognizes any ASCII characters within `EOL Sequence` including the escape characters shown previously in Table 5-2.

Multi-field As

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

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

Array Separator

The `Array Separator` field specifies the character string used to separate elements of an array written by `WRITE TEXT` transactions. The entry in this field must be a a single character surrounded by double quotes. "Double quote" means ASCII 34 decimal. HP VEE-Test recognizes any ASCII character as an `Array Separator` as well as the escape characters shown previously in Table 5-2.

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

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

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

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

Array Format

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

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

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

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

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

5

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

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

```
WRITE TEXT A
```

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

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

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

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

Note

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

END On EOL (HP-IB Only)

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

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

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

Conformance

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

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

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

- Binblock
- State
- Upload String
- Download String

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

IEEE 728 block headers are of the following forms:

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


where:

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

<Data> is a stream of arbitrary bytes.

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

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

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

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

Serial Interface Settings

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

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

Note that setting `Handshake` to `Xon/Xoff` enables start/stop control for the associated serial interface. Output is stopped by sending ASCII DC3 and started by sending ASCII DC1.

Data Width (GPIO only)

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

A16 Space Configuration Dialog Box (VXI only)

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

Byte Access

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

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

Word Access

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

LongWord Access

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

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

Add Register

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

- `Name` - The symbolic name of the register, which is used to refer to the particular register in a `Direct I/O` object using `READ REGISTER` or `WRITE REGISTER` transactions.
- `Offset` - The offset in *bytes* from the *relative* base of a device's A16 memory for the register being configured.

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

Delete Register

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

A24/A32 Space Configuration Dialog Box (VXI only)

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

Note

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

Byte Access

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

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

Word Access

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

LongWord Access

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

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

Byte Ordering

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

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

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

Add Location

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

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

- REAL32 - Read or write a 32-bit word. The 32-bits are represented as a IEEE 754 32-bit floating-point number. HP VEE-Test supports this format even if the LongWord Access field is specified as NONE (by using two D16 accesses to read or write all 32 bits). If the LongWord Access field is specified as D32 ACCESS, all 32 bits are accessed.

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

Delete Location

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

Notes on Configuring a VXI Device

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

Figure 5-8 shows the A16 Space Configuration dialog box with the register configuration of an HP E1411B Multimeter. The Offset field is configured with the offset in bytes of each register from the *relative* base of the device's A16 space. Notice that there are two registers with an offset of four bytes, status and control. Register status is configured for READ mode only, while register control is configured for WRITE mode only. While two separate register locations could have the same mode, the Name field must be unique.

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

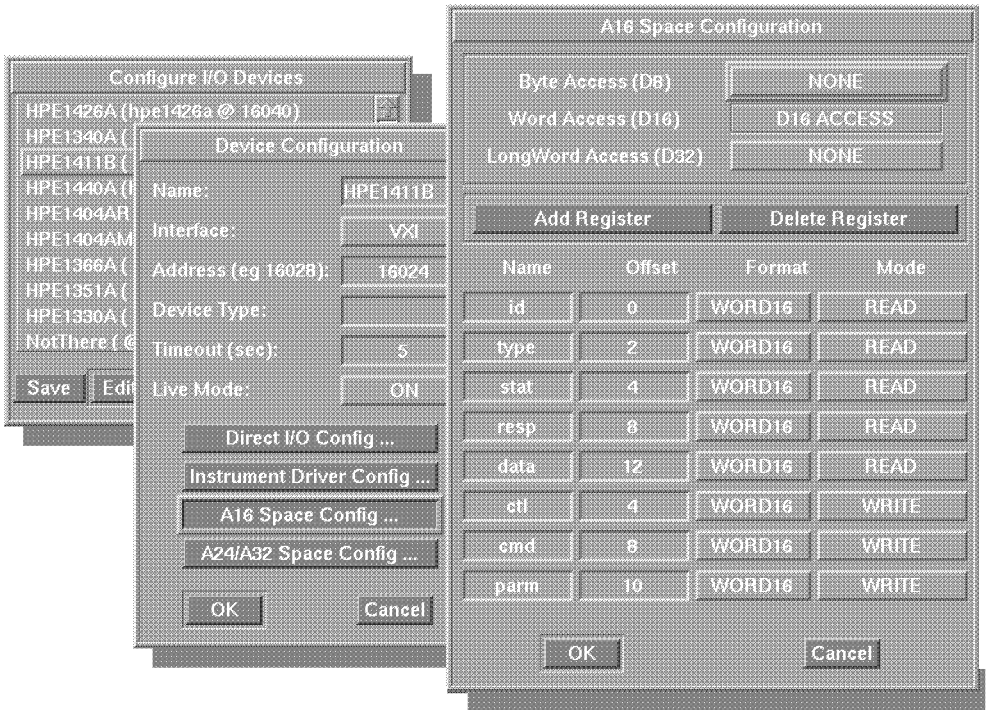


Figure 5-8. A16 Configuration for the HP E1411B Multimeter

Advanced Topic - I/O Configuration File

This section contains information that is beyond the needs of most first-time users of HP VEE-Test.

This section discusses a special file that you may occasionally need to modify. This file is `.veeio`, the I/O configuration file. It is stored in your `$HOME` directory (typically your `/users` directory). If you have difficulty understanding this section, ask your system administrator for help.

When you configure instruments using `I/O` ⟹ `Configure I/O`, you click on the `Save` button in the `Device Configuration` dialog box to save the settings. These settings are saved not only in memory for the remainder of your work session, but also in `.veeio`. This way, the next time you execute `veetest`, you can continue working with your existing I/O configuration.

If you do not have a `.veeio` file in your `$HOME` directory when you execute `veetest`, HP VEE-Test creates a default `.veeio` for you. This happens when you execute `veetest` for the first time. This default configuration contains the instruments used in the online examples included with HP VEE-Test.

You cannot open any model 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 `Device Configuration` dialog box, not the text in the object's title bar. Furthermore, if the object under consideration is a `State 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 models, although other setting affect how the models *execute*.

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

Sharing Models

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

Running Example Models

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

1. Manually add the default instrument configuration to your current configuration using `I/O` ⟹ `Configure I/O`.
2. Rename your `.veeio` file and restart HP VEE-Test.

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

Default I/O Configuration

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

If you choose method 2, follow this procedure:

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

Using State Drivers

Using State Drivers Interactively

The open view of a `State Driver` provides a graphical control panel that you can use to interactively construct a measurement state. If you connect the corresponding physical instrument to your computer and turn `Live Mode` on, you can control the physical instrument interactively as you build the measurement state. To change an individual setting, click on the corresponding field (*not the label*) in the graphical control panel and complete the resulting dialog box. To make a measurement and view the result, click on the display region of a numeric or XY display. Note that XY displays may take a few seconds to update.

Using State Drivers in a Model

To add a `State Driver` to your model:

1. Click on `I/O` ⟹ `Instrument`. The `Select an I/O Device` dialog box appears.
2. Click on the name of the desired instrument to highlight it, then click on the `State Driver` button.

   If the `State Driver` button is disabled (flat), this instrument has not yet been configured for use as a `State Driver`. Refer to the section "Configuring Instruments" earlier in this chapter 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 `State Drivers` in a model, you will often need to add input or output terminals. Each input or output terminal actually corresponds to a component in the driver. There are two ways to add a terminal:

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

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

Using Component Drivers

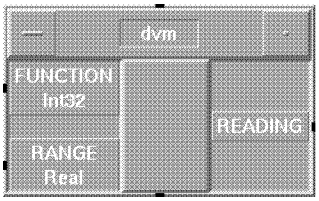


Figure 5-9. A Typical Component Driver

Using Component Drivers in a Model

To add a `Component Driver` to a model:

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

   If the `Comp Driver` button is disabled (flat), this instrument has not yet been configured for use as a `Component Driver`. Refer to the section "Configuring Instruments" earlier in this chapter for configuration procedures.
3. When the object outline appears, position the pointer and click once to place the object in the work area.

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

Figure 5-10 illustrates this type of situation. This model measures the frequency response of a filter by sweeping the input frequency sourced by `fgen` and measuring the response using `dvm`. Since the subthread attached to `For Log Range` executes repeatedly, component drivers are used to improve execution speed. Note that state drivers are still appropriate for the initial set up of `fgen` and `dvm`.

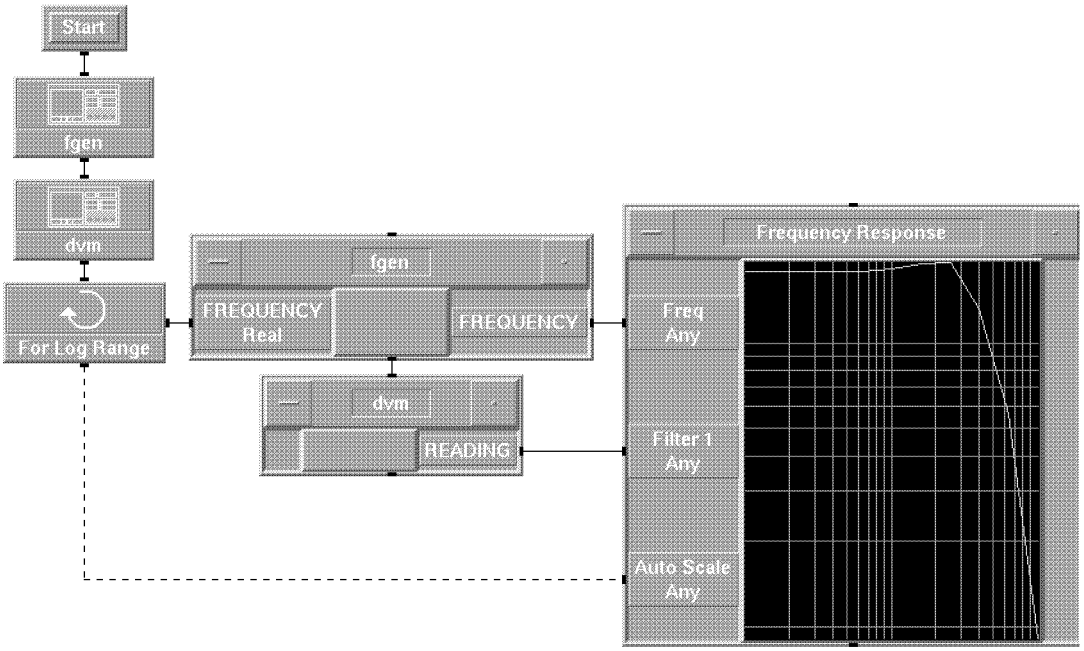


Figure 5-10. Using State and Component Drivers

The model shown in Figure 5-10 is stored in this file:

```
/usr/lib/veetest/examples/instr_io/manual15.ex
```

Advanced I/O Control

This section explains the objects accessed via the I/O ⟹ Advanced I/O menu and how to download measurement subprograms to instruments. This menu choice was previously called Advanced HP-IB.

Polling

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

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

Object	Serial Poll Behavior
Device Event	The Device Event object can poll the specified instrument once and output a scalar integer, which is the serial poll response using the NO WAIT option. The Device Event object can also wait for a specific bit pattern within the serial poll response byte by using a user supplied bit mask and the ALL CLEAR and ANY SET options.
Direct I/O	Direct I/O objects for HP-IB instruments support a WAIT SPOLL transaction. This transaction repeatedly polls an instrument until the serial poll response byte matches a specific bit patteren, using a user-supplied bit mask and the ALL CLEAR or ANY SET options. Refer to Chapter 12 for details about Direct I/O.

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

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

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

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

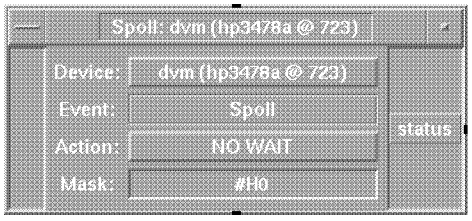


Figure 5-11. Device Event Configured for Serial Polling

Service Requests

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

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

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

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

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

The model shown in Figure 5-12 shows how to handle service requests. In the case shown, it is possible that either dvm or scope is responsible for a service request. The model determines the originator of the service request by using Device Event to obtain the status byte of each instrument. Each status byte is tested using If/Then/Else and the bit(x,n) function to determine if bit 6 is true. If bit 6 is set, then the corresponding instrument is responsible for the service request. The Until Break object automatically re-enables the entire thread to handle any subsequent service requests. The Device Event object is configured for NO WAIT, meaning the status byte is returned without using the mask value. If a mask value of 64 is used and the Device Event object is configured for ANY SET, the If/Then/Else and bit(x,n) function will not have to be used.

Note that different instruments have different requirements for clearing and re-enabling service requests. In Figure 5-12, dvm requires only a serial poll to clear and re-enable its SRQ capability. However, scope requires the additional step of clearing the originating event register.

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

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

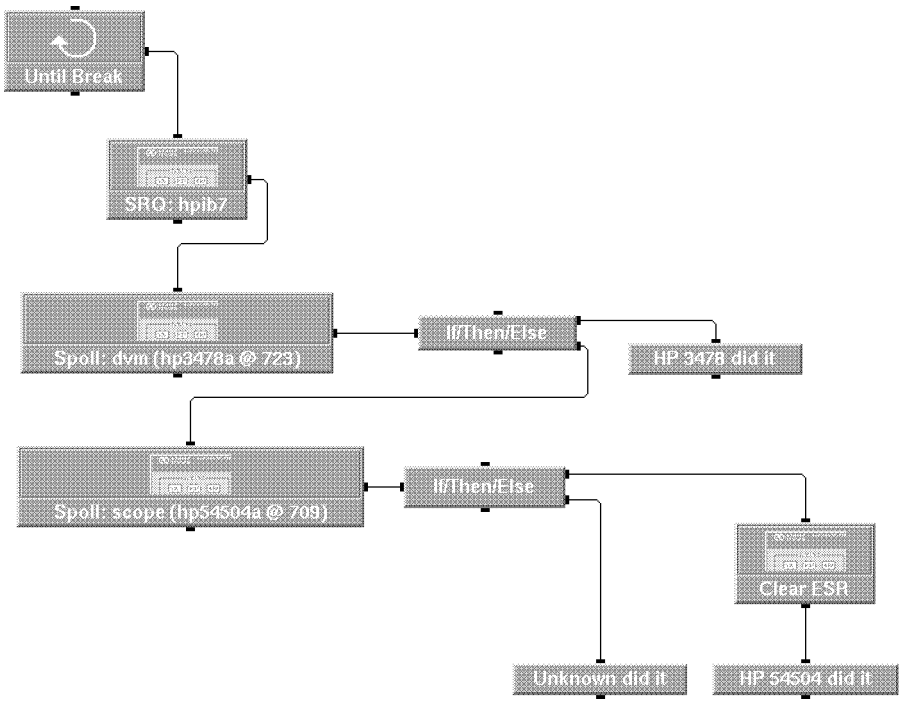


Figure 5-12. Handling Service Requests

The model shown in Figure 5-12 is saved in this file:

```
/usr/lib/veetest/examples/instr_io/manual16.ex
```

Monitoring Bus Activity

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

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

Figure 5-13 shows the bus messages sent to write `*RST` to an instrument at HP-IB address 717.

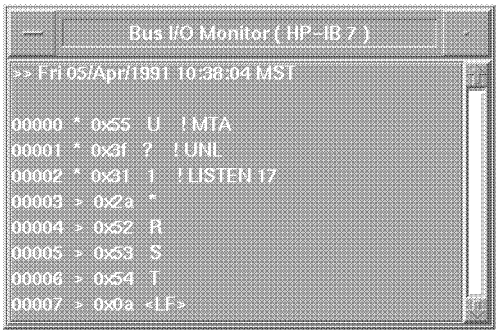


Figure 5-13. The Bus I/O Monitor

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

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

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

Low-level Bus Control

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

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

For details about `Interface Operations` and `Direct I/O`, please refer to "Communicating with Instruments" in Chapter 12.

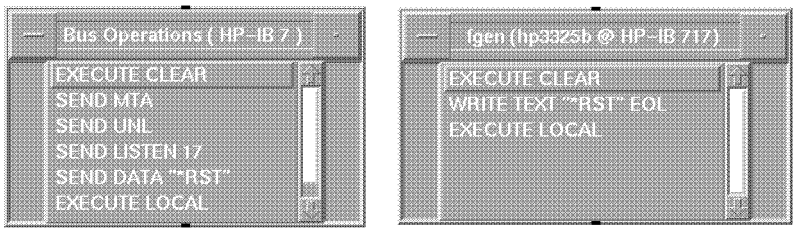


Figure 5-14. Two Methods of Low-Level HP-IB Control

Instrument Downloading

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

1. Create and maintain your measurement routine using a text editor, such as `vi`. Save the measurement routine in an ordinary text file.
2. Use `From File` to read the file.

3. Use `Direct I/O` to write the contents of the file to the instrument.

The following section presents a complete example of downloading using this approach. Please refer to Chapter 12 for details about `From File` and `Direct I/O`.

Example of Downloading

Figure 5-15 shows a model that downloads a measurement subprogram to the HP 3852A. This example downloads a simple subprogram, `BEEP2`, that beeps twice and displays a message.

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

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

   `Name: HP 3852A`

   `Interface: HP-IB`

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

   `Device Type: HP 3852A`

   `Timeout: 5`

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

2. Click on the `Save` button.

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

```
DISP MSG "LOADING BEEP2"
WAIT 1

SUB BEEP2
DISP "SUB BEEP2 CALLED"
BEEP
BEEP
SUBEND

DISP MSG "BEEP2 LOADED"
```

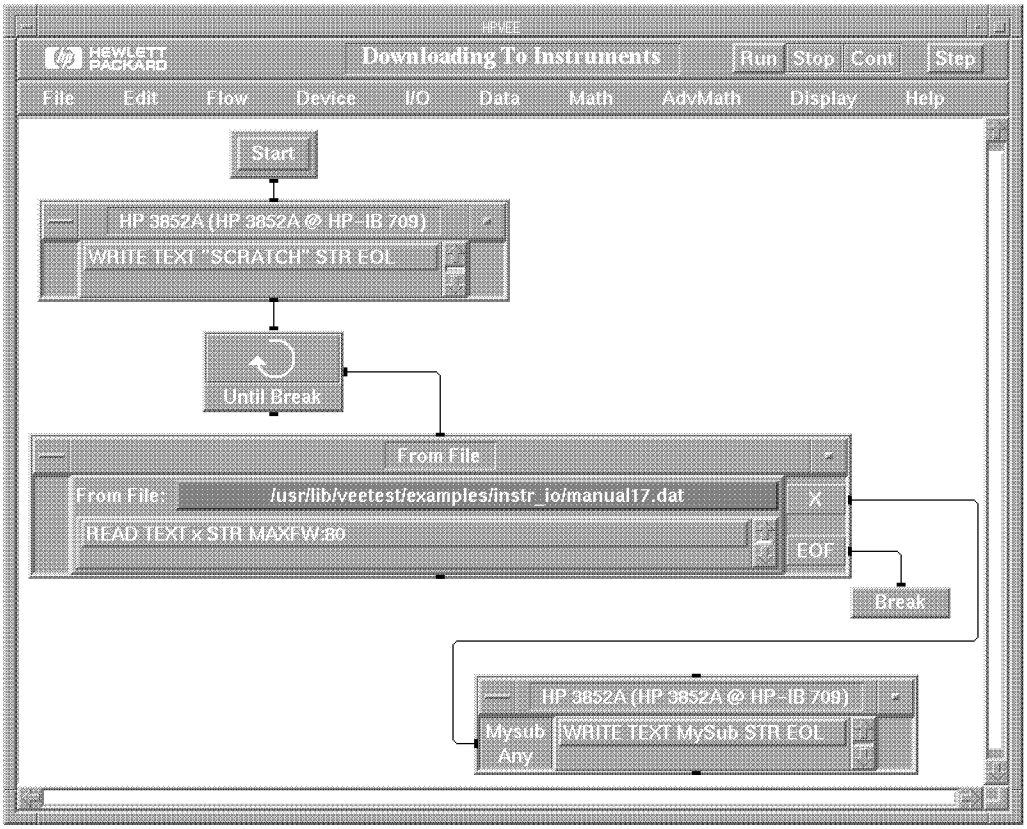


Figure 5-15. Downloading To An Instrument

The model shown in Figure 5-15 is saved in this file:

`/usr/lib/veetest/examples/instr_io/manual17.ex`

Troubleshooting

Instrument Control Troubleshooting

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

Instrument Control Troubleshooting (continued)

Problem	Remedy/Cause
Cannot determine the instrument address.	For GPIO and serial interfaces, the instrument address is the same as the interface select code. HP-IB instrument addresses are set by hardware switches or front panel commands. Older instruments use small switches located on the rear panel near the HP-IB connector. Newer instruments set and query the address via front panel buttons. Consult your instrument's programming manual for details. Refer to the section "Device Configuration Dialog Box" earlier in this chapter for examples of specifying addresses. VXI devices have logical addresses set by switches on the outside of the cards (usually the cards will have to be removed from the card cage to access the switches).
Cannot determine the interface select code.	In some cases, the select code is printed on the rear panel of the interface itself. Contact your system administrator for details about your hardware configuration. Refer to the section "Device Configuration Dialog Box" earlier in this chapter for examples of specifying addresses. These are the factory default select codes for commonly used interfaces: ■ HP-IB : 7 ■ Serial: 9, 17 ■ GPIO : 12 ■ VXI : 16
Errors when opening a model containing instrument objects, such as `Configuration Error: Device` *Name* `not configured`.	You do not have an `I/O` ⟹ `Configure I/O` entry with `Name` and `ID Filename` fields that match the entries saved in the model. Refer to the section "Advanced Topic - I/O Configuration File" earlier in this chapter for details.

Related Reading

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

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

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

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

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

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

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

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

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

6

Building UserObjects

To create modular, well-designed models, you can build a `UserObject` to perform each logical function.

This chapter covers the following topics:

- Benefits of using `UserObject`s
- Definition of `UserObject`s and how they work
- Creating `UserObject`s
- Exiting `UserObject`s
- Adding `UserObject`s to a library

Benefits of UserObjects

A `UserObject` provides a means for you to encapsulate a group of objects that perform a particular function. This encapsulation allows you to:

- Use modular design techniques necessary for an organized approach towards designing and building complex models.

  `UserObjects` allow you to use top-down design techniques to create a more flexible and maintainable model.

- 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 models so that common functions are built only once.

- Create pop-up panels to dynamically display information.

To create an interface element such as a dialog box, you can specify that the panel of a `UserObject` appear in the work area only when the `UserObject` operates. This topic is discussed in Chapter 7.

Understanding UserObjects

A `UserObject` is an enclosed environment that provides a virtual HP VEE work area. Any model that works properly in the main HP VEE environment can be completely encapsulated into a `UserObject` and it will work the same way.

When the model runs, a `UserObject` operates like any other object; data is sent to the `UserObject` through the input terminals, the internal function operates, and data is sent out over the output terminals.

It is possible to nest `UserObjects` within `UserObjects`. Therefore a hierarchy of environments can be formed. Each `UserObject` has its own context separate from the `UserObjects` which are external or internal to it.

Understanding Contexts

A **context** is a work area that includes all objects except those inside nested `UserObjects`. The main work area is a context (the root context) and every `UserObject` is a context. Any action that is context-sensitive affects only the objects on the context's work area; it excludes the objects inside nested `UserObjects`.

In the figure below, the main work area, `UserObject1`, `UserObject2`, and `UserObject3` are all separate contexts. The main work area's context does not include `UserObject2`. `UserObject1`'s context includes all objects inside of it, including `UserObject2`, but not the objects inside of `UserObject2`.

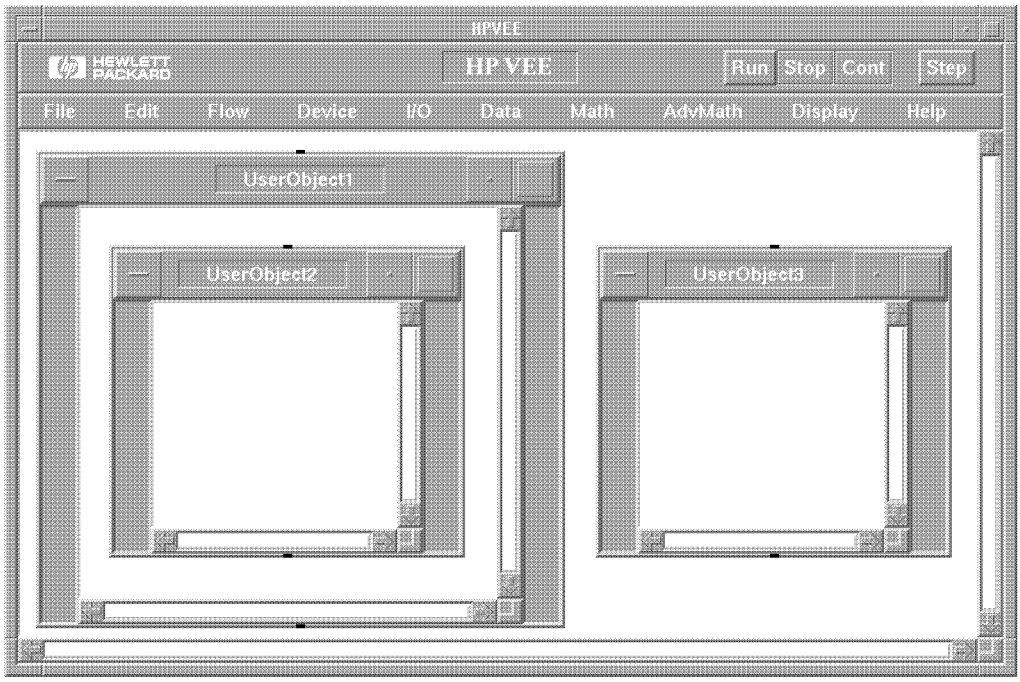


Figure 6-1. Four Different Contexts

Because each context is a work area, you can access a pop-up `Edit` menu inside each `UserObject` work area. An `Edit` menu is also available from the `UserObject` object menu. The following `Edit` actions are context-sensitive and operate only within each work area:

- `Clean Up Lines`
- `Add To Panel` (each context has its own panel)
- `Move Objects`
- `Create UserObject`

For example, if you select `Clean Up Lines` from the main `Edit` menu, only the lines in the main work area are rerouted. If you select `Clean Up Lines` from the `UserObject1 Edit` menu, only the lines in the `UserObject1` work area are rerouted; `UserObject2` is not affected.

The following actions are also context-sensitive:

- `Clear At Activate` is available on certain objects from their object menu. If selected, this action occurs every time the `UserObject` containing the object operates.
- `Initialize At Activate` is available on various `Data` objects from their object menu. If selected, this action occurs every time the `UserObject` containing the object operates.
- `Trig Mode` is available from the `File` ⟹ `Preferences` ⟹ feature of the main work area or from the object menu of the `UserObject`.

Objects within the `UserObject` (internal objects) can only communicate with objects outside the `UserObject` boundary (external objects) through the `UserObject` data input and output terminals. (The exception is that Global variables can be used within any context of the model, including a UserObject. Refer to "Using Global Variables in UserObjects," later in this chapter, for details.)

Understanding Propagation in UserObjects

Each `UserObject`, or context, is a group of objects that provide a particular function. The propagation rules of a `UserObject` are as follows:

- All data inputs and the sequence input (if connected) of the `UserObject` must be activated before any internal objects operate (even if you have objects without input dependencies or on independent threads).
- When the internal objects operate, they follow the rules of propagation as listed in "Understanding Propagation" in Chapter 3. Internal objects timeshare with external objects on different subthreads. The `UserObject` does *not* block the operation of external objects on different subthreads.
- If the optional XEQ input pin is activated, the `UserObject` immediately begins operation of its internal objects, using whatever old data that may still be on the `UserObjects`'s unactivated input pins. XEQ is rarely necessary for most `UserObjects`.
- Within a `UserObject`, input terminals operate with the same precedence as an unconstrained device.

- All internal objects must finish operating before any data outputs are activated (unless the `UserObject` is exited prematurely by an error or an `Exit UserObject`). Only those output pins activated from inside the `UserObject` pass data out to other objects.

Refer to "Understanding Propagation" in Chapter 3 for more information about propagation rules.

Note

If you have a `Start` object in a `UserObject` (to handle feedback), pressing `Start` runs only the internal objects; it will not read the data from the `UserObjects`'s input terminals or activate the `UserObject`'s output terminals (data or sequence), therefore no propagation outside the `UserObject` takes place.

Short Cut

You can connect an object inside a UserObject to another object that is outside the UserObject, or even inside a different UserObject. When you do this, the appropriate input and output terminals on the UserObject(s) will automatically appear.

Creating UserObjects

You create a `UserObject` in either of the following ways:

- Select `UserObject` from the `Device` menu. Place the objects you want within the `UserObject`.

  This method is useful when building a new `UserObject` (top-down method).

- Build the `UserObject`'s desired function as a model. When the model runs as you want it to, select all the objects (with `Edit` ⟹ `Select Objects`) and then select `Create UserObject` from the `Edit` menu (bottom-up method).

  This method is useful when you have an existing structure to incorporate into a `UserObject`. By using the principle of bottom-up design, you can design the function needed and then incorporate it into the `UserObject`.

Both methods give you the same result. To add objects to an existing `UserObject`, move the objects in to the existing `UserObject` work area.

Note

Problems can arise if the objects selected are already part of a model. There are subtle differences in the way the objects interact when they are communicating across `UserObject` boundaries. For example, a `For Range` object connected to the output terminal of a `UserObject` will activate the `UserObject` output pin only once, since the `UserObject` buffers its terminals.

Adding Inputs and Outputs

Add data input and output terminals to the `UserObject` to get data from other objects or output data to other objects. You can add data input and output terminals two ways:

- Use the object menu to add terminals (as you would for any object). Or you can use the short cut CTRL-A to add a terminal. Once the terminals are created, you must connect the internal and external pins of the terminal.
- Draw a line to connect the external object and the internal object; the terminal is automatically created.

Note

A `UserObject` transmits only *data* across its boundary, therefore only data terminals are created to connect external objects to internal objects. If you try to connect control, sequence, or trigger pins across the `UserObject` boundary, HP VEE creates a *data* terminal. This may not operate in the desired manner for your model.

You can add other terminals to a `UserObject` to affect how it operates. The terminals you can add are as follows:

- `XEQ` - an input pin that forces the objects in the `UserObject` to start executing before all the data or sequence input terminals of the `UserObject` are activated. Input terminals that have not been activated contain old data or `nil`.

- `Print` - a control input that graphically prints the open view of the `UserObject`.
- `Error` - an output pin that traps any errors that are generated from any object within the `UserObject` that does not have an error pin or by a `Raise Error` object. If an error is trapped by an error pin, the model continues to run and you can process the error. For more information about trapping errors, refer to "Trapping Errors" in Chapter 4

Exiting UserObjects Early

To exit a `UserObject` *before* all internal objects have operated, use the `Flow` ⟹ `Exit UserObject` or `Flow` ⟹ `Raise Error` object.

Exit UserObject

When an `Exit UserObject` operates, the `UserObject` stops running. Any data on those data output pins which have been activated from inside the `UserObject` is sent and propagation continues outside the `UserObject`.

Use `Exit UserObject` to output data without waiting for all internal objects to operate.

Figure 6-2 shows a dialog box that queries for a name from inside a `UserObject`. `Exit UserObject` terminates execution of all threads within the `UserObject` when either button (`OK` or `Cancel`) is pressed.

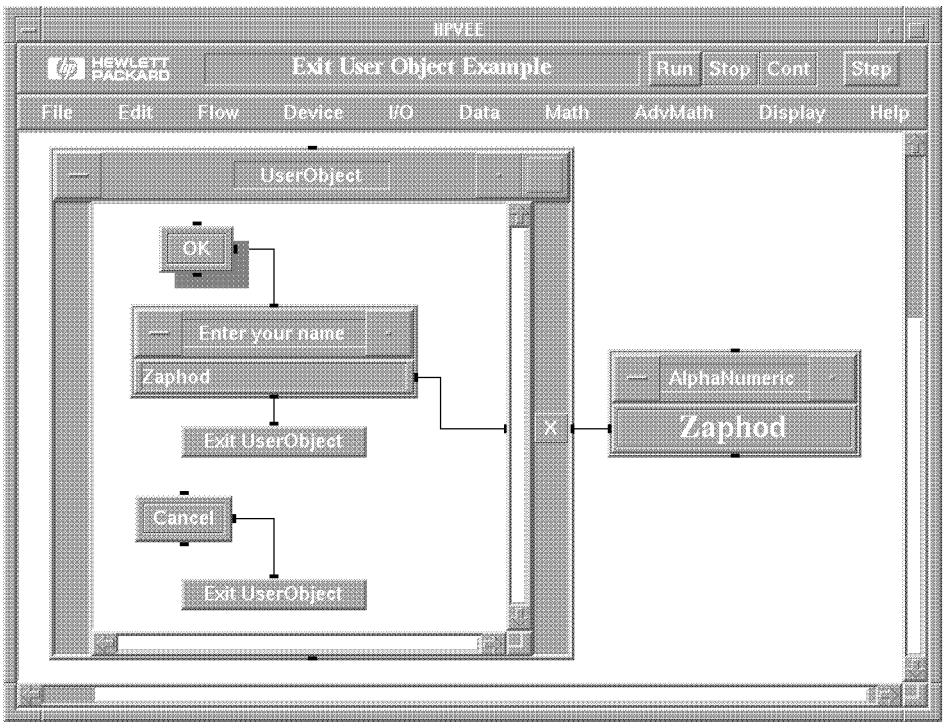


Figure 6-2. Example of `Exit UserObject`

The model shown in Figure 6-2 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual18.ex
 - or -
/usr/lib/veetest/examples/concepts/manual18.ex
```

Raise Error

The `Raise Error` object allows you to exit a `UserObject` with an error condition. When `Raise Error` operates, the `UserObject` stops running. Any data on the data output pins is lost. `Raise Error` sends a message and error number upward through contexts until they encounter an error pin or the the main work area's context (the root context).

If an error pin is encountered, the error number is output on that pin and may be handled like any trapped error. The sequence output pin of the trapping context activates after the error pin thread completes.

If the root context is encountered, the entire model stops running and an error dialog box is displayed. It contains the message and the error number.

To create a robust and usable model, it is important to trap errors with an error pin and to define your own error numbers and messages with `Raise Error`. User-generated error numbers should not be in the range of 300 to 1000 since HP VEE error numbers are in that range.

Figure 6-3 shows how to use `Raise Error` to generate your own error messages. When an out of range value is entered and `OK` is pressed, the `Raise Error` generates an error.

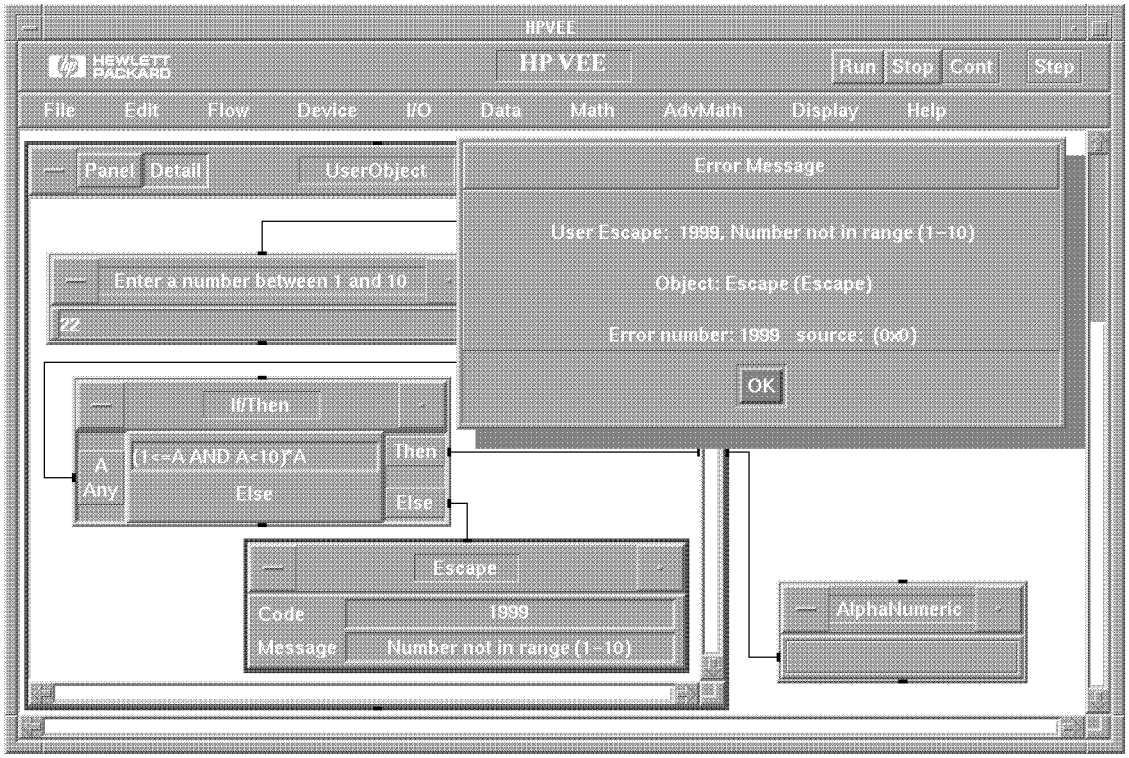


Figure 6-3. Using a `Raise Error`

The model shown in Figure 6-3 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual19.ex
 - or -
/usr/lib/veetest/examples/concepts/manual19.ex
```

Creating a Library of Functions

You can build a library of self-contained functions with `UserObjects`. When you create the functions as `UserObjects`, they are easy to use and easy to incorporate into other models. A library saves you time and energy by allowing you to leverage existing functions instead of re-creating them.

Building Panel Views

To create a custom user interface to the `UserObject`, you can build a panel view that contains the objects with which the user will interact. Each `UserObject` has a panel view available. For information about building a panel view on a `UserObject`, refer to Chapter 7.

Securing UserObjects

After the functionality of a `UserObject` is final, and you don't want anyone to change the internal operations, secure the `UserObject`. (Users will use only the input and output terminals to interact with the `UserObject`.)

After selecting `Secure` from the `UserObject` object menu, you'll be prompted to save the unsecured version of the `UserObject`.

Caution

Make sure you keep an unsecured version so that in the future you can edit the internal objects. Once secured, `UserObjects` *cannot* be unsecured.

If the `UserObject` does not have a panel view, securing it minimizes the `UserObject` to an icon that cannot be opened. If it has a panel view, the secured view of the `UserObject` is the panel view.

To put the secured `UserObject` in your library of functions, select the `UserObject` and select `Save Objects` from the `File` menu. Save the secured version under a different name than the unsecured version.

Merging and Saving UserObjects

You do not need to secure a `UserObject` to store it in your library of functions. To save a `UserObject`, select it then select `Save Objects` from the `File` menu.

Note

HP VEE provides a directory for you to store useful objects: `/usr/lib/veeengine/lib/contrib/` or `/usr/lib/veetest/lib/contrib/`.

The initial path for `Save Objects` (until you select `Save Preferences`) is `/usr/lib/veeengine/lib/` or `/usr/lib/veetest/lib`.

To retrieve functions from your library, select `Merge` from the `File` menu. `Merge` allows you to keep the existing model on the work area so you may add to it.

Occasionally, you may want to merge a library object or model inside of a `UserObject`. Be sure that the white outline box of the merging object fits *completely* inside the `UserObject` work area. Otherwise the merged object will be placed outside of the `UserObject` and you will have to move the object into the `UserObject`.

Using Global Variables in UserObjects

We've introduced the concept of global variables in chapter 3. Now let's look at how to use global variables in UserObjects. A global variable is a named variable that is set globally, and which can be used by name in any context of an HP VEE model. For example, a global variable can be set with `Set Global` in the root context of the model, and can be accessed by name within the context of any UserObject or User Function (even though nested recursively) *within the same HP VEE process*. Thus, global variables can be used throughout your HP VEE model, but not in a remote process such as a Remote Function or Compiled Function.

Let's look at some examples of using global variables in UserObjects. In the model of the following figure, a Real array is output to the `Set Global` object,

which sets the global variable `globalA`. The global variable is accessed by name three times within the UserObject, once by the `Get Global` object, once by the expression `sort(globalA)` in the `Formula` object, and once by the expression in the transaction of the `To File` object.

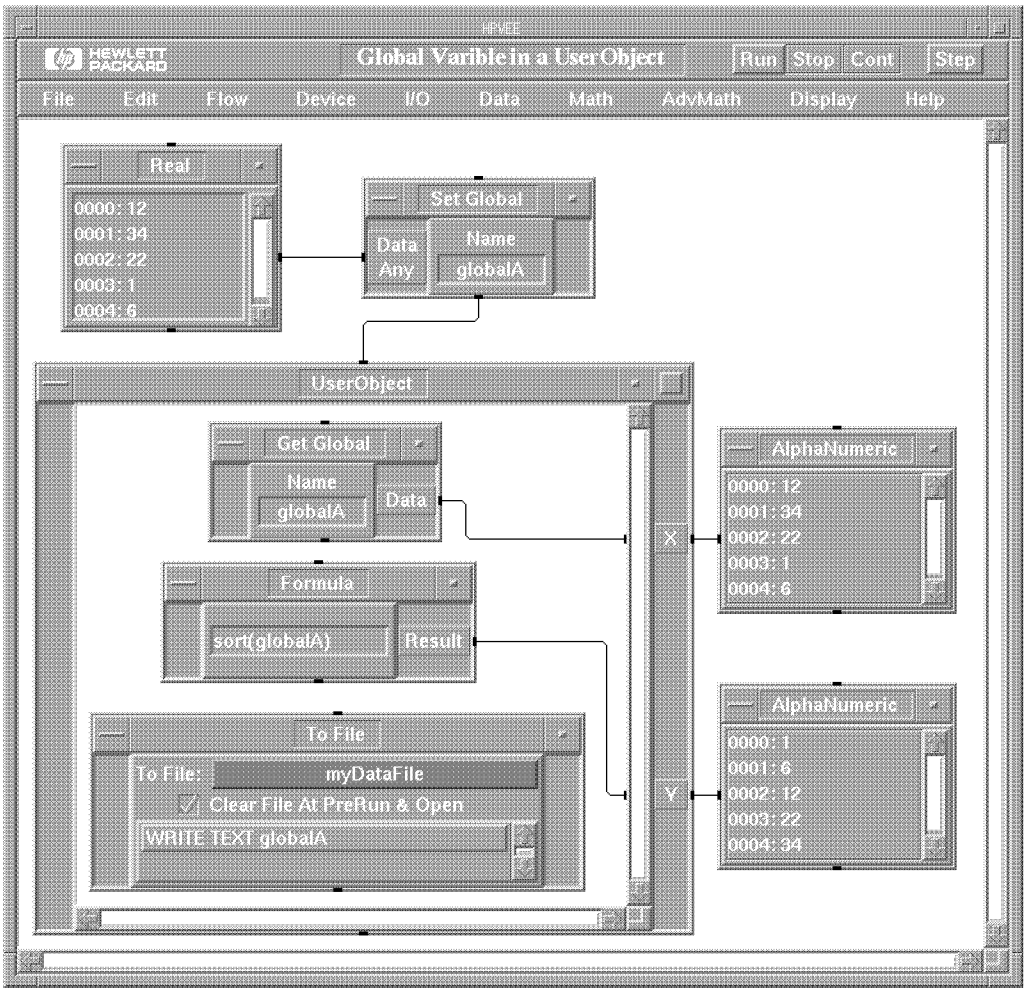


Figure 6-4. Retrieving a Global Variable in a UserObject

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

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

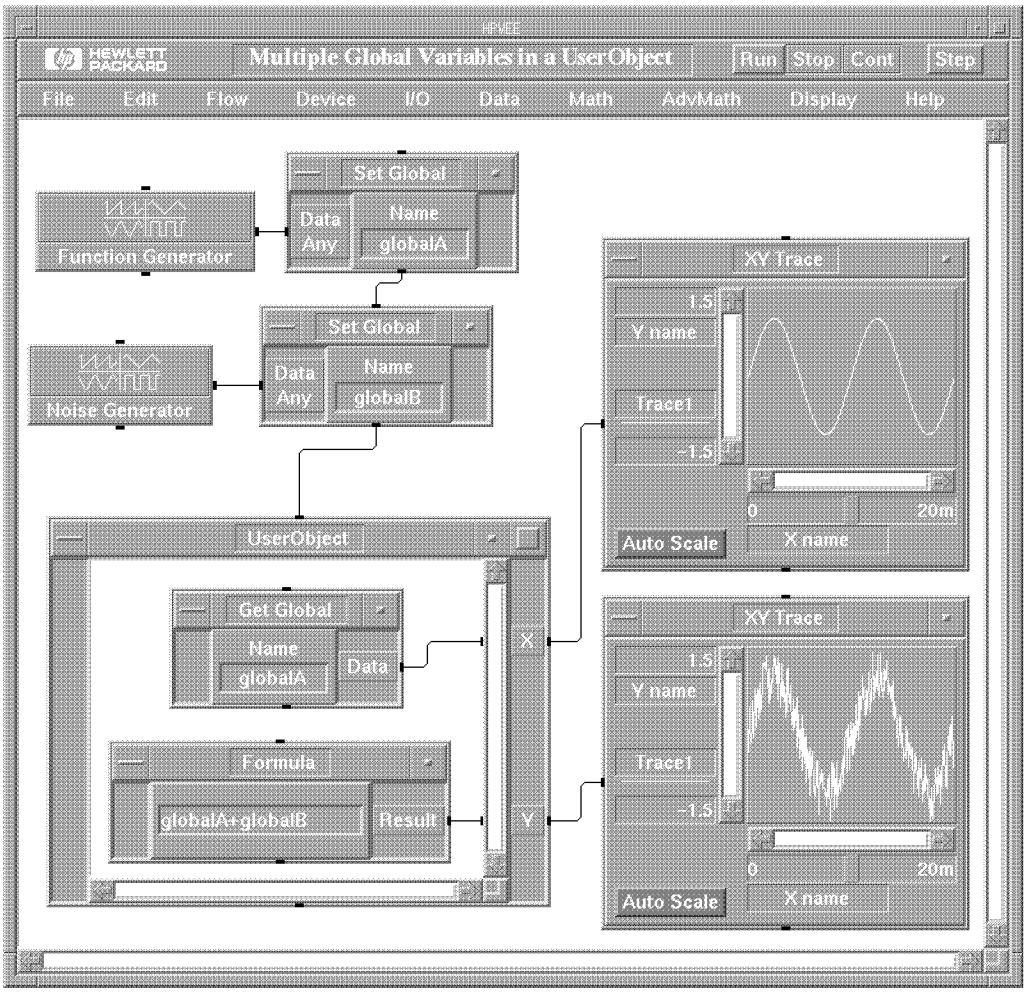


Figure 6-5. Retrieving Multiple Global Variables in a UserObject

The sine wave becomes `globalA` and the noise waveform becomes `globalB`, respectively. These waveforms are retrieved within the UserObject by the `Get Global` object (which retrieves and outputs the sine wave) and the `Formula` object (which retrieves both `globalA` and `globalB`, adds them, and outputs the combined waveform). Again the sequence input pin of the UserObject is used to hold off retrieval of the global variables until they have been set. However,

in this case, you must ensure that *both* `Set Global` objects execute before the UserObject executes. To do this, you can "chain" the sequence pins of the two `Set Global` objects. The order in which the two `Set Global` objects execute does not matter, but the both must execute before the UserObject executes.

For further information about global variables, refer to "Using Global Variables" in chapter 3, "Using Global Records" in chapter 8, and to the `Set Global` and `Get Global` reference sections in the *HP VEE Reference* manual.

7

Building Panel Views

If others will be running your model, you may want to create a panel view as a user interface to your model.

This chapter covers the following topics:

- Benefits of panel views
- Definition of panel views
- Building panel views
- Adding dynamic (pop-up) elements to panel views
- Saving and securing panel views

Benefits of Panel Views

A panel view provides the following benefits:

- Shows only the objects necessary for operation.

  A panel view contains only what is needed to run a model. (This could be as little as the data input fields and a display).

- Protects the model from intervention by a user.

  From a panel view, a user does not see the details of a model. From a secured panel view, a user cannot change the model in any way.

- Allows you to create a complete application from your model.

  A panel view provides a user interface to the complex model, making an application that is easier to use and understand.

- Improves the performance of models by decreasing run time.

Updating object views takes time. Depending on the complexity of a model, running from the simpler panel view may increase performance.

Understanding Panel Views

A **panel view** is an alternate view of the model that provides a user interface to your model (you created your model in the detail view). You choose which objects from the detail view to include on the panel view, which view of the object (icon or open view) to use on the panel view, and where on the panel to place them; the lines connecting objects on the detail view are not shown on the panel view. Once you create a panel view, it is part of your model and is affected by the `File` features (such as `New` or `Save`).

There are two different types of panel views: the main panel and `UserObject` panel views.

- The main panel view is the alternate view of the main work area. The main panel view can contain objects and `UserObject` panels.
- A `UserObject` panel view is the alternate view of a `UserObject`. Each `UserObject` may have an associated panel view, which may be statically or dynamically displayed. A `UserObject` panel view can contain objects and nested `UserObject` panel views.

Both types of panel views are created and used the same way, except `UserObject` panel views may be dynamically displayed (as pop-ups) when they operate.

Figure 7-1 and Figure 7-2 show the detail view and the panel view of a model that calculates position, velocity, and acceleration for an object in motion in the presence of drag. Notice the differences in size and configuration of the `X vs Y Plot` between the detail view and panel view.

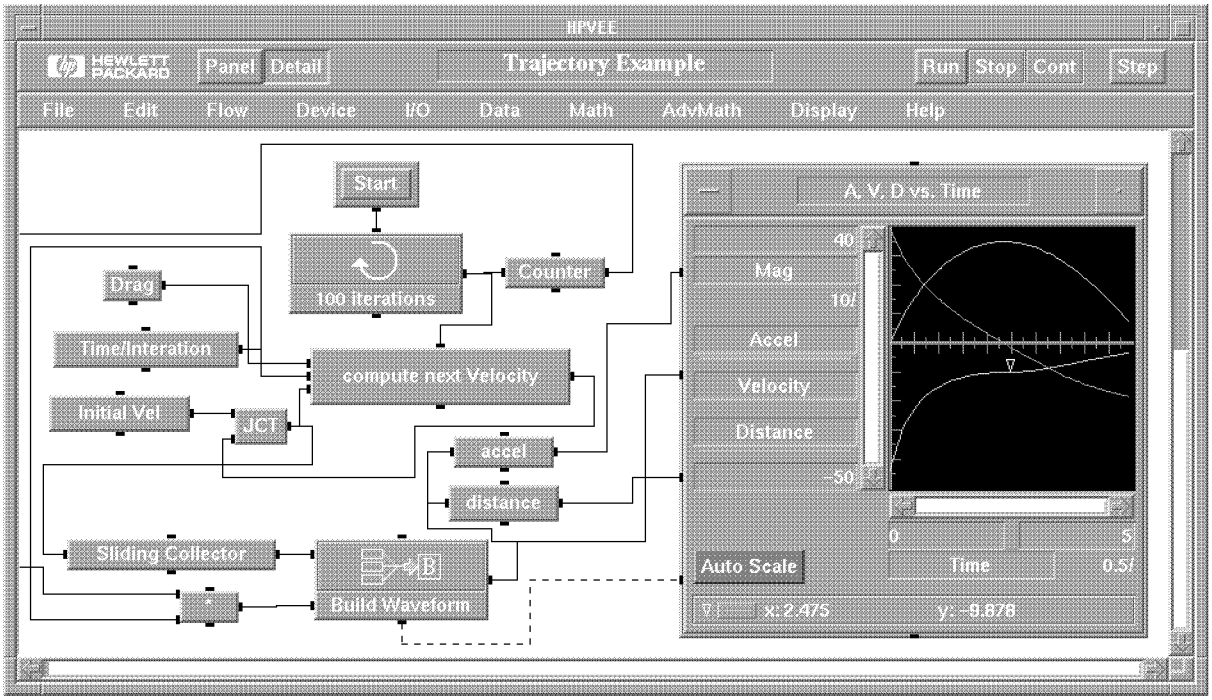


Figure 7-1. Detail View of Trajectory Example

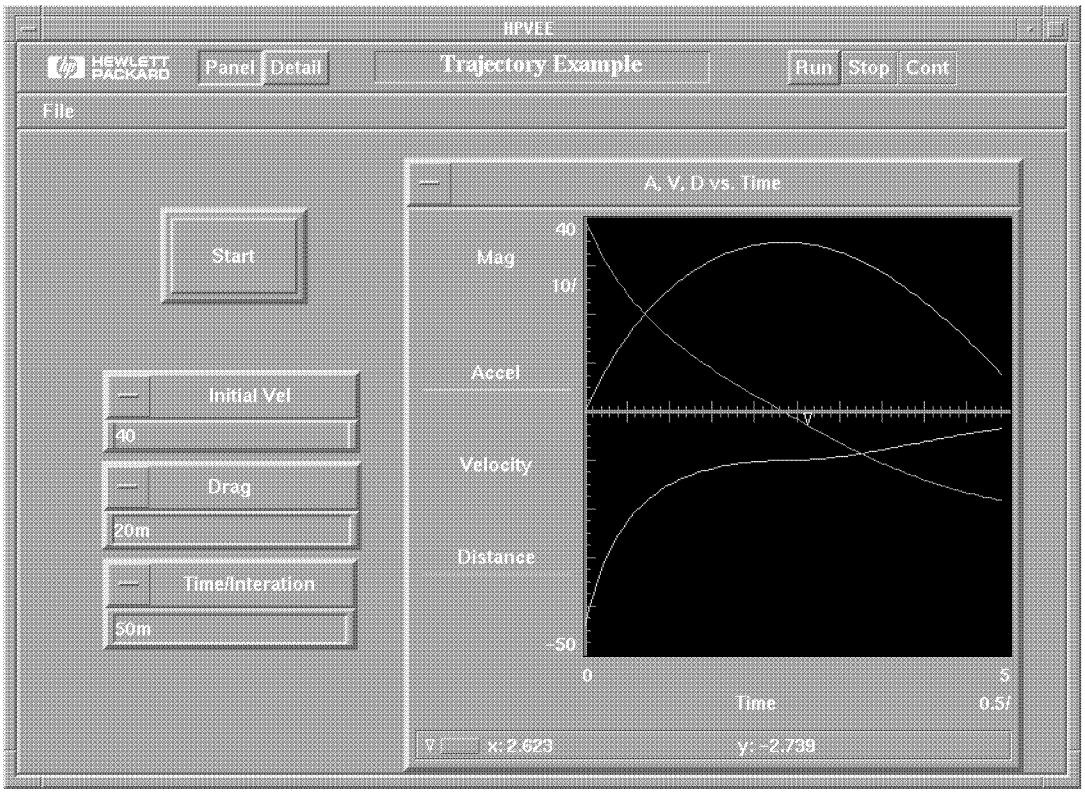


Figure 7-2. Panel View of Trajectory Example

7

The model shown in Figure 7-1 and Figure 7-2 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual20.ex
 - or -
/usr/lib/veetest/examples/concepts/manual20.ex
```

Before You Start

Before you construct a panel view, the model should run correctly. Since you often need to edit both the detail view and the panel view, it is difficult to build a detail view and the panel view at the same time due to the iterative nature of constructing both. Construct the model first in the detail view, then create the panel view.

Creating Panel Views

After your model runs properly and you've prepared the objects on the detail view, you can create the panel view. To create the panel view, select the objects you want on the panel view and select `Add to Panel` from the `Edit` menu. Note that once the object is on the panel view, you cannot change its view (icon or open view).

`Add to Panel` is context-sensitive. `Add to Panel` from the main work area's `Edit` menu creates the main panel view that contains only the selected objects from the main work area. `Add to Panel` from a `UserObject`'s `Edit` menu creates a `UserObject` panel view that contains only the selected objects from the `UserObject`. If no objects are selected, `Add to Panel` is not available.

After you've added objects to the panel view, press the `Panel` and `Detail` buttons in the upper left corner of the window to move between the different views. You can continue to add objects from the detail view.

Note

Once an object is added to the panel view, it must have a corresponding object on the detail view. If you `Cut` an object from the detail view, the corresponding object on the panel view is gone. If you `Paste` the object back to the detail view, you'll have to add the object to the panel view again.

Objects visible on the detail view appear on the panel view in the same position as they were on the detail view. If objects are not visible on the detail view (located off the visible part of the work area), the objects will appear justified against an edge of the panel view. The panel view work area does not

scroll, although you may resize it. The sizes of the panel view and the detail view may be different.

Note

A large-sized panel view created on a high-resolution monitor will not be fully displayed if it is displayed on a low-resolution monitor.

The title and menu bar are different on the panel view and the detail view. Figure 7-3 shows that on the panel view, only the `File` menu choice is available and the `Step` button is gone because debugging can be done only in the detail view.

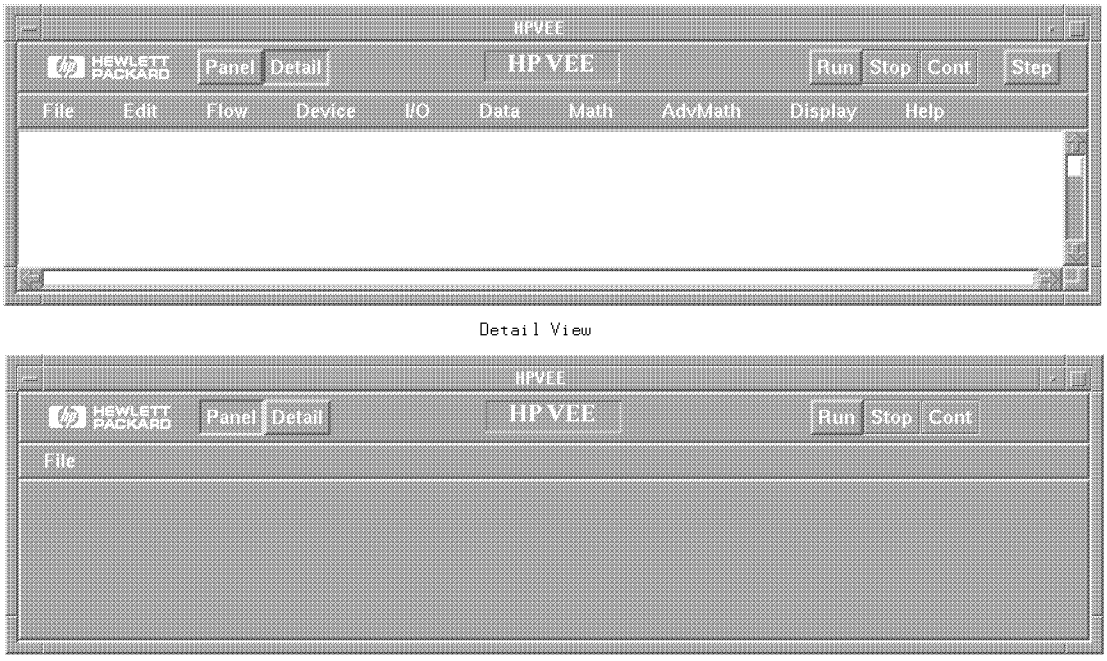


Figure 7-3. Differences Between Detail View and Panel View

After you place the objects on the panel view, you can modify the objects to construct the layout and alter the configuration of the objects using features from their object menus. The next section explains how to layout panel views.

Laying Out Panel Views

The panel view creates an interface for the users of your model. After you've added objects to the panel view, modify the look of the panel view to meet the needs of the users and their interaction with the model. Some layout techniques are:

- Logically group objects. For example, you may want to group all data input objects or displays together.
- Chronologically order objects. For example, if the user needs to fill in an entry field and then press a button, make sure the objects are close together and that the entry field precedes the button.
- Resize objects to fit together. For example, if you have a row of displays, make them all the same size.
- Resize objects to denote importance. For example, you may want to increase the size of buttons such as `Stop`.

The best way to make sure that the panel view layout meets your user's needs is to check your layout by having someone (perhaps a potential user) run your model from the panel view and give you feedback.

Note

This section explains layout of static objects on the panel view. If you put pop-up elements on the panel view, be sure to consider their layout. Pop-up elements are discussed later in this chapter.

You can change an object's size, location, and appearance by using the following features from its object menu:

- `Move` - Move the object.
- `Size` - Resize the object.
- `Show Title` - Hide or display the title bar.
- `Layout` $\Longrightarrow$ (icons only) - Change the bitmap displayed.
- `Delete` - Delete the object from the panel view (the corresponding object remains on the detail view).

The object menus of some open views contain other features that allow you to change the appearance of the objects on the panel view. For example, on graphical displays the `Grid Type` and `Panel Layout` choices change the appearance of the object on the view (panel or detail view) without affecting the appearance of the object on the other view.

Note

The panel view of your model is to a large extent *independent* of the detail view. Not only can you show only selected objects in the panel view, you can show the objects in a different size or location than in the detail view. You can show the title in one view, but not the other. Or you can select a different scale or grid type for a display object in each view. For the `Note Pad` object, you can enable editing in the detail view, but disable editing in the panel view.

Figure 7-4 shows how the the appearance of an `X vs Y Plot` on the panel view was changed without affecting the corresponding object on the detail view.

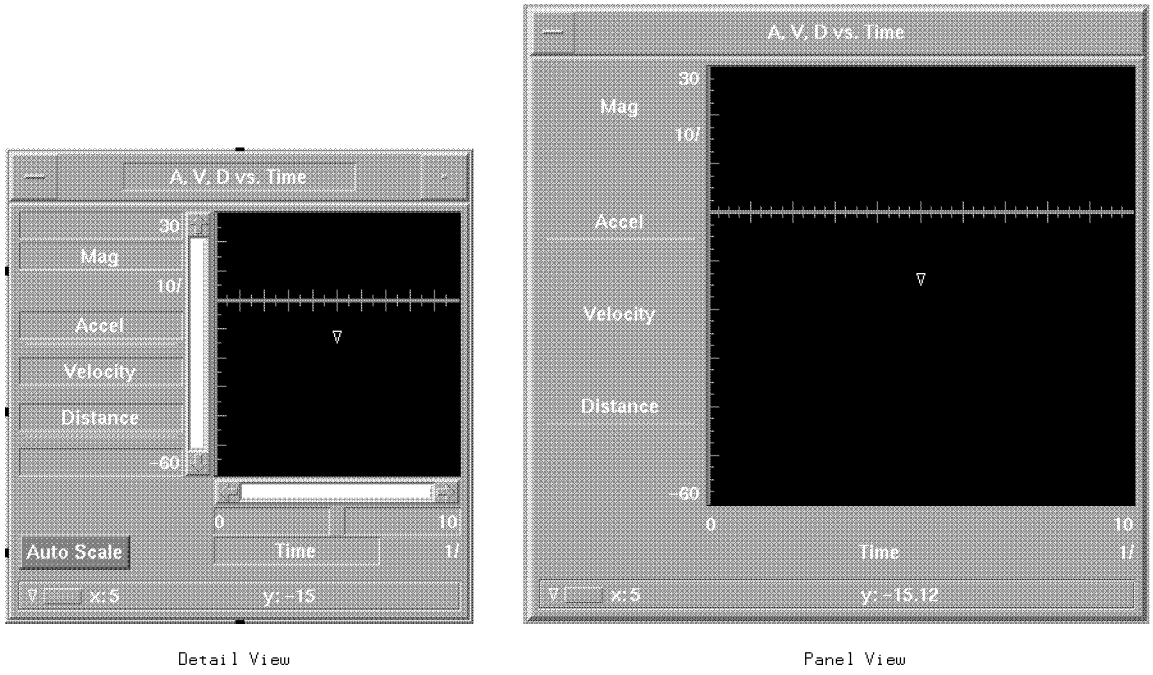


Figure 7-4. Panel View vs. Detail View of `X vs Y Plot`

Setting Values and States

Although the appearance of an object on the panel view is not connected to the appearance of the object on the detail view, the entry values and states *are* shared between the panel view and detail view.

For example, if you change the the size of a `Slider` on the panel view, the size of the associated `Slider` on the detail view does not change. But if you change the `Slider's` maximum and minimum values from the open view on the panel view, the values change on the associated `Slider` on the detail view.

The object menus of many open view objects contain features that allow you to set values and states that affect the way the objects operate. Some

features that are shared between the open view object on a panel view and its associated detail view are:

- `Initialize At PreRun`
- `Initialize At Activate`
- `Clear At PreRun`
- `Clear At Activate`
- `Auto Execute`
- `Slider Detents`
- `Slider` limit values
- Graphical display trace names

Saving Panel Views

When you save a model that has both a panel view and detail view, both views are saved in the file. When you `Open` that file, you'll see whichever view was visible when it was saved (but both views are still present).

When you save a secured panel view, only the panel view is saved.

Securing Panel Views

Once you are satisfied with the panel view's functionality and appearance, the model or `UserObject` can be secured. As mentioned previously, securing a model or `UserObject` prevents the user from altering the model and accessing the detail view. It may also enhance the model's performance. Before you secure a panel view, make sure you don't want to modify it or the detail view. After you secure the panel view, you will not be able to edit it; you won't be able to access any features that modify objects' appearance, settings, or the way they operate.

Main Panel View

Secure the model by selecting `Secure` from the `File` menu (on either the panel view or detail view). Before the model is secured, you'll be prompted to save the unsecured model. Note that this method secures the *entire* model, including any `UserObjects`.

After the model is secured, save it. Use a name that is different from the unsecured model name.

Note

The secured panel view and the unsecured model are unconnected. If you change the unsecured model, you'll have to `Secure` it again. Once a model is secured, it *cannot* be unsecured.

UserObject Panel View

To secure a `UserObject` without securing the rest of the model, select `Secure` from the `UserObject`'s object menu. Before the `UserObject` is secured, you'll be prompted to save the unsecured `UserObject`.

After the `UserObject` is secure, save it using the `Save Object` feature under the `File` menu. Use a name that is different from the unsecured object.

Note

The secured object and the unsecured object are unconnected. If you change the unsecured object, you'll have to `Secure` it again.

Adding Pop-up Elements

You can add a dynamically-displayed element to the main panel view by selecting `Show Panel on Exec` from the `UserObject` object menu *instead* of adding the `UserObject` panel view to the main panel view.

A pop-up is a `UserObject` panel view that is displayed when the `UserObject` starts to operate and stays displayed until the `UserObject` has finished operating.

It is very useful to add dynamically-displayed (pop-up) elements to a panel view. Pop-ups allow you to:

- Save space.

  Because a pop-up doesn't permanently occupy space on the panel view, you can overlap the pop-ups to save space.

- Page information.

When you've got multiple actions to perform, you can "page" through them using pop-ups.

- Help users focus on information.

  If all information is displayed at all times, users may not know which fields are getting updated. If new or critical information is put in a pop-up, users may be more aware of changes.

  Another use for pop-ups is to let the user specify the information wanted; the model only displays what the user asks for.

- Create dialog boxes.

  Pop-ups are very useful when asking for user input. When you use a dialog box, you can prevent users from changing information later and more easily perform error checking.

Before You Start

As with any aspect of panel views, your model should work *before* you create any pop-ups. If you want objects to pop-up, but they are not in `UserObjects`, you *must* put them in a `UserObject` and make sure your model still works. Many times the action of creating a `UserObject` will change the way your model operates. For more information about `UserObjects`, refer to Chapter 6.

`UserObject` panel views follow the same layout and functionality guidelines discussed earlier in this chapter.

Creating Pop-up Panel Views

Any `UserObject` panel view can be popped-up. Select objects in the `UserObject` and then, from the `UserObject`'s `Edit` menu, select `Add to Panel`.

The `UserObject`'s title remains on the pop-up so you can use the title bar to label the pop-up panel view.

Pop-up Layout

Size the `UserObject` panel view to the size you want the pop-up to be. Arrange the objects on the panel view using the techniques described earlier in this chapter.

After you have completed the `UserObject` panel view layout, set the `Show Panel on Exec` option on the `UserObject`'s object menu.

When the `UserObject` operates, the default pop-up position is in the center of the work area. To set the pop-up's location, switch to the panel view and run the model. When the `UserObject` panel view pops-up, move it to the desired location by dragging the title bar.

It is useful to use a `Confirm (OK)` button on the `UserObject` and the `UserObject`'s panel view. Because a `UserObject` operates until all objects within it finish operating, the `Confirm (OK)` button allows the pop-up to stay displayed until `OK` is clicked.

Note

The `UserObject` pop-up is displayed on both the detail view and the panel view. If you move the position of the pop-up on either view, it will also move on the other. The position of the pop-up is relative to the HP VEE work area; moving the `UserObject` does not change the location of the pop-up.

Pop-up Examples

The following examples show some uses for pop-ups.

Informational Messages

Pop-ups are useful to display informational messages. In Figure 7-5, after the user has read the message, he or she clicks `OK` and the pop-up disappears.

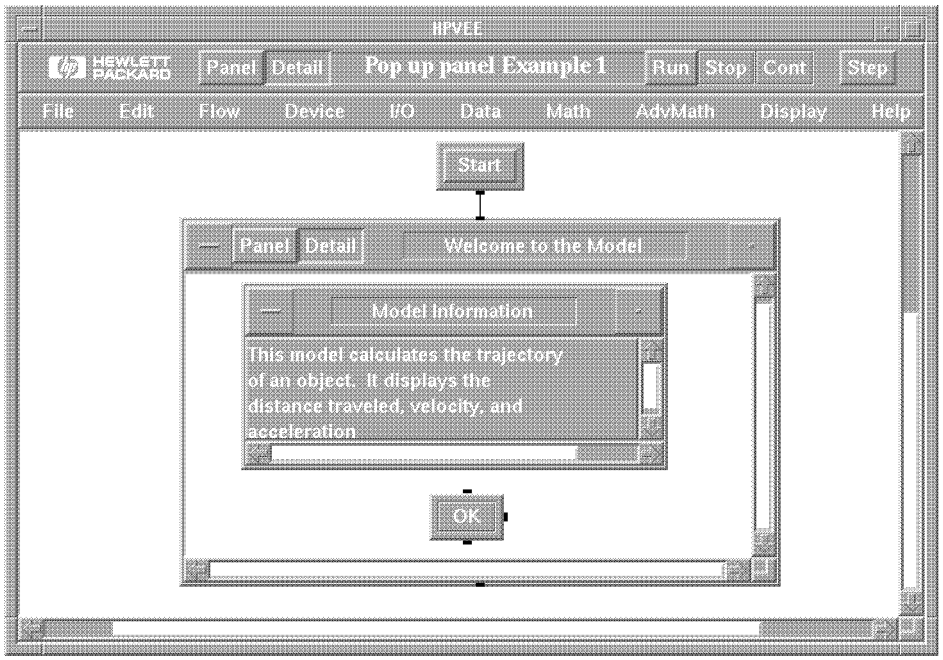


Figure 7-5. Informational Message (Detail View)

Figure 7-6 shows the panel after the user presses `Start`.

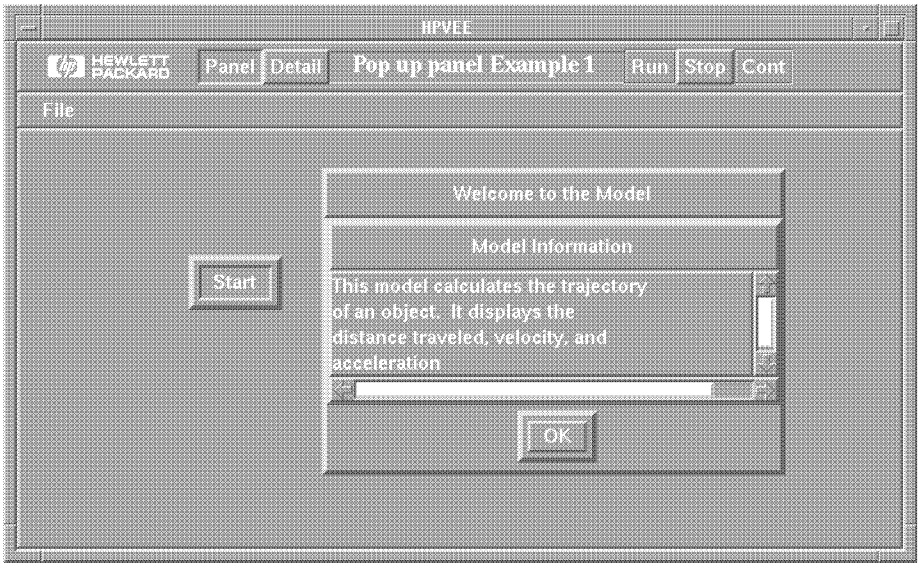


Figure 7-6. Informational Message (Panel View)

The model shown in Figure 7-5 and Figure 7-6 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual21.ex
 - or -
/usr/lib/veetest/examples/concepts/manual21.ex
```

Overlaying Displays

When you overlay displays, you allow the user to select the type of display wanted. Only the selected display is shown.

In Figure 7-7, the display shows the data you select.

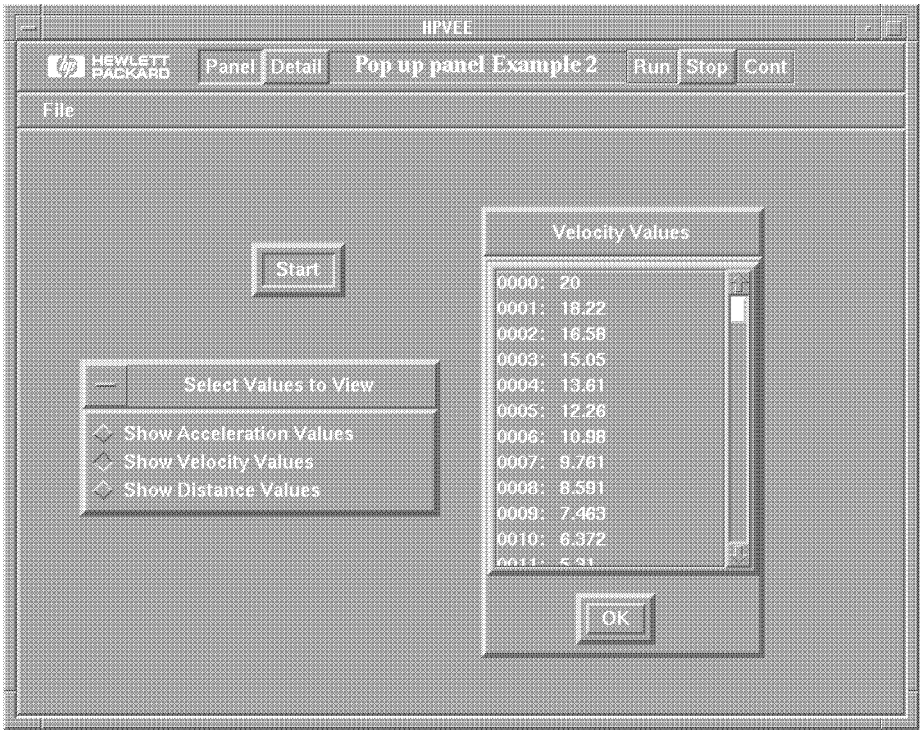


Figure 7-7. Overlaying Displays on a Panel View

7

The model shown in Figure 7-7 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual22.ex
 - or -
/usr/lib/veetest/examples/concepts/manual22.ex
```

Dialog Boxes

Dialog boxes allow you to get user input at a particular time without letting the user change the information later. Dialog boxes also allow you to perform error checking on the information the user entered.

The model in Figure 7-8 and Figure 7-9 asks for the user's name. The user can fill in the name or cancel the operation. If the user presses `OK`, the name is displayed. If the user presses `Cancel`, the message `No Name Entered` is displayed.

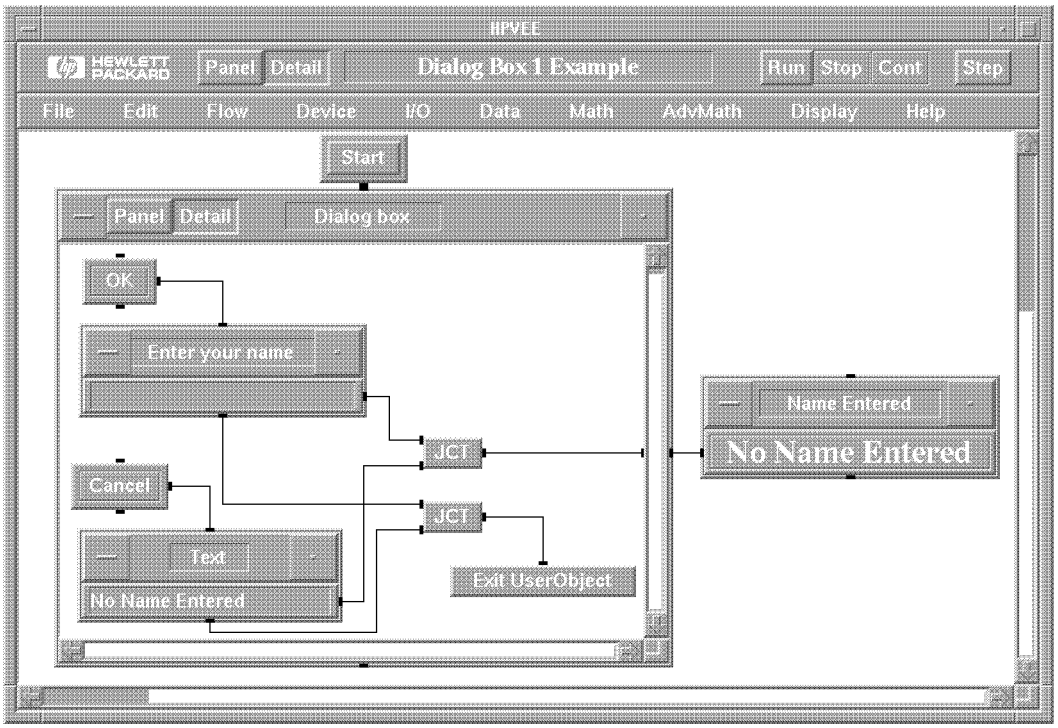


Figure 7-8. Dialog Box (Detail View)

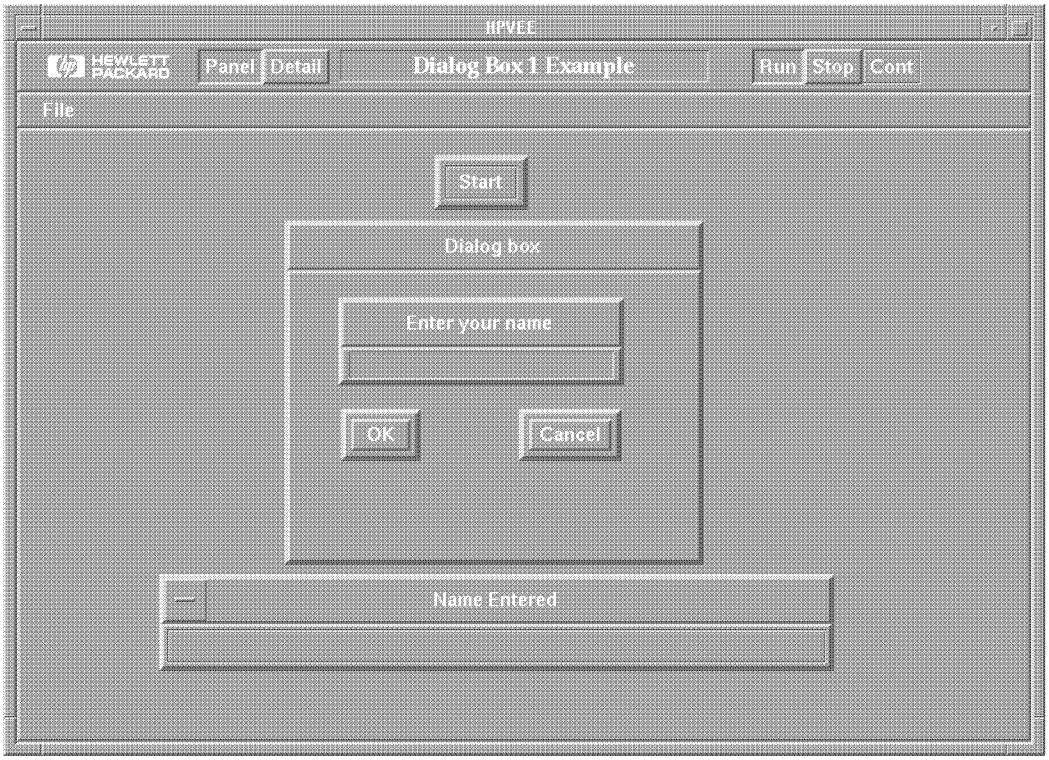


Figure 7-9. Dialog Box (Panel View)

The model shown in Figure 7-8 and Figure 7-9 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual23.ex
 - or -
/usr/lib/veetest/examples/concepts/manual23.ex
```

Note

Some other commonly-used dialog boxes are in `/usr/lib/veeengine/concepts/` or `/usr/lib/veetest/concepts/`

8

Optimizing Models

Although the time to run a model varies (depending on the current load on your computer system), the following techniques may help you improve execution speed:

- Leave input terminals set to type/shape `Any` where possible. HP VEE will convert data types only when necessary.
- Turn off `Clear At Prerun` and `Clear At Activate` on displays where not needed.
- Use `Initialize At PreRun` and `Initialize At Activate` instead of setting defaults with control pins.
- Collect data for graphical displays and plot the *entire* array at once rather than plotting each individual scalar point. If the `X` values of a plot are regularly spaced, use an `XY Trace` display rather than an `X vs Y Plot`.
- Run the model from the panel view (if the panel view contains fewer objects then the detail view).
- Set graphical displays to be as plain as possible. The settings that allow the fastest update times are `Grid Type` ⟹ `No Grid` and `Panel Layout` ⟹ `Graph Only`.
- Iconify those objects that continuously update their displays (such as `Timer`, `Counter`, `Accumulator`, and `Display` objects).
- Use parallel operations (processing an array at a time) rather than iterators (processing each element separately).
- Use Complex data in expressions rather than PComplex. Most of the math libraries will convert PComplex to Complex, calculate the answer, and convert Complex back to PComplex.

- To display PComplex data, set `Trig Mode` (under `Edit` ⟹ `Preferences`) to `Radians`. HP VEE internally stores PComplex values as radians.
- In general, the fewer objects that need to operate, the faster the model will run. Perform as many functions as possible in each object.
- Connect the sequence input pins on displays so that displays do not operate on intermediate values; the displays wait to update until the final values are sent.

Many objects can perform a set of logical functions. Your models will be more compact and easier to maintain if you use the following techniques to use these objects to their fullest:

- Type equations in a `Formula` object instead of using multiple `Constant` ⟹ and single-function `Math` and `AdvMath` objects. You can nest functions in the `Formula` object. For example `(sin(ramp(100, 0, 360)))`.
- Use one `If/Then/Else` object with multiple conditions instead of multiple `If/Then/Else` or `Conditional` ⟹ objects.
- To input a one-dimensional array of data, use a `Constant` ⟹ object configured as an array instead of using one `Constant` ⟹ object per value and then building an array.
- To read all of the "rest" of the data available from a file or other source, you can use the `ARRAY 1D TO END:(*)` transaction. This is simpler than looping on single-element reads and collecting the result into an array.
- Use the `Sequencer` (chapter 13) to control the flow of execution of several User Functions.
- When using the `Sequencer`, only enable logging for transactions where the `Log` record is required. If the `Log` output pin is not used, delete it to speed up execution slightly.

Examples

The following examples illustrate some of the techniques listed above.

Parallel Operations

HP VEE can process data in any data shape. If you have an array of data and want to perform an operation on each element, you don't have to iterate through each element of the array. You just operate on the array as a whole.

Parallel operations are useful because they allow you to easily analyze your data in the shape that makes the most sense.

For example, if you want to multiply each element of an array by 100, you don't have to use an iterator to do it. Simply multiply the array by `100` as shown in Figure 8-1.

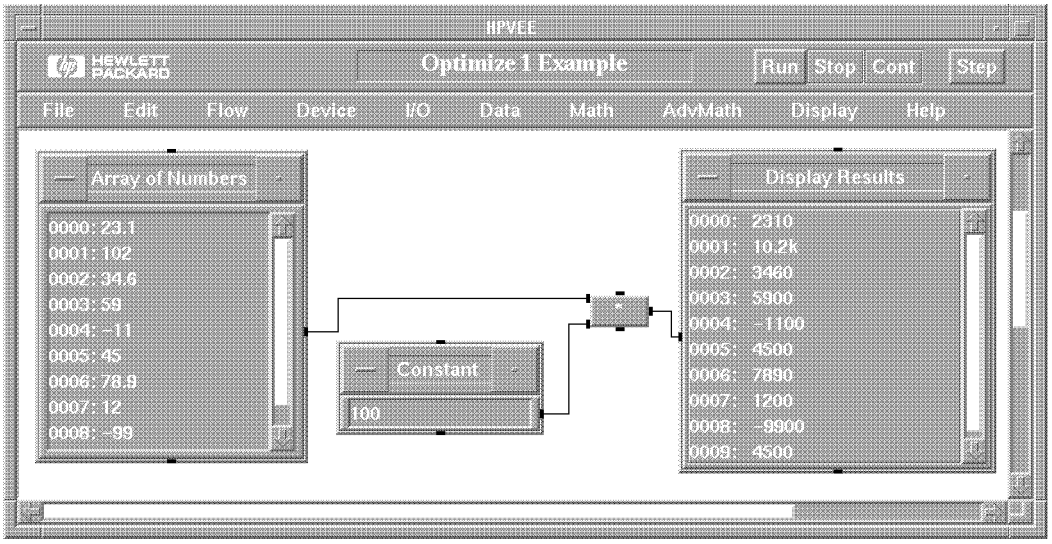


Figure 8-1. Example of a Parallel Operation

The model shown in Figure 8-1 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual24.ex
 - or -
/usr/lib/veetest/examples/concepts/manual24.ex
```

Figure 8-2 shows another example of saving time with parallel operations. In the top half of the example, an array with 100 elements is sent to the `Sin(X)`, `Cos(X)`, and `X vs Y Plot` objects, so each of these objects only executes once. In the bottom half of the example, a Scalar Real value is sent to the `Sin(X)`, `Cos(X)`, and `X vs Y Plot` objects 100 times. The graphical result of both methods is the same, but the top version runs more than 25 times faster.

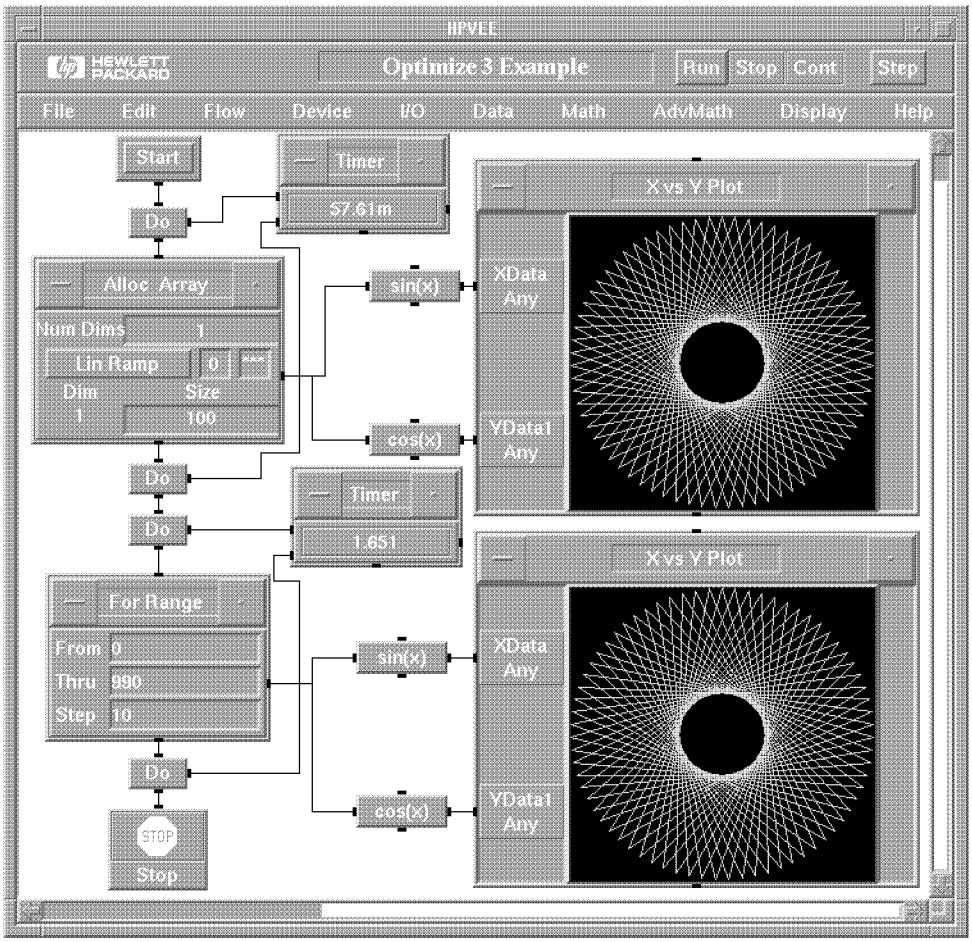


Figure 8-2. Another Parallel Operation Example

The model shown in Figure 8-2 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual25.ex
 - or -
/usr/lib/veetest/examples/concepts/manual25.ex
```

Showing the Icon Instead of the Open View

Figure 8-3 shows that updating any display (including the open view of a `Counter`) takes time: you can increase the speed of your model by iconifying display objects.

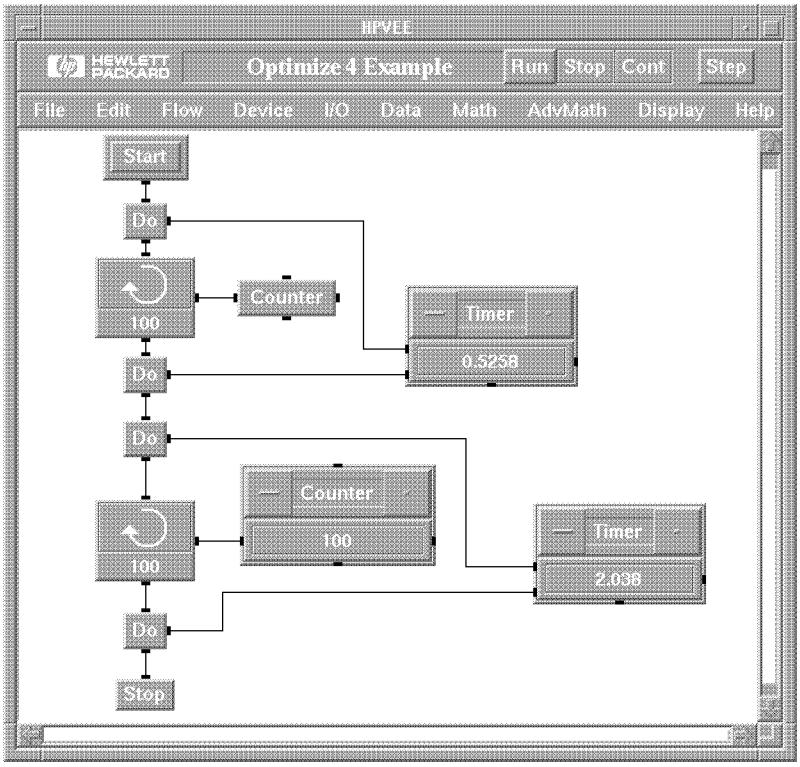


Figure 8-3. Increasing Speed with An Icon

8

The model shown in Figure 8-3 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual26.ex
 - or -
/usr/lib/veetest/examples/concepts/manual26.ex
```

Compacting Math Equations

The more objects on your work area, the more difficult it is to see the connections between objects and understand exactly the operations taking place. By compacting a math equation to a single `Formula` object, the model becomes easier to maintain.

Figure 8-4 shows you two ways to complete the same operation. The second thread (on the bottom) is more compact than the first thread.

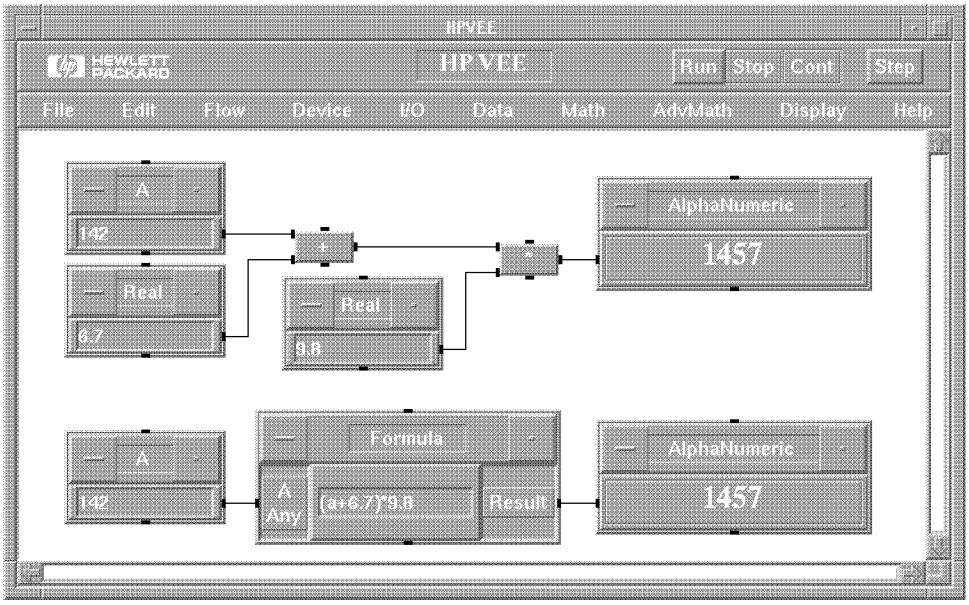


Figure 8-4. Compact Math Example

9

Understanding Common Structures

This chapter shows some common structures that may be useful in your models that allow you to perform the following tasks:

- Outputting values from an `If/Then/Else` object
- Specifying messages from Conditional objects
- Displaying one of multiple outputs
- Resetting buttons

For more examples of common structures, browse through the `/usr/lib/veeengine/examples/concepts/` or `/usr/lib/veetest/examples/concepts/` directory.

Outputting Values from If/Then/Else

Figure 9-1 shows how to get the value that satisfies a condition. For example, if `A > B`, then output `A`; if `B > A`, then output `B`. Figure 9-1 works only for Scalar data.

The `If/Then/Else` object evaluates each expression as a formula. The `If/Then/Else` returns a `1` if the condition is true. Since each condition is a formula box, you can operate on the result of the condition. If the expression evaluates as non-zero, the value propagates to the output pin. Otherwise it evaluates the next expression.

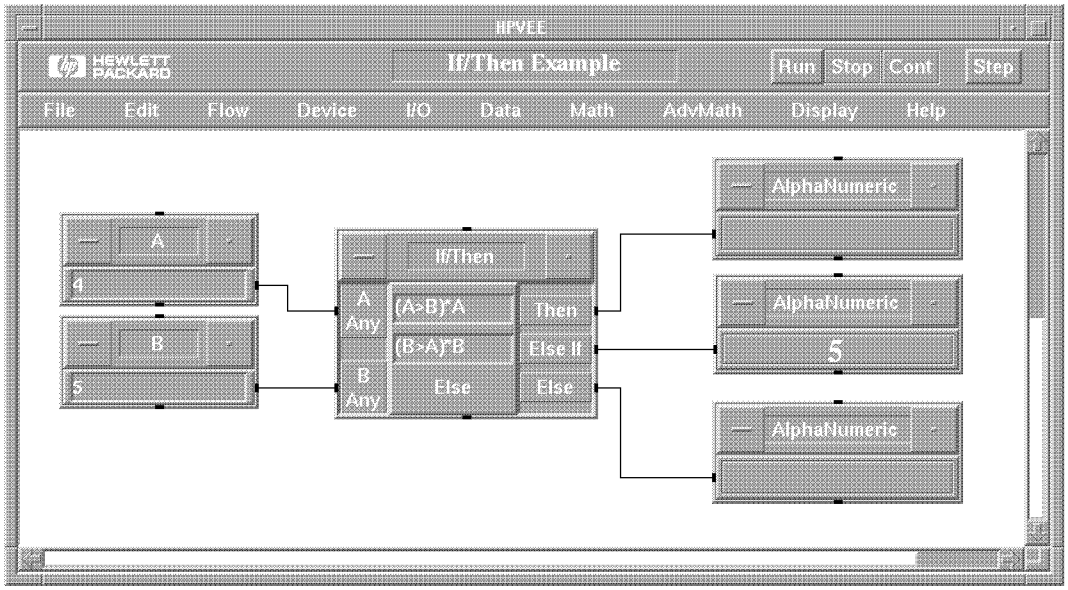


Figure 9-1. Getting a Value from If/Then/Else

Specifying Messages from Conditionals

Figure 9-2 shows how to output a specific message that is dependent on the condition met in any `If/Then/Else` or `Conditional` ⟹ object. Use a `Gate` to allow the message associated with the true condition to be propagated. Figure 9-2 works for both Scalar and Array data.

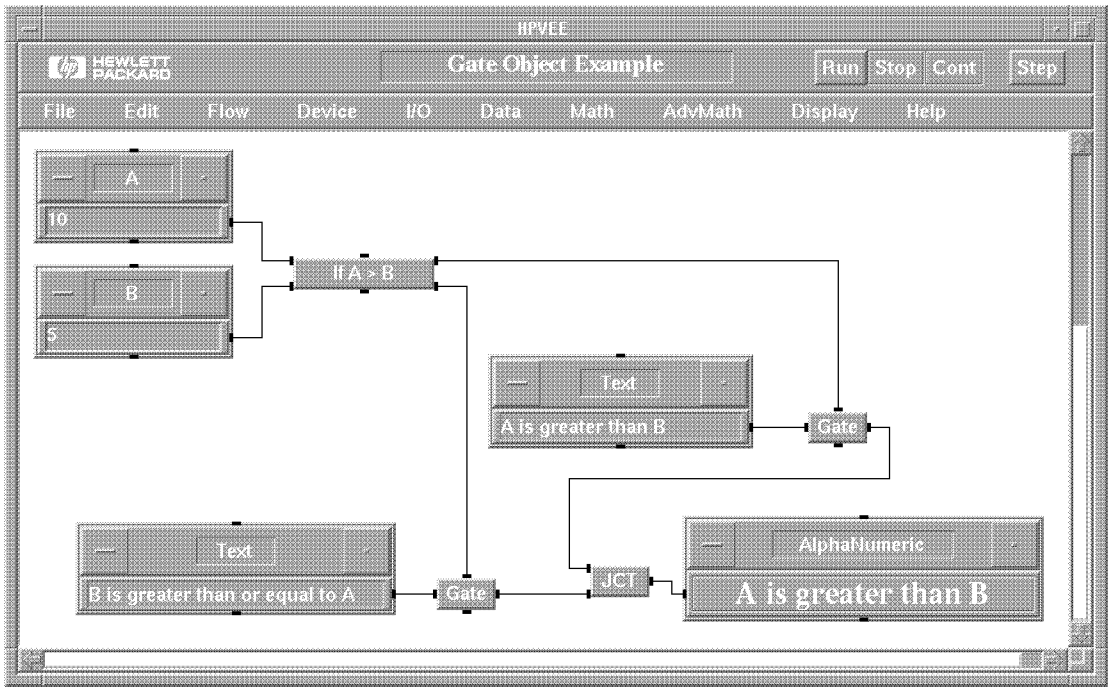


Figure 9-2. Specifying a Conditional Message

The model shown in Figure 9-2 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual36.ex
 - or -
/usr/lib/veetest/examples/concepts/manual36.ex
```

Displaying One of Multiple Outputs

The previous two examples also show how to use a `JCT` object to display whichever statement or value is true from multiple possibilities.

Resetting Buttons

The `Toggle` button is very useful for getting user input (a one or zero), but it must be reset to allow a user to repeat an action (such as clearing a display). The feedback on each `Toggle` button resets it by activating the `Reset` control pin.

Figure 9-3 shows two `Toggle` buttons. The first clears the display whenever the button is pushed; the second stops the model. Notice the use of `If/Then/Else` objects to check the `Toggle` output.

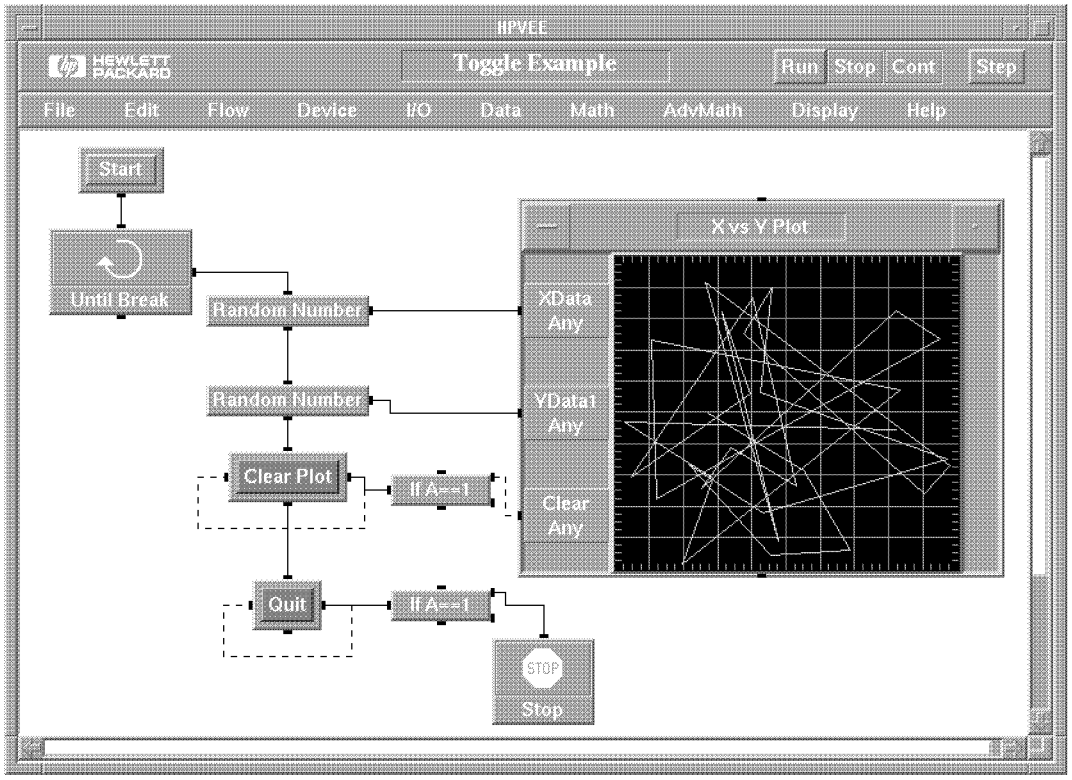


Figure 9-3. Using a `Toggle`

The model shown in Figure 9-3 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual37.ex
 - or -
/usr/lib/veetest/examples/concepts/manual37.ex
```

10

Using Records and DataSets

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

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

Record Containers

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

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

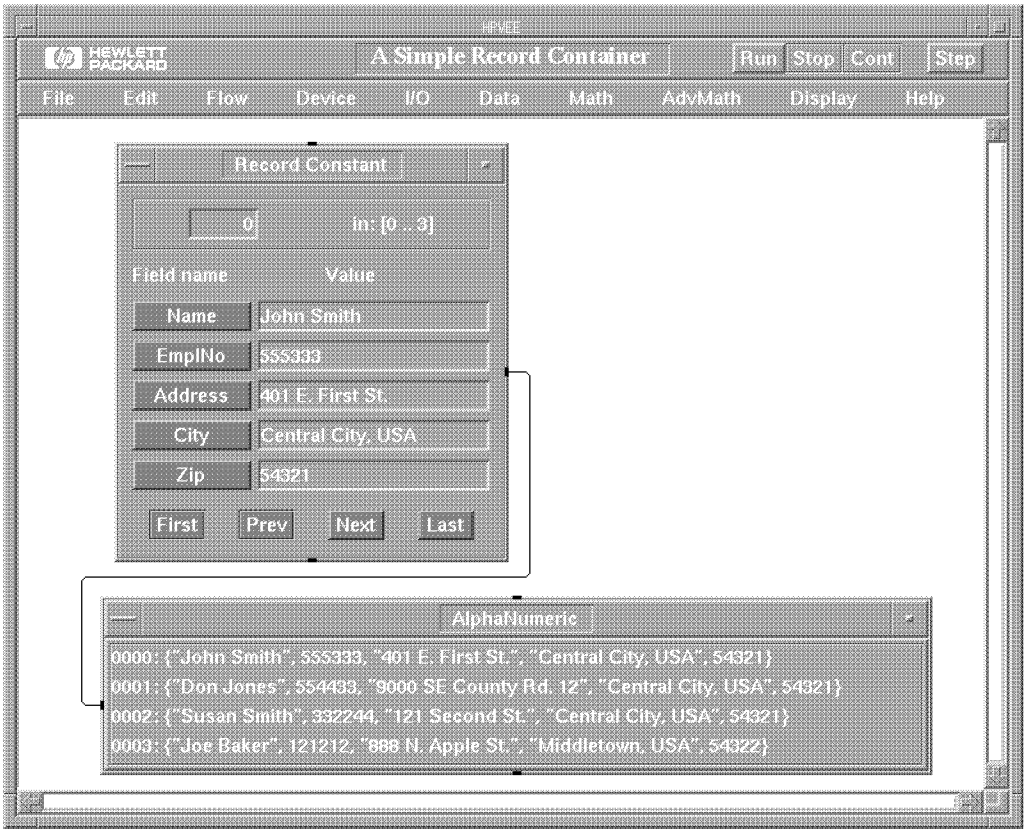


Figure 10-1. A Simple Record Container

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

Accessing Records

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

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

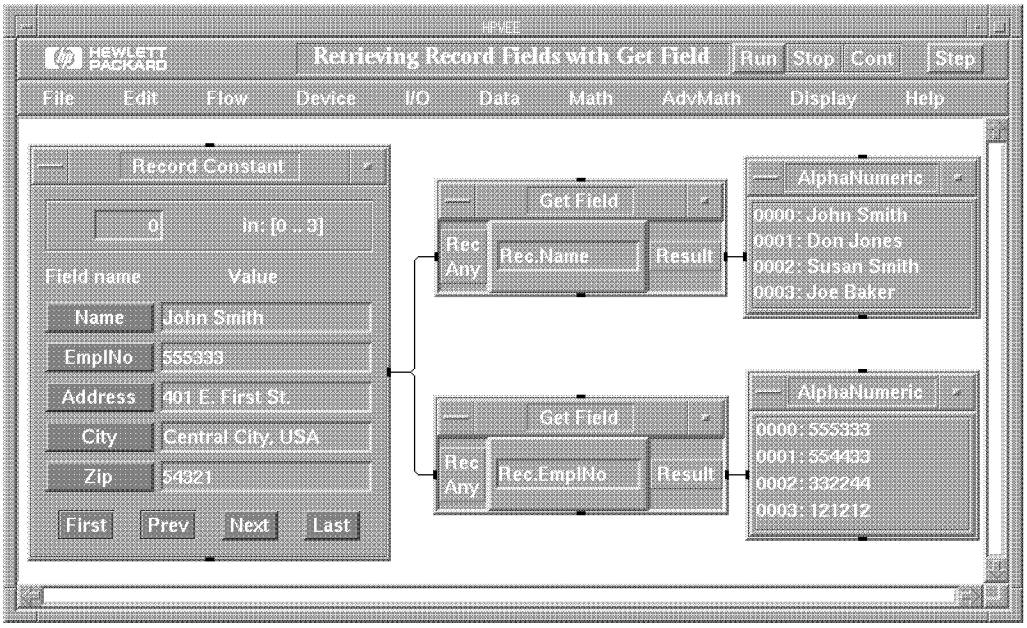


Figure 10-2. Retrieving Record Fields with Get Field

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

example, you could use this syntax in a transaction in the `To String` object, but more about that later.

In the previous example, the entire `Name` and `EmplNo` fields were obtained, that is, the entire array for each field. But suppose you want the `Name` and `EmplNo` fields from a single array element. You can use the array syntax `Rec[1].Name` and `Rec[1].EmplNo` to obtain just the second element ("element `1`") of each field:

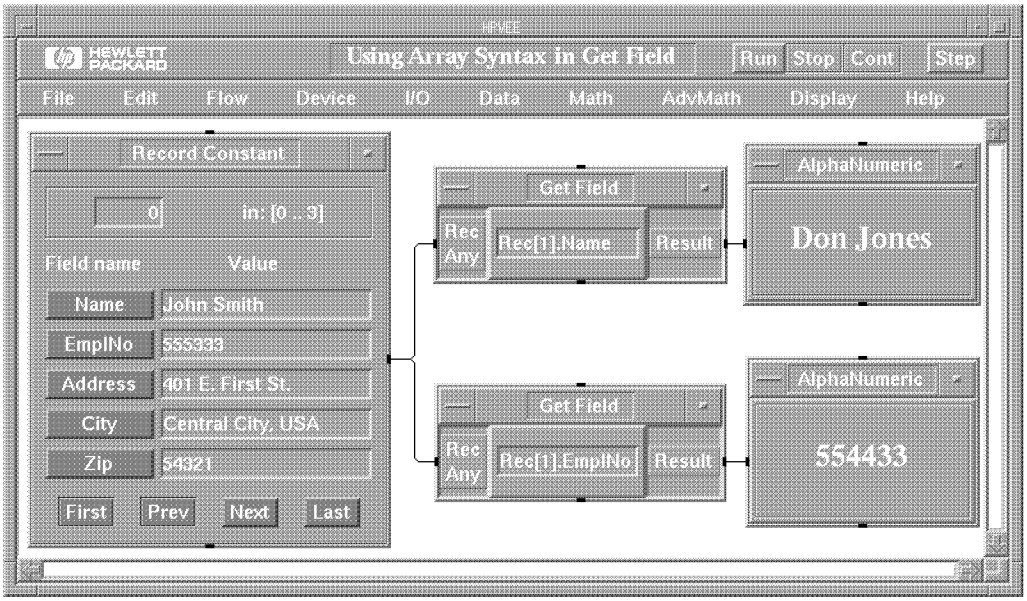


Figure 10-3. Using Array Syntax in Get Field

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

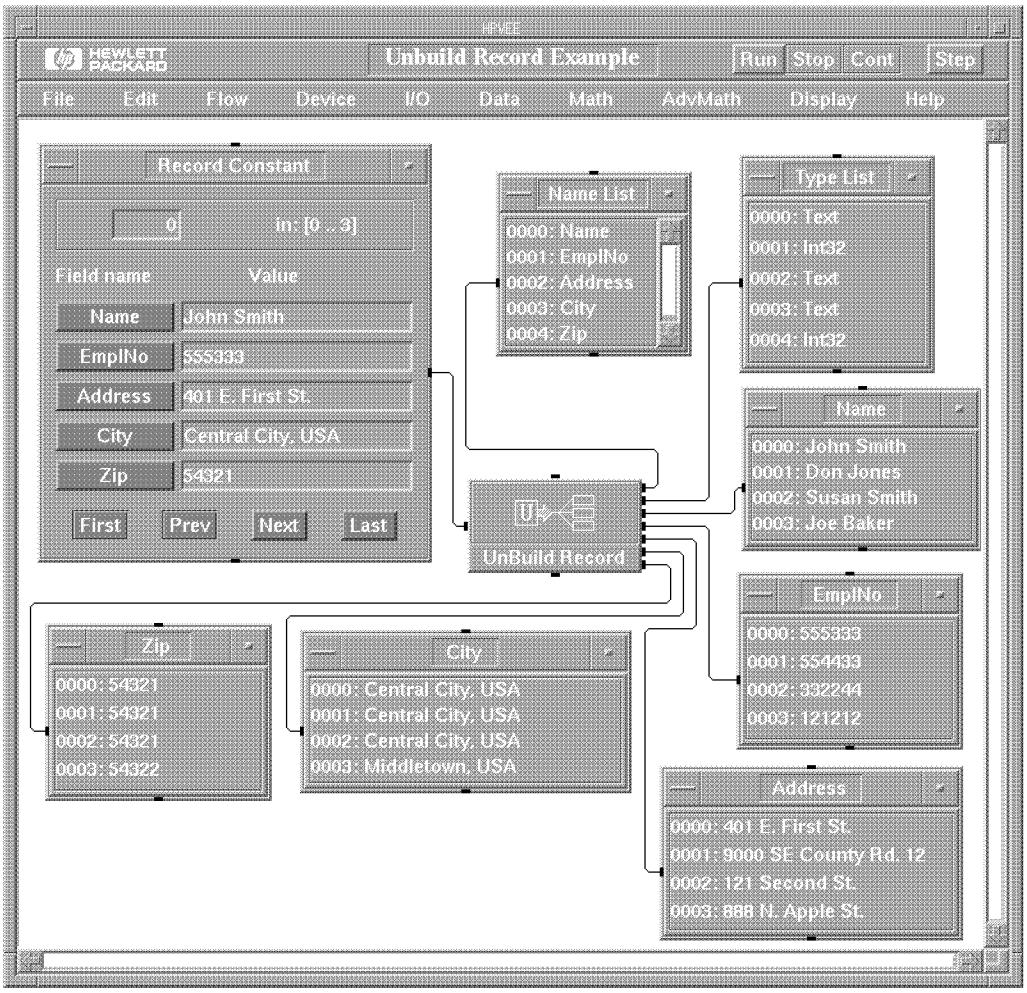


Figure 10-4. Retrieving Record Fields with UnBuild Record

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

```
/usr/lib/veeengine/examples/concepts/manual38.ex
 - or -
/usr/lib/veetest/examples/concepts/manual38.ex
```

Building Records

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

Note

The Record data type has the highest precedence of all HP VEE data types. However, data cannot be converted to and from the Record data type through the automatic promotion/demotion of data types described in chapter 3 of this manual. For example, you cannot send Record data into an input terminal contrained to be Real. Instead, you must perform these conversions by using Build Record and Unbuild Record, or with the Rec.A "dot" syntax described earlier.

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

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

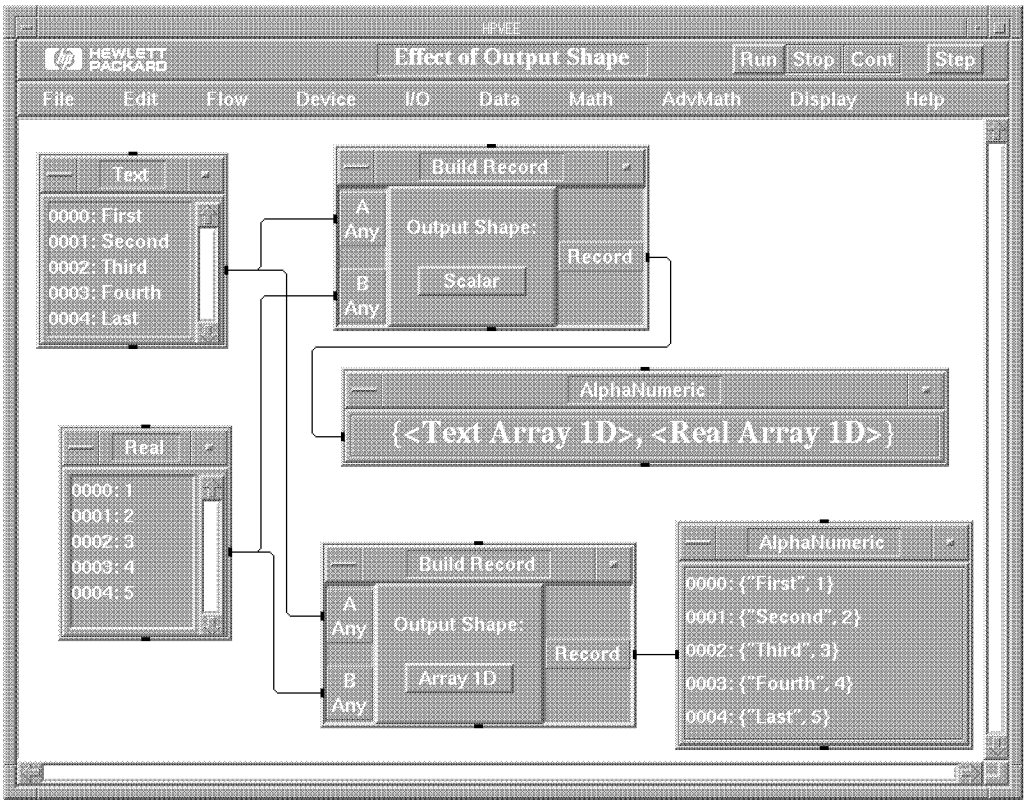


Figure 10-5. The Effect of Output Shape in Build Record

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

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

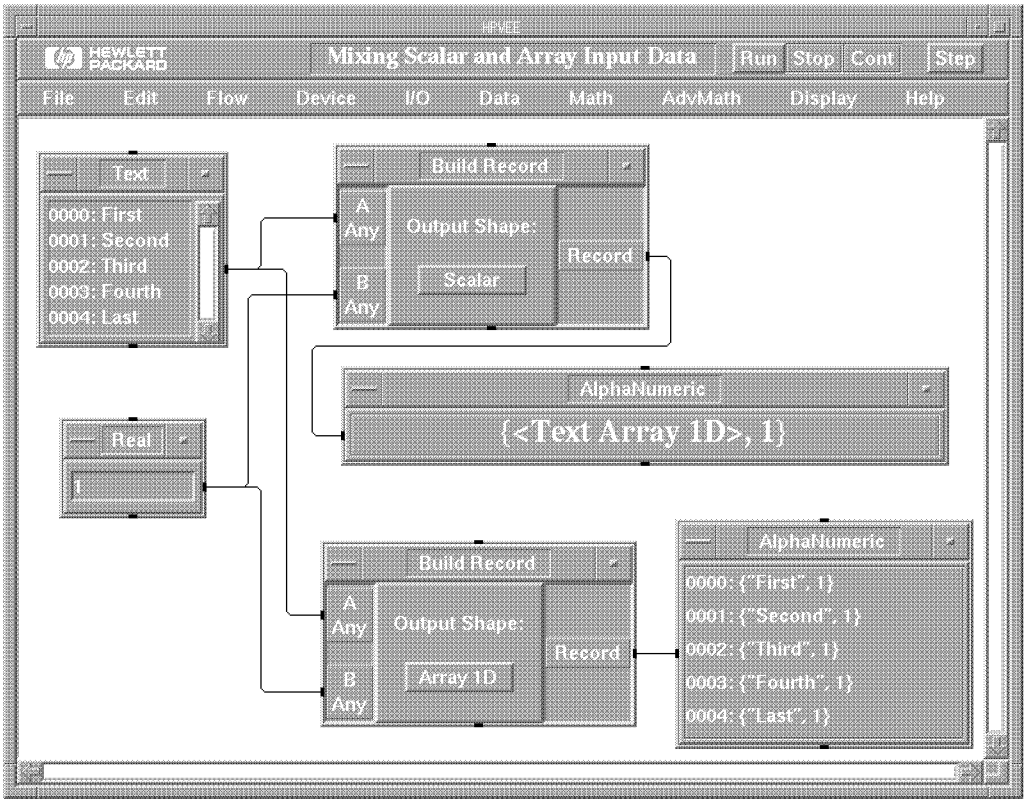


Figure 10-6. Mixing Scalar and Array Input Data

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

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

Editing Record Fields

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

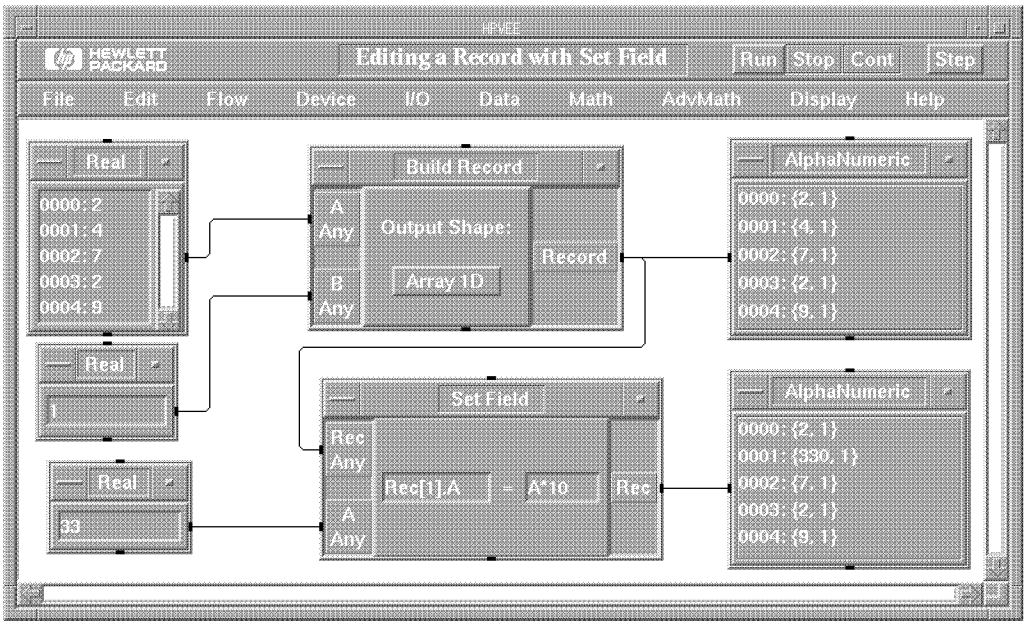


Figure 10-7. Using Set Field to Edit a Record

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

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

is very similar to a BASIC assignment statement such as:

```
A=B*10
```

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

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

Building Records Containing Waveforms

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

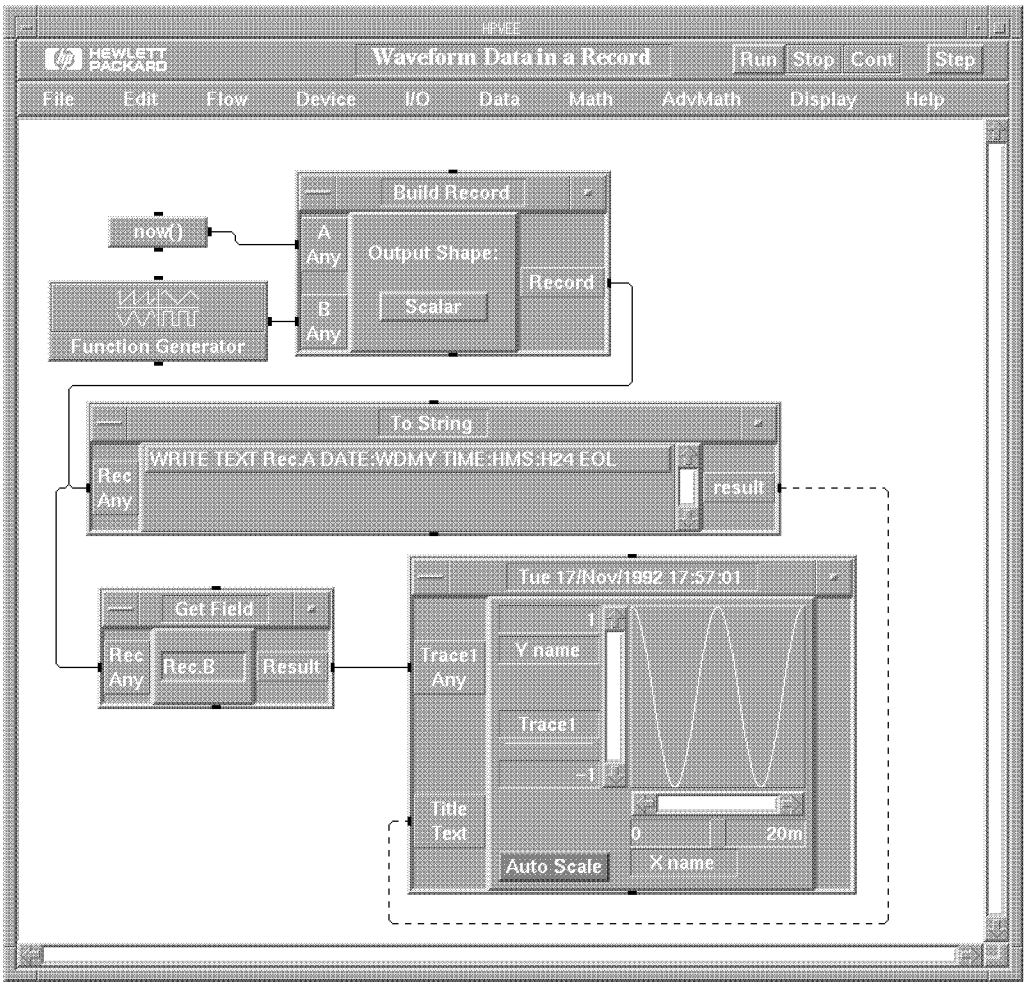


Figure 10-8. Building a Record from Waveform Data

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

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

The above model is saved in:

```
/usr/lib/veeengine/examples/concepts/manual39.ex
 - or -
/usr/lib/veetest/examples/concepts/manual39.ex
```

Using Global Records

In chapter 3 we briefly looked at global variables, and the Set Global and Get Global objects. In chapter 6 we looked at using global variables in UserObjects. In many cases you may have several related global variables of different data types and shapes. It is often useful to group these global variables together into one global variable, which is a record. Let's look at how to create and use global variables of the Record data type. The process is really quite simple. Just build the record with Build Record and output it to the Set Global object, as shown in the following example:

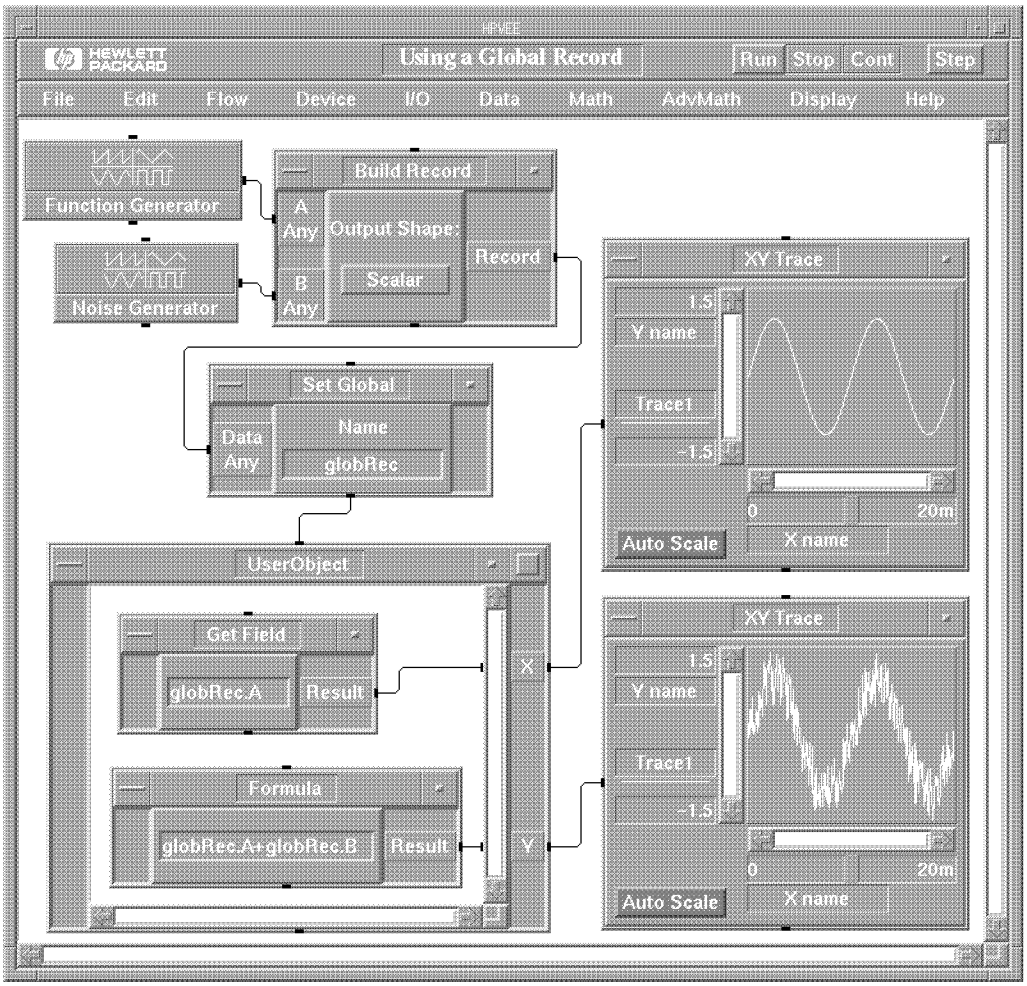


Figure 10-9. Using a Global Record

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

In the example, expressions in the `Get Field` and `Formula` objects, inside the UserObject, retrieve the waveform data from the global record. Note that a

global record, like any global, is valid in any context, including a UserObject or User Function.

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

Using DataSets

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

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

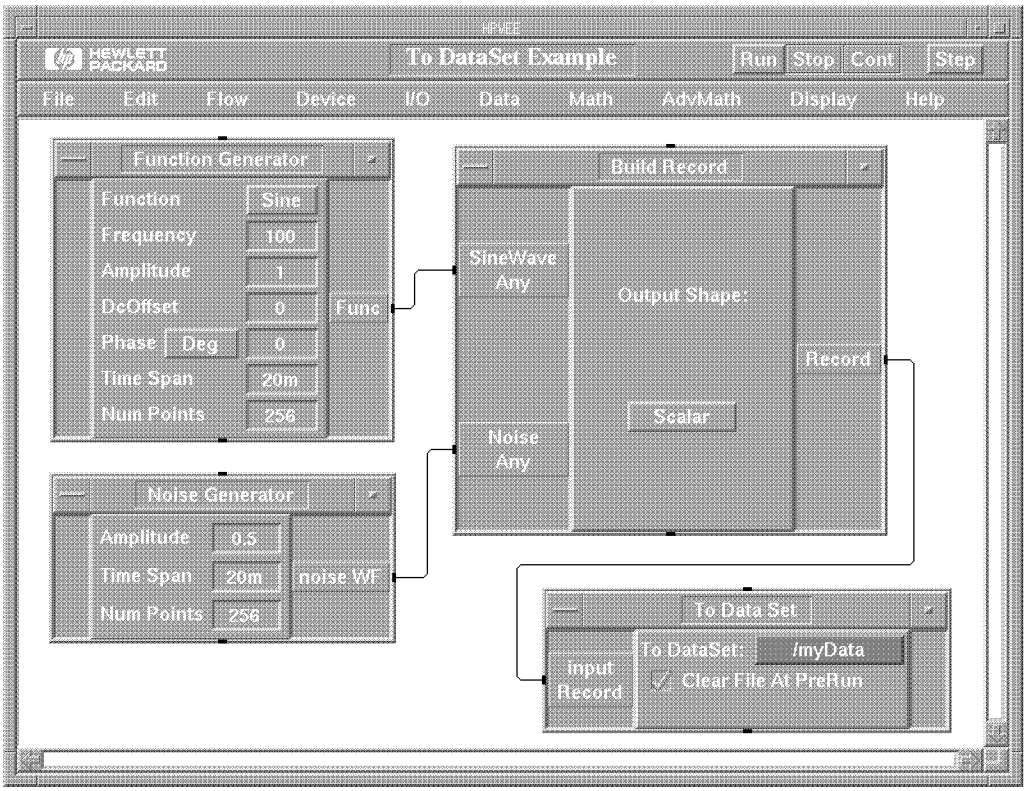


Figure 10-10. Using To DataSet to Save a Record

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

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

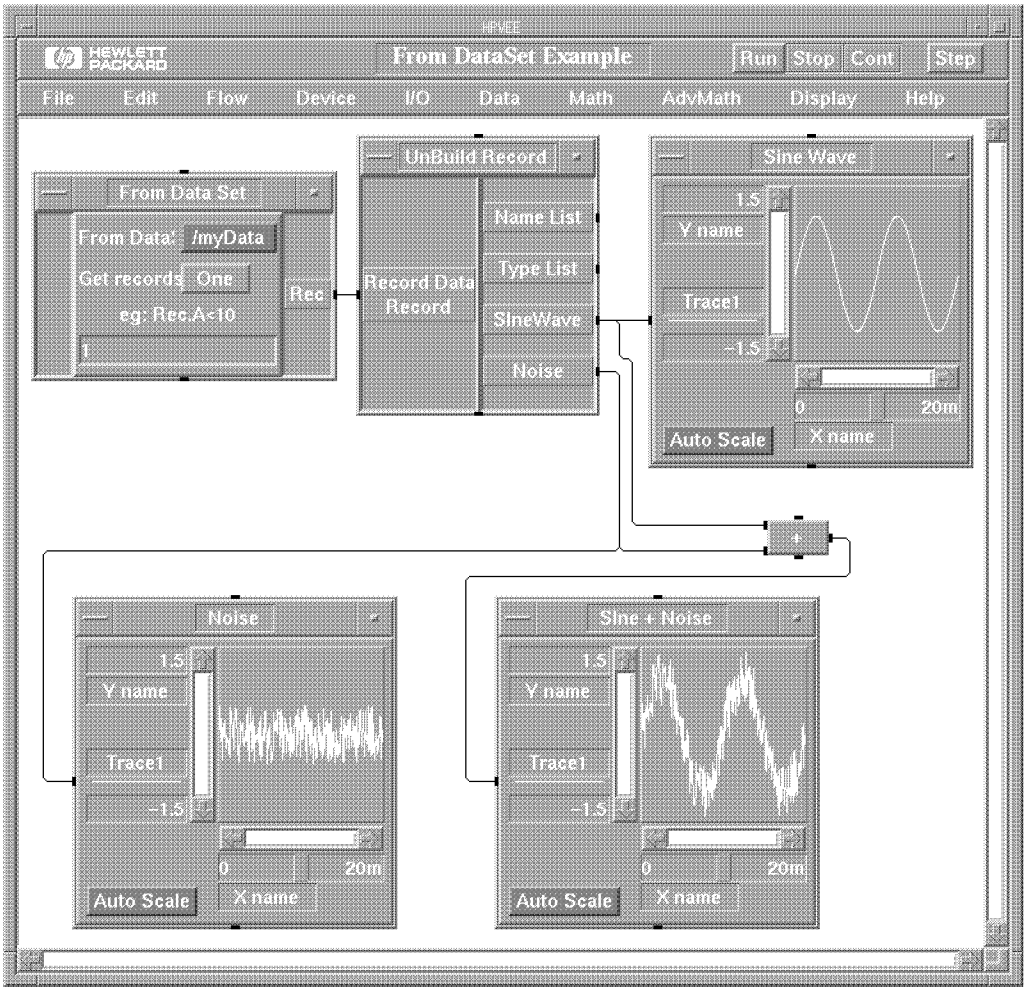


Figure 10-11. Using From DataSet to Retrieve a Record

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

The pair of models of this last example are saved in:

```
/usr/lib/veeengine/examples/concepts/manual40.ex
/usr/lib/veeengine/examples/concepts/manual41.ex
 - or -
/usr/lib/veetest/examples/concepts/manual40.ex
/usr/lib/veetest/examples/concepts/manual41.ex
```

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

11

Creating User-Defined Functions

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

User Functions

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

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

11 Creating a User Function

The first step in creating any User Function is to create a UserObject (refer to chapter 6, "Building UserObjects"). The following model contains a UserObject that adds a noise component to the waveform on its `Y` input, and then outputs the modified waveform on its `Y+Noise` output terminal. Note that the amplitude of the noise component is controlled by the `Real Slider` connected to the `Amplitude` input of the UserObject.

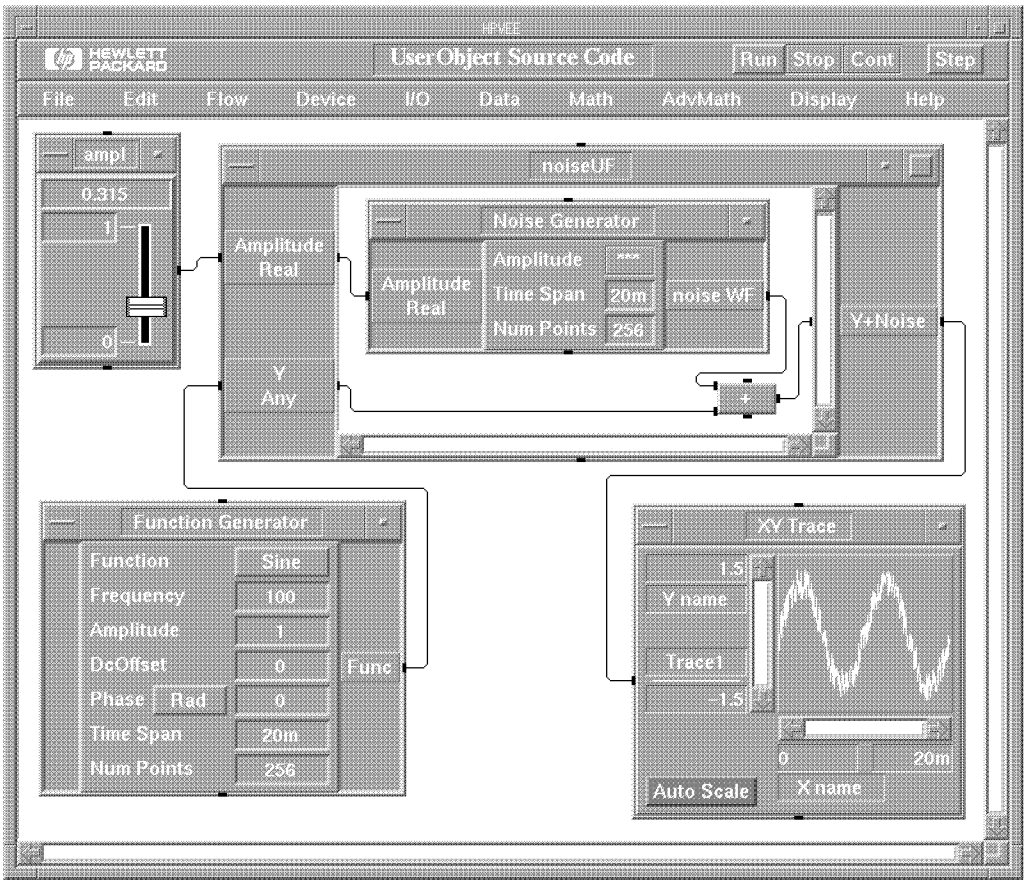


Figure 11-1. Model with UserObject

In our example, the UserObject adds a noise component to the 100 Hz sine wave output by the `Function Generator`. You may want to load this model and follow along with our example. The example is saved in:

```
/usr/lib/veeengine/examples/concepts/manual42.ex
 - or -
/usr/lib/veetest/examples/concepts/manual42.ex
```

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

Note

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

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

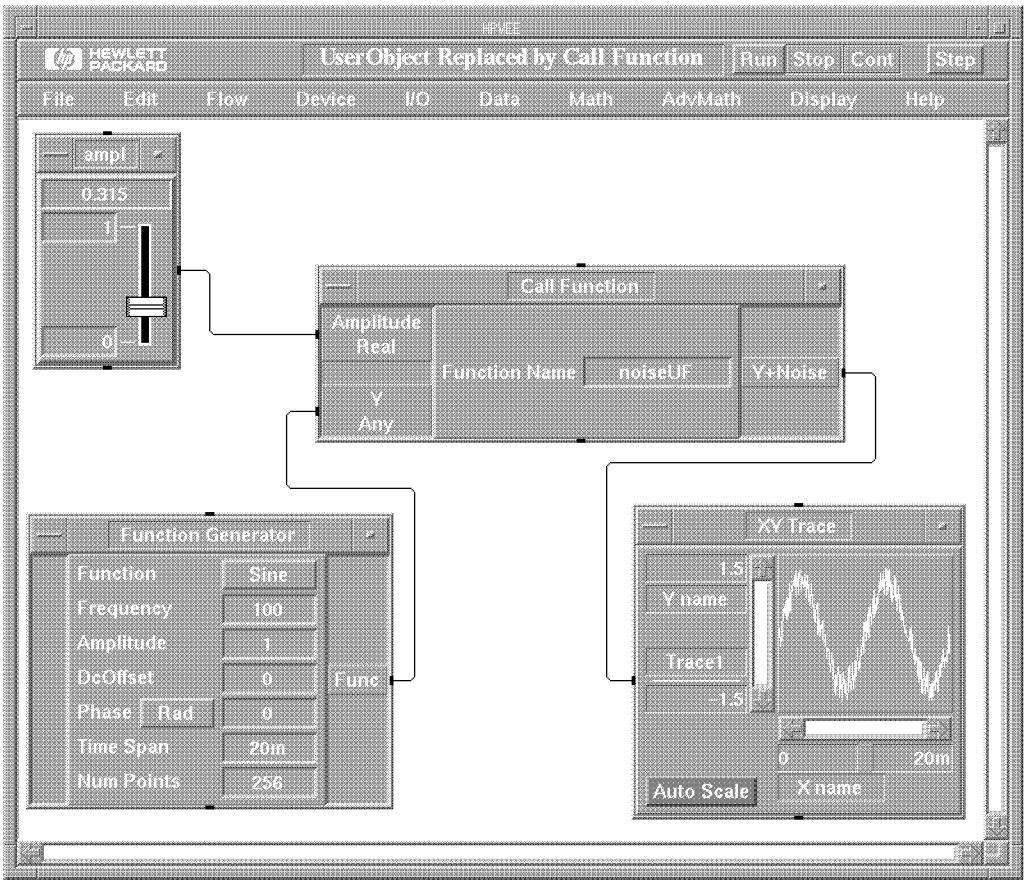


Figure 11-2. UserObject Replaced by Call Function

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

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

Editing a User Function

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

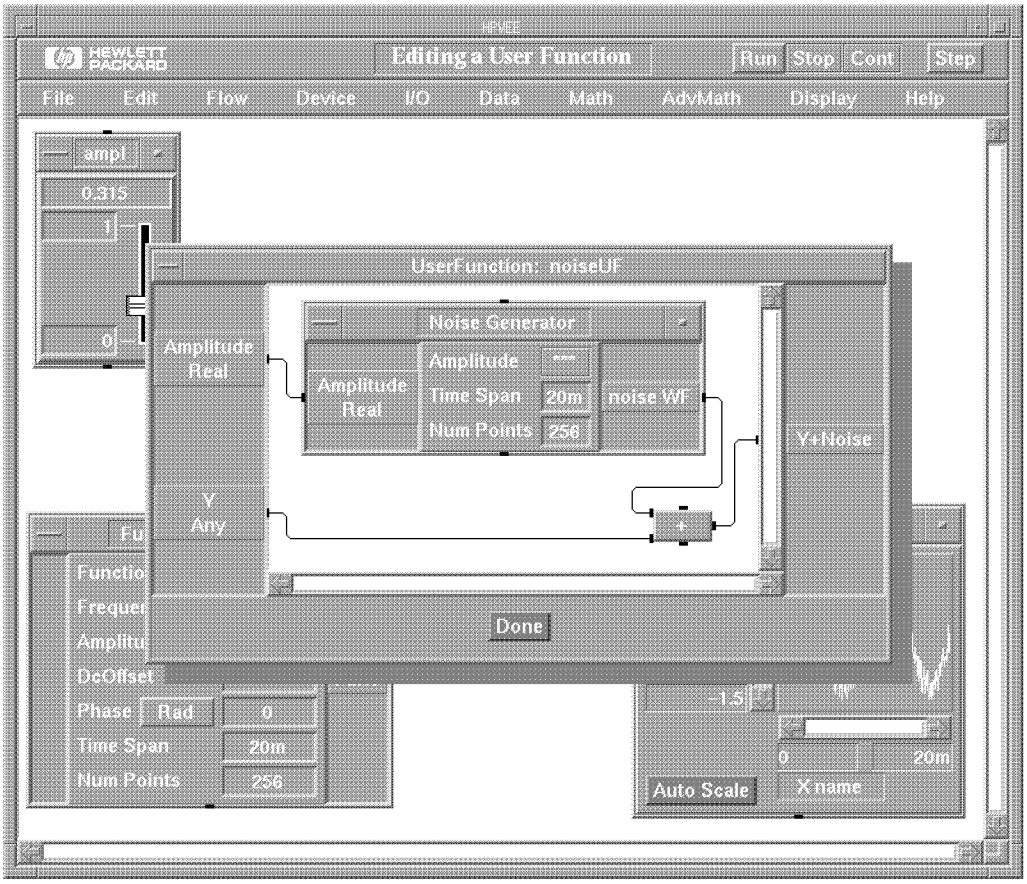


Figure 11-3. Editing a User Function

Note

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

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

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

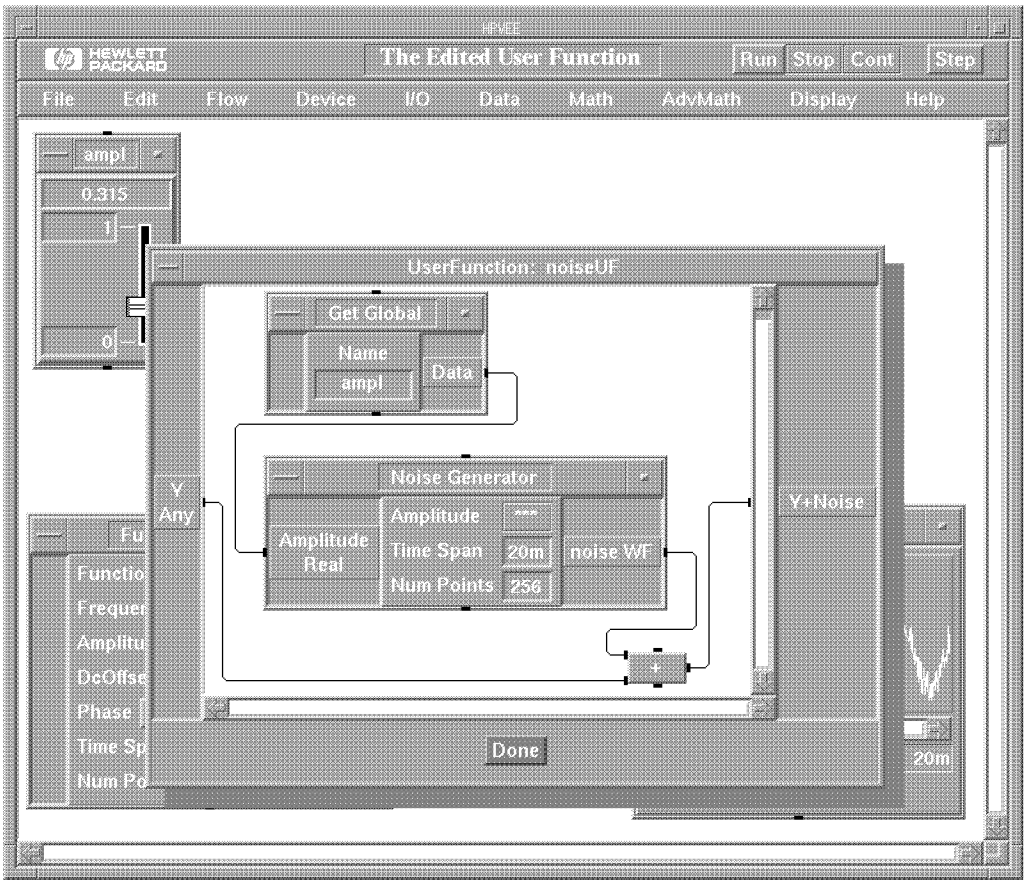


Figure 11-4. The Edited User Function

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

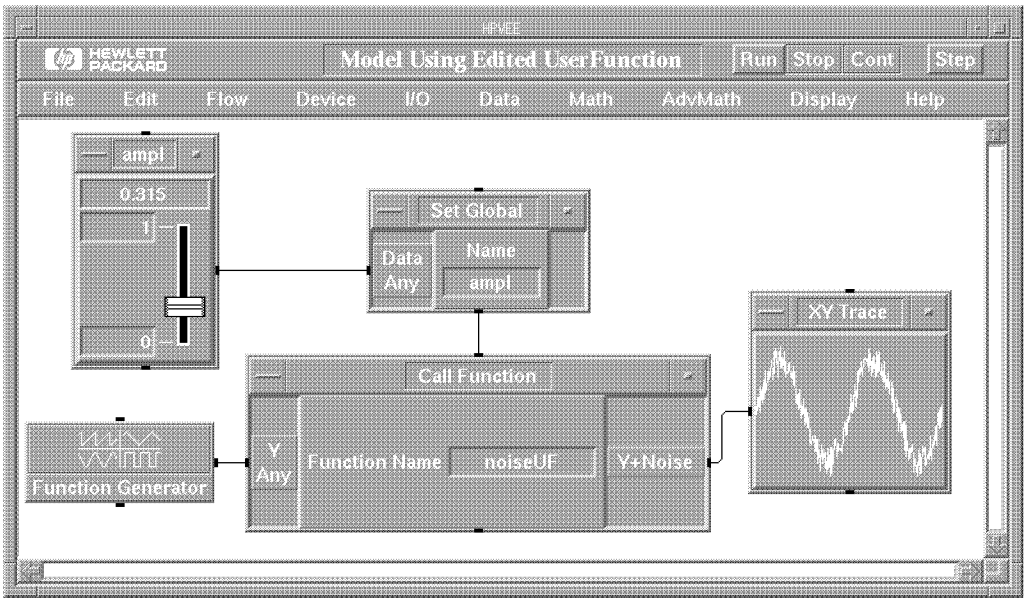


Figure 11-5. Model Using Edited User Function

This model performs the same task as the model of Figure 11-2, but uses a global variable to set the amplitude of the noise component. Note that the sequence output pin of the `Set Global` is connected to the sequence input pin of the `Call Function` object. This ensures that the global variable `ampl` will be set before it is called by the `Get Global` within the User Function.

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

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

can reconfigure the `Call Function` pinout at any time by selecting `Configure pinout` from the object menu.)

Note

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

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

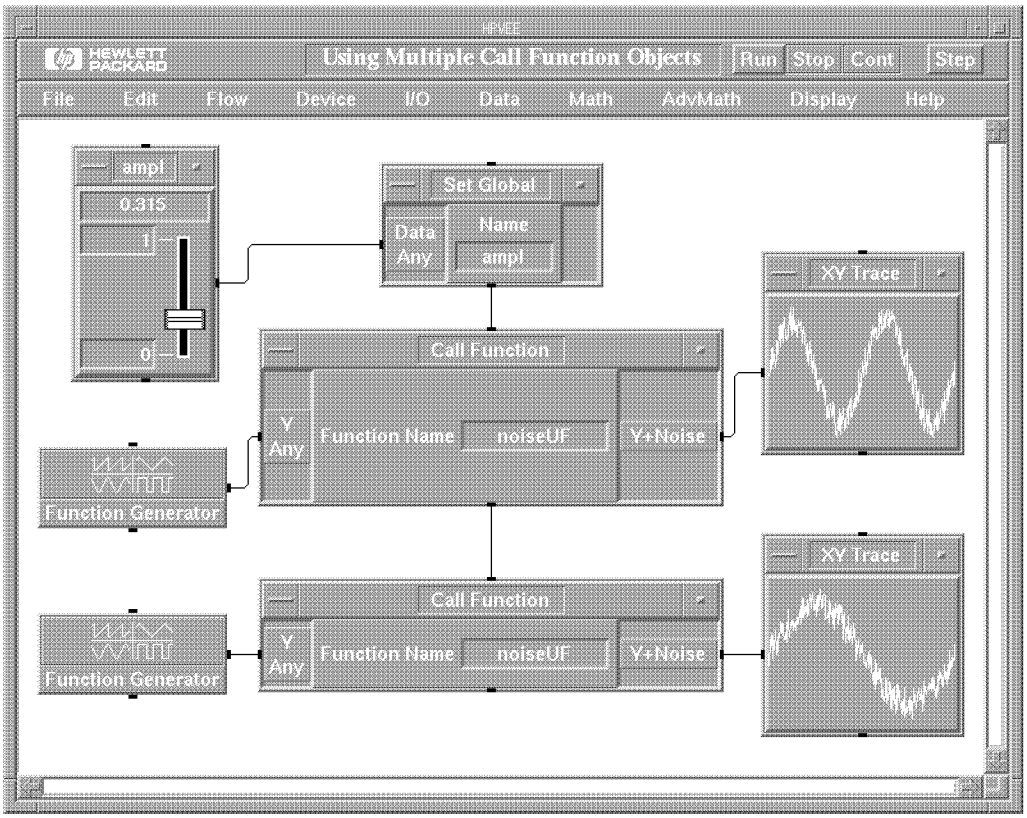


Figure 11-6. Using Multiple Call Function Objects

11

Calling a User Function from an Expression.

You don't have to use the `Call Function` object to call a User Function. In fact you can call a User Function from an expression in a `Formula` object, or from any expression evaluated at run time. The following model extends the example of Figure 11-5 by adding two `Formula` objects.

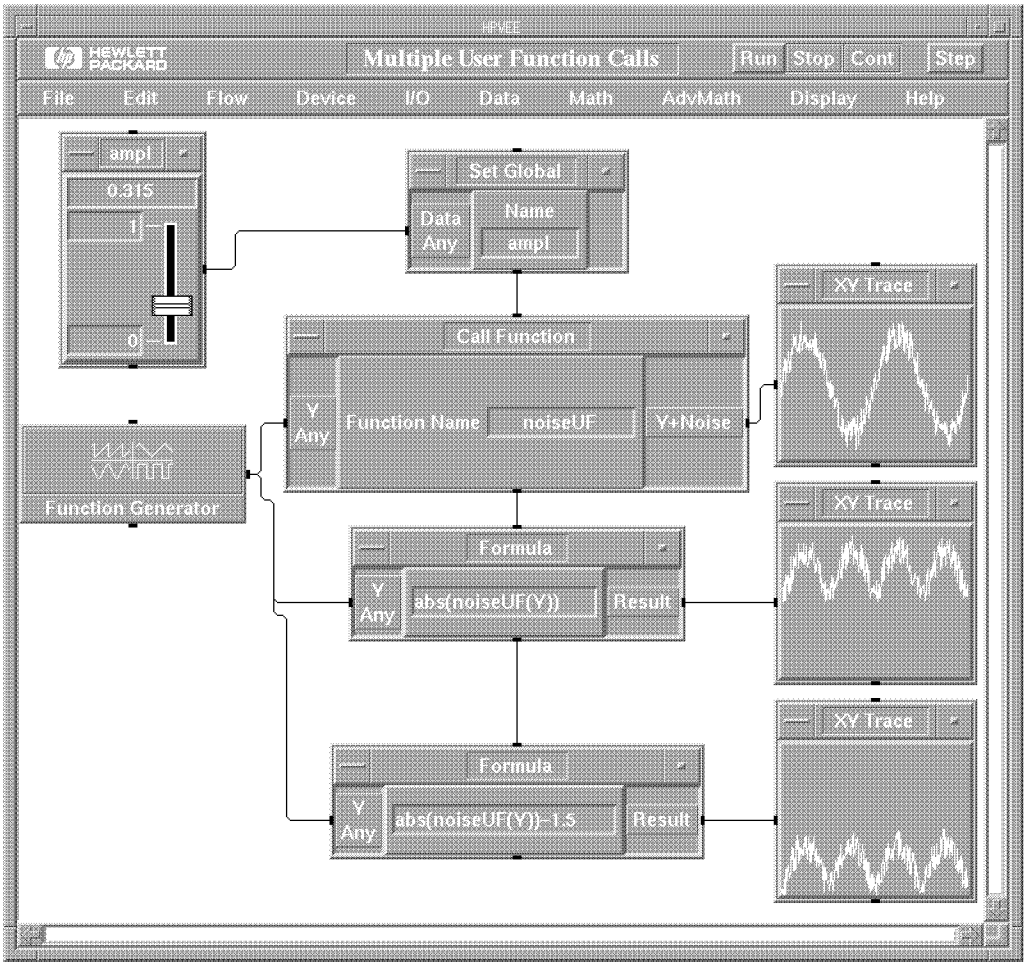


Figure 11-7. Calling a User Function from Expressions

In the model, the `Call Function` object calls the User Function `noiseUF` and returns a sine wave with an added noise component, as before. The expression `abs(noiseUF(Y))` in the first `Formula` object returns the absolute value of the waveform returned by the User Function `noiseUF`. Thus, the displayed noisy sine wave is "rectified" in the positive direction. The expression `abs(noiseUF(Y))-1.5` in the second `Formula` object does the same, but also adds a negative "dc offset" to the waveform. Note that, as in the previous example, the sequence pins are used to ensure correct propagation with the global variable.

This model is saved in:

```
/usr/lib/veeengine/examples/concepts/manual43.ex
 - or -
/usr/lib/veetest/examples/concepts/manual43.ex
```

The ability to call a User Function from an expression is very useful — especially when you include such an expression in a transaction in the `Sequencer` object. Refer to chapter 13 for more information about this topic.

Creating a User Function Library

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

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

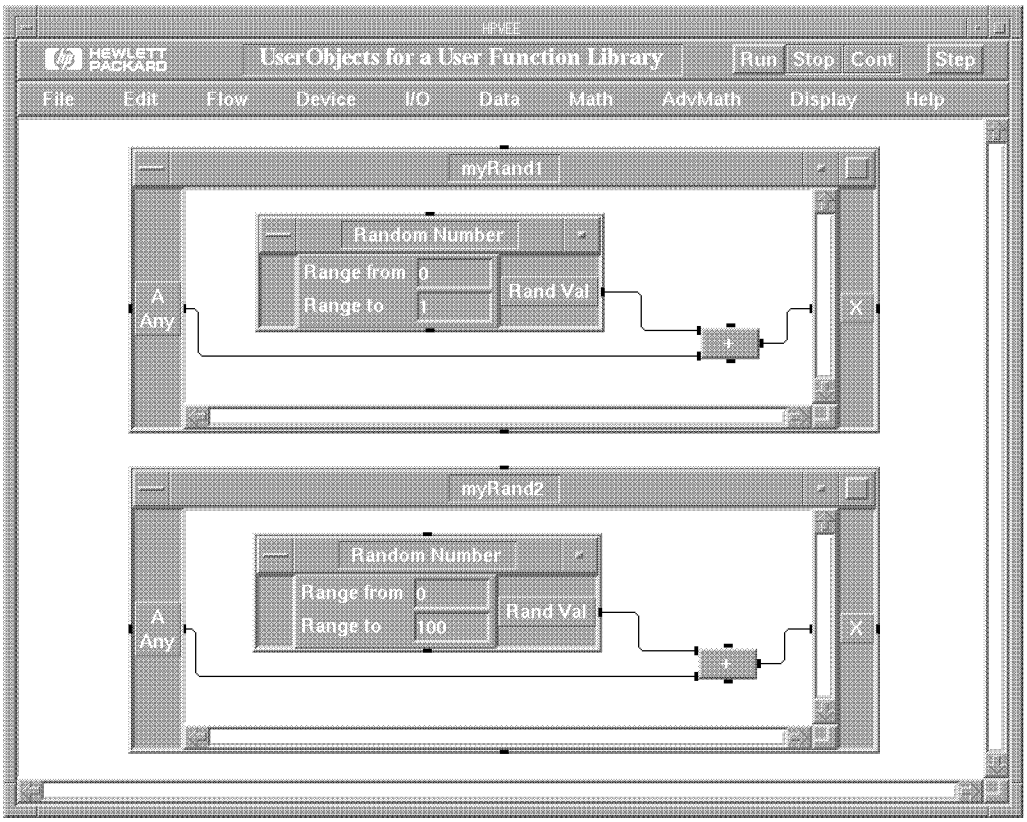


Figure 11-8. Creating UserObjects for a User Function Library

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

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

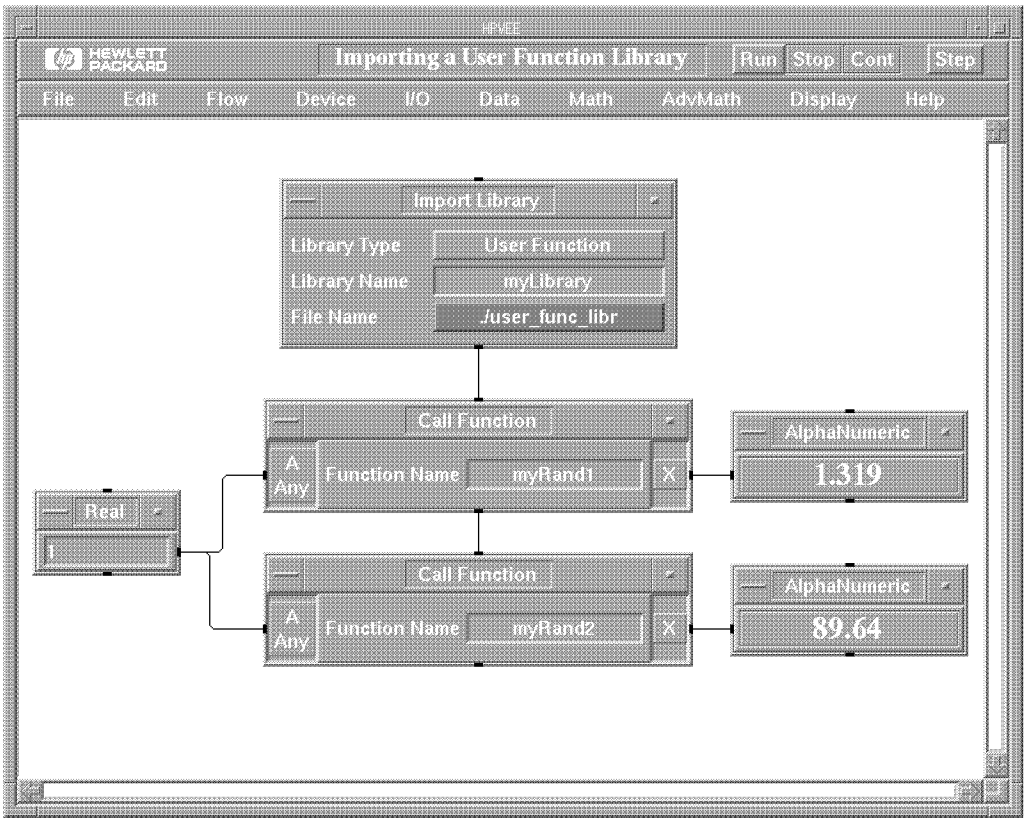


Figure 11-9. Importing a User Function Library

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

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

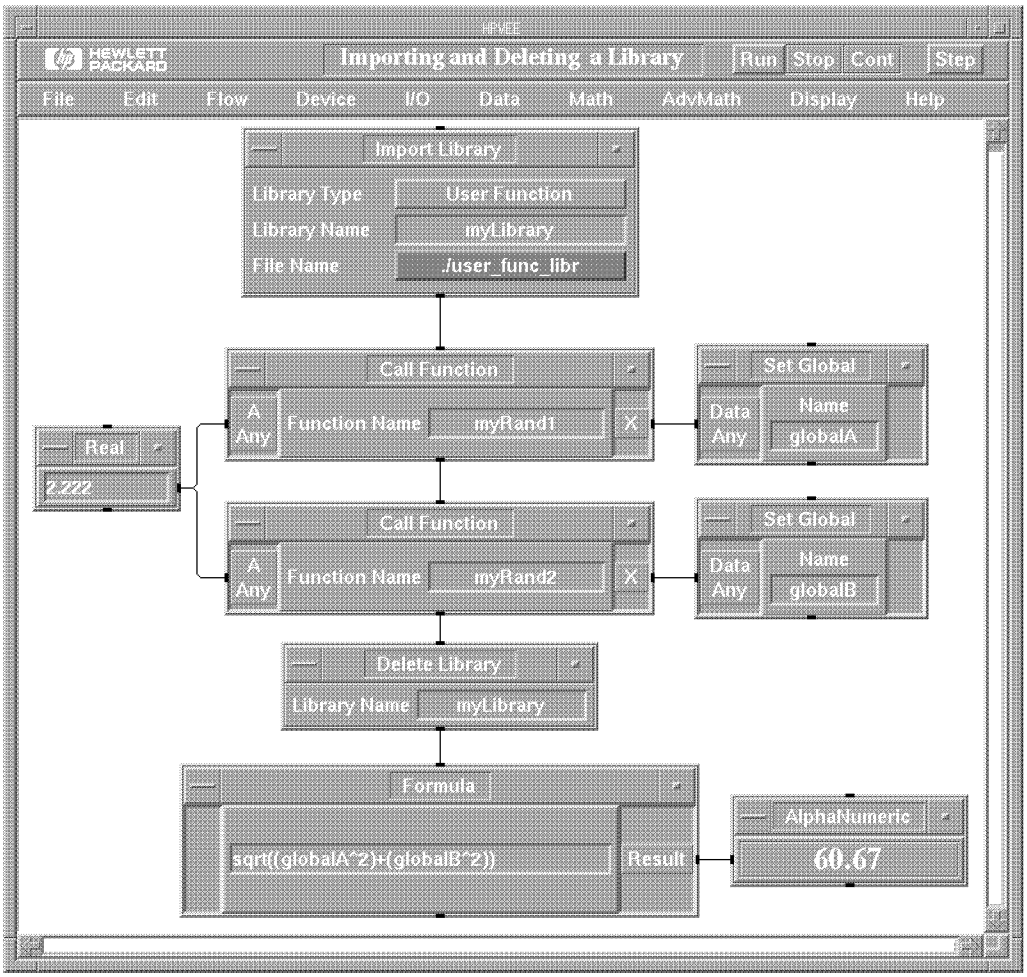


Figure 11-10. Importing and Deleting a User Function Library

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

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

Note

You cannot edit the User Functions imported with `Import Library` — you can only call them. However, you can *merge* a library of User Functions using `Merge Library` from the `File` menu. Once the library is merged into your model, you can edit the individual User Functions with `Edit UserFunction`.

Compiled Functions

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

Note

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

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

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

Design Considerations for Compiled Functions

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

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

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

- HP VEE can't trap errors originating in the external routine. Because your external routine becomes part of the HP VEE process, any errors in that routine will propagate back to HP VEE, and a failure in the external routine may cause HP VEE to "hang" or otherwise fail. Thus, you need to be sure of what you want the external routine to do, and provide for error checking in the routine. Also, if your external routine exits, so will HP VEE.
- Your routine must manage all memory that it needs. Be sure to deallocate any memory that you may have allocated when the routine was running.
- Your external routine cannot convert data types the way HP VEE does. Thus, you should configure the data input terminals of the `Call Function` object to accept *only* the type and shape of data that is compatible with the external routine.

- If your external routine accepts arrays, it must have a valid pointer for the type of data it will examine. Also, the routine must check the size of the array on which it is working. The best way to do this is to pass the size of the array from HP VEE as an input to the routine, separate from the array itself. If your routine overwrites values of an array passed to it, use the return value of the function to indicate how many of the array elements are valid.
- System I/O resources may become locked. Your external routine is responsible for timeout provisions, and so forth.

Importing and Calling a Compiled Function

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

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

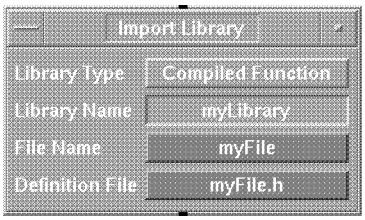


Figure 11-11. Using Import Library for Compiled Functions

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

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

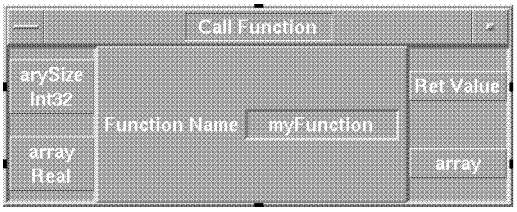


Figure 11-12. Using Call Function for Compiled Functions

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

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

Creating a Compiled Function

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

The Definition File

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

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

Where:

- `<return type>` can be: `long`, `double`, or `char*`.
- `<function name>` can be a string consisting of an alpha character followed by alphanumeric characters, up to a total of 512 characters.

  string of from 1 to 32 characters.
- `<type>` can be: `long`, `double`, `char*`, `long*`, `double*`, or `char**`.
- `<paramname>` can be a string consisting of an alpha character followed by alphanumeric characters, up to a total of 512 characters. The parameter names are optional, but it is recommended to include them. If a parameter is to be passed by reference, the parameter name must be preceded by the indirection symbol (`*`).

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

If you specify "pass by reference" for a parameter by preceding the parameter name with `*`, HP VEE will pass the address of the information to your function. If you specify "pass by value" for a parameter by leaving out the

*, HP VEE will copy the value (rather than the address of the value) to your function. You'll want to pass the data by reference if your external routine changes that data for propagation back to HP VEE. *Also, all arrays must be passed by reference.*

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

HP VEE allows up to 20 parameters to be passed by reference to a Compiled Function. On the other hand, up to 20 long integer parameters, or up to 10 double precision floating point parameters, may be passed by value.

Note

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

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

```
/*comments*/
```

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

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

Building a C Function

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

The following C function accepts a real array and adds 1 to each element in the array. The modified array is returned to HP VEE on the `Array` terminal, while the size of the array is returned on the `Ret Value` terminal. This function, once linked into HP VEE, becomes the Compiled Function called in the HP VEE model shown in Figure 11-13.

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

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

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

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

The definition file for this function is as follows:

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

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

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

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

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

The example program uses the ANSI C function prototype. This isn't necessary, but it makes things a little easier to understand. The function prototype declares the data types that HP VEE should pass into the function.

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

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

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

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

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

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

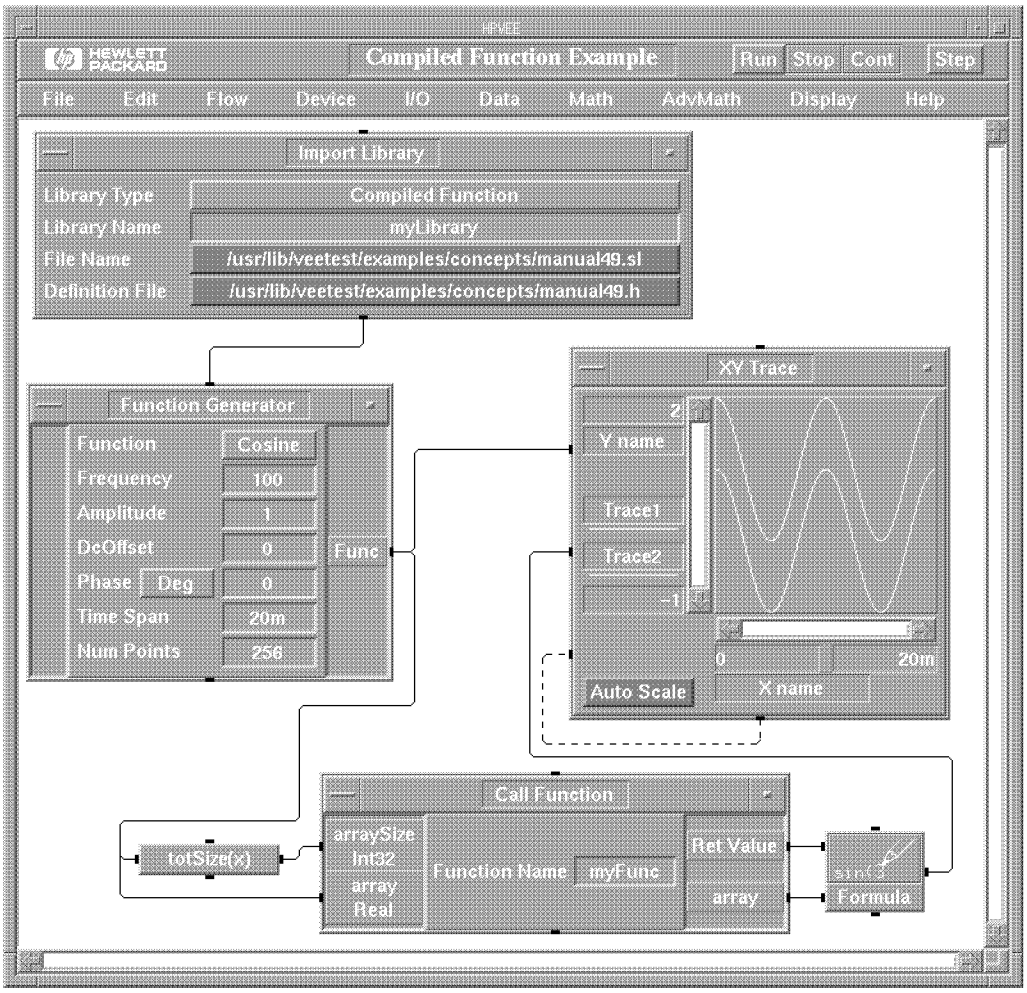


Figure 11-13. Model Calling a Compiled Function

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

The example in Figure 11-13 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual49.ex

or

/usr/lib/veetest/examples/concepts/manual49.ex
```

Also, the C file is saved as `manual49.c`, the definition file as `manual49.h`, and the shared library as `manual49.sl`.

Creating a Shared Library

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

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

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

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

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

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

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

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

Binding the Shared Library

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

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

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

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

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

Remote Functions

The third type of user-defined function is the Remote Function. A Remote Function is actually a User Function that runs in another HP VEE process on a remote host computer. The remote function is called from the local HP VEE process over the LAN (Local Area Network). Just as for User Functions and Compiled Functions, you can import a library of Remote Functions with the `Import Library` object. Once one or more Remote Functions have been imported, they can be called by either using the `Call Function` object, or by including function names in expressions. A library of Remote Functions can be deleted with the `Delete Library` object, again just as for User Functions or Compiled Functions. Thus, you can include Remote Function calls in your model just as you would User Functions. However, there are some differences, and some networking technicalities, which are described in this section.

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

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

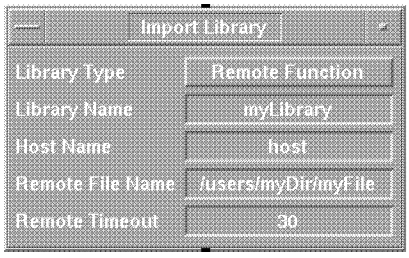


Figure 11-14. Import Library for Remote Functions

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

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

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

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

Note

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

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

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

UNIX Security, UIDs, and Names.

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

The basic requirement is that, in order to invoke the service HP VEE process, you must have a user name on the remote host which is the same as your user name on the computer running the client HP VEE process. (However,

the passwords need not be the same.) Also, you must have a directory in the `/users` directory. In addition, in order to establish network communication between the two hosts, either the remote host must have an `/etc/hosts.equiv` file with an entry for the client host, *or* the user must have an `.rhosts` file in the `$HOME` directory on the remote host, which contains an entry for the client host.

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

Client host: `myhost`

User: `mike`

Password: `twoheads`

and the service host can be identified as follows:

Service host: `remhost`

User: `mike`

Passwd: `arebetter`

Directory: `/users/mike`

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

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

  or

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

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

Note

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

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

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

The .veeio and .veerc files

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

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

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

Timeouts

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

Errors

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

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

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

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

Note

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

12

Using Transaction I/O

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

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

Using Transactions

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

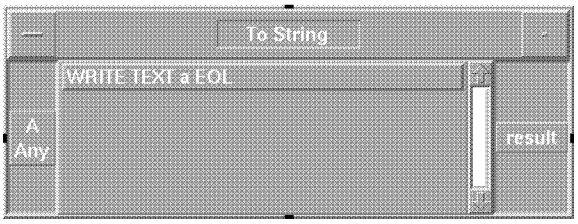


Figure 12-1. Default Transaction in `To String`

The default transaction in `To String` is:

```
WRITE TEXT a EOL
```

Before exploring too many details, consider a simple model using the `To String` object to illustrate how transactions operate. The model in Figure 12-2 uses two transactions, one to write a string literal and one to write a number in fixed decimal format.

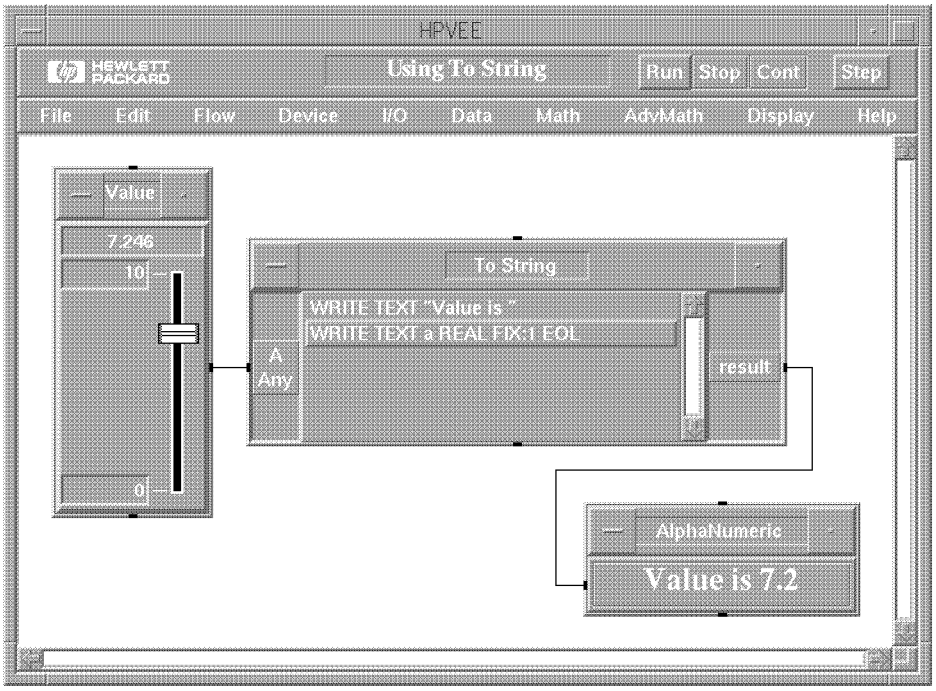


Figure 12-2. A Simple Model Using `To String`

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

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

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

Creating and Editing Transactions

Table 12-1. Editing Transactions With A Mouse

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

Table 12-2. Editing Transactions With the Keyboard

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

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

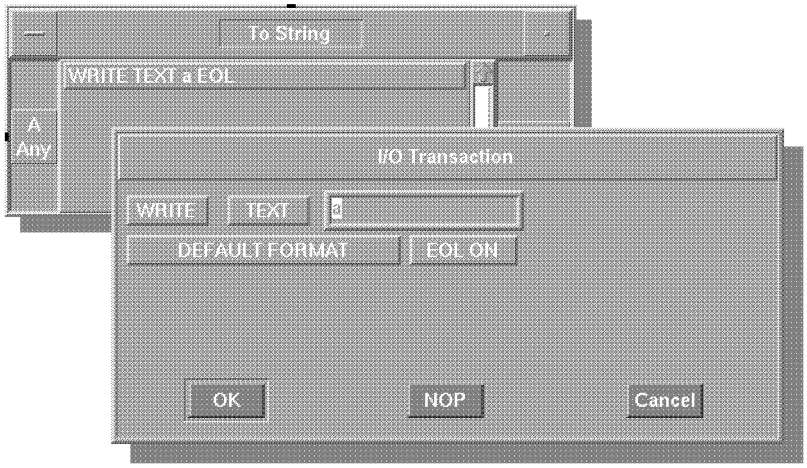


Figure 12-3. Editing the Default Transaction in `To String`

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

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

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

Editing the Data Field

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

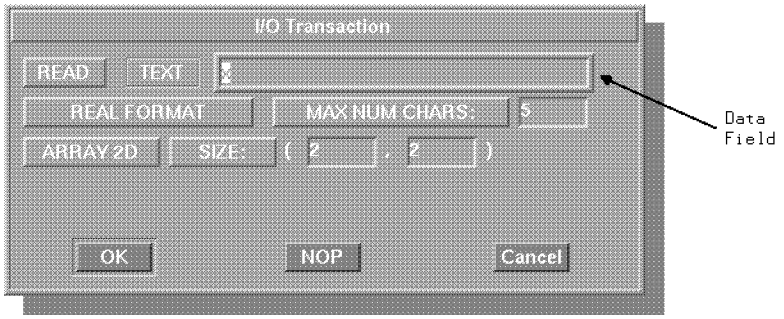


Figure 12-4. The Data Field

You must type in the proper list of what you wish to read or write. Table 12-3 lists typical entries for the data field. Note that `WRITE` transactions allow you to specify an expression list (variables, constants, and operators), but `READ` allows only a variable list.

Table 12-3. Typical Data Field Entries

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

The expressions allowed in a `WRITE` data field are the same as those allowed in `Formula` objects. Note that you may include the escape characters shown in Table 12-4 in any field that accepts Text input in the form of a string delimited by double quotes.

Note

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

Table 12-4. Escape Characters

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

Adding Terminals

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

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

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

The variable names that appear on the terminal must match the variable names in the transaction specification to achieve useful results. This is easy to

overlook, because HP VEE automatically assigns variable names such as `X`,`Y`, or `Z` when you add a terminal.

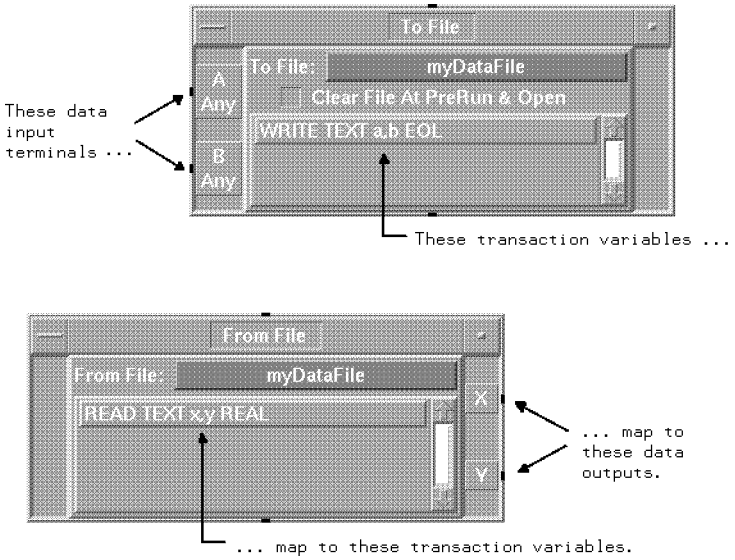


Figure 12-5. Terminals Correspond to Variables

To edit the variable name of a terminal:

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

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

Reading Data

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

Transactions that Read a Specified Number of Data Elements

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

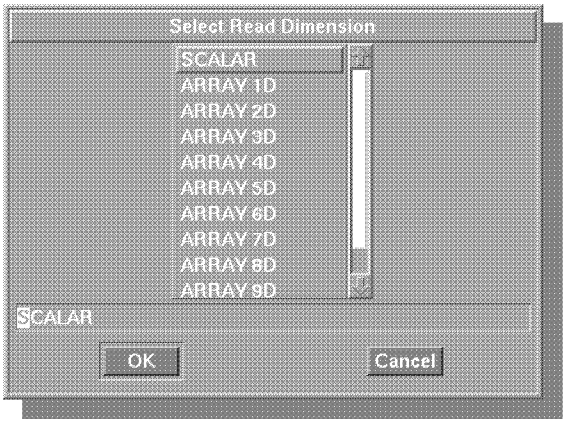


Figure 12-6. Select Read Dimension Dialog Box

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

When you click on the `OK` button in the `Select Read Dimension` dialog box, the transaction dialog box will reconfigure itself with a fill-in field for each of the dimensions specified. Figure 12-7 shows the transaction dialog box configured to read a three-dimensional array of binary integers into the variable named `matrix`. Each of the three fields after `SIZE:` contains the number of integers for the corresponding dimension. (In this case, each dimension has two elements.)

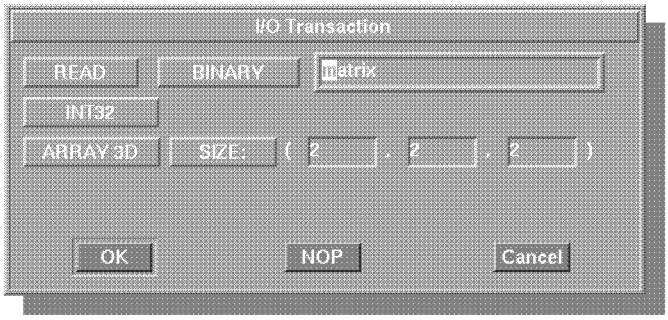


Figure 12-7. Transaction Dialog Box for Multi-Dimensional Read

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

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

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

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

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

```
READ BINARY x INT32
```

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

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

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

Read-To-End Transactions

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

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

- `From File`
- `From String`
- `From Stdin`
- `To/From Named Pipe`
- `To/From HP BASIC/UX` (HP VEE-Test only)
- `Execute Program`

Figure 12-8 shows the transaction dialog box of a `From File` object, reading a three dimensional array of binary integers, but configured for read-to-end:

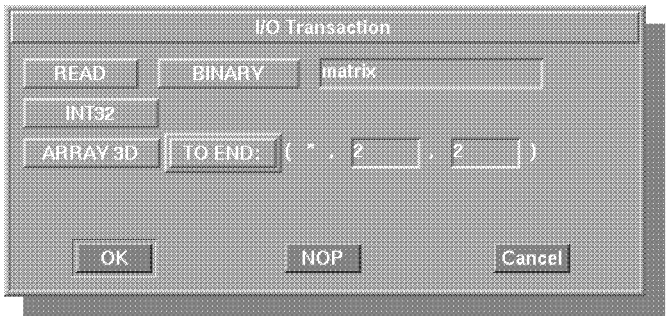


Figure 12-8. Transaction Dialog Box for Multi-Dimensional Read-To-End

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

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

On the other hand, reading a multi-dimensional array to end is somewhat more complicated. In this case the number of elements must be supplied for each dimension, except the left-most or "outer" dimension. Figure 12-8 shows that this dimension has an (`*`) in place of a size in the transaction. This dimension size is unknown until the read-to-end is transaction complete.

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

array has n dimensions, the size of n-1 of those dimensions must be specified. Only one (the left-most) dimension can be of unknown size.

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

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

Note

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

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

Non-blocking Reads

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

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

- To/From Named Pipe
- To/From HP BASIC/UX (HP VEE-Test only)
- From StdIn

Note

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

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

Figure 12-9 shows a model where READ IO STATUS DATA READY is used to detect data on the StdIn pipe.

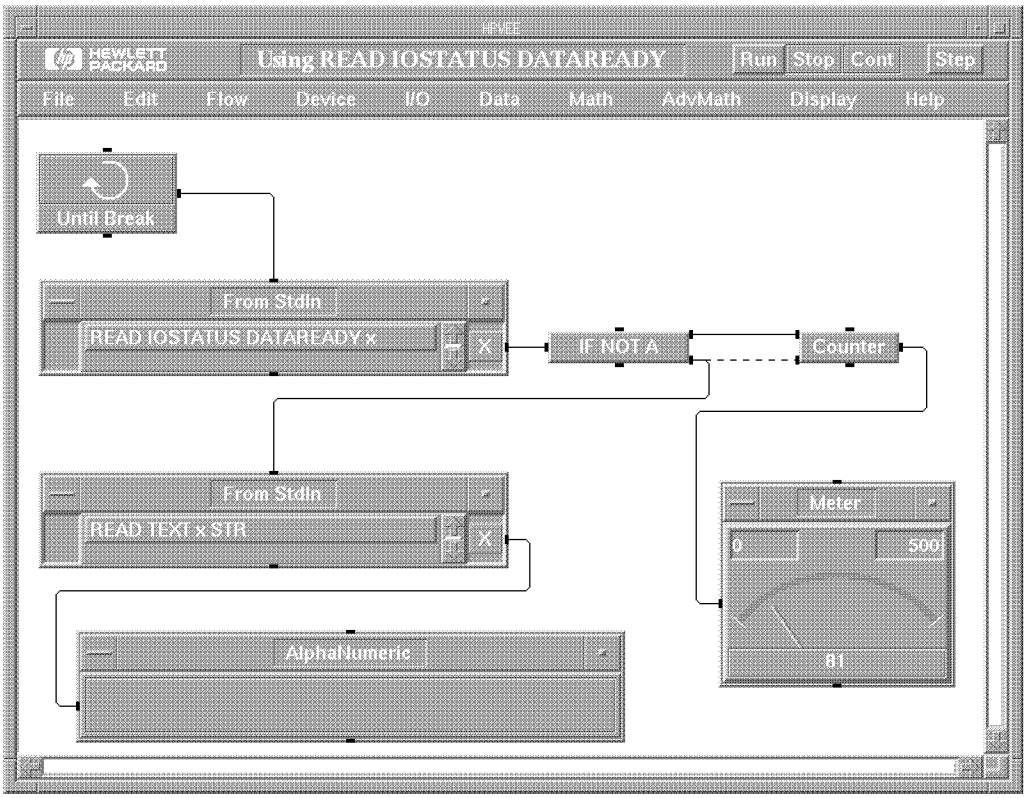


Figure 12-9. Using `READ IOSTATUS DATAREADY` for a Non-Blocking Read

This model is saved in:

```
/usr/lib/veeengine/examples/concepts/manual47.ex
-or-
/usr/lib/veetest/examples/concepts/manual47.ex
```

The model in Figure 12-9 shows the use of a `READ IOSTATUS DATAREADY` transaction in `From StdIn`. The transaction returns a zero (`0`) if no data is present on the **stdin** pipe. If data is present, a one (`1`) is returned. The `If/Then/Else` is used to test the returned value of the `READ IOSTATUS DATAREADY` transaction. If the result is `1`, then the second `From StdIn` is allowed to execute, reading the data typed into the HP VEE start-up terminal window. If no data has been typed into the start-up terminal window

(or a `Return` has not been typed), execution continues again at the start of the thread. Note the use of `Until Break` to iterate the thread so the `From StdIn` with the `READ IOSTATUS DATAREADY` transaction is continually tested.

To view complete models that illustrate how to read arrays from files, open and run these models:


```
/usr/lib/veeengine/examples/concepts/manual27.ex
/usr/lib/veeengine/examples/concepts/manual28.ex
-or-
/usr/lib/veetest/examples/concepts/manual27.ex
/usr/lib/veetest/examples/concepts/manual28.ex
```

Suggestions for Experimentation

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

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

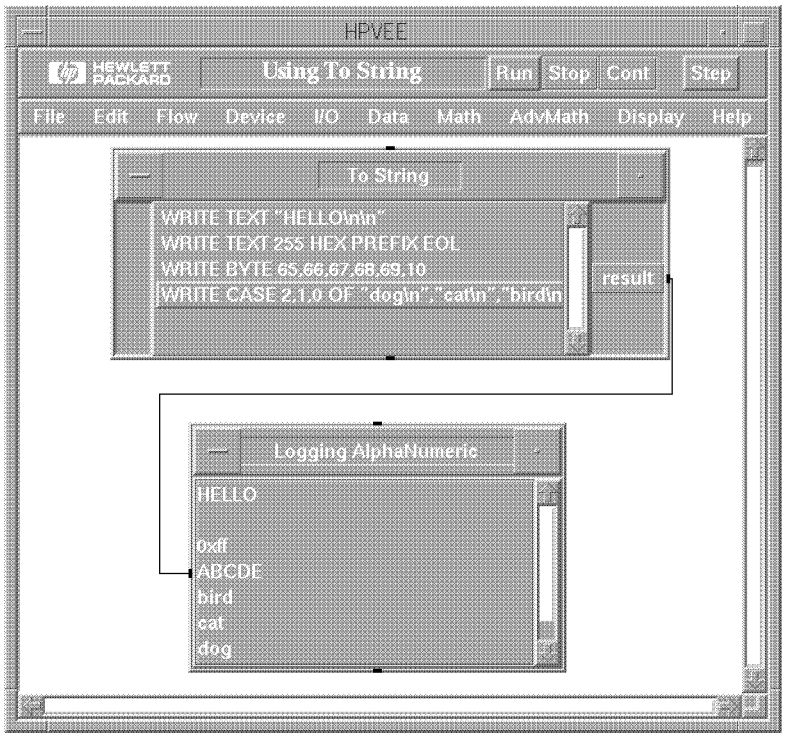


Figure 12-10. Experimenting with `To String`

Details About Transaction-Based Objects

Execution Rules

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

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

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

Object Configuration

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

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

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

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

`Direct I/O` objects behave differently. `Direct I/O` objects include a `Show Config` feature in their object menu that allows you to view (but not edit) configuration settings. To edit the configuration of a `Direct I/O` object, you must use `I/O` ⟹ `Configure I/O`. Please refer to "Details of Configure I/O Dialog Boxes" in Chapter 5 for details.

Clicking on `Config` in the object menu of a transaction I/O object yields a `Format Configuration` dialog box like the one in Figure 12-11.

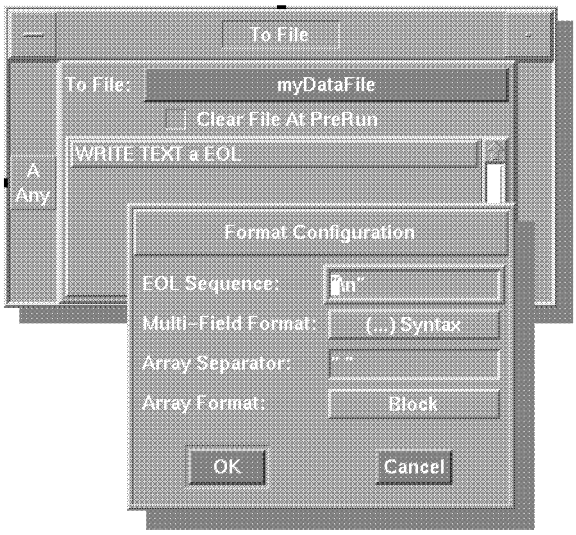


Figure 12-11. The `Format Configuration` Dialog Box

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

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

EOL Sequence

The `EOL Sequence` field specifies the characters that are sent at the end of `WRITE` transactions that use `EOL ON`. The entry in this field must be zero or more characters surrounded by double quotes. "Double quote" means ASCII 34 decimal. HP VEE recognizes any ASCII characters within `EOL Sequence` including the escape characters shown previously in Table 12-4.

Multi-field As

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

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

Array Separator

The `Array Separator` field specifies the character string used to separate elements of an array written by `WRITE TEXT` transactions. The entry in this field must be surrounded by double quotes. "Double quote" means ASCII 34 decimal. HP VEE recognizes any ASCII character as an `Array Separator` as well as the escape characters shown previously in Table 12-4.

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

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

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

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

Array Format

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

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


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

```
1 2 3
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.

READ and WRITE Compatibility

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

Choosing the Correct Transaction

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

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

Table 12-5. Summary of Transaction-Based Objects

Object	Description
`To File`	Writes data to a file.
`From File`	Reads data from a file.
`To String`	Writes text to an HP VEE container.
`From String`	Reads text from an HP VEE container.
`Execute Program`	Spawns an executable file; writes to standard input and reads from standard output of the spawned process.
`To Printer`	Writes text to the HP VEE text printer.
`To StdOut`	Writes data to HP VEE standard output.
`To StdError`	Writes data to HP VEE standard error.
`From StdIn`	Reads data from HP VEE standard input.
`Direct I/O`	Communicates directly with HP-IB, VXI, serial, or GPIO instruments.
`Interface Operations`	Transmits low-level bus commands and data bytes on an HP-IB or VXI interface.
`To/From Named Pipe`	Transmits data to and from named pipes to support interprocess communications.
`To/From HP BASIC/UX`	Transmits data to and from an HP BASIC/UX process via HP-UX named pipes.

Table 12-6. Summary of Transaction Types

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

Selecting the Correct Object and Transaction

1. Determine the source or destination of your I/O operation and the form in which data is to be transmitted.
2. Determine the type of object that supports the source or destination using Table 12-5.
3. Determine the correct type of transaction using Table 12-6.
4. To determine the remaining specifications for the transaction, such as encodings and formats, consult Appendix E.

Example of Selecting an Object and Transaction

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

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


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

1. The source is a text file. The data consists of a time stamp in 24-hour hours-minutes-seconds notation and signed real numbers in scientific and decimal notation.
2. Consulting Table 12-5, note that the object used to read a file is `From File`.
3. Consulting Table 12-6, note that the type of transaction used to read data from a file is `READ`.
4. The desired transactions are:

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

Using To String and From String

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

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

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

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

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

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

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

Communicating with Files

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

Details About File Pointers

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

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

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

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

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

Read Pointers

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

Write Pointers

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

Closing Files

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

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

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

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

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

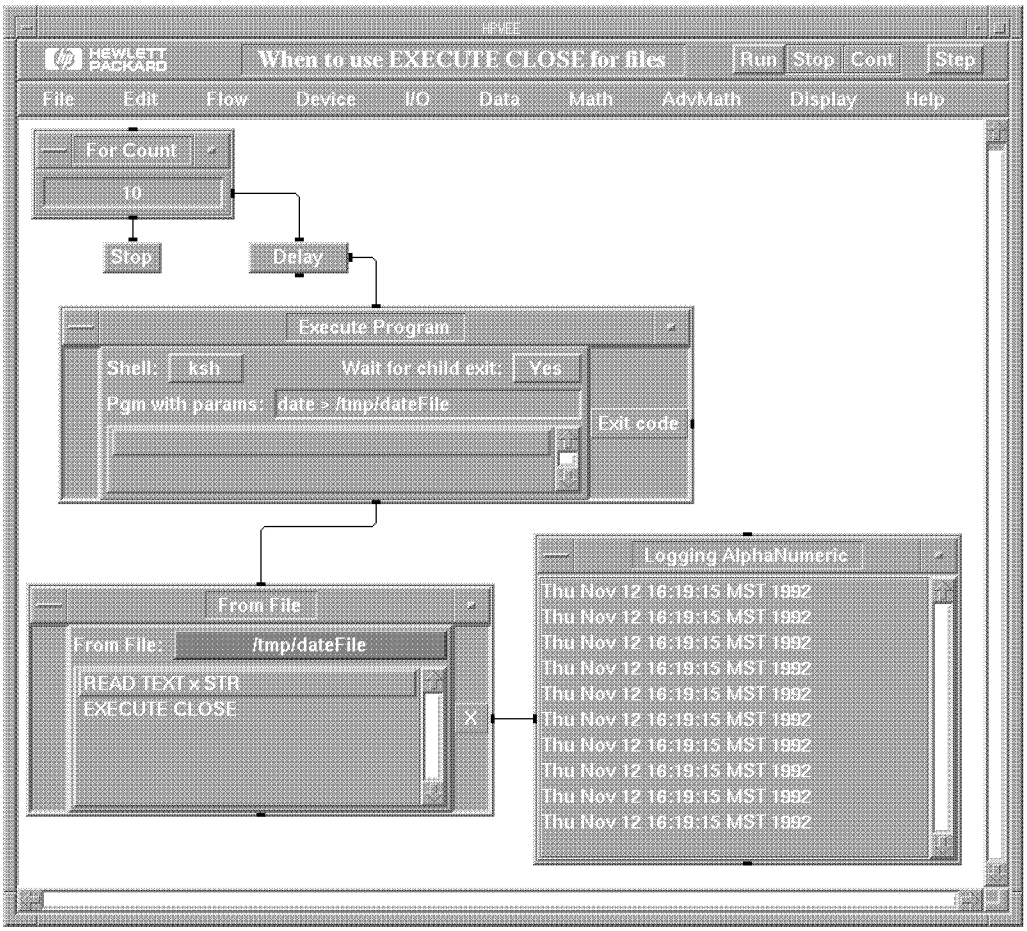


Figure 12-12. Using the `EXECUTE CLOSE` Transaction

This model is saved in:

```
/usr/lib/veeengine/examples/concepts/manual48.ex
-or-
/usr/lib/veetest/examples/concepts/manual48.ex
```

In Figure 12-12 **`Execute Program`** executes a shell command (**`date`**) that creates and writes the date and time to a file (**`/tmp/dateFile`**). Within the same thread, a **`From File`** reads the date from that file using a

READ TEXT x STR transaction. The EXECUTE CLOSE transaction is necessary because the subthread is executed multiple times by For Count. Succeeding executions of Execute Program will overwrite the file. However, since From File only opens the file once, upon the second execution of From File the read pointer will be *stale* — it will no longer point to the file since Execute Program has *re-created* the file. An error will occur.

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

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

The EOF Data Output

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

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

Figure 12-13 illustrates a typical use of `EOF`. The file being read contains a list of `X-Y` data of unknown length. Here are typical contents of the file:

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

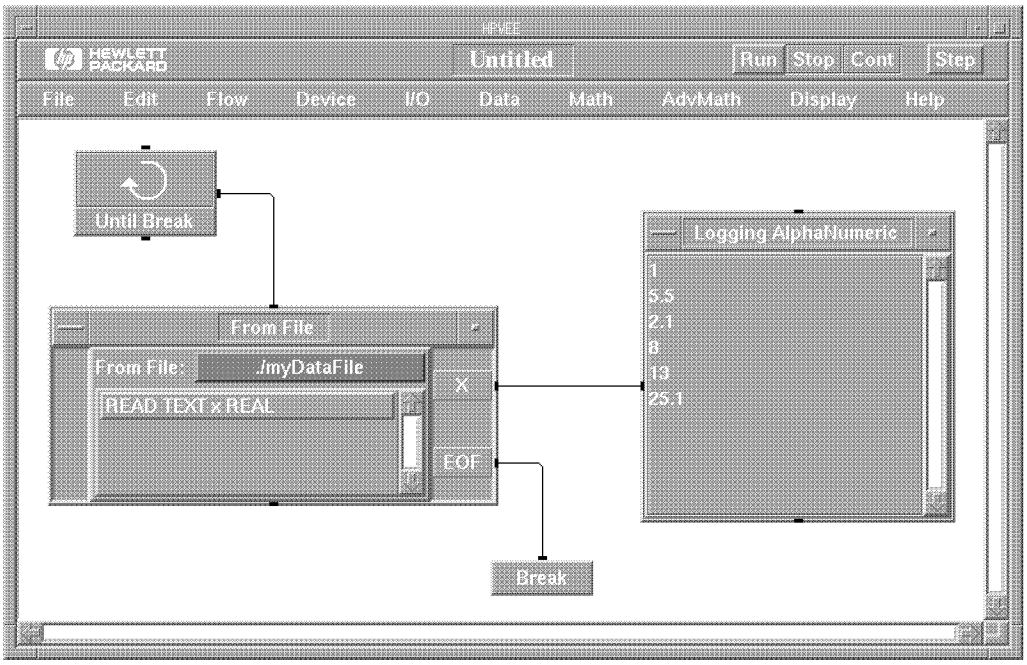



Figure 12-13. Typical Use of EOF to Read a File

Common Tasks for Importing Data

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

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

Importing X-Y Values

One very common problem is reading a text file containing an unknown number of `X` and `Y` values and plotting them. The model shown in Figure 12-14 solves this problem.

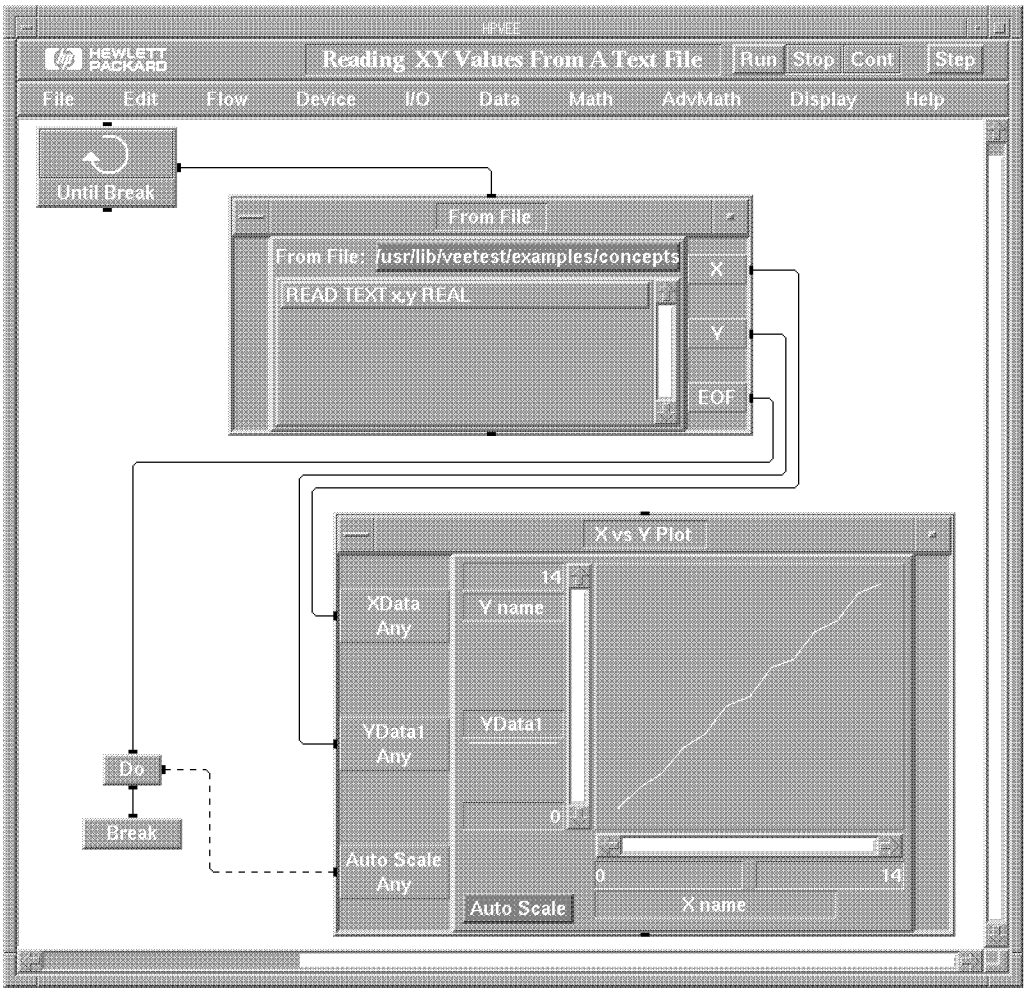


Figure 12-14. Importing `XY` Values

The model shown in Figure 12-14 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual29.ex
-or-
/usr/lib/veetest/examples/concepts/manual29.ex
```

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


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

Importing Waveforms

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

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

Fixed-Format Header. Here is a portion of the data file read by the model in Figure 12-15:

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

            .243545
            .2345776
.
.
.
```

Since this is a fixed-format header, labels such as `NAME` and `SAMPLES` are irrelevant. The waveform attributes *always appear and are in the same position*. Figure 12-15 shows a model that reads the waveform data file.

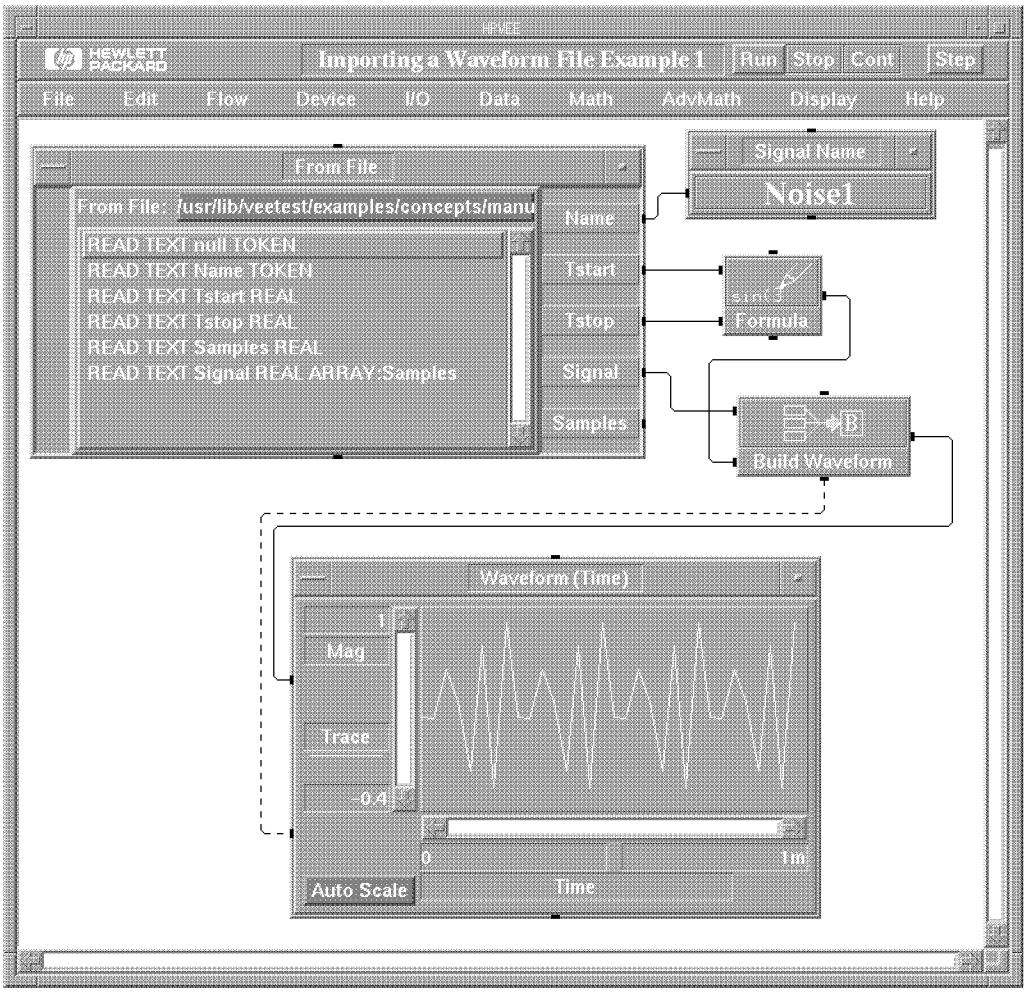


Figure 12-15. Importing a Waveform File

The model shown in Figure 12-15 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual30.ex
-or-
/usr/lib/veetest/examples/concepts/manual30.ex
```

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

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

Variable-Format Header. Here is a portion of the data file read by the model in Figure 12-16:

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

<DATA>

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

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

To simplify this example, the model in Figure 12-16 finds only the data associated with <DATA>. In your own applications, you might need to search for several markers.

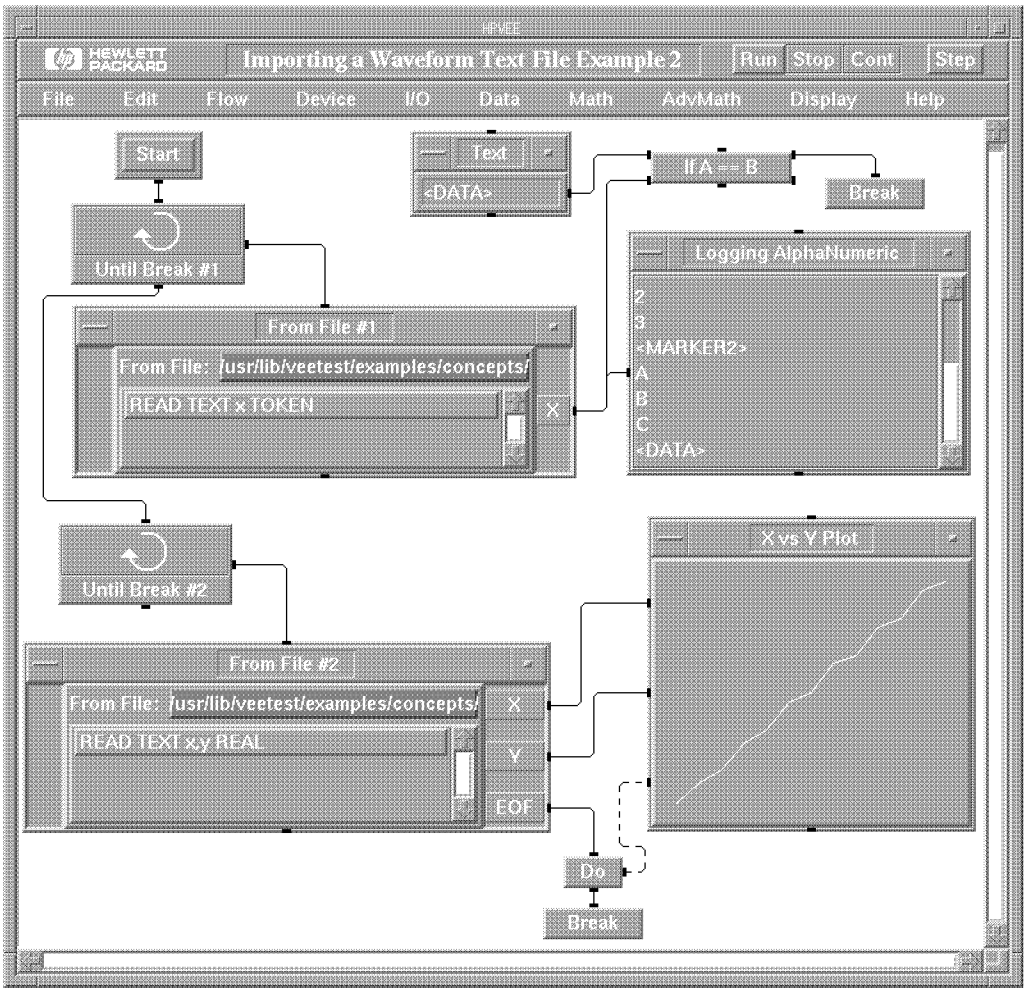


Figure 12-16. Importing a Waveform File

The model shown in Figure 12-16 is saved in:

```
/usr/lib/veeengine/examples/concepts/manual31.ex
-or-
/usr/lib/veetest/examples/concepts/manual31.ex
```

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

Communicating with Programs

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

Execute Program

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

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

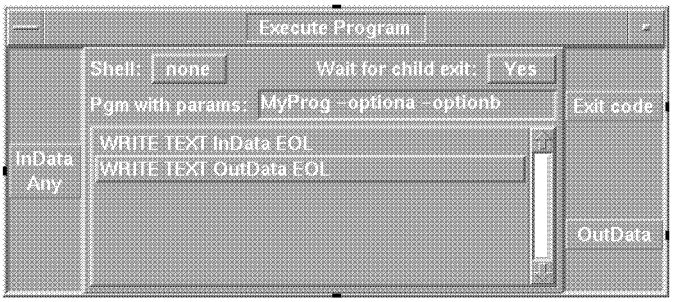


Figure 12-17. The `Execute Program` Object

Execute Program Fields

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
MyProg -optionA -optionB
```

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

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

Running a Shell Command

`Execute Program` can be used to run shell commands such as `ls`, `mkdir`, and `rm`. Figure 12-18 shows one method for obtaining a list of files in a directory using an HP VEE model.

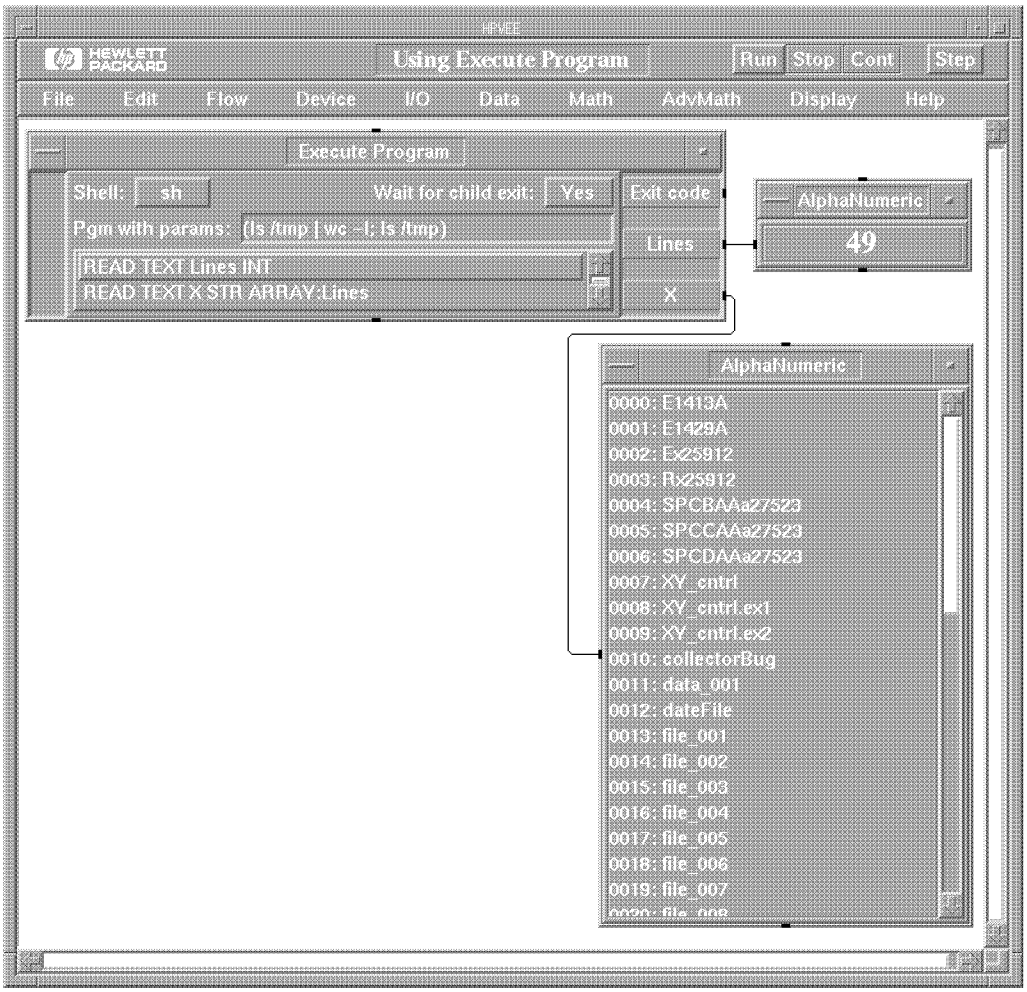


Figure 12-18. `Execute Program` **Running a Shell Command**

The model shown in Figure 12-18 is saved in this file:

```
/usr/lib/veeengine/examples/escapes/manual32.ex
-or
/usr/lib/veetest/examples/escapes/manual32.ex
```

In Figure 12-18, the `Execute Program` determines the number of file names in the `/tmp` directory by listing the names in a single column (`ls -1`) and piping this list to a line counting program (`wc -l`). Because the pipe is used, the command contained in the `Pgm with params` field must be sent to a shell for interpretation. Thus, the `Shell` field is set to `sh`. The number of lines is read by the `READ TEXT` transaction and passed to the output terminal names `Lines`.

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

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

Figure 12-19 shows another method for obtaining a list of files in a directory using an HP VEE model.

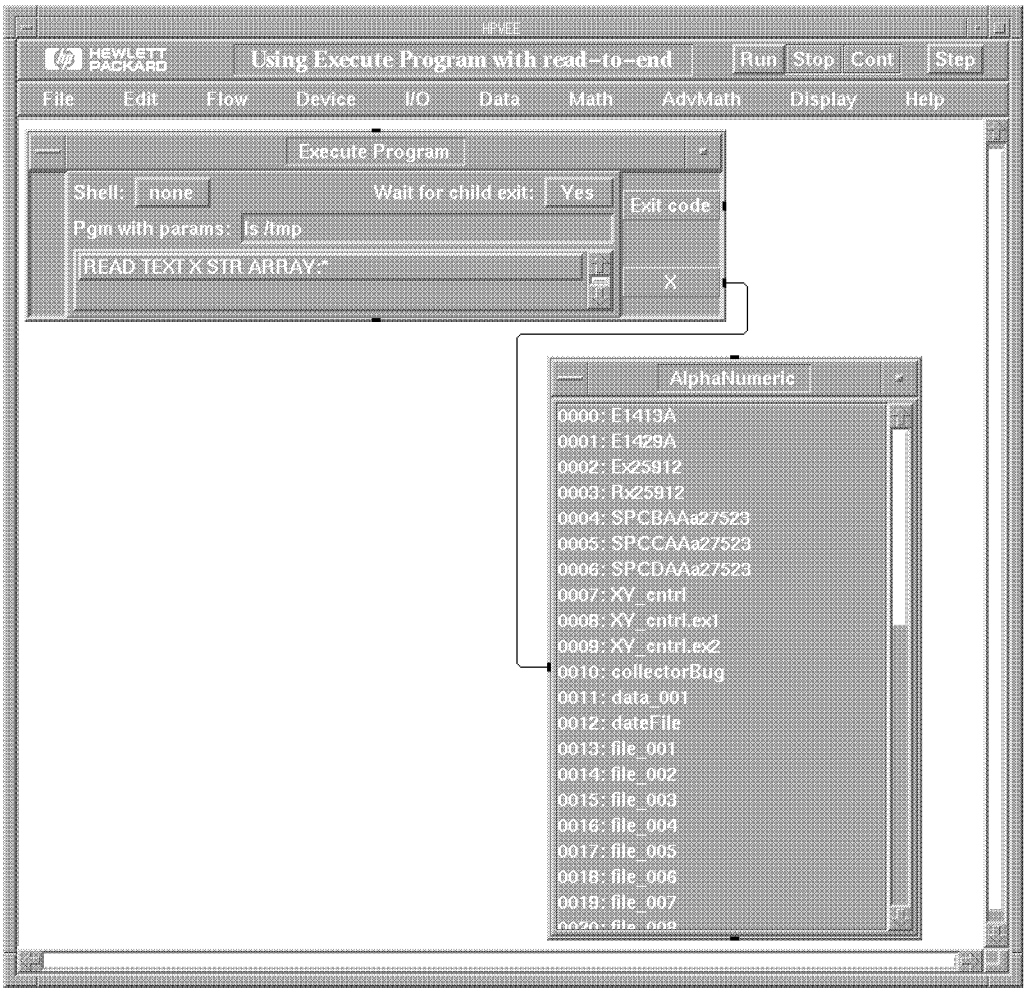


Figure 12-19. `Execute Program` **Running a Shell Command using Read-To-End**

This model is saved in:

```
/usr/lib/veeengine/examples/escapes/manual50.ex
-or
/usr/lib/veetest/examples/escapes/manual50.ex
```

In Figure 12-19 the HP VEE model displays the contents of the `/tmp` directory in a simpler fashion than in Figure 12-18.

In Figure 12-19, `Execute Program` has in the `Pgm with params` field the single shell command `ls /tmp`. There is no need to first obtain the number of files in the directory, as was done in the model in Figure 12-18, because the transaction `READ TEXT x STR ARRAY:*` uses the read-to-end feature discussed in "Reading Data". The shell command, when it is done executing, will close the pipe that `Execute Program` is using to read the list of files. This sends an end-of-file (`EOF`) which terminates the the transaction.

Running a C Program

The model shown in Figure 12-20 illustrates one way to share data with a C program using `stdin` and `stdout` of the C program. In this case, the C program simply reads a real number from HP VEE, adds one to the number, and returns the incremented value.

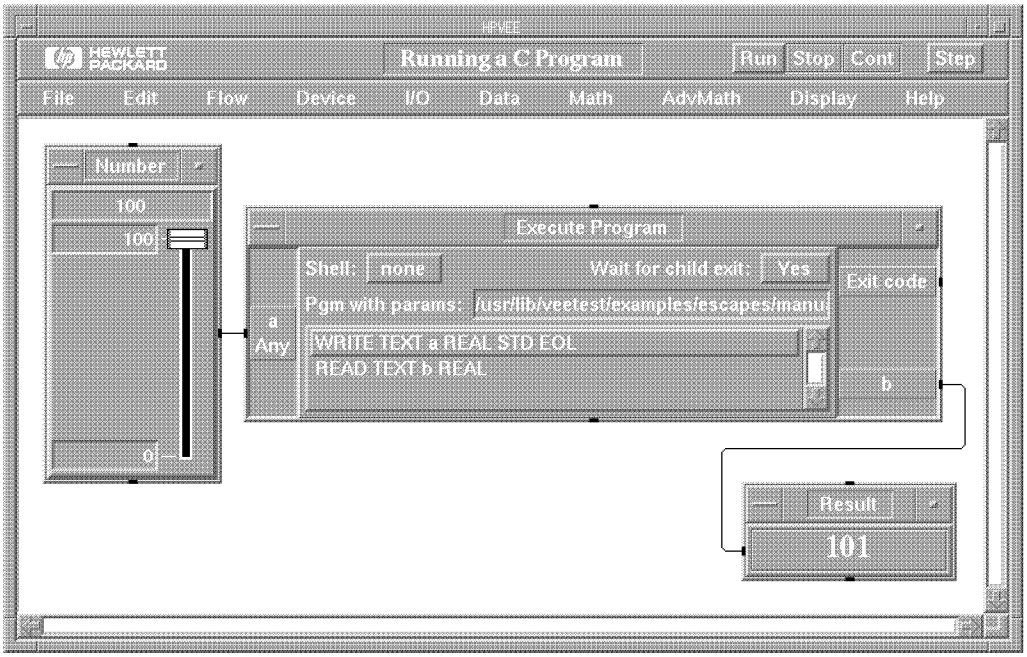


Figure 12-20. `Execute Program` **Running a C Program**

The model shown in Figure 12-20 is saved in this file:

```
/usr/lib/veeengine/examples/escapes/manual33.ex
-or-
/usr/lib/veetest/examples/escapes/manual33.ex
```

Figure 12-21 contains a listing of the C program called by the HP VEE model in Figure 12-20.

The program listing in Figure 12-21 uses both `setbuf` and `fflush` to force data through `stdout` of the C program; in practice, either `setbuf` or `fflush` is sufficient. Using `setbuf(`*file*`,NULL)` turns off buffering for all output to *file*. Using `fflush(`*file*`)` flushes any already buffered data to *file*.


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

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

Figure 12-21. C Program Listing

To/From Named Pipe

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

All `To/From Named Pipe` objects contain the same default names for read and write pipes. Be certain that you correctly specify the names of the pipes you want to read or write. This can be a problem if you run HP VEE on a diskless workstation. You must be sure that the named pipes in your model are not being accessed by another user.

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

Hints for Using Named Pipes

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

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

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

or

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

or

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

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

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

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

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

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

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

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

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

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

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

Init HP BASIC/UX

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

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

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

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

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

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

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

To/From HP BASIC/UX

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

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

Examples Using To/From HP BASIC/UX

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

1. Start HP BASIC.
2. Run a specific HP BASIC program.
3. Send a single number to HP BASIC for analysis.
4. Retrieve the analyzed data.
5. Terminate HP BASIC.

Here are typical To/From HP BASIC/UX settings and the corresponding HP BASIC/UX program:

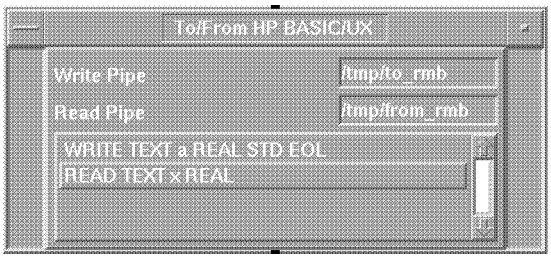


Figure 12-22. To/From HP BASIC/UX **Settings**

Here is the HP BASIC/UX program:

```
100  ASSIGN @From_vee TO "/tmp/to_rmb"
110  ASSIGN @To_vee TO "/tmp/from_rmb"
120  ! Your analysis code here
130  ENTER  @From_vee;Vee_data
140  OUTPUT @To_vee;Rmb_data
150  END
```

To view an example model that solves this problem, open this model:

```
/usr/lib/veetest/examples/rmb/manual34.ex
```

Sharing Array Data. To share array data between HP VEE-Test and HP BASIC using TEXT encoding, you must modify the default Array Separator in To/From HP BASIC/UX. To do this, click on Config in the To/From HP BASIC/UX object menu and set the Array Separator field to ", " (a comma followed by a blank).

Be sure that HP VEE-Test and HP BASIC use the same size arrays.

Note that the order in which HP VEE-Test and HP BASIC read and write array elements is compatible. If HP VEE-Test and HP BASIC share an array using READ and WRITE transactions in To/From HP BASIC/UX, each element will have the same value in HP VEE-Test as in HP BASIC.

To view an example model that shares arrays between HP VEE-Test and HP BASIC, open this model:

```
/usr/lib/veetest/examples/rmb/manual35.ex
```

Sharing Binary Data. It is possible to share numeric data between HP VEE-Test and HP BASIC without converting the numbers to text. To do this, you must select `BINARY` encoding in the `To/From HP BASIC/UX` transactions and `FORMAT OFF` for the `ASSIGN` statements that reference the named pipes in HP BASIC.

There are only two cases where it is possible to share numeric data in binary form:

- HP VEE-Test `BINARY REAL` is equivalent to HP BASIC `REAL`
- HP VEE-Test `BINARY INT16` is equivalent to HP BASIC `INTEGER`

Communicating with Instruments

Task	Object
Transmit data via HP-IB, VXI, serial, and GPIO interfaces.	`Direct I/O`
Send low-level HP-IB or VXI messages, commands, and data.	`Interface Operations`

Note

You must properly configure HP VEE-Test to communicate with instruments before you can use `Direct I/O` objects. Please read "Configuring Instruments" in Chapter 5 if you have not already done so.

Direct I/O

`Direct I/O` allows you control an instrument directly using the instrument's built-in commands. You do not need an HP VEE-Test driver file to use `Direct I/O` to control an instrument.

Sending Commands

The most important `WRITE` transactions for `Direct I/O` use with HP-IB, message-based VXI, and serial instruments are:

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

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

`Direct I/O` to register-based and some message-based VXI instruments use `WRITE REGISTER` and `WRITE MEMORY` transactions. These transactions are the *only* method of communicating with register-based VXI instruments. Refer to Appendix E for further information about these transactions.

WRITE TEXT Transactions. Most HP-IB, message-based VXI, and serial instruments use human-readable strings for programming commands. Such commands are easily sent to instruments using `WRITE TEXT` transactions. For example, all instruments conforming to IEEE 488.2 recognize `*RST` as a reset command. Here is the transaction used to reset such an instrument:

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


Note that instruments often define very precise "punctuation" in their syntax. They may demand that you send specific characters after each command or at the end of a group of commands. In addition, HP-IB instruments vary in their use of the signal line End-Or-Identify (EOI). If you suspect that you are having problems in this area, examine the `END (EOI) on EOL` and `EOL Sequence` fields in the `Direct I/O Configuration` dialog box for the object in question. If you do not know how to access the `Direct I/O Configuration` dialog box, refer to "Configuring Instruments" in Chapter 5.

Please consult your instrument programming manual to determine the proper command syntax for your instrument.

`WRITE TEXT` transactions are all that is needed to set up instruments for the majority of all situations where `Direct I/O` is required. `Direct I/O` allows you to use `WRITE` encodings other than `TEXT` when it is required by the instrument. The encodings other than `TEXT` that are most often useful are `BINBLOCK` and `STATE`.

WRITE BINBLOCK Transactions. `BINBLOCK` encoding writes data to instruments using IEEE-defined block formats. These block formats are typically used to transfer large amounts of related data, such as trace data from oscilloscopes and spectrum analyzers. The block formats supported by HP VEE-Test are discussed in greater detail in Appendix E. Instruments usually require a significant number of commands before accepting `BINBLOCK` data; consult your instrument's programming manual for details.

To use `BINBLOCK` transactions, you *must* properly configure the `Conformance` field (and possibly `Binblock`) in the instrument's `Direct I/O Configuration`. Please refer to "Direct I/O Configuration" in Chapter 5 for more detailed information.

WRITE STATE Transactions. Some HP-IB and message-based VXI instruments support a learn string capability, which allows you to upload all of the instrument settings. Later, you can recall the measurement state of the instrument by downloading the learn string using a `WRITE STATE` transaction. Learn strings are particularly useful when you wish to download measurement states but an instrument driver is unavailable.

Note that `WRITE STATE` transactions are available for HP-IB and message-based VXI instruments only.

Here is the typical procedure for using learn strings:

1. Configure the instrument to the desired measurement state; typically this is done using the instrument front panel.
2. Click on `Upload` in the object menu of a `Direct I/O` object configured for the instrument. The instrument state is now associated with this particular instance of the `Direct I/O` object.
3. Add a `WRITE STATE` transaction to the `Direct I/O` object.

When it is used, `WRITE STATE` is generally the first transaction in a `Direct I/O` object. `WRITE STATE` writes the `Upload`ed learn string to the instrument, thus setting all instrument functions simultaneously. Subsequent `WRITE` transactions can modify the instrument setup in an incremental fashion.

The behavior of `Upload` and `WRITE STATE` for HP-IB and message-based VXI instruments is affected by the `Direct I/O Configuration` settings for `Conformance` and `State (Learn String)`. If `Conformance` is `IEEE 488.2`, HP VEE-Test will automatically handle learn strings using the IEEE 488.2 `*LRN?` definition. If `Conformance` is `IEEE 488`, `Upload String` specifies the command used to query the state, and the `Download String` specifies the command that precedes the string when it is downloaded. Note that message-based VXI instruments are `IEEE 488.2` compliant.

Clicking on `Upload` in the `Direct I/O` object menu has these results:

- The learn string is uploaded *immediately*.
- The learn string remains with that particular `Direct I/O` object as long as it exists, until the next `Upload`. *The learn string is saved with the model.*
- If you clone a `Direct I/O` object, its associated learn string is included in the clone.

Learn String Example. Assume you wish to program the HP 54100A digitizing oscilloscope using learn strings. The oscilloscope's programming manual contains these important facts:

- The oscilloscope conforms to IEEE 488; it does not conform to IEEE 488.2.
- The command used to query the oscilloscope's learn string is `SETUP?`.
- The command that must precede a learn string that is downloaded to the instrument is `SETUP` . Note that a space must come between the `P` in `SETUP` and the first character in the downloaded learn string.

You must use I/O ⟹ Configure I/O to specify the proper direct I/O configuration for the oscilloscope. The settings important to learn strings are shown in Figure 12-23.

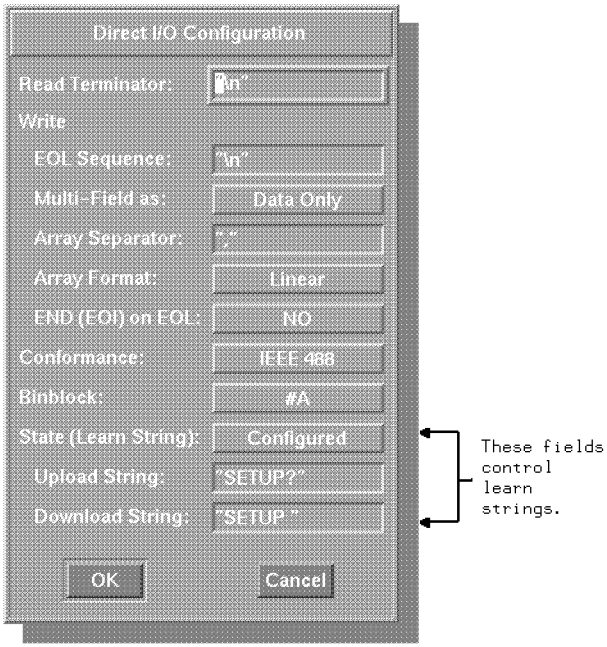


Figure 12-23. Configuring For Learn Strings

To upload a learn string from the oscilloscope, click on Upload in the object menu of a Direct I/O object that controls the oscilloscope. To download the learn string, use this transaction:

```
WRITE STATE
```

Reading Data

Instruments return data in a variety of formats. In general, you must know *what kind* of data and *how much* data you want HP VEE-Test to read from an instrument. The kind of data determines the encoding and format you must specify in the transaction. The amount of data being read determines the configuration you must use for the SCALAR or ARRAY fields in the transaction

dialog box. `Direct I/O` does not implement read-to-end reads as discussed in "Reading Data". (Note that the size configuration for arrays appears as `ARRAY` when the settings in the `I/O Transaction` dialog box are copied to the transaction list.)

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

- `READ TEXT`
- `READ BINBLOCK`

`Direct I/O` to GPIO instruments uses only `READ BINARY` and `READ IOSTATUS`.

`Direct I/O` to register-based and some message-based VXI instruments use `READ REGISTER` and `READ MEMORY` transactions. These transactions are the *only* method of communicating with register-based VXI instruments. Refer to Appendix E for further information about these transactions.

If you have difficulty reading data from instruments, try using the `Bus I/O Monitor` to examine that data to determine its format. You may wish to consult Appendix E to determine HP VEE-Test's rules for interpreting incoming data.

READ TEXT Transactions. Frequently, the data you read from an instrument as the result of a query is a single numeric value that is formatted as text. For example, a particular voltmeter returns each reading as a single number in exponential notation, such as `-1.234E+00`. Here is the transaction to read a value from the voltmeter:

```
READ TEXT a REAL
```

Some instruments respond to a query with alphabetic information combined with the numeric measurement data. In general, this not a problem; `READ TEXT REAL` transactions throw away preceding alphabetic characters and extracts the numeric value.

Interface Operations

`Interface Operations` allows you to control HP-IB and VXI interfaces and instruments using low-level commands. `Interface Operations` supports two actions that provide this low-level control:

- `EXECTUTE`
- `SEND`

`EXECUTE` commands are of the form:

`EXECUTE` *Command*

where *Command* is one of the bus commands summarized in Table 12-7. While the commands listed in Table 12-7 have the same names as the `EXECUTE` commands in `Direct I/O`, there is an important difference.

- `Direct I/O EXECUTE` commands address an instrument to receive the command.
- `Interface Operations EXECUTE` commands may affect multiple instruments. For HP-IB, these instruments must be addressed to listen.

Please refer to Appendix E for details about the exact bus actions corresponding to each `EXECUTE` command.

Table 12-7.
Summary of `EXECUTE` Commands (`Interface Operations`)

Command	Description
`CLEAR`	Clears all HP-IB devices by sending DCL (Device Clear). For VXI, resets the interface and runs the Resource Manager.
`TRIGGER`	For HP-IB, triggers all devices addressed to listen by sending GET (Group Execute Trigger). For VXI, triggers TTL, ECL, or external triggers.
`LOCAL`	For HP-IB, releases the REN (Remote Enable) line. There is no counterpart for VXI.
`REMOTE`	For HP-IB, asserts the REN (Remote Enable) line. There is no counterpart for VXI.
`LOCAL LOCKOUT`	For HP-IB, sends the LLO (Local Lockout) message. Any device in remote mode at the time LLO is sent will lock out front panel operation. There is no counterpart for VXI.
`ABORT`	Clears the HP-IB interface by asserting the IFC (Interface Clear) line. To clear and reset the VXI interface use `CLEAR`.

`SEND` transactions are of this form:

`SEND` *BusCmd*

BusCmd is one of the bus commands listed in Table 12-8. These messages are defined in detail in IEEE 488.1. *BusCmd* is HP-IB specific only. There are no counterparts for VXI.

Table 12-8. SEND Bus Commands

Command	Description
COMMAND	Sets ATN true and transmits the specified data bytes. ATN true indicates that the data represents a bus command.
DATA	Sets ATN false and transmits the specified data bytes. ATN false indicates that the data represents device dependent information.
TALK	Addresses a device at the specified primary bus address (0-31) to talk.
LISTEN	Addresses a device at the specified primary bus address (0-31) to listen.
SECONDARY	Specifies a secondary bus address following a TALK or LISTEN command. Secondary addresses are typically used by cardcage instruments where the cardcage is at a primary address and each plug-in module is at a secondary address.
UNLISTEN	Forces all devices to stop listening; sends UNL.
UNTALK	Forces all devices to stop talking; sends UNT.
MY LISTEN ADDR	Addresses the computer running HP VEE to listen; sends MLA.
MY TALK ADDR	Addresses the computer running HP VEE to talk; sends MTA.
MESSAGE	Sends a multi-line bus message. Consult IEEE 488.1 for details. The multi-line messages supported by HP VEE-Test are: DCL Device Clear SDC Selected Device Clear GET Group Execute Trigger GTL Go To Local LLO Local Lockout SPE Serial Poll Enable SPD Serial Poll Disable TCT Take Control

Related Reading

1. Haviland, Keith and Salama, Ben, *UNIX System Programming*. (Addison-Wesley Publishing Company, Menlo Park, California, 1987).

This book contains information of general interest to programmers using UNIX. In particular, it contains explanations of interprocess communications and pipes that are applicable to with `To/From Named Pipe`, `To/From HP BASIC/UX`, and `Execute Program`.

13

Using the Sequencer Object

If you are using HP VEE-Engine, please skip this chapter. HP VEE-Engine does not include the `Sequencer` object.

If you are using HP VEE-Test, you'll need to understand several topics covered earlier in this manual in order to use the `Sequencer` object effectively. These topics include instrument I/O operations (Chapter 5), UserObjects (Chapter 6), Records and DataSets (Chapter 10), and User Functions (Chapter 11). Also, for information on how to configure a transaction, refer to "Using Transactions" in Chapter 12.

You can use the `Sequencer` object, provided by HP VEE-Test under the `Device` menu, to control the order of calling of a series of tests. The `Sequencer` object executes a series of sequence transactions. Each of these transactions evaluates an HP VEE expression, which may contain calls to User Functions, Compiled Functions, Remote Functions, or other HP VEE functions. After evaluating the HP VEE expression, the transaction compares the value returned by that expression against a test specification. Depending on whether the test passes or fails, the transaction then evaluates different expressions and selects the next transaction to be executed. Transactions may optionally log their results to the `Log` output pin, or to a User Function, Compiled Function, or Remote Function specified in the `Logging Config` dialog box.

Sequence Transactions

The `Sequencer` object, in its open view, shows a list of sequence transactions. Each transaction is similar to the other types of transactions shown in Chapter 12. To see how the `Sequencer` uses transactions to execute expressions and call functions, let's look at a simple example.

In the following model there are two User Functions in the background: `myRand1`, which adds a random number from 0 to 1 to the value of its input, and `myRand2`, which adds a random number from 0 to 100 to its input. (Refer to Chapter 11 for further information on creating and using User Functions.)

13

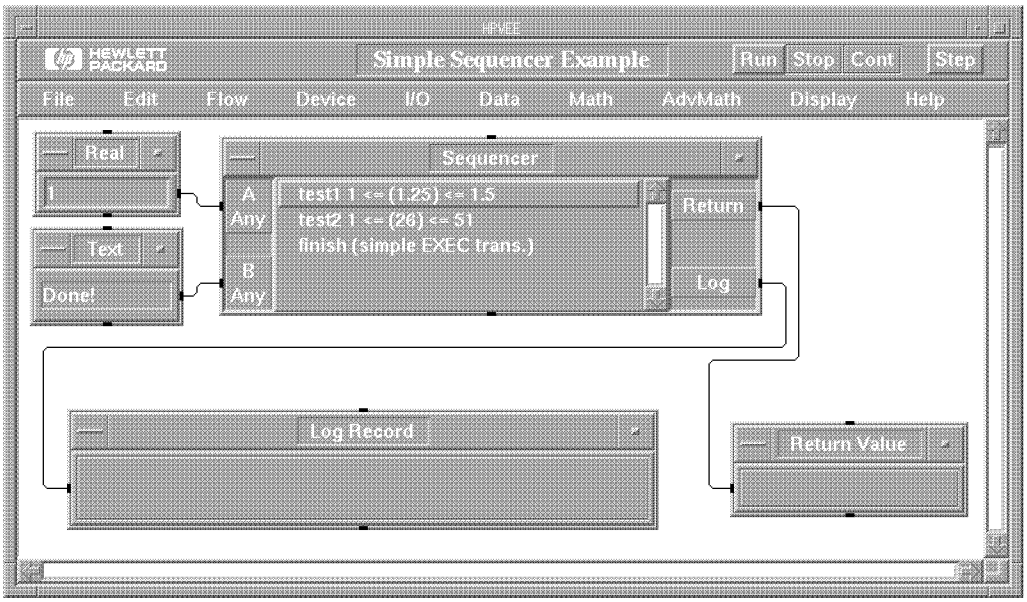
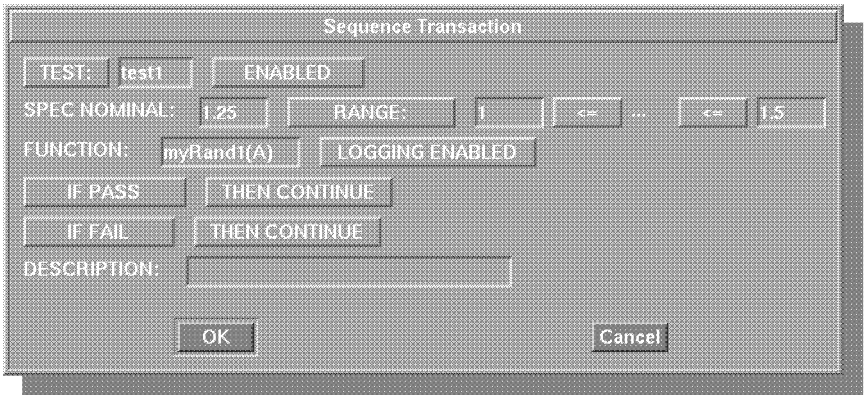


Figure 13-1. A Simple Sequencer Model

When you click on a transaction with the mouse, a dialog box "expands" the transaction so you can view and edit it. The following dialog box shows the first transaction, `test1`:



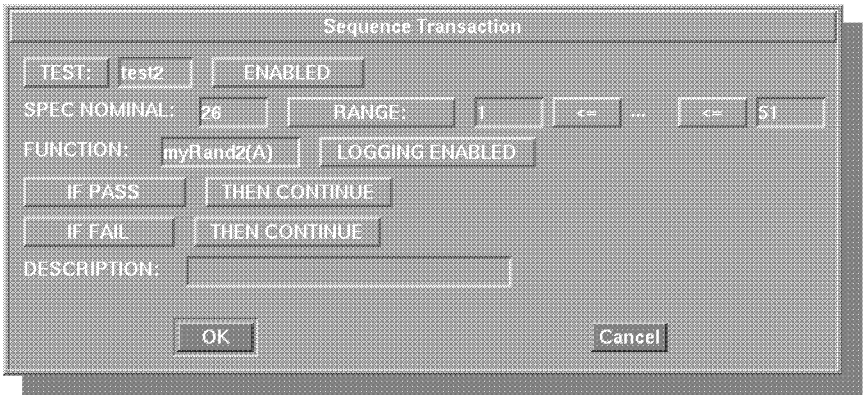
A sequence transaction can either be a TEST transaction or an EXEC transaction. In this transaction, the type is `TEST:`, the name field is `test1`, the nominal specification is `1.25`, a `RANGE:` specification is used, and the range is `1 <= ... <= 1.5`. Thus, only values from 1 to 1.5 will pass the test. The expression `myRand1(A)` calls the user function using the value on the `A` input terminal of the `Sequencer` as its input parameter. The transaction has logging enabled, so a local variable named `Test1` will be automatically created, which contains the log record of the results of this test. This log record will also be available as part of the `Log` output terminal. The `IF PASS` and `IF FAIL` conditions are both `THEN CONTINUE`. This means that, pass or fail, once `test1` is done, the next transaction, `test2`, will be executed.

The `DESCRIPTION` field is simply a comment area for this test.

Note

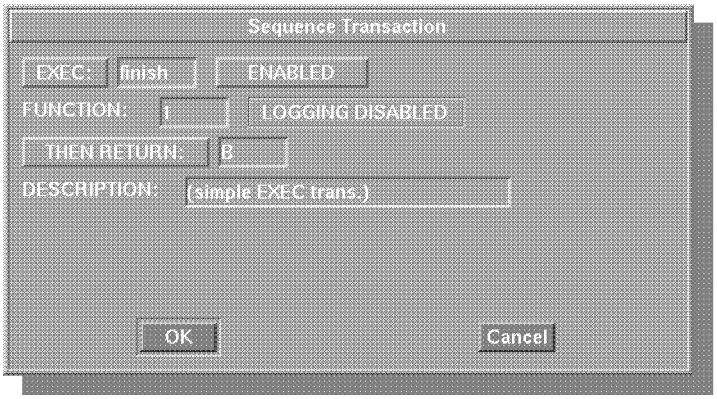
For `RANGE` or `LIMIT` tests, the `SPEC NOMINAL` value is not used, except for "documentation" purposes. However, if you use tests based on `TOLERANCE` or `%TOLERANCE` values, the tolerance will be calculated relative to the `SPEC NOMINAL` value.

The second transaction, `test2`, is also a TEST transaction:



This second test is similar to the first. The User Function `myRand2` is called with the expression `myRand2(A)` and the resulting value is tested to see if it is in the range 1 through 51, with a nominal specification of 26. Again, pass or fail, the `Sequencer` continues to the next transaction.

The third transaction is an EXEC transaction:



An EXEC transaction, unlike a TEST transaction, performs no comparison of the function result to a specification or range. EXEC transactions are used to perform an action that does not require a pass/fail test. For example, an EXEC transaction could call a routine that sets up an external configuration before a TEST transaction is performed, or it could execute a power down procedure after a series of tests. (An EXEC transaction is a short cut for specifying an "always pass" test condition.)

In our example, the transaction named `finish` returns the value of `B` to the `Return` output terminal of the `Sequencer` object. Since no test is performed, logging does not occur for an EXEC transaction.

Note that you can use the `DESCRIPTION` field to briefly describe any transaction.

When you run the model, the three transactions are executed in sequence:

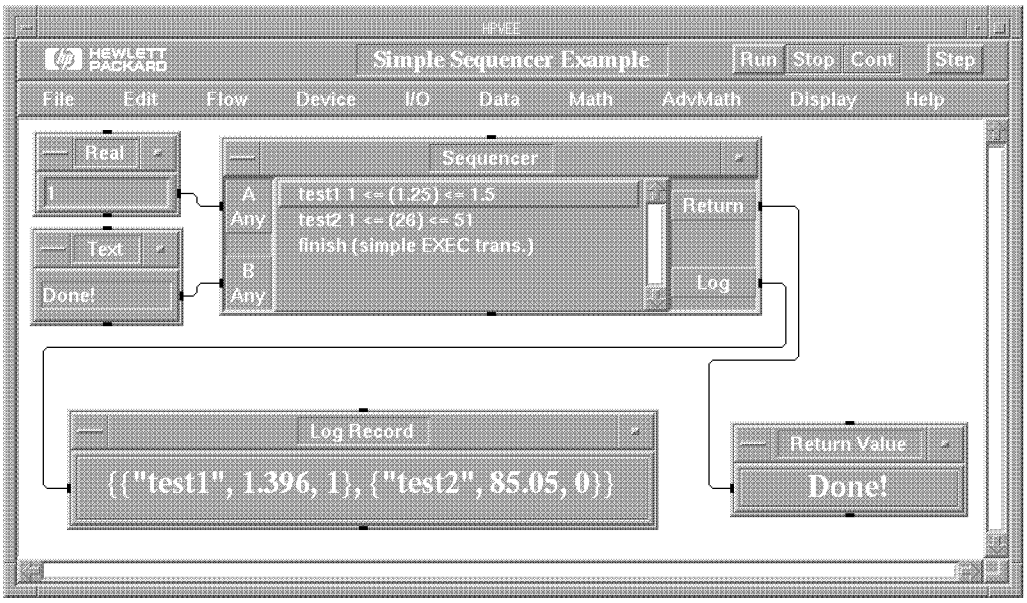


Figure 13-2. Running the Model

The logged test results are output on the `Return` output terminal and displayed. Note that the results are logged as the Record data type, in fact a

record of records. In this case, `test1` has passed with a value of `1.396` and `test2` has failed with a value of `85.05`. The third transaction returns the value on the `B` input, which is the string `Done!`.

Let's look more closely at how logging works. Each transaction that has logging enabled creates a log record and attaches it to the transaction name. In our example, logging is enabled for the first two tests, so local variables named `Test1` and `Test2` contain the log records for those transactions. The fields contained in the log records are defined by the `Logging Config` device menu selection. By default, log records contain `Name`, `Result`, and `Pass` fields.

The `Test1` and `Test2` local variable names can be used in any expression *within* the `Sequencer` to access the results of the current or a previously executed transaction. For example, `Test3` could have called a function with `Test1.Result` as a parameter to pass the result of the first test. Or `Test2.Pass` could be used as an expression, which would evaluate to `1` if `Test2` passed, or `0` if `Test2` failed.

There is one more local variable, `thisTest`, available to access the logging records. The value of `thisTest` is always the same as the logging record for the currently executing transaction. This allows you to write transaction expressions that can be used in many transactions without having to include the name of each transaction.

Now let's examine the data structure produced by the `Log` output terminal on the `Sequencer`, which is a record of records:

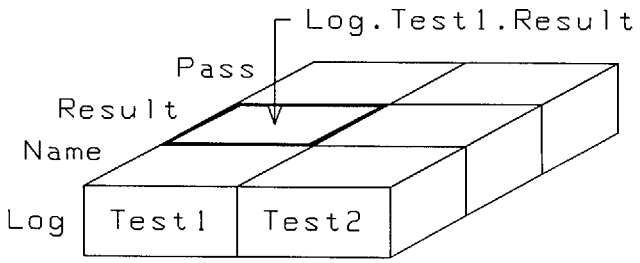


Figure 13-3. A Logged Record of Records

The record produced by the `Log` output pin contains a field for each transaction that has logging enabled — `Test1` and `Test2` in our example. Each of these fields is simply the log record for the specified transaction, containing the fields `Name`, `Result`, and `Pass`. This record of records is available on the `Log` output pin and can be used by other objects by using the record "dot" syntax. For example, the expression `Log.Test1.Result` would, in this case, return the value `1.396` (see Figure 13-2). Likewise, `Log.Test1.Name` would return `test1` and `Log.Test1.Pass` would return `1`.

Note that the data logged on the `Log` output pin is always the data from the *last* execution of each transaction. If you wish to log the results of *every* execution of each transaction, use the `Log Each Transaction To:` option in the `Logging Config` dialog box. This option will call the specified function (or expression) at the completion of every transaction. This option can also be useful if you wish to log test results to a file or printer *as they happen*, rather than waiting until the `Sequencer` has completed. The local variable `thisTest` can be used as a parameter to the logging function to pass the log record of the transaction that has just completed.

Logging Test Results

Now let's look at a more practical example of logging test results, where an iterator causes the `Sequencer` to repeat the tests over and over, and to log the results:

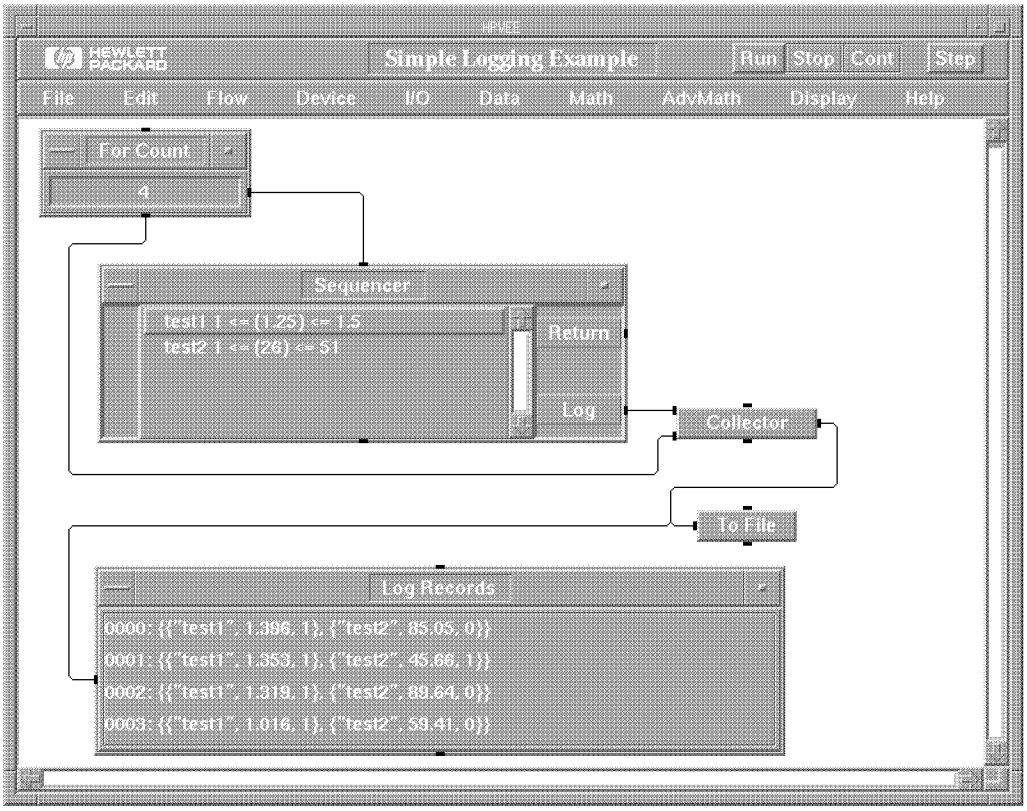


Figure 13-4. A Simple Logging Example

In this example, the `For Count` object causes the `Sequencer` to execute its series of tests (`test1` and `test2` of the previous example) four times. For example, if four "widgets" are being tested on an assembly line, each execution of the `Sequencer` tests one widget. The resulting series of records from the `Log` output terminal is collected by the `Collector` and displayed as an array of

records. Note, also, that you can use the `To File` object to output this array to a file using a `WRITE CONTAINER` I/O transaction.

Conceptually, the output of the `Collector` in this example can be viewed as an array of records of records, as shown below:

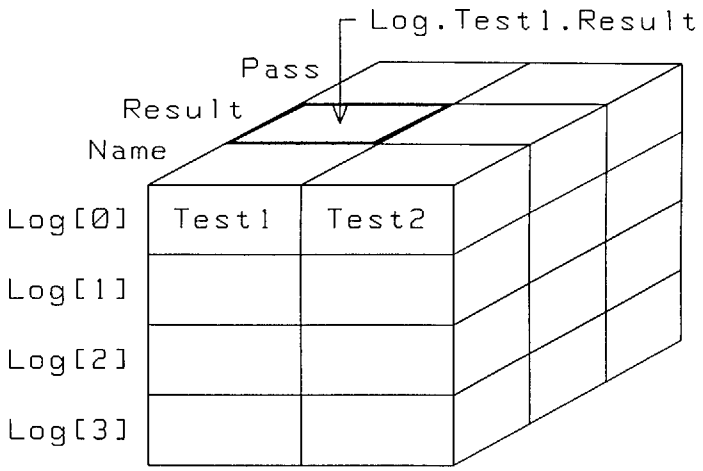


Figure 13-5. A Logged Array of Records of Records

Each array element (`Log[0]`, `Log[1]`, etc.) represents a single iteration of the sequencer, and is a record of records as shown in Figure 13-3. As mentioned before, the logged output is available for analysis in expressions. In this case, `Log.Test1.Result` is a "core sample" from the array. In fact, `Log.Test1.Result` would return an array of values (1.396, 1.353, 1.319, and 1.016 for the example results shown in Figure 13-4).

Note

The logged array is *not* a three-dimensional array, but is rather an array that consists of records of records. This is important because the individual fields of a record can be of differing data types. For example, while the `Name` field is Text, the `Result` field could be a Waveform, and so forth. Also, the `Test2.Result` field could be a Waveform, while the `Test1.Result` field is a Real value.

However, each individual field must be of a consistent data type throughout the array. For example, the field `Test1.Result` can't be a Real value for `Log[0]` and a Waveform for `Log[1]`.

Let's extend our example to 10 iterations of the `Sequencer`, and add some analysis of the logged data. In the following example, the expression `log.test1.result` in the `Formula` object returns a 10 element Real Array, which contains the results of `test1`. This array is then statistically analyzed by means of the `min(x)`, `max(x)`, `mean(x)`, and `sdev(x)` objects.

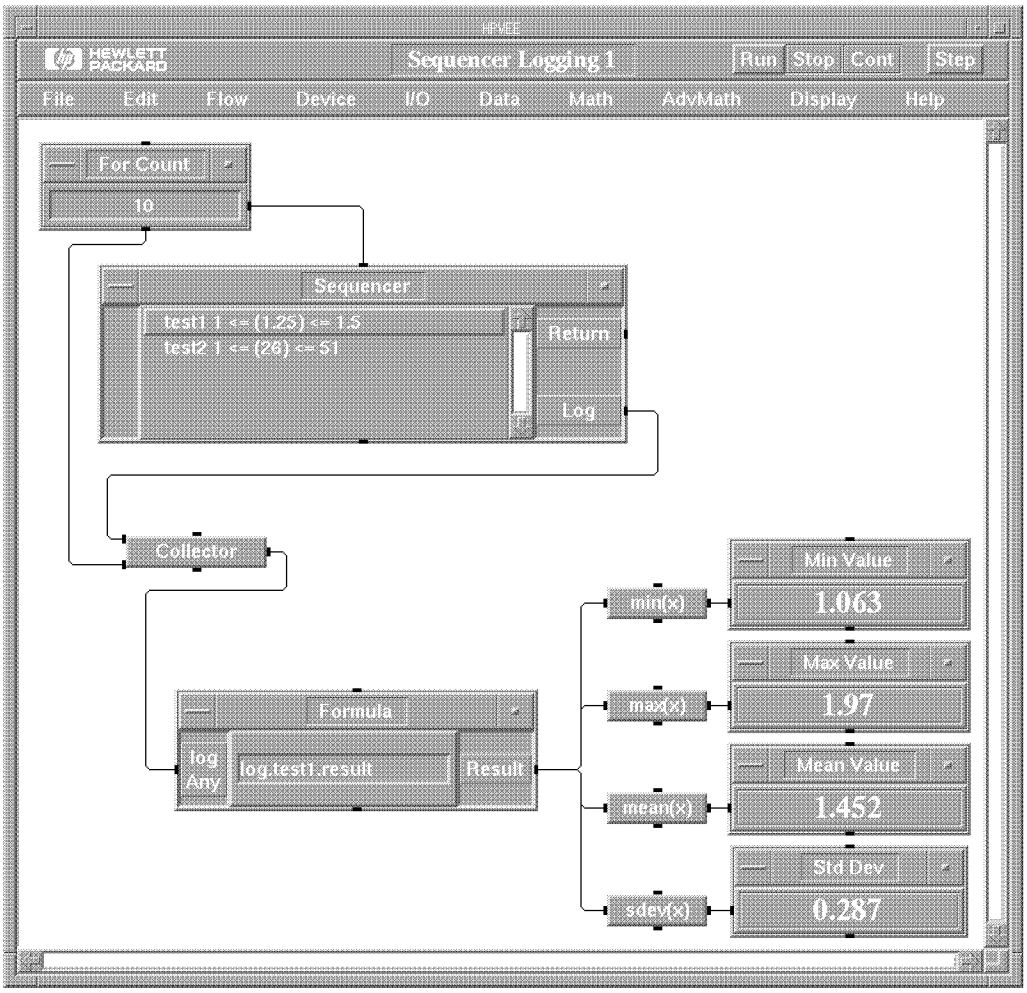


Figure 13-6. Analyzing the Logged Test Results

This example is saved in:

```
/usr/lib/veetest/examples/mfgtest/manual44.ex
```

Logging to a DataSet

You can use a DataSet to store your logged test results. In the following model, the `Sequencer` object `Log` output terminal is connected to the `To DataSet` object.

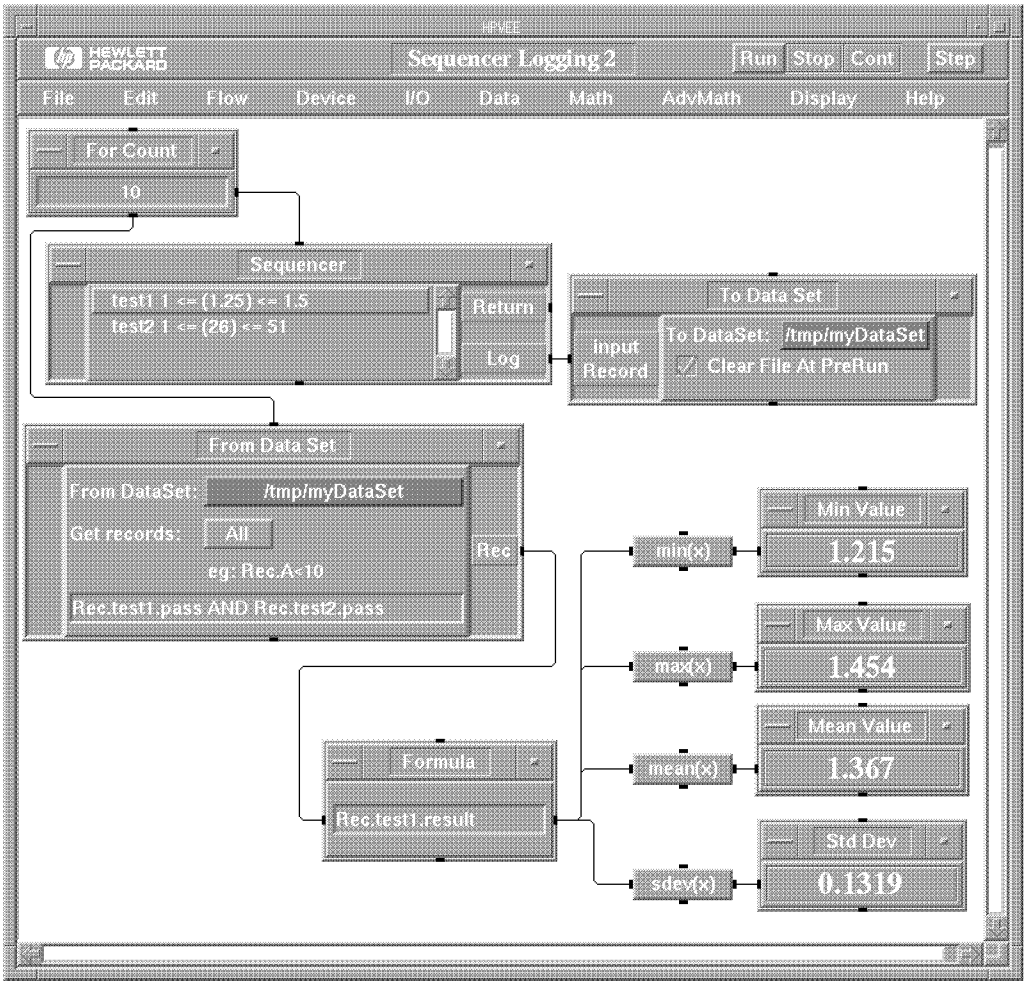


Figure 13-7. Logging to a DataSet

Once the For Count object is finished, it causes the From DataSet object to retrieve the stored DataSet (myDataSet). From DataSet is configured to retrieve ALL records from myDataSet, but to test each record against the condition Rec.test1.pass AND Rec.test2.pass. In other words, a particular record is retrieved only if *both* test1 and test2 passed for that record.

Of the retrieved records, if any, the expression Rec.test1.result returns all of the test1.result record fields, which are then statistically analyzed. (Note that this model will error if none of the records satisfy the expression Rec.test1.pass AND Rec.test2.pass.)

This example is saved in:

```
/usr/lib/veetest/examples/mfgtest/manual45.ex
```


Some Restrictions in Logging Test Results

There are some situations where you must be careful in collecting Sequencer log records into an array of records. As explained in Chapter 10, to build an array of records, all of the array elements of a given field must be of the same type, shape, and size. For a record of records, as is generated by the Log output terminal of the Sequencer, the type, shape, and size of each field must match for sub-records as well.

For example, suppose you are collecting the logged results of several executions of a Sequencer, either by using the Collector to build an array (see Figure 13-6) or by sending the results to a DataSet (see Figure 13-7). In either case, if any of the logged values of a given transaction were to change type, shape, or size between executions of the Sequencer, an error will occur. The error will be generated by the Collector or To DataSet object because the array of records cannot be built.

This situation could easily occur if a transaction is not executed on every execution of the Sequencer; for example, if an ENABLED IF condition is specified. If the transaction is not executed, a log record will still be generated, but the NAME and DESCRIPTION fields will be empty strings and all the other fields will contain a Real scalar value of zero. If the same transaction, on a subsequent execution of the Sequencer, is executed and logs a result that is not a Real scalar, an error will occur. You might want to consider, in this situation, just writing each logged record out to a file in container format with To File, instead of using To DataSet.

An error could also occur if your tests return arrays of different sizes; for example, if the test returns an array of the failed data points. In this case, you might want to design the test so that it pads the array so as to always return the same size array.

A Practical Test Example

So far, we've just looked at how the `Sequencer` works, and how you might store, retrieve, and analyze the logged data. But normally, you'll want to use the `Sequencer` to control a series of "real world" tests. So let's look at a simple practical example.

In the old days, carbon resistors were manufactured by a rather imprecise process, and then tested, sorted, and marked. The trick was that the standard resistance values (for example, 220, 270, and 330 ohms) were chosen to overlap at the 10 percent tolerance. Thus, you didn't have to throw any resistors away. If a resistor was more than 10 percent greater than 220 ohms, it could be labeled as a 270 ohm resistor, and so forth.

So our problem is to construct a model in which the `Sequencer` calls a User Function, which returns a resistance value. The `Sequencer` will then run a series of tests to determine which nominal resistance value and percent tolerance the resistor satisfies. This is a "bin sort" problem. That is, the sequencer returns a result that identifies the bin in which to put the resistor.

One of the big advantages of using the `Sequencer` to call a User Function is that different User Functions can be substituted. For our problem, we'll just use a User Function (`simResist`) that returns a random resistance value in the expected range during development. You can easily substitute another User Function that executes instrument I/O and returns real resistance values once you've tested your solution.

The simplest solution to our problem is to use an extended series of sequence transactions, each testing the resistance value against a nominal value and tolerance.

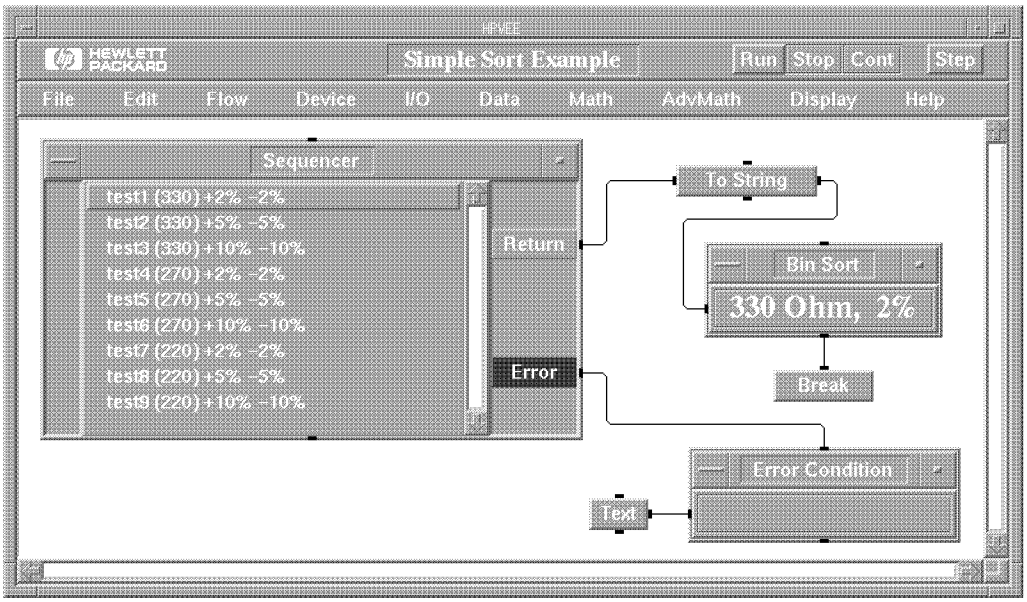
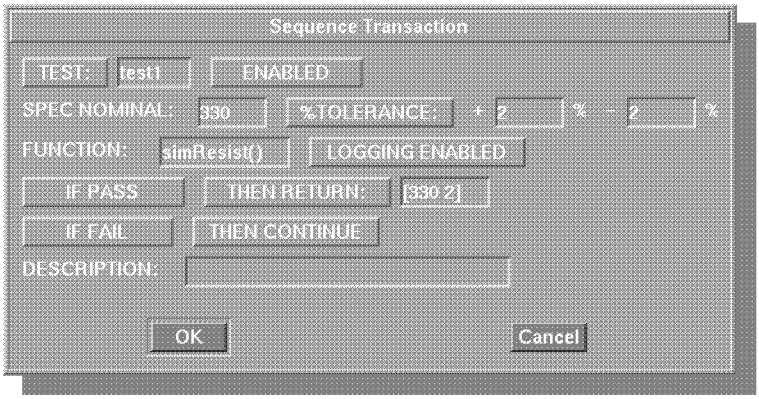


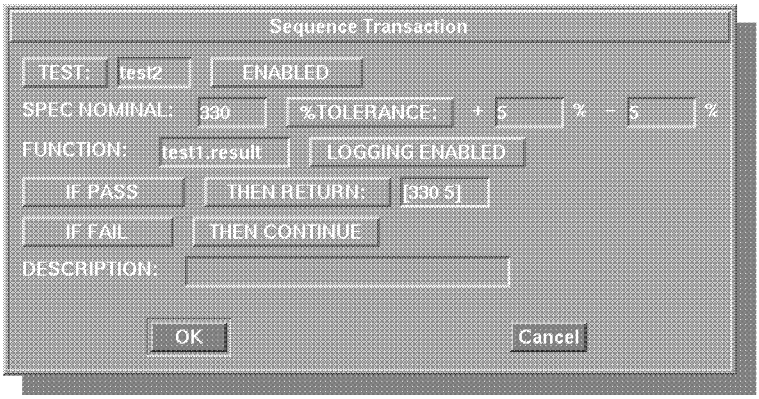
Figure 13-8. Simple Bin Sort Example

In this example, the first sequence transaction (`test1`) calls the User Function `simResist` with the expression `simResist()`. (This User Function requires no inputs.)



Note that `test1` tests to see if the resistance value returned by `simResist` is within ±2 percent of the nominal value 330. If it is, the two-element Real array `[330 2]` is returned on the `Return` output terminal, and the `To String` object converts this value to the string `330 Ohm, 2%`. If the test fails, the `Sequencer` goes on to the next test.

The second transaction, `test2`, works just like the first except that instead of calling `simResist` again, the `FUNCTION` field contains the expression `test1.result`:



Key Idea

Any transaction with logging enabled creates a "local" Record variable with the same name as the test. This record contains the fields specified for the logging record. Thus, for the transaction `test1`, the expression `test1.result` returns the value returned by the function called in `test1`.

There are two reasons for using the expression `test1.result` in our example. First, by using `test1.result` in transactions `test2` through `test9` we can ensure that each transaction uses the same function result, even if we later change `test1` to call a different function. More importantly in this example, each time you call the User Function, a *new* resistance value will be returned. Instead, we want to continue testing the original resistance value against successive nominal values and tolerances. So the transactions `test2` through `test9` all include the expression `test1.result` in the `FUNCTION` field. These transactions work like the first, returning the appropriate array (`[330 5]`, `[330 10]`, `[270 2]`, and so forth) if passed.

The first eight tests simply continue to the next test if failed. However, an indication is needed if *all* of the tests are failed. Thus, `test9` is configured `IF FAIL THEN ERROR`. The `Error` output terminal causes the `AlphaNumeric` display entitled `Error Condition` to execute, displaying the text `Out of Range`.

Although this approach is simple, it is not very efficient. You would have to create quite a large number of sequence transactions to test several resistance values, with three tolerances in each case. Let's look at an improved version of our "bin sort" example.

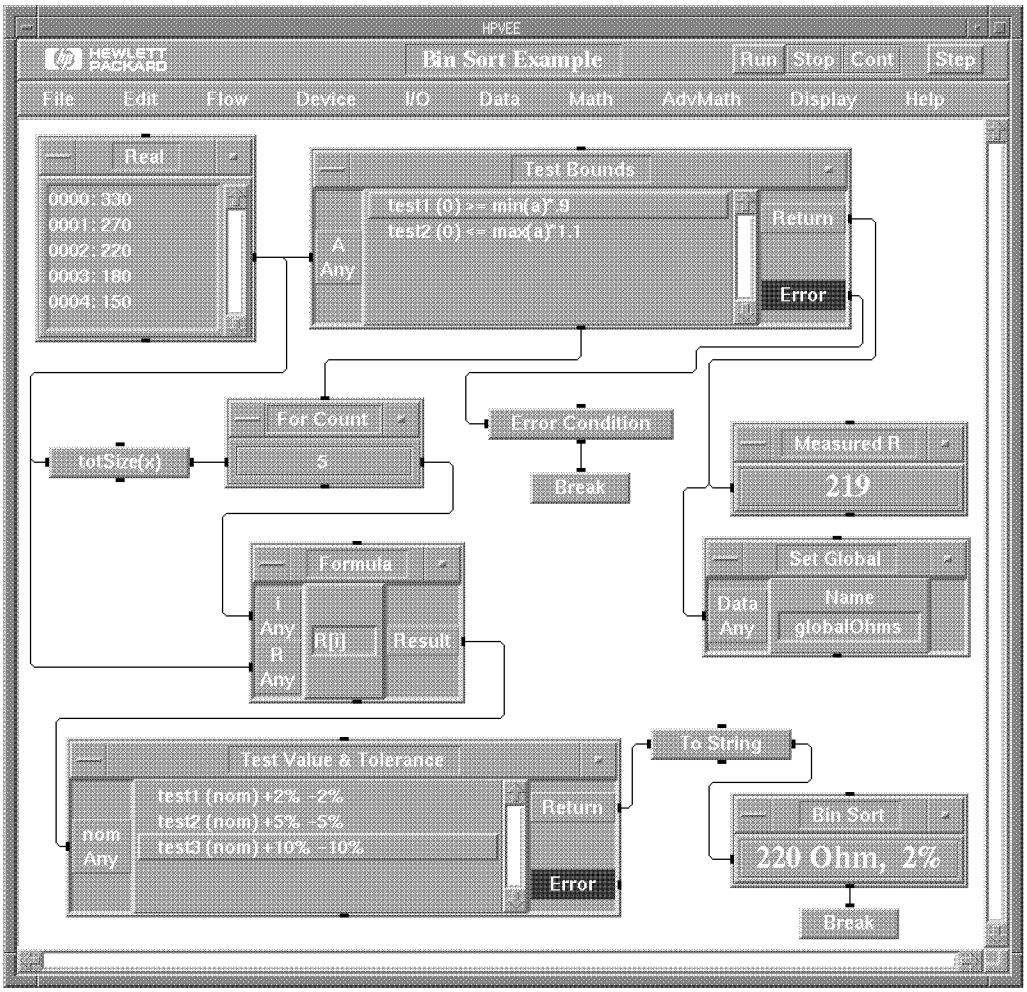


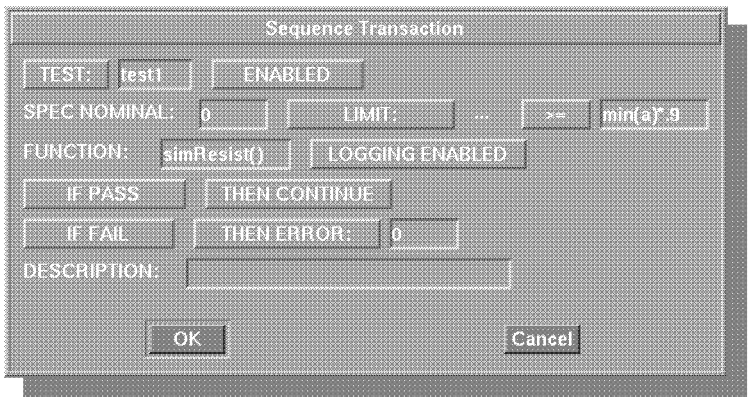
Figure 13-9. Improved Bin Sort Example

This example is saved in:

```
/usr/lib/veetest/examples/mfgtest/manual46.ex
```

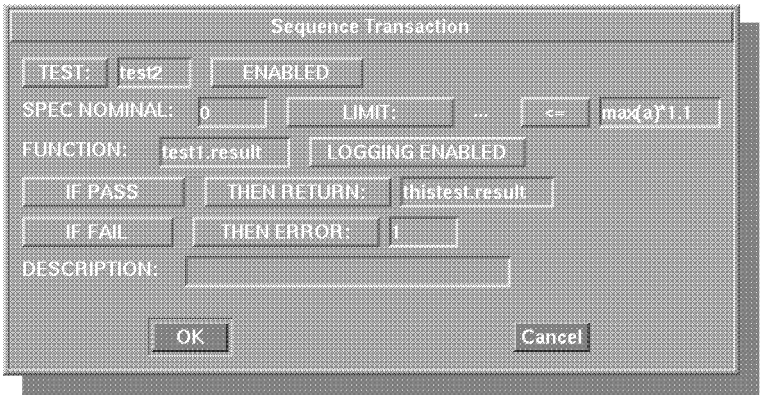
You may want to load this model and explore how it works. Here are some key points:

- This model uses two `Sequencer` objects. The first one (labeled `Test Bounds`) "re-uses" the tests in the second one (labeled `Test Value & Tolerance`).
- The `Real` array in the upper left corner of the model contains five elements, each representing a standard resistance value. However, the list of values is *extensible* in this example. Regardless of the number of array elements, the `TotSize(x)` function returns that number so that the `For Count` object will iterate the correct number of times. The expression `R[i]` in the `Formula` object takes care of the indexing.
- In the `Sequencer` named `Test Bounds`, the first transaction (`test1`) calls the User Function `simResist` with the expression `simResist()`:



A simulated resistance test value is returned and tested to see if it is at least 90 percent of the lowest value (150 Ohms) in the array. (Note that any value field in a sequence transation can contain an expression such as `min(a)*.9`.)

The second transaction (`test2`) tests to see if the value (`test1.result`) is less than or equal to 110 percent of the highest value (330 Ohms) in the array.

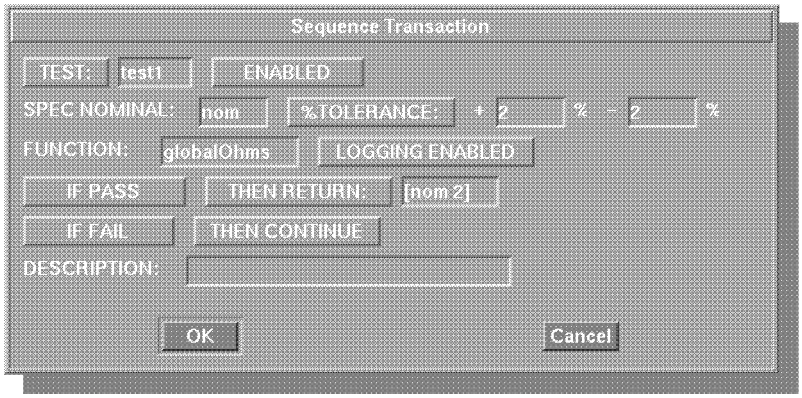


If either test fails, an error occurs.

- If an error does occur, the UserObject named `Error Condition` uses a `Triadic` expression to ascertain whether to display `Out of Range: LOW` or `Out of Range: HIGH`. The UserObject is configured as `Show Panel on Exec`, so if either error condition occurs, a display "pops up" to show the error. You'll find that this happens once every few times you run the model because the User Function `simResist` returns random values in the range 100–400. (To continue, just press `OK` in the pop-up box.)
- The transaction `test1` in the first `Sequencer` is the only transaction that calls the User Function `simResist`. (Instead, `test2` includes the expression `test1.result`.) This is necessary in this case because we want to run multiple tests on just one resistance value. Otherwise, a new value would be returned every time the User Function was called. However, there is another reason. Since the User Function `simResist` is only called once, you can easily replace it with a call to a different User Function. The example (`manual46.ex`) contains a second User Function named `measResist`, which uses an HP Instrument Driver to call an HP 3478A Digital Voltmeter configured for resistance measurements. If you have an HP 3478A meter,

just connect it to your HP-IB, change the `FORMULA` field in `test1` to the expression `measResist()`, and run the model.

- Regardless of whether simulated or measured resistance values are taken, the `Test Bounds` return value is displayed, and is set as a global variable (`globalOhms`). The three transactions in the `Sequencer` labeled `Test Value & Tolerance` each call this global variable using the expression `globalOhms`, for example:



If a test passes, the appropriate real array (e.g., `[220 2]`) is output. The `To String` object converts the data to a string (e.g., `220 Ohm, 2%`). The `Sequencer` will be executed as many times as necessary until a `Bin Sort` result is found.

- Note that we are not using the `Log` output terminal in either `Sequencer`, so we've deleted it to speed up execution.
- If you want to see the flow of this model, try running it a few times with `Show Exec Flow` and `Show Data Flow` turned on.

For further information about transaction options, using control pins, and so forth, refer to the `Sequencer` section in the *HP VEE Reference* manual.

For some further examples using the `Sequencer`, look in the directory:

```
/usr/lib/veetest/examples/mfgtest/
```

14

Troubleshooting Problems

This chapter explains common situations and recovery actions.

Table 14-1. Problems, Causes, and Solutions

Problem	Cause	Solution
Your `UserObject` doesn't operate when you think it should.	You might be crossing the context boundaries with asynchronous data (such as connecting to an XEQ pin on an object inside the `UserObject`).	Possible Solution 1: Move any asynchronous dependencies to outside the `UserObject`. Possible Solution 2: Use `Show Exec Flow` or `Show Data Flow` to view the order of operation in your model.
You want to change the functionality of an object.		Use the object menu features (including control pins).

Table 14-1. Problems, Causes, and Solutions (continued)

Problem	Cause	Solution
You only get one value output from an iterator within a `UserObject`.	A `UserObject` only activates its outputs once.	Take the iterator out of the `UserObject`.
An iterator only operates once.	Your iteration subthread is connected to the sequence output pin, not the data output pin.	Start the iteration subthead from the data output pin.
`For Count` doesn't operate.	The value of `For Count` is `0` or negative.	Change the value; if you need a negative value, negate the output or use `For Range`.
`For Range` or `For Log Range` doesn't operate.	The sign of the step size is wrong. If `From` is less than `Thru`, `Step` must be positive. If `Thru` is less than `From`, `Step` must be negative.	Change `Step`.
You move objects when you try to connect them.	You're clicking too close to the pin. To connect to a pin, the pointer must not be touching the object.	Click just outside the object, near the pin.

Table 14-1. Problems, Causes, and Solutions (continued)

Problem	Cause	Solution
You get the UNIX message `sh:`*name* `- not found`.	You mistyped the name of the executable.	Possible Solution 1: Check your spelling and type it again. Possible Solution 2: Make sure `veeengine` has three *e*'s.
HP VEE appears to hang—the pointer is an hourglass.	Possible Cause 1:HP VEE is rerouting lines because you have `Auto Line Routing` set on and you moved an object. Possible Cause 2: HP VEE is printing the screen or the model. Possible Cause 3: You just `Cut` a large object or a large number of objects. HP VEE is saving the objects to the `Paste` buffer.	Wait. If the pointer doesn't change back to the crosshairs within a few minutes, type CTRL-C (or whatever your `intr` setting is in the terminal window from which you started HP VEE), close the HP VEE window, or kill the HP VEE process.

Table 14-1. Problems, Causes, and Solutions (continued)

Problem	Cause	Solution
You can't **Open** a model, **Cut** objects, or delete a line (the feature is grayed).	The model is still running.	Press **Stop** twice to stop the model, then try the action again.
You can't **Paste** (the feature is grayed).	The **Paste** buffer is empty.	**Cut**, **Copy**, or **Clone** the object(s) again.
You can't **Cut**, **Create UserObject**, or **Add to Panel** (the feature is greyed).	No objects are selected.	Select the objects using **Edit** $\Longrightarrow$ **Select Objects** and try the action again.
A **UserObject** only outputs the last data element generated.	**UserObjects** do not accumulate data in the output terminal buffer. It only holds the last data element received.	Use a **Collector** to gather all of the data generated into an array. Send this data to the output terminal.
You cannot add inputs or modify the condition or formula.	You have a pre-defined object (from **Data** $\Longrightarrow$ **Conditional** $\Longrightarrow$, the **Math** menu or **AdvMath** menu).	Use an **If/Then/Else** or **Formula** object. You can modify them to complete your task.
You can't get out of line drawing mode		Double-click to end line drawing mode.

Table 14-1. Problems, Causes, and Solutions (continued)

Problem	Cause	Solution
You get a `Parse Error` object when you `Open` a model.	Possible Cause 1: You `Save`d a model that contained an object with invalid data, such as an `If/Then/Else` object with an invalid expression.	Replace the `Parse Error` object with a new object.
	Possible Cause 2: You have HP VEE-Engine and `Open`ed a model that was created in HP VEE-Test. `Parse Error` replaces objects that are unique to HP VEE-Test.	Run the model in HP VEE-Test.
Your characters are not appearing correctly.	You have a non-USASCII keyboard.	Refer to Appendix A for recovery information.
Your colors outside of HP VEE are changing (although when you're in HP VEE, the HP VEE colors look normal).	Your color map planes are all used.	Refer to Appendix A for recovery information.

A

Configuring HP VEE

This appendix explains how to configure and customize HP VEE for your environment by changing X11 and HP VEE options. This appendix discusses the following topics:

- Changing X11 attributes (such as colors, fonts, and window size and placement)
- Using two (or more) color palettes
- Customizing your icon bitmaps
- Selecting a bitmap for a panel view
- Recovering from X11 color plane limitations
- Using non-USASCII keyboards
- Using HP-GL Plotters
- Using Two-Byte Character Sets

Changing X11 Attributes

HP VEE provides a variety of font and color palette files that you can use to customize the look of HP VEE. The palette files are stored in `/usr/lib/veeengine/config/` or `/usr/lib/veetest/config/`. The beginning of each file contains a short description of the color and font scheme. To use a palette file, you must install it into your X11 resources database.

If you are using `xrdb`, install the palette file by typing `xrdb -merge` *palette_filename* before starting HP VEE.

If you are not using `xrdb`, merge the palette file into your X11 resources file. Your X11 resources file is usually `.Xdefaults in your $HOME` directory, but may be in a file identified with the environment variable `$XENVIRONMENT`.

To change the color and font scheme for all user of HP VEE on your system, your system administrator may replace the file `/usr/lib/X11/app-defaults/Vee` with one of the palette files.

To change other X11 resources, you can change or add to your X11 resources file. For example, to change the default geometry of the HP VEE window so that it always started in the lower right corner of your screen and the window was sized to 640 by 480 pixels, you would add the following line to your X11 resources file (probably `.Xdefaults`): `Vee*geometry: =640x480-0-0`.

For more information about customizing an X11 environment, refer to *Beginner's Guide to the X Window System*.

Using Multiple Color Sets/Fonts

It is often convenient to keep two (or more) sets of HP VEE color and font combinations. For example, you may want to use a different font and color palette for printing the screen. (The default printer colors are selected to give a desirable printout after gray-scale conversion, and may not be the best choice for a color printer.)

You can use two different options to start HP VEE with a different "instance" name. These are the `-name` option, which controls the palette and font for the screen, and the `-prname` option, which controls the palette for printing.

The default HP VEE resource file contains a `VeePrint` palette for printing, in addition to the `Vee` palette for the screen. HP VEE will automatically read in the `VeePrint*` color resources and use them for printing, but will use the standard `Vee*` colors on the screen. For example, lines will be white on the screen, but will be printed black.

If you want the printer to use the same colors as the screen, start HP VEE with the `-prname` option and the "instance" name `Vee` as follows:

```
veeengine -prname Vee
-- or --
veetest -prname Vee
```

HP VEE will now use the `Vee*` color resources for printing, instead of the `VeePrint*` color resources.

If you want to create a custom palette and font selection for your display, you'll need to merge the appropriate palette into your X11 environment. You'll have to change the application class name from `Vee*` to another name, such as `myVee*`. (Refer to "Changing X11 Attributes" for further information.)

To use the `myVee*` resources instead of `Vee*` for your display screen, execute one of the following:

```
veeengine -name myVee
-- or --
veetest -name myVee
```

To use the `myVee*` resources both for the screen and for printing:

```
veeengine -name myVee -prname myVee
-- or --
veetest -name myVee -prname myVee
```

Several palettes are found in the directory:

```
/usr/lib/veeengine/config
-- or --
/usr/lib/veetest/config
```

Customizing Icon Bitmaps

You can change the bitmap displayed on any object icon to a bitmap or pixmap of your choice. HP VEE supports the X11 bitmap, xwd, and GIF formats.

To create your own bitmaps, use the `/usr/bin/X11/bitmap` program. For example, type:

`/usr/bin/X11/bitmap` *name*`.icon` *size*x*size*

name is the name of the file to create or edit. *size* is the number of pixels for the width and height of the bitmap. `48x48` is a good size for an icon. You can also use any other bitmap editor that outputs the X11 bitmap format.

You can also use an X11 Window Dump (xwd) format file. Just use `/usr/bin/X11/xwd` to capture an image from a window on the screen. Or you can use a paint program such as IslandPaint to create a pixmap and save it in a file in xwd format. HP VEE can also import GIF files.

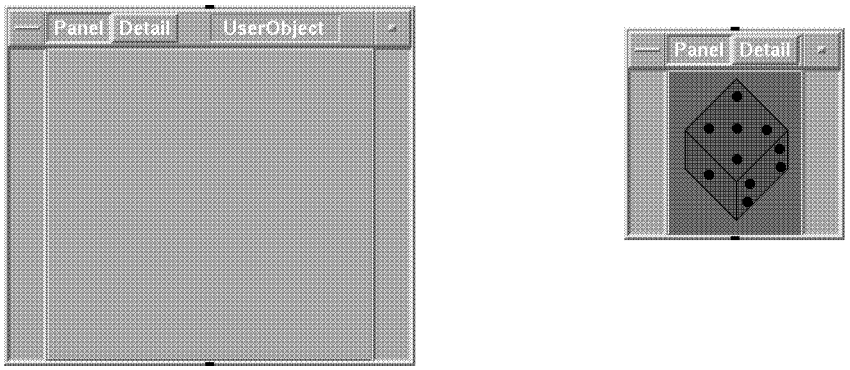
To replace the default bitmap on a single object, select your bitmap from the icon's object menu (`Layout` ⟹ `Select Bitmap`).

To replace the default bitmap for a category of objects (such as `Formula` objects), rename your bitmap file to the default bitmap file name (such as `formula.icon`) and keep it in your HP VEE startup directory.

Selecting a Bitmap for a Panel View

You can select a bitmap to use as the "background" for a panel view. The bitmap can be an `XWD` or `GIF` file, or any X11 bitmap. You can create a bitmap as described in the previous section.

To select a bitmap for the panel view of a UserObject, choose `Select Bitmap` from the object menu. For example, in the illustration below the figure on the left is the panel view of a UserObject, with a blank background. If you choose `Select Bitmap` from the object menu and choose the bitmap named `die5.xwd`, the figure on the right will result (the die will be in color on the screen):



To select a bitmap for the panel view of an HP VEE model, the process is the same except that you choose `Select Bitmap` from the `Edit` menu for the panel view. (Just click on the right mouse button with the pointer in the work area to display the edit menu.)

If You See Colors Changing On Your Screen

Your workstation is equipped with a certain number of color planes (usually 1, 4, 6, or 8). X11 uses the information in these color planes to color your application's window. If you have more than one application running (each in its own window), and you notice the screen colors changing as you move from one application's window to another, then one of two things may be happening. Either all the applications, together, use more colors than your display has available, or one or more of the applications allocates its own private color map (for example, HP BASIC/UX).

HP VEE uses at least 39 colors (this varies depending on how you define the colors and which colors HP VEE actually uses while running), so you may experience this behavior when HP VEE is one of your applications. The symptoms are: when you are in the HP VEE window, the HP VEE colors will be correct for HP VEE, but may be wrong in other application's windows. When you move to another application's window, the colors will be correct for that application, but may be wrong for HP VEE. *This is typical X11 behavior—it is not a problem with HP VEE.*

This behavior does not affect the performance of HP VEE or any other application. However, if it bothers you, there are some things you can do to help, depending on the cause.

There are two causes of this behavior:

- You have requested more colors than your workstation can simultaneously display.
- One of the applications you are running controls a local color map.

Too Many Colors

Your workstation can display some number of colors at one time, based on the number of color planes for your display. This number is:

$$2^{number\ of\ color\ planes}$$

For example, if you have 4 color planes, you can use as many as 16 colors at a time on your display.

$$2^4 = 16$$

If you exceed this number, you may see the screen flashing as you change from one window to another.

If you exceed your total available colors, the first step in eliminating the "flashing" is to reduce your colors to be within the limits of your workstation. Some tips on reducing colors are:

- Remove any "extra" colors. If two applications can use the same color scheme, then customize them to do this.
- Use "reduced-color" color schemes in applications. For example, HP VEE is shipped with several optional color schemes. The color scheme called `Safari` uses the fewest colors in HP VEE (39).
- Stop, or do not even start, any applications that you do not currently need. Often, each application uses its own color scheme. This can quickly increase your requested colors to exceed your color map limit. Once you have stopped other applications, you probably need to stop, then re-start, HP VEE before the behavior goes away.
- Reduce the number of colors allocated by the `xinitcolormap` command. Because these colors remain permanently in the color map, there is room for fewer temporary colors.

Some X11 window managers have a colormap focus directive (for example, `*colormapFocusPolicy`). The value to which this is set may also contribute to how colors are used on the screen. In particular, if you exceed the total number of colors you can simultaneously display, do not set this to be `explicit` or you may not see correct colors in your application's window.

Applications that Use a Local Color Map

Some applications use a local color map. This means that when you run this application, it saves the current color map and switches over to its own, local color map. When this happens you may see the "flashing" between windows.

One way to circumvent this is to pre-allocate the HP VEE colors using the `xinitcolormap` command. To do this, you create an ASCII file listing the colors you wish to pre-allocate. This file is described in the `man` page for `xinitcolormap`. Basically, though, the file cannot contain blank lines, must start with the colors `Black` and `White`, and the color format can be either pre-defined word colors or the actual RGB hex values, preceded by the

symbol—**#**. For example, the following two examples contain black, white, and a shade of light gray:

```
Black
White
LightGray
```

Figure A-1. Color Map File Using Words

```
#000000
#ffffff
#a8a8a8
```

Figure A-2. Color Map File Using Hex Numbers

HP BASIC/UX is one application that uses a local color map and recommends that you pre-allocate the HP BASIC/UX colors at startup using the `xinitcolormap` command (refer to the `/usr/lib/rmb/newconfig/rgb.README` file).

Because of this, if you will use HP VEE with HP BASIC/UX (or other applications that allocate colors in the same way HP BASIC/UX does), you need to also pre-allocate HP VEE colors at startup. If you do not, you may see the colors flash on the screen as you move from one window to another.

To do this:

1. Create a "colormap" file that contains all the different HP VEE colors you will use. HP has supplied the following colormap files in HP VEE's configuration directory:
   - `DefColorMap` if you are using the default HP VEE colors.
   - `SafColorMap` if you are using the `Safari` color scheme (use this if you need to use fewer colors).
2. Change to your `$HOME` directory:

   ```
   cd $HOME
   ```

3. Concatenate the HP BASIC/UX and the HP VEE colormap files:

```
cat /usr/lib/rmb/newconfig/xrmbcolormap vee-colormapfile > .xveecolormap
```

   Note that the HP BASIC/UX colors must go first, because HP BASIC/UX assumes that they are the first 16 entries in the colormap. You can mix the word colors and the hex number colors in one file.

4. You must use the `xinitcolormap` command before you allocate any colors for other applications. This means that it should be placed near the beginning of your .x11start file.

   For example, if you use the `.x11start` file, your colors are in `$HOME/.xveecolormap`, and you have 55 colors listed in the file (16 from HP BASIC/UX + 39 from HP VEE), you would add the following line to `.x11start`:

   ```
   /usr/bin/X11/xinitcolormap -c 55 -f $HOME/.xveecolormap
   ```

5. Restart X11. To do this, stop the window manager by pressing the following three keys at the same time: Shift-CTRL-Break, or selecting `Reset` from your root menu (if it is configured for this choice), then type:

   ```
   x11start
   ```


Using Non-USASCII Keyboards

If you are using a non-USASCII keyboard, you need to use a special HP VEE palette so that your characters appear correctly in HP VEE. You need to modify your X11 environment to use the `Iso` or `Katakana` palette. Install the palette using the instructions in "Changing X11 Attributes".

Use the `Iso` palette if you have one of the following keyboards:

- Belgian
- Canadian English
- Canadian French
- Danish
- Dutch
- European Spanish

- Finnish
- French
- German
- Italian
- Latin Spanish
- Norwegian
- Swedish
- Swiss French
- Swiss German
- UK English

Use the `Katakana` or `Kanji` palette if you have one of the following keyboards:

- Katakana
- Kanji

Note

If you are accessing data that was created with the `Roman8` character set, you must translate any special characters (above ASCII 127) used.

Your terminal window may use `Roman8`; therefore `TEXT` written to stdout, file names (such as specified by `To File` and `From File`), and programs names must use ASCII characters 0-127 to match with those specified with HP VEE.

Using HP-GL Plotters

HP Vee supports graphics output to plotters and files using HP-GL. Before you can send plots to a plotter (either local or networked) your system administrator must add the plotter as a spooled device on your system.

In addition to standard HP-GL plotters such as the HP 7475, the HP ColorPro (HP 7440), or the HP 7550, some printers can be used as plotters, such as the PaintJet XL, and the LaserJet III. The HP ColorPro plotter requires the Graphics Enhancement Cartridge in order to plot polar or Smith Chart graticules, or an Area-Fill line type. The PaintJet XL requires the HP-GL/2 Cartridge in order to make any plots. In order to make plots on the LaserJet III, at least two megabytes of optional memory expansion is required, and the

Page Protection configuration option should be enabled. Plots of many vectors, especially with Polar or Smith chart graticules, may require even more optional memory in the LaserJet III. Any plot intended for a printer requires the plotter type to be set to HP-GL/2, which causes the proper HP-GL/2 setup sequence to be included with the plot information.

Any of the following graphical two-dimensional displays can be plotted to an HP-GL or HP-GL/2 plotter, or to a file:

```
XY Trace, Strip Chart, Complex Plane,
X vs Y Plot, Polar Plot, Waveform,
Magnitude Spectrum, Phase Spectrum,
Magnitude vs Phase.
```

You can specify the appropriate plotter configuration by selecting:
`File` ⟹ `Preferences` ⟹ `Plotter Config`.

To generate a plot, just select `Plot` on the display's object menu, specify the required parameters in the `Plot Display` dialog box, and then press `OK`. You can also add `Plot` as a control input to generate plots programmatically. The entire view of the display object will be plotted, and scaled to fill the defined plotting area, while retaining the aspect ratio of the original display object. By re-sizing the display object, you can control the aspect ratio of the plotted image. By making the display object larger, you can reduce the relative size of the text and numeric labels around the plot.

For an explanation of the plotter configuration parameters, refer to the `Plotter Config` section in the *HP VEE Reference* manual. Also, refer to the reference sections for the appropriate two-dimensional display devices.

Using Two-Byte Character Sets

Two-byte characters (such as Kanji for Japanese) can be entered into any field in HP-VEE where you can enter text. This includes all titles, text (string) constants, input/output pin names, and note pads. To use two-byte characters you must have the UNIX NLIO subsystem installed and initialized, the $LANG shell variable set to your local language (for example: `export LANG=japanese`), and an appropriate set of fonts selected (such as the Kanji app-defaults file):

```
/usr/lib/veeengine/config/Kanji

-- or --

/usr/lib/veetest/config/Kanji
```

Refer to "Changing X11 Attributes" at the beginning of this chapter for further information.

Note

The Kanji app-defaults file specifies a two-byte "stroke" font. A stroke font consists of characters drawn as strokes, rather than as a raster image. The stroke font is required for output to most plotters, since most plotters do not directly support Kanji.

The following are some limitations to two-byte character support:

- When using the `Plot` command to send a graphical two-dimensional display (e.g., `XY Trace`) to a plotter or file, you must first specify that labels are to be output using the two-byte "stroke" font. To do this, go to the `Plotter Config` dialog box:

  `File` ⟹ `Preferences` ⟹ `Plotter Config`.

  The `Label Using` field gives you two choices: `Stroke Fonts` and `Plotter ROM`. You must select `Stroke Fonts` in order to output a display with two-byte characters. Otherwise, all two-byte characters in field labels will be encoded into HP-GL label commands as two-byte characters in the HP15 character set, which is not supported by most plotters.

  For further information, refer to the `Plotter Config` section in the *HP VEE Reference* manual.

- When reading text that includes two-byte characters from a `Direct I/O` object, the two-byte character rules are not used when looking for the EOL string. Thus, an EOL character may be incorrectly found in the second byte of a two-byte character. (This is only a problem if an EOL character has an ASCII value greater than 32 decimal.)
- Date/Time parsing and formatting have not been globalized, and continue to only execute in English. To obtain a localized date string, use an `Execute Program` object with a program of "date" and a transaction of `READ TEXT x STR`.

B

Example Models and Library Objects

Both HP VEE-Engine and HP VEE-Test include several examples of HP VEE models that you can use. A library of objects that you can `Merge` into your models is also included. The example models and library objects are installed as part of the normal HP VEE installation process.

Using the Examples

The example models are included with HP VEE for several purposes. Many of the examples from the manuals are included in the examples directories (with file names like `manual01.ex`, etc). If you want to try running a model shown in one of the manuals, you can save time by opening and running it from a file. Other examples, not referenced in any of the manuals, are included to illustrate specific HP VEE concepts, or to illustrate solutions to engineering problems using HP VEE. To help you find the example you want, the examples are divided into subdirectories. Each subdirectory has a `CONTENTS` file, which is really an HP VEE model that you can load like any other model.

For HP VEE-Engine the examples are installed in subdirectories under:

```
/usr/lib/veeengine/examples/
```

For HP VEE-Test the examples are installed in subdirectories under:

```
/usr/lib/veetest/examples/
```

Under `examples` you will find several subdirectories, each containing a class of example models. (The exact selection of subdirectories depends on which version of HP VEE you have.) Just load the `CONTENTS` model for a particular subdirectory to determine its contents. The `CONTENTS` model describes the

class of examples in the subdirectory and provides a brief description of each example model.

Once you have selected an example model of interest, just load it with `File` ⟹ `Open` as with any other model. If you want to modify an example model and save it, you'll want to save it in a different directory. (You can't write to the `examples` subdirectories unless you are logged on as "root.")

Using Library Objects

The object library provides several objects that you can merge into your own model. Just select `Merge` from the `File` menu and a list box will appear for the appropriate library directory:

```
/usr/lib/veeengine/lib/

- or -

/usr/lib/veetest/lib/
```

The directory provides a `CONTENTS` model, which describes the available objects. To merge an object, just double click on its name and insert the object in your model.

Most of the library objects are actually UserObjects that encapsulate individual objects. You can create your own UserObjects for the library, but you'll need to save them in the `contrib` subdirectory:


```
/usr/lib/veeengine/lib/contrib/

- or -

/usr/lib/veetest/lib/contrib/
```

(You can't write to the `lib` directory unless you are logged on as "root.")

The `contrib` subdirectory is empty at installation — it provides a place for your own library of "contributed" objects.

There is another subdirectory under `lib`, named `conversions`:

```
/usr/lib/veeengine/lib/conversions/

- or -

/usr/lib/veetest/lib/conversions/
```

This subdirectory contains several formula objects that you can `Merge` into your model. Each of these objects performs a useful conversion function such as degrees to radians. This subdirectory contains a `CONTENTS` model that describes each of the objects.

C

ASCII Table

This appendix contains reference tables of ASCII 7-bit codes.

ASCII 7-bit Codes

	Binary	Oct	Hex	Dec	HP-IB Msg
NUL	0000000	000	00	0	
SOH	0000001	001	01	1	GTL
STX	0000010	002	02	2	
ETX	0000011	003	03	3	
EOT	0000100	004	04	4	SDC
ENQ	0000101	005	05	5	PPC
ACK	0000110	006	06	6	
BEL	0000111	007	07	7	
BS	0001000	010	08	8	GET
HT	0001001	011	09	9	TCT
LF	0001010	012	0A	10	
VT	0001011	013	0B	11	
FF	0001100	014	0C	12	
CR	0001101	015	0D	13	
SO	0001110	016	0E	14	
SI	0001111	017	0F	15	
DLE	0010000	020	10	16	
DC1	0010001	021	11	17	LLO
DC2	0010010	022	12	18	
DC3	0010011	023	13	19	
DC4	0010100	024	14	20	DCL
NAK	0010101	025	15	21	PPU
SYN	0010110	026	16	22	
ETB	0010111	027	17	23	

	Binary	Oct	Hex	Dec	HP-IB Msg
CAN	0011000	030	18	24	SPE
EM	0011001	031	19	25	SPD
SUB	0011010	032	1A	26	
ESC	0011011	033	1B	27	
FS	0011100	034	1C	28	
GS	0011101	035	1D	29	
RS	0011110	036	1E	30	
US	0011111	037	1F	31	
space	0100000	040	20	32	listen addr 0
!	0100001	041	21	33	listen addr 1
"	0100010	042	22	34	listen addr 2
#	0100011	043	23	35	listen addr 3
$	0100100	044	24	36	listen addr 4
%	0100101	045	25	37	listen addr 5
&	0100110	046	26	38	listen addr 6
'	0100111	047	27	39	listen addr 7
(	0101000	050	28	40	listen addr 8
)	0101001	051	29	41	listen addr 9
*	0101010	052	2A	42	listen addr 10
+	0101011	053	2B	43	listen addr 11
,	0101100	054	2C	44	listen addr 12
-	0101101	055	2D	45	listen addr 13
.	0101110	056	2E	46	listen addr 14
/	0101111	057	2F	47	listen addr 15

ASCII 7-bit Codes (continued)

	Binary	Oct	Hex	Dec	HP-IB Msg
0	0110000	060	30	48	listen addr 16
1	0110001	061	31	49	listen addr 17
2	0110010	062	32	50	listen addr 18
3	0110011	063	33	51	listen addr 19
4	0110100	064	34	52	listen addr 20
5	0110101	065	35	53	listen addr 21
6	0110110	066	36	54	listen addr 22
7	0110111	067	37	55	listen addr 23
8	0111000	070	38	56	listen addr 24
9	0111001	071	39	57	listen addr 25
:	0111010	072	3A	58	listen addr 26
;	0111011	073	3B	59	listen addr 27
<	0111100	074	3C	60	listen addr 28
=	0111101	075	3D	61	listen addr 29
>	0111110	076	3E	62	listen addr 30
?	0111111	077	3F	63	UNL
@	1000000	100	40	64	talk addr 0
A	1000001	101	41	65	talk addr 1
B	1000010	102	42	66	talk addr 2
C	1000011	103	43	67	talk addr 3
D	1000100	104	44	68	talk addr 4
E	1000101	105	45	69	talk addr 5
F	1000110	106	46	70	talk addr 6
G	1000111	107	47	71	talk addr 7
H	1001000	110	48	72	talk addr 8

	Binary	Oct	Hex	Dec	HP-IB Msg
I	1001001	111	49	73	talk addr 9
J	1001010	112	4A	74	talk addr 10
K	1001011	113	4B	75	talk addr 11
L	1001100	114	4C	76	talk addr 12
M	1001101	115	4D	77	talk addr 13
N	1001110	116	4E	78	talk addr 14
O	1001111	117	4F	79	talk addr 15
P	1010000	120	50	80	talk addr 16
Q	1010001	121	51	81	talk addr 17
R	1010010	122	52	82	talk addr 18
S	1010011	123	53	83	talk addr 19
T	1010100	124	54	84	talk addr 20
U	1010101	125	55	85	talk addr 21
V	1010110	126	56	86	talk addr 22
W	1010111	127	57	87	talk addr 23
X	1011000	130	58	88	talk addr 24
Y	1011001	131	59	89	talk addr 25
Z	1011010	132	5A	90	talk addr 26
[	1011011	133	5B	91	talk addr 27
\	1011100	134	5C	92	talk addr 28
]	1011101	135	5D	93	talk addr 29
^	1011110	136	5E	94	talk addr 30
_	1011111	137	5F	95	UNT
‘	1100000	140	60	96	secondary addr 0

ASCII 7-bit Codes (continued)

	Binary	Oct	Hex	Dec	HP-IB Msg
a	1100001	141	61	97	secondary addr 1
b	1100010	142	62	98	secondary addr 2
c	1100011	143	63	99	secondary addr 3
d	1100100	144	64	100	secondary addr 4
e	1100101	145	65	101	secondary addr 5
f	1100110	146	66	102	secondary addr 6
g	1100111	147	67	103	secondary addr 7
h	1101000	150	68	104	secondary addr 8
i	1101001	151	69	105	secondary addr 9
j	1101010	152	6A	106	secondary addr 10
k	1101011	153	6B	107	secondary addr 11
l	1101100	154	6C	108	secondary addr 12
m	1101101	155	6D	109	secondary addr 13
n	1101110	156	6E	110	secondary addr 14
o	1101111	157	6F	111	secondary addr 15
p	1110000	160	70	112	secondary addr 16
q	1110001	161	71	113	secondary addr 17
r	1110010	162	72	114	secondary addr 18
s	1110011	163	73	115	secondary addr 19
t	1110100	164	74	116	secondary addr 20
u	1110101	165	75	117	secondary addr 21
v	1110110	166	76	118	secondary addr 22
w	1110111	167	77	119	secondary addr 23
x	1111000	170	78	120	secondary addr 24
y	1111001	171	79	121	secondary addr 25
z	1111010	172	7A	122	secondary addr 26
{	1111011	173	7B	123	secondary addr 27
|	1111100	174	7C	124	secondary addr 28
}	1111101	175	7D	125	secondary addr 29
~	1111110	176	7E	126	secondary addr 30
[del]	1111111	177	7F	127	

D

HP VEE Utilities

HP VEE provides some utility programs, accessible from a UNIX command line in any X11 window.

The veedoc Utility for Documenting Models

HP VEE includes a utility program `veedoc`, accessible from a UNIX command line, which extracts information from a model created with HP VEE-Test or HP VEE-Engine. The `veedoc` utility prints a line for every object or local User Function in your model. Each line contains an identification number and the name of the object or User Function. This identification number denotes the relative "nesting" position of the object in the model and is unique to a particular object. The identification number, once assigned, will not change and will not be reused. The `veedoc` utility also extracts the `Show Description` information for the root context level (accessible through the `File` $\Longrightarrow$ `Show Description` menu), all objects, and all local User Functions. If a `Note Pad` object is present, its content is also extracted.

To use this utility, go to a UNIX command line and execute:

```
/usr/lib/veetest/veedoc filename [ ... ]
- or -
/usr/lib/veeengine/veedoc filename [ ... ]
```

where *filename* is the name of your model, including path. For example, to run `veedoc` on the example model `mfgtest.ex`, execute:

```
veedoc examples/applications/mfgtest.ex
```

To print this same information, execute:

```
veedoc examples/applications/mfgtest.ex | lp
```

Additional information can be found by looking at the UNIX manual page for `veedoc`, which was added when you installed HP VEE .

Note

The `veedoc` utility is compatible with models created with HP VEE Release A.00.01 and later versions. If you want to use `veedoc` with models created with Release A.00.00, you must first load the model into the current version of HP VEE and then re-save it.

The HP Driver Writer's Tool for Creating Instrument Drivers

The HP Driver Writer's Tool (HP DWT) is a utility program that allows you to create your own HP Instrument Driver (or ID). This utility is installed as part of the normal HP VEE-Test installation, but it is not shipped with HP VEE-Engine.

To start the HP DWT, execute the following command from the UNIX command line in an X11 window.

```
/usr/lib/veetest/dwt
```

A menu driven window, similar to the HP VEE window, will appear. The pull-down menus `File`, `Edit`, `Create`, `Other`, and `Help` allow you to navigate within the HP DWT. For instructions on how to create, edit, save, compile, and use an ID, refer to the online Help for the HP DWT. For the advanced ID programmer, the ID syntax is described in the *HP Instrument Driver Language Reference* manual. Ordering information for this manual is given in the online Help.

Note

The HP Driver Writer's Tool starts up in "Learning Mode" by default. In this mode, the `File` menu selections are "grayed out." Thus, to exit the program you will first have to turn off "Learning Mode" (`Other` ⟹ `Learning Mode`), and then exit (`File` ⟹ `Exit`).

E

I/O Transaction Reference

This appendix contains details about the behavior of all I/O transaction actions, encodings, and formats. This appendix is organized by the transaction actions summarized in Table E-1. For example, if you need detailed information about `TEXT` encoding, do this:

- Look in the `WRITE` section for details about `WRITE TEXT` transactions.
- Look in the `READ` section for details about `READ TEXT` transactions.

Table E-1. Summary of Transaction Types

Action	Description
`WRITE`	Writes data to the destination specified in the object.
`READ`	Reads data from the source specified in the object.
`EXECUTE`	Executes low-level commands to control the file, device, or interface associated with the object. `EXECUTE` is used to adjust file pointers, to close pipes and files, and to provide low-level control of devices and hardware interfaces.
`WAIT`	Waits for the specified number of seconds before executing the next transaction. For `Direct I/O`[1] objects, `WAIT` can also wait for a specific serial poll response, or for specific values in accessable VXI device registers.
`SEND`[1]	Sends IEEE 488-defined bus messages (bus commands and data) to an HP-IB interface.

1 HP VEE-Test only.

Table E-2. Summary of I/O Transaction Objects

Objects	Supported Transactions				
	EXECUTE	WAIT	READ	WRITE	SEND[1]
To File	X	X		X	
From File	X	X	X		
To Printer		X		X	
To String		X		X	
From String		X	X		
To StdOut		X		X	
From StdIn		X	X		
To StdErr		X		X	
Execute Program	X	X	X	X	
To/From Named Pipe	X	X	X	X	
Direct I/O[1]	X	X	X	X	
Interface Operations[1]	X				X
To/From HP BASIC/UX[1]	X	X	X	X	

1 HP VEE-Test only.

WRITE Transactions

This section is organized by the `WRITE` encodings summarized in Table E-3. Topics that apply to all `WRITE` encodings are summarized at the beginning of this section.

Path-Specific Behaviors

Some `WRITE` transactions behave differently depending on the I/O path of the destination. For example, `WRITE TEXT HEX` transactions format hexadecimal numbers differently depending on whether the destination is a UNIX file or an instrument. To distinguish these behaviors, this section uses the following terms:

Term	Meaning
UNIX paths	Any destination other than an instrument, such as a UNIX file, a string, the printer, or a UNIX pipe.
direct I/O paths	Any instrument accessed using `Direct I/O`.

The behaviors described in the following sections apply to all paths, except as specifically noted.

Table E-3. WRITE Encodings and Formats

Encodings	Formats
TEXT	DEFAULT STRING QUOTED STRING INTEGER OCTAL HEX REAL COMPLEX PCOMPLEX COORD TIME STAMP
BYTE	Not Applicable
CASE	Not Applicable
BINARY	STRING BYTE INT16 INT32 REAL32 REAL64 COMPLEX PCOMPLEX COORD
BINBLOCK	BYTE INT16 COMPLEX INT32 PCOMPLEX REAL32 REAL64 COORD

Table E-3. `WRITE` Encodings and Formats (continued)

Encodings	Formats

`CONTAINER`	`DEFAULT`
	`STRING`
	`INTEGER`
	`REAL`
	`COMPLEX`
	`PCOMPLEX`
	`COORD`
	`WAVEFORM`
	`SPECTRUM`

`STATE`[1]	Not Applicable

`REGISTER`[2]	`BYTE`
	`WORD16`
	`WORD32`
	`REAL32`

`MEMORY`[2]	`BYTE`
	`WORD16`
	`WORD32`
	`REAL32`

`IOCONTROL`[3]	Not Applicable

1 Direct I/O to HP-IB only (HP VEE-Test only).

2 Direct I/O to VXI only (HP VEE-Test only).

3 Direct I/O to GPIO only (HP VEE-Test only).

TEXT Encoding

`WRITE TEXT` transactions are of this form:

`WRITE TEXT` *ExpressionList [Format]*

ExpressionList is a single expression or a comma-separated list of expressions.

Format is an optional setting that specifies one of the formats listed in Table E-4.

Table E-4. Formats for `WRITE TEXT` Transactions

Format	Description
`DEFAULT`	HP VEE automatically determines an appropriate text representation based on the data type of the item being written.
`STRING`	Writes Text data without any conversion. Writes numeric data types as Text with maximum numeric precision.
`QUOTED STRING`	Writes data in the the same format as `STRING`, except the data is surrounded by double quotes (ASCII 34 decimal).
`INTEGER`	Writes data as a 32-bit two's complement integer in decimal form.
`OCTAL`	Writes data as a 32-bit two's complement integer in octal form.
`HEX`	Writes data as a 32-bit two's complement integer in hexadecimal form.
`REAL`	Writes data as a 64-bit floating point number in a variety of notations including fixed decimal and scientific notation.
`COMPLEX`	Writes a comma-separated pair of 64-bit floating point numbers that represent a complex number. The first number represents the real part and the second number represents the imaginary part.
`PCOMPLEX`	Writes a comma-separated pair of 64-bit floating point numbers that represent a complex number. The first number represents the magnitude and the second number represents the phase angle in the phase units specified in the transaction.
`COORD`	Writes a comma-separated series of 64-bit floating point numbers that represent a rectangular coordinate.
`TIME STAMP`	Converts a real number (for example, the output of the `now()` function) to a meaningful form and writes it in a variety of combinations of year, month, day, and time.

DEFAULT Format

`WRITE TEXT` (default) transactions are of this form:

`WRITE TEXT` *ExpressionList*

ExpressionList is a single expression or a comma-separated list of expressions.

The transaction converts each item in *ExpressionList* to a meaningful string and writes it. Consider the simple case of writing the scalar variable `X`:

```
WRITE TEXT X
```

Figure E-1. A `WRITE TEXT` Transaction

If `X` in Figure E-1 contains text, such as:

```
bird cat dog
```

then no conversion is performed and the transaction writes exactly 12 characters.

If `X` in Figure E-1 contains a scalar Integer, such as:

```
8923
```  *the value of X (decimal notation)*

then the numeric value is converted to text and HP VEE writes exactly four characters.

If `X` in Figure E-2 contains a scalar real value, such as:

```
1.2345678901234567
```

Figure E-2. Numeric Data

then each significant digit up to 16 significant digits is written. (The least significant digit is approximate because of the conversion between HP VEE's internal binary form and decimal notation).

For example, if you write the data in Figure E-2 using this transaction:

```
WRITE TEXT a EOL
```

then HP VEE writes this:

```
1.234567890123457
```

If the absolute value of the number is sufficiently large or small, exponential notation is used. The Reals that form the sub-elements of Coord, Complex, and PComplex behave the same way.

If EOL ON is specified for any WRITE TEXT DEFAULT transaction, the character specified in the EOL Sequence field for that object is written following the last character in *ExpressionList*.

STRING Format

WRITE TEXT STRING transactions are of this form:

```
WRITE TEXT ExpressionList STR
```

ExpressionList is a single expression or a comma-separated list of expressions.

WRITE TEXT STRING transactions behave basically the same as WRITE TEXT (default) transactions (one exception will be discussed). The significant difference is that STRING allows you to specify additional details about output formatting including field width, justification, and number of characters.

Field Width and Justification. If a transaction specifies DEFAULT FIELD WIDTH, only those characters resulting from the conversion of items within *ExpressionList* to Text are written.

If a transaction specifies FIELD WIDTH: F, then the converted Text is written right- or left-justified within a space F characters wide.

The transactions in Figure E-3 specify that all characters are to be written within a field of twenty characters with left justification.

```
WRITE TEXT X STR FW:20 LJ EOL
WRITE TEXT Y STR FW:20 LJ EOL
```

Figure E-3. Two WRITE TEXT STRING Transactions

If `X` and `Y` in Figure E-3 have these values:

```
bird cat dog          the Text value of X
12345678901234567     the Real value of Y
```

then HP VEE writes this:

```
bird cat dog
12345678901234567
^                  ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the right of `dog` and to the right of the second `7` are spaces (ASCII 32 decimal).

If justification is changed to `RIGHT JUSTIFY`, then the transactions appear as shown in Figure E-4.

```
WRITE TEXT X STR FW:20 RJ EOL
WRITE TEXT Y STR FW:20 RJ EOL
```

Figure E-4. Two `WRITE TEXT STRING` Transactions

If `X` and `Y` in Figure E-4 have these values:

```
bird cat dog          the Text value of X
12345678901234567     the Real value of Y
```

then HP VEE writes this:

```
          bird cat dog
   12345678901234567
^                  ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the left of `bird` and to the left of the first `1` are spaces (ASCII 32 decimal).

If the length of a string exceeds the specified field width, the entire string is written. The field width specification never truncates; only `MAX NUM CHARS` can truncate characters.

The transaction in Figure E-5 specifies that all characters are to be written in a field width of four characters with left justification.

```
WRITE TEXT X STR FW:4 LJ
```

Figure E-5. A `WRITE TEXT STRING` Transaction

If `X` in Figure E-5 has this value:

`bird cat dog` *the Text value of X, 12 characters*

then HP VEE writes this:

`bird cat dog` *all 12 characters*

Even though the specified field width is four characters, the transaction writes all twelve characters of the string.

Number of Characters. If you specify `ALL CHARS`, then all of the characters generated by the conversion of each item in *ExpressionList* are written. If you specify `MAX NUM CHARS:` *M*, then only the first *M* characters of each item in *ExpressionList* are written.

The transactions in Figure E-6 specify that a maximum of seven characters are written in each field, the field width is twenty characters, and field entries are left justified.

```
WRITE TEXT X STR:7 FW:20 LJ EOL
WRITE TEXT Y STR:7 FW:20 LJ EOL
```

Figure E-6. Two `WRITE TEXT STRING` Transactions

If `X` and `Y` in Figure E-3 have these values:

```
bird cat dog          the Text value of X
12345678901234567     the Real value of Y
```

then HP VEE writes this:

```
bird ca
1234567
^                   ^
```

Notice that the numeric value of `Y` is first converted to Text and characters are truncated. Numeric values are not rounded by `MAX NUM CHARS`.

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the right of `bird` and to the right of the first `1` are spaces (ASCII 32 decimal).

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

Note

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

QUOTED STRING Format

WRITE TEXT QUOTED STRING transactions are of this form:

```
WRITE TEXT ExpressionList QSTR
```

ExpressionList is a single expression or a comma-separated list of expressions.

In general, the behaviors previously discussed for the STRING format apply to QUOTED STRING format. There are two differences between STRING and QUOTED STRING:

- For QUOTED STRING, a double quote (ASCII 34 decimal) is added to the beginning and the end of the string. Note that the double quotes are applied before any padding spaces are added to justify the string within the specified field width.
- Control characters (ASCII 0-31 decimal), escape characters (Table E-5), and the characters ' (ASCII 39 decimal) and " (ASCII 34 decimal) embedded inside a double-quoted string receive special treatment.

Field Width and Justification. If you specify DEFAULT FIELD WIDTH, only those characters resulting from the conversion of items within *ExpressionList* to Text and the surrounding double quotes are written.

If you specify FIELD WIDTH: *F*, then the converted Text and the surrounding quotes are written right or left justified within a space *F* characters wide.

The transactions in Figure E-7 specify that all characters are to be written as quoted strings in a field 20 characters wide with left justification.

```
WRITE TEXT X QSTR FW:20 LJ EOL
WRITE TEXT Y QSTR FW:20 LJ EOL
```

Figure E-7. Two WRITE TEXT QUOTED STRING Transactions

If X and Y in Figure E-7 have these values:

```
bird cat dog          the Text value of X
12345678901234567     the Real value of Y
```

then HP VEE writes this:

```
"bird cat dog"
"12345678901234567"
^                    ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the right of dog" and to the right of 7" are spaces (ASCII 32 decimal).

If justification is changed to RIGHT JUSTIFY, then the transactions appear as shown in Figure E-8.

```
WRITE TEXT X QSTR FW:20 RJ EOL
WRITE TEXT Y QSTR FW:20 RJ EOL
```

Figure E-8. Two WRITE TEXT QUOTED STRING Transactions

If X and Y in Figure E-8 have these values:

```
bird cat dog          the Text value of X
12345678901234567     the Real value of Y
```

then HP VEE writes this:

```
      "bird cat dog"
 "12345678901234567"
^                    ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the left of "bird and to the left of "1 are spaces (ASCII 32 decimal).

If the length of a string exceeds the specified field width, the entire string is output. The field width specification never truncates strings that are written; only MAX NUM CHARS can truncate characters.

The transactions in Figure E-9 that specifies that all characters are to be written within a field of four characters with left justification.

```
WRITE TEXT X QSTR FW:4 LJ
```

Figure E-9. A WRITE TEXT QUOTED STRING Transaction

If X in Figure E-9 has this value:

`bird cat dog` *the Text value of X, 12 characters*

then HP VEE writes this:

`"bird cat dog"` *all 12 characters*

Number of Characters. If you specify ALL CHARS, then all of the characters generated by the conversion of each item in *ExpressionList* as well as the surrounding double quotes are written. If you specify MAX NUM CHARS: *M*, then only the first *M* characters of each item in *ExpressionList plus* the surrounding double quotes are written. In other words, a total of *M*+2 characters are written for each item in *ExpressionList*.

The transaction in Figure E-10 that specifies MAX NUM CHARS:7 (field width 20, left justified).

```
WRITE TEXT X QSTR:7 FW:20 LJ EOL
WRITE TEXT Y QSTR:7 FW:20 LJ EOL
```

Figure E-10. Two WRITE TEXT QUOTED STRING Transactions

If `X` and `Y` in Figure E-10 have these values:

```
bird cat dog          the Text value of X
12345678901234567     the Real value of Y
```

then HP VEE writes this:

```
"bird ca"
"1234567"
^                    ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the right of `ca"` and to the right of `7"` are spaces (ASCII 32 decimal).

Embedded Control and Escape Characters. In this discussion, the terms **control character** and **escape character** have specific meaning. A control character is a single byte of data corresponding to one of the ASCII characters 0-31 decimal. For example, linefeed is ASCII 10 decimal and the symbol `<LF>` denotes linefeed character in this discussion. The string `\n` is a human-readable escape character representing linefeed that is recognized by HP VEE. HP VEE uses escape characters to represent control characters within quoted strings.

Table E-5. Escape Characters

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

Consider the effects of various embedded escape characters on the transaction in Figure E-11.

```
WRITE TEXT X QSTR EOL
```

Figure E-11. A `WRITE TEXT QUOTED STRING` Transaction

If `X` in Figure E-11 has this value:

```
bird\ncat dog
```

then HP VEE writes this to UNIX paths:

```
"bird\ncat dog"
```

For the same transaction and data, HP VEE-Test writes this to direct I/O paths:

```
"bird<LF>cat dog"
```

Note that `<LF>` means the single character, linefeed (ASCII 10 decimal).

If `X` in Figure E-11 has this value:

```
bird \"cat\" dog
```

then HP VEE writes this to UNIX paths and direct I/O paths for serial interfaces:

```
"bird \"cat\" dog"
```

For the same transaction and data, HP VEE-Test writes this to direct I/O paths for HP-IB interfaces:

```
"bird ""cat"" dog"
```

This unique behavior for HP-IB interfaces is provided to support the requirements of IEEE 488.2.

INTEGER Format

`WRITE TEXT INTEGER` transactions are of this form:

`WRITE TEXT` *ExpressionList* `INT`

ExpressionList is a single expression or a comma-separated list of expressions.

The type of integer generated by this transaction is a 32-bit two's complement integer. The range of these integers is `2 147 483 647` to `-2 147 483 648`. The only characters written to represent these numbers are `+-0123456789`.

HP VEE attempts to convert each item in *ExpressionList* to the Int32 data type before converting it to Text for final formatting. HP VEE follows the usual conversion rules; refer to "Converting Data Types on Input Terminals" in Chapter 3 for details.

If a Real is written using `INTEGER` format:

- Real values outside the valid range of Int32 generate an error.
- Real values within the valid range of Int32 are converted by truncating the fractional portion of the Real.

Number of Digits. If you specify `DEFAULT NUM DIGITS`, the transaction writes only the digits required to express the value of the integer; leading zeros are not used.

If you specify `MIN NUM DIGITS:` *M*, the transaction pads the output with leading zeros as required to give a total of exactly *M* digits.

Consider the two transactions in Figure E-12 which differ only in their specification for the number of output digits.

```
WRITE TEXT X INT EOL       default number of digits
WRITE TEXT X INT:6 EOL     six digits
```

Figure E-12. Two `WRITE TEXT INTEGER` Transactions

If `X` in Figure E-12 has this value:

```
4567
```

then HP VEE writes this:

```
4567
004567
```

MIN NUM DIGITS never causes truncation of the output string. The transaction in Figure E-13 specifies the minimum number of digits to be 1.

```
WRITE TEXT X INT:1 EOL
```

Figure E-13. A WRITE TEXT INTEGER Transaction

If X in Figure E-13 has a value of:

```
12345678
```

then HP VEE writes this:

```
12345678    all eight digits
```

Sign Prefixes. You may optionally specify one of the sign prefixes listed in Table E-6 as part of a WRITE TEXT INT transaction.

Table E-6. Sign Prefixes

Prefix	Description
/-	Positive numbers are written with no prefix, neither a + nor a space. All negative numbers are written with a - prefix.
+/-	All positive numbers are written with a + prefix. All negative numbers are written with a - prefix.
" "/-	All positive numbers are written with a space (ASCII 32 decimal) prefix. All negative numbers are written with a - prefix.

Any prefixed signs do not count towards MIN NUM DIGITS. The transaction shown in Figure E-14 specifies explicit leading signs for positive and negative numbers.

```
WRITE TEXT X INT:6 SIGN:"+/-" EOL
WRITE TEXT Y INT:6 SIGN:"+/-" EOL
```

Figure E-14. Two WRITE TEXT INTEGER Transactions

If X and Y in Figure E-14 have values of:

```
123    the Integer value of X
-123   the Integer value of Y
```

then HP VEE writes this:

```
+000123    six digits plus sign
-000123
```

OCTAL Format

WRITE TEXT OCTAL transactions are of this form:

```
WRITE TEXT ExpressionList OCT
```

ExpressionList is a single expression or a comma-separated list of expressions.

The type of integer written by this transaction is a 32-bit two's complement integer. The range of these integers is 2 147 483 647 to -2 147 483 648. The only characters written to represent these octal numbers are 01234567. An optional prefix may be specified which may include other characters.

HP VEE attempts to convert any data written using OCTAL format to the Int32 data type before converting it to Text for final formatting. The usual HP VEE conversion rules are followed.

If a Real is written using OCTAL format:

- Real values outside the valid range of Int32 generate an error.
- Real values within the valid range of Int32 are converted by truncating the fractional portion of the Real.

Number of Digits. The behavior of `DEFAULT NUM DIGITS` and `MIN NUM DIGITS` is the same as described previously in the "Number of Digits" section for `WRITE TEXT INTEGER` transactions.

Octal Prefixes. You may specify one of the prefixes listed in Table E-7 as part of a `WRITE TEXT OCTAL` transaction.

Table E-7. Octal Prefixes

Prefix	Description
`NO PREFIX`	HP VEE writes each octal number without any prefix; only the digits `01234567` appear in the output.
`DEFAULT PREFIX`	For direct I/O paths, HP VEE prefixes each octal number with `#Q`. This supports the octal Non-Decimal Numeric data format defined by IEEE 488.2. For UNIX paths, HP VEE prefixes each octal number with a `0` (zero). If leading zeros are added to achieve the specified `MIN NUM DIGITS`, `DEFAULT PREFIX` will not add additional leading zeros.
`PREFIX:` *string*	HP VEE prefixes each octal number with the characters specified in *string*.

The transaction in Figure E-15 specifies the default prefix and six digits:

```
WRITE TEXT X OCT:6 PREFIX EOL
```

Figure E-15. A `WRITE TEXT OCTAL` Transaction

If `X` in Figure E-15 has this value:

`15` *the value 15 decimal*

then HP VEE-Test writes this to direct I/O paths:

`#Q000017` *exactly six digits plus prefix*

Using the same transaction and data, HP VEE writes this to UNIX paths:

```
000017    exactly six digits
```

The transaction in Figure E-16 specifies a custom prefix and ten digits:

```
WRITE TEXT X OCT:10 PREFIX:"oct>" EOL
```

Figure E-16. A `WRITE TEXT OCTAL` Transaction

If `X` in Figure E-16 has this value:

```
15    the Integer value 15 decimal
```

then HP VEE writes this to UNIX paths and direct I/O paths:

```
oct>000017
```

Note that the prefix written by `DEFAULT PREFIX` depends on the destination, but the prefix written by `PREFIX:` *string* is independent of the destination.

HEX Format

`WRITE TEXT HEX` transactions are of this form:

`WRITE TEXT` *ExpressionList* `HEX`

The type of integer written by this transaction is a 32-bit two's complement integer. The range of these integers is `2 147 483 647` to `-2 147 483 648`. The only characters written to represent these hexadecimal numbers are `0123456789abcdef`. An optional prefix may be specified that may include other characters.

The behavior of `WRITE TEXT HEX` is nearly identical to that of `WRITE TEXT OCTAL`. The only difference is the set of prefixes available and the behavior of `DEFAULT PREFIX`.

Hexadecimal Prefixes. You may specify one of the prefixes listed in Table E-8 as part of a `WRITE TEXT HEX` transaction.

Table E-8. Hexadecimal Prefixes

Prefix	Description
`NO PREFIX`	HP VEE writes each hexadecimal number without any prefix; only the digits `0123456789abcdef` appear in the output.
`DEFAULT PREFIX`	For direct I/O paths, HP VEE prefixes each hexadecimal number with `#H`. This supports the hexadecimal Non-Decimal Numeric data format defined by IEEE 488.2. For UNIX paths, HP VEE prefixes each hexadecimal number with `0x`.
`PREFIX:` *string*	HP VEE prefixes each hexadecimal number with the characters specified in *string*.

The transaction in Figure E-17 specifies the default prefix and six digits:

```
WRITE TEXT X HEX:6 PREFIX EOL
```

Figure E-17. A `WRITE TEXT HEX` Transaction

If `X` in Figure E-15 has this value:

`15` *the Integer value 15 decimal*

then HP VEE-Test writes this to direct I/O paths:

`#H00000f` *exactly six digits plus prefix*

Using the same transaction and data, HP VEE this to UNIX paths:

`0x00000f` *exactly six digits plus prefix*

The transaction in Figure E-18 specifies a custom prefix and three digits:

```
WRITE TEXT X HEX:3 PREFIX:"hex>" EOL
```

Figure E-18. A `WRITE TEXT HEX` Transaction

If `X` in Figure E-18 has this value:

`15` *the Integer value 15 decimal*

then HP VEE writes this to UNIX paths and direct I/O paths:

`hex>00f` *exactly three digits plus prefix*

Note that the prefix written by `DEFAULT PREFIX` depends on the destination, but the prefix written by `PREFIX:` *string* is independent of the destination.

REAL Format

`WRITE TEXT REAL` transactions are of this form:

`WRITE TEXT` *ExpressionList* `REAL`

The type of Real number generated by this transaction is a 64-bit IEEE 754 floating-point number. The range of these numbers is:

```
-1.797 693 134 862 315E+308
-2.225 073 858 507 202E-307
 0
 2.225 073 858 507 202E-307
 1.797 693 134 862 315E+308
```

The only characters written to represent these numbers are `+-.0123456789E`.

Notations and Digits. You may optionally specify one of the notations in Table E-9 as part of a `WRITE TEXT REAL` transaction.

Table E-9. `REAL` Notations

Notation	Description
`STANDARD`	HP VEE automatically determines whether each Real value should be written in fixed-point notation (decimal points as required, no exponents) or in exponential notation. Non-significant zeros are never written.
`FIXED`	HP VEE writes each Real value as a fixed-point number. Numbers with fractional digits are automatically rounded to fit the number of fractional digits specified by `NUM FRACT DIGITS`. Trailing zero digits are added as required to give the specified number of fractional digits.
`SCIENTIFIC`	HP VEE writes each Real value using exponential notation. Each exponent includes an explicit sign (`+` or `-`) and the upper-case `E` is always used. Numbers with fractional digits are automatically rounded to fit the number of fractional digits specified by `NUM FRACT DIGITS`. Trailing zero digits are added as required to give the specified number of fractional digits.

The transactions in Figure E-19 specify `STANDARD` notation and four significant digits.

```
WRITE TEXT X REAL STD:4 EOL
WRITE TEXT Y REAL STD:4 EOL
WRITE TEXT Z REAL STD:4 EOL
```

Figure E-19. Three `WRITE TEXT REAL` Transactions

If X, Y, and Z in Figure E-19 have these values:

1.23456E2	*the Real value of X*
1.23456E09	*the Real value of Y*
1.23	*the Real value of Z*

then HP VEE writes this:

123.5	*mantissa rounded as required*
1.235E+09	*large numbers in exponential notation*
1.23	*never any trailing zeros*

The transactions in Figure E-20 specify FIXED notation and four fractional digits.

```
WRITE TEXT X REAL FIX:4 EOL
WRITE TEXT Y REAL FIX:4 EOL
WRITE TEXT Z REAL FIX:4 EOL
```

Figure E-20. Three WRITE TEXT REAL Transactions

If X, Y, and Z in Figure E-20 have these values:

1.2345678E2	*the Real value of X*
1.2345678E-09	*the Real value of Y*
1.23	*the Real value of Z*

then HP VEE writes this:

123.4568	*mantissa rounded as required*
0.0000	*small numbers round to zero*
1.2300	*trailing zeros added as required*

The transactions in Figure E-21 specify SCIENTIFIC notation and four fractional digits.

```
WRITE TEXT X REAL SCI:4 EOL
WRITE TEXT Y REAL SCI:4 EOL
WRITE TEXT Z REAL SCI:4 EOL
```

Figure E-21. Three WRITE TEXT REAL Transactions

If X, Y, and Z in Figure E-21 have these values:

1.2345678E2	*the Real value of X*
-1.2345678E-09	*the Real value of Y*
0	*the Real value of Z*

then HP VEE writes this:

1.2346E+02	*exponent is E plus two signed digits*
-1.2346E-09	*last digit rounded as required*
0.0000E+00	*trailing zeros padded as required*

COMPLEX, PCOMPLEX, and COORD Formats

COMPLEX, PCOMPLEX, and COORD correspond to the HP VEE multi-field data types with the same names. The behavior of all three formats is very similar. The behaviors described in this section apply to all three formats except as noted.

Just as the HP VEE data types Complex, PComplex, and Coord are composed of multiple Real numbers, the COMPLEX, PCOMPLEX, and COORD formats are essentially compound forms of the REAL format. Each constituent Real value of the multi-field data types is written with the same output rules that apply to an individual REAL formatted value.

The final output of transactions involving multi-field formats is affected by the `Multi-Field Format` setting for the object in question. `Multi-Field Format` is accessed via `I/O` ⟹ `Configure I/O` for `Direct I/O` objects and via `Config` in the object menu for all other objects. The two possible settings for `Multi-Field Format` are:

- `Data Only`. This writes multi-field data formats as a list of comma-separated numbers *without* parentheses.
- `( ... ) Syntax`. This writes multi-field data formats as a list of comma-separated numbers grouped by parentheses.

Subsequent examples will illustrate these behaviors.

COMPLEX Format. `WRITE TEXT COMPLEX` transactions are of this form:

`WRITE TEXT` *ExpressionList* `CPX`

The transaction in Figure E-22 specifies a fixed-decimal notation, explicit leading signs, a field width of 10 characters, and right justification.

```
WRITE TEXT X CPX FIX:3 SIGN:"+/-" FW:10 RJ EOL
```

Figure E-22. A `WRITE TEXT COMPLEX` Transaction

If the `Multi-Field Format` is set to `( ... ) Syntax`, and `X` in Figure E-22 has this value:

```
( -1.23456 , 9.8 )    the Complex value of X
```

then HP VEE writes this:

```
(    -1.235 ,     +9.800)
 ^         ^   ^         ^
```

If the `Multi-Field Format` is set to `Data Only` and `X` in Figure E-22 has the same value, then HP VEE writes this:

```
     -1.235,    +9.800
^         ^ ^         ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the left of + are spaces (ASCII 32 decimal).

Note that with `( ... ) Syntax`, a space-comma-space sequence separates the ten-character wide fields that contain the real and imaginary parts of the Complex number. With either `Multi-Field Format` there is a separate ten-character field for both the real and the imaginary part. Neither parentheses nor the separating comma and spaces are included in the field.

PCOMPLEX Format. `WRITE TEXT PCOMPLEX` transactions are of this form:

`WRITE TEXT` *ExpressionList* `PCX`

`PCOMPLEX` format allows you to specify the phase units for the polar complex number it writes. Note that phase units are independent of the units set by `File` ⟹ `Preferences` ⟹ `Trig Mode`.

Table E-10. `PCOMPLEX` Phase Units

Unit	Description
`DEG`	Degrees
`RAD`	Radians
`GRAD`	Gradians

The first transaction in Figure E-23 specifies phase measurement in degrees, and the second transaction specifies phase measurement in radians.

```
WRITE TEXT X PCX:DEG STD EOL
WRITE TEXT X PCX:RAD STD EOL
```

Figure E-23. Two `WRITE TEXT PCOMPLEX` Transactions

If the Multi-Field Format is set to Data Only, and X in Figure E-23 has this value:

```
(-1.23456 , @90)    the PComplex value of X, phase in degrees
```

then HP VEE writes this:

```
1.23456,-90
1.23456,-1.570796326794897
```

The transaction in Figure E-24 specifies phase measurement in radians, fixed-decimal notation, three fractional digits, explicit leading signs, a field width of ten characters, and right justification.

```
WRITE TEXT X PCX:RAD FIX:3 SIGN:"+/-" FW:10 RJ EOL
```

Figure E-24. A WRITE TEXT PCOMPLEX Transaction

If the Multi-Field Format is set to (...) Syntax, and X in Figure E-24 has this value:

```
(-1.23456 , @9.8)    the PComplex value of X,  angle in radians
```

then HP VEE writes this:

```
(    +1.235 , @    +0.375)
 ^         ^    ^         ^
```

Note that HP VEE normalizes all PComplex numbers to yield a positive magnitude and a phase between $+\pi$ and $-\pi$.

If the Multi-Field Format is set to Data Only, and X in Figure E-24 has the same value, then HP VEE writes this:

```
      +1.235,    +0.375
^         ^ ^         ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the left of - and to the left of + are spaces (ASCII 32 decimal).

COORD Format. `WRITE TEXT COORD` transactions are of this form:

`WRITE TEXT` *ExpressionList* `COORD`

`COORD` format has all the same behaviors of `COMPLEX` format. The only difference is that `COORD` may contain an arbitrary number of fields while `COMPLEX` has exactly two fields.

TIME STAMP Format

`WRITE TEXT TIME STAMP` transactions are of this form:

`WRITE TEXT` *ExpressionList* *[*`DATE:` *DateSpec]* *[*`TIME:` *TimeSpec]*

ExpressionList is a single expression or a comma-separated list of expressions.

DateSpec is one of the following pre-defined date and time combinations:

- `Date`
- `Time`
- `Date&Time`
- `Time&Date`
- `Delta Time`

If you specify a transaction that includes `Date`, you may also specify a *DateSpec* of `Weekday DD/Month/YYYY` or `DD/Month/YYYY`.

If you specify a transaction that includes `Time`, you may also specify a *TimeSpec*. *TimeSpec* is a combination of the following pre-defined time formats:

- `HH:MM` (hours and minutes)
- `HH:MM:SS` (hours, minutes, and seconds)
- `12 HOUR`
- `24 HOUR`

Each item in *ExpressionList* is converted to a Real and interpreted as a date and time. This Real number represents the number of seconds that have elapsed since midnight, January 1, AD 1 UTC. The most common source for this Real number is the output of a `Time Stamp` object. You use the `TIME`

STAMP format to convert this Real number to a meaningful string that contains a human-readable date and/or time.

TIME STAMP supports a variety of notations for writing dates and times. If a Real variable contains this value:

```
62806574669.31164
```

then TIME STAMP can write it using any of these Time and Date notations:

Notation	Result
Date with Weekday DD/Month/YYYY	Thu 04/Apr/1991
Time with HH:MM:SS and 24 HOUR	15:44:29
Date&Time with Weekday DD/Month/YYYY, HH:MM:SS, and 24 HOUR	Thu 04/Apr/1991 15:44:29
Time&Date with HH:MM:SS, 24 HOUR, and Weekday DD/Month/YYYY	15:44:29 Thu 04/Apr/1991
Delta Time with HH:MM:SS	17446270:44:29
Date with Weekday DD/Month/YYYY	Thu 04/Apr/1991
Date with DD/Month/YYYY	04/Apr/1991
Time with HH:MM:SS and 24 HOUR	15:44:29
TIME with HH:MM and 24 HOUR	15:44
TIME with HH:MM:SS and 24 Hour	15:44:29
TIME with HH:MM:SS and 12 Hour	3:44:29 PM

BYTE Encoding

`BYTE` transactions are of this form:

`WRITE BYTE` *ExpressionList*

ExpressionList is a single expression or a comma-separated list of expressions.

HP VEE converts each item in *ExpressionList* to an Int16 (16-bit two's complement integer) and writes the least-significant 8-bits. This is a transaction for writing single characters to a device. Each expression in *ExpressionList* must be a scalar.

The transactions in Figure E-25 produce the output shown in Figure E-26.

```
WRITE BYTE 65,66,67
WRITE BYTE 65+1024,65+2048
```

Figure E-25. Two `WRITE BYTE` Transactions

```
ABCAA
```

Figure E-26. Character Data

CASE Encoding

`WRITE CASE` transactions are of this form:

`WRITE CASE` *ExpressionList1* `OF` *ExpressionList2*

ExpressionList is a single expression or a comma-separated list of expressions.

HP VEE converts each item in *ExpressionList1* to an integer and uses it as an index into *ExpressionList2*. The indexed item(s) in *ExpressionList2* are written in a string format that is the same as `WRITE TEXT` (default).

Note that the indexing of items in *ExpressionList2* is zero-based.

The transactions in Figure E-27 illustrate the behavior of CASE format.

```
WRITE CASE 2,1  OF "Str0","Str1","Str2"
WRITE CASE X  OF  1,1+A,3+A
```

Figure E-27. Two WRITE CASE Transactions

If the variables in Figure E-27 have these values:

2 *the Real value of X*
0.1 *the Real value of A*

then HP VEE writes this:

```
Str2Str1
3.1
```

BINARY Encoding

WRITE BINARY transactions are of this form:

WRITE BINARY *ExpressionList DataType*

ExpressionList is a single expression or a comma-separated list of expressions.

*DataType*s is one of the following pre-defined HP VEE data types:

- BYTE - 8-bit byte
- INT16 - 16-bit two's complement integer
- INT32 - 32-bit two's complement integer
- REAL32 - 32-bit IEEE 754 floating-point number
- REAL64 - 64-bit IEEE 754 floating-point number
- STRING - null terminated string
- COMPLEX - equivalent to two REALs
- PCOMPLEX -equivalent to two REALs
- COORD - equivalent to two or more REALs

Note

HP VEE stores and manipulates all integer values as the `INT32` data type, and all real numbers as the `Real` data type, also known as `REAL64`. Thus, the `INT16` and `REAL32` data types are provided for I/O only. HP VEE-Test performs the following data-type conversions for instrument I/O:

- On an output transaction `INT32` values are individually converted to `INT16` values, which are output to the instrument. However, since the `INT16` data type has a range of -32768 to 32767, values outside this range will be truncated to 16 bits.
- On an output transaction `REAL64` values are individually converted to `REAL32` values, which are output to the instrument. However, since the `REAL32` data type has a smaller range than `REAL64` data type, values outside this range cannot be converted to `REAL32` and will result in an error.

`BINARY` encoded transactions convert each of the values specified in *ExpressionList* to the HP VEE data type specified by *DataType*. Each converted item is then written in the specified binary format. However, since the binary data written is a copy of the representation in computer memory, it is not easily shared by different computer architectures or hardware.

`BINARY` encoded data has the advantage of being very compact. `READ BINARY` transactions can read any corresponding `WRITE BINARY` data.

Note that `BINARY` encoding writes only the numeric portion of each data type. For example, the parentheses and comma that can be included when writing Complex and Coord data with `TEXT` encoding are never written with `BINARY` encoding. Similarly, when writing arrays, `BINARY` encoding does not write any `Array Separators`. `WRITE BINARY` transactions do allow you to specify `EOL ON`. There is rarely a need to write `EOL` with `BINARY` transactions because numeric data types are of fixed length and strings are null-terminated.

BINBLOCK Encoding

`WRITE BINBLOCK` transactions are of this form:

`WRITE BINBLOCK` *ExpressionList DataType*

ExpressionList is a single expression or a comma-separated list of expressions.

DataType is one of these pre-defined HP VEE data types:

- `BYTE` - 8-bit byte
- `INT16` - 16-bit two's complement integer
- `INT32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating-point number
- `REAL64` - 64-bit IEEE 754 floating-point number
- `COMPLEX` - equivalent to two `REAL`s
- `PCOMPLEX` -equivalent to two `REAL`s
- `COORD` - equivalent to two or more `REAL`s

`BINBLOCK` writes *each item* in *ExpressionList* as a separate data block. The block header used depends on the type of object performing the `WRITE` and the object's configuration.

Non-HP-IB BINBLOCK

If the object is *not* `Direct I/O` to HP-IB, a `WRITE BINBLOCK` always writes an IEEE 488.2 Definite Length Arbitrary Block Response Data block. This data format is primarily used for communicating with HP-IB instruments using `Direct I/O`, although it is supported by other objects.

Each Definite Length Arbitrary Block is of the form:

```
#<Num_digits><Num_bytes><Data>
```

where:

`#` is literally the `#` character as shown.

`<Num_digits>` is an ASCII character that is a single digit (decimal notation) indicating the number of digits in `<Num_bytes>`.

`<Num_bytes>` is a list of ASCII characters that are digits (decimal notation) indicating the number of bytes that follow in `<Data>`.

`<Data>` is a sequence of arbitrary 8-bit data bytes.

HP-IB BINBLOCK

If the object is `Direct I/O` to HP-IB, the behavior of `WRITE BINBLOCK` transactions depends upon the `Direct I/O Configuration` settings for `Conformance` and `Binblock`; these settings are accessed via the `I/O` $\Longrightarrow$ `>Configure I/O` menu selection.

If `Conformance` is set to `IEEE 488.2`, then `WRITE BINBLOCK` *always* writes an IEEE 488.2 Definite Length Arbitrary Block Response Data block.

If `Conformance` is set to `IEEE 488`, then the type of header used depends on `Binblock`. `Binblock` may specify IEEE 728 `#A`, `#T`, or `#I` block headers. If `Binblock` is `None`, `WRITE BINBLOCK` writes an IEEE 488.2 Definite Length Arbitrary Block Response Data block.

IEEE 728 block headers are of the following forms:

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

where:

`#` is the character as shown.

`A`,`T`, `I` are the characters as shown.

`<Byte_Count>` consists of two bytes which together form a 16-bit unsigned integer that specifies the number of bytes that follow in `<Data>`. (HP VEE calculates this automatically.)

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

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

CONTAINER Encoding

`WRITE CONTAINER` transactions are of this form:

`WRITE CONTAINER` *ExpressionList*

ExpressionList is a single expression or a comma-separated list of expressions.

A `WRITE CONTAINER` transaction writes each item in *ExpressionList* using a special HP VEE text representation.

This representation retains all the HP VEE attributes associated with the data type written, such as shape, size, and name. Any `WRITE CONTAINER` data can be retrieved without any loss of information using `READ CONTAINER`.

For example, this transaction:

```
WRITE CONTAINER 1.2345
```

writes this:

```
(Real
 (data 1.2345)
)
```

If DEFAULT formatting is specified, HP VEE writes a container of the same data type as each item in *ExpressionList*.

If a format is specified, HP VEE converts each item in *ExpressionList* to the specified format using the standard data type conversion rules. The converted data is written as a container of the specified type. Please refer to "Converting Data Types on Input Terminals" in Chapter 3 for details about automatic conversion of data types.

In general, it is best to use DEFAULT formatting. If a format is specified, conversion of incoming data may produce undesireable results.

STATE Encoding

STATE encoding is supported by HP VEE-Test only.

WRITE STATE transactions are of the form:

WRITE STATE *[DownloadString]*

DownloadString is an optional string that allows you to specify a download string if you have not previously specified one in the direct I/O configuration for the corresponding instrument. This explained in greater detail in the sections that follow.

WRITE STATE transactions are used by Direct I/O objects to download a learn string to an instrument. There is exactly one learn string associated with each instance of a Direct I/O object. This learn string is uploaded by clicking on Upload in the Direct I/O object menu. The learn string contains the null string before Upload is selected for the first time.

The behavior of WRITE STATE is affected by the Direct I/O Configuration settings for Conformance and Download String. These settings are accessed via the I/O $\Longrightarrow$ Configure I/O menu selection. If Conformance is IEEE 488, the WRITE STATE transaction writes the Download String followed by the learn string. If Conformance is IEEE 488.2, the learn string is downloaded without any prefix as defined by IEEE 488.2. Please refer to "WRITE STATE Transactions" in Chapter 12 for a detailed example using learn strings.

REGISTER Encoding

REGISTER encoding is supported by HP VEE-Test only.

WRITE REGISTER is used to write values into a VXI device's A16 memory.

WRITE REGISTER transactions are of this form:

```
WRITE REG: SymbolicName ExpressionList INCR
-or-
WRITE REG: SymbolicName ExpressionList
```

where:

SymbolicName is a name defined during configuration of a VXI device. The name refers to a specific address within a device's register space. Specific data types for WRITE REGISTER transactions are:

- BYTE - 8 bit byte
- WORD16 - 16-bit two's complement integer
- WORD32 - 32-bit two's complement integer
- REAL32 - 32-bit IEEE 754 floating point number

These data types are also specified during configuration of a VXI device and do not appear in the transaction.

ExpressionList is a single expression or a comma-separated list of expressions.

INCR specifies that array data is to be written incrementally starting at the register address specified by *SymbolicName*. The first element of the array is written at the starting address, the second at that address plus an offset equal to the length in bytes of the data type, and so forth until all array elements have been written. If INCR is not specified in the transaction, the entire array is written to the single location specified by *SymbolicName*.

MEMORY Encoding

`MEMORY` encoding is supported by HP VEE-Test only.

`WRITE MEMORY` is used to write values into a VXI device's A24 or A32 memory.

`WRITE MEMORY` transactions are of this form:

```
WRITE MEM: SymbolicName ExpressionList INCR
-or-
WRITE MEM: SymbolicName ExpressionList
```

where:

SymbolicName is a name defined during configuration of a VXI device. The name refers to a specific address within a device's extended memory. Specific data types for `WRITE MEMORY` transactions are:

- `BYTE` - 8 bit byte
- `WORD16` - 16-bit two's complement integer
- `WORD32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating point number

These data types are also specified during configuration of a VXI device and do not appear in the transaction.

ExpressionList is a single expression or a comma-separated list of expressions.

`INCR` specifies that array data is to be written incrementally starting at the memory location specified by *SymbolicName*. The first element of the array is written at that location, the second at that location plus an offset equal to the length in bytes of the data type, and so forth until all array elements have been written. If `INCR` is not specified in the transaction, the entire array is written to the single memory location specified by *SymbolicName*.

IOCONTROL Encoding

`IOCONTROL` encoding is supported by HP VEE-Test only.

`WRITE IOCONTROL` transactions are of this form:

```
WRITE IOCONTROL CTL ExpressionList
-or-
WRITE IOCONTROL PCTL ExpressionList
```

ExpressionList is a single expression or a comma-separated list of expressions.

`IOCONTROL` encoding is used only for `Direct I/O` to GPIO interfaces.

This transaction sets the control lines of a GPIO interface:

```
WRITE IOCONTROL CTL a
```

HP VEE-Test converts the value of `a` to an Integer. The least X significant bits of the Integer value are mapped to the control lines of the interface, where X is the number of control lines.

For example, the HP 98622A GPIO interface uses two control lines, `CTL0` and `CTL1`.

Value Written	CTL1	CTL0
0	0	0
1	0	1
2	1	0
3	1	1

In the preceding table, `1` indicates that a control line is asserted, a `0` indicates that it is cleared.

This transaction controls the computer-driven handshake line of a GPIO interface:

```
WRITE IOCONTROL PCTL a
```

If the value of a is non-zero, the PCTL line is set. If the value is zero, no action is taken. PCTL is cleared automatically by the interface when the peripheral meets the handshake requirements.

READ Transactions

Table E-11. READ Encodings and Formats

Encodings	Formats
TEXT	CHAR TOKEN STRING INTEGER OCTAL HEX REAL COMPLEX PCOMPLEX COORD TIME STAMP
BINARY	STR BYTE INT16 INT32 REAL32 REAL64 COMPLEX PCOMPLEX COORD

Table E-11. `READ` Encodings and Formats (continued)

Encodings	Formats

`BINBLOCK`	`BYTE`
	`INT16`
	`INT32`
	`REAL32`
	`REAL64`
	`COMPLEX`
	`PCOMPLEX`
	`COORD`

`CONTAINER`	Not Applicable
`IOSTATUS`	Not Applicable

`REGISTER`[1]	`BYTE`
	`WORD16`
	`WORD32`
	`REAL32`

`MEMORY`[1]	`BYTE`
	`WORD16`
	`WORD32`
	`REAL32`

1 Direct I/O to VXI only (HP VEE-Test only).

TEXT Encoding

`READ TEXT` transactions are generally very easy to use. This is because they are able to read and discard what is irrelevant and selectively read what is important. This works well most of the time, but occasionally you must analyze very carefully what HP VEE considers to be irrelevant and what it considers to be important. This will rarely (if ever) be a problem if you are reading text files written by HP VEE, as long as you read them using the same format used to write them. Problems are most likely to occur when you are trying to import a file from another software application.

Table E-12 describes `READ TEXT` behavior in a general way only; be sure to read all the sections that follow to understand all the possible variations.

Table E-12. Formats for `READ TEXT` Transactions

Format	Description
`CHAR`	Reads *any* 8-bit character.
`TOKEN`	Reads a contiguous list of characters as a unit; this unit is called a token. Tokens are separated by specified delimiter characters (you specify the delimiters). For example, in normal written English, words are tokens and spaces are delimiters.
`STRING`	Reads a list of 8-bit characters as a unit. Most control characters are read and discarded. The end of the string is reached when the specified number of characters has been read, or when a newline character is encountered.
`INTEGER`	Reads a list of characters and interprets them as a decimal or non-decimal representation of an integer. The only characters considered to be part of a decimal `INTEGER` are `0123456789-+`. HP VEE recognizes the prefix `0x` (hex) and all the Non-Decimal Numeric formats specified by IEEE 488.2: `#H` (hex), `#Q` (octal), `#B` (binary).
`OCTAL`	Reads a list of characters and interprets them as the octal representation of an integer. The characters considered to be part of an `OCTAL` are `01234567`. HP VEE also recognizes the IEEE 488.2 Non-Decimal Numeric prefix `#Q` for octal numbers.

Table E-12. Formats for `READ TEXT` Transactions (continued)

Format	Description
`HEX`	Reads a list of characters and interprets them as the hexadecimal representation of an integer. The only characters considered to be part of a `HEX` are `0123456789abcdefABCDEF`. The character combination `0x` is the default prefix; it is not part of the number and is read and ignored. HP VEE also recognizes `0x` and the IEEE 488.2 Non-Decimal Numeric prefix `#H` for hexadecimal numbers.
`REAL`	Reads a list of characters and interprets them as the decimal representation of a Real (floating-point) number. All common notations are recognized including leading signs, signed exponents, and decimal points. The characters recognized to be part of a `REAL` are `0123456789-+.Ee`. HP VEE also recognizes certain characters as suffix multipliers for Real numbers (refer to Table E-13).
`COMPLEX`	Reads the equivalent of two `REAL`s and interprets them as a complex number. The first number read is the real part and the second number read is the imaginary part.
`PCOMPLEX`	Reads the equivalent of two `REAL`s and interprets them as a complex number in polar form. Some engineering disciplines refer to this as "phasor notation". The first number read is considered to be the magnitude and the second is the angle. You may specify units of measure for phase in the transaction.
`COORD`	Reads the equivalent of two or more `REAL`s and interprets them as rectangular coordinates.
`TIME STAMP`	Reads one of the specified HP VEE time stamp formats which represent the calendar date and/or time of day.

General Notes for READ TEXT

Read to End. The `READ TEXT` formats support a choice between reading a specified number of elements or reading until EOF is encountered. In a transaction, *NumElements* is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the first expression is an asterisk (`*`), the transaction will read data until an EOF is encountered. Read to end is supported only for `From File`, `From String`, `From StdIn`, `Execute Program`, `To/From Named Pipe`, and `To/From HP BASIC/UX` transactions. Only the first dimension can have an asterisk rather than a number.

For example, the following transaction, reading from a file:

```
READ TEXT a REAL ARRAY:*,10
```

will read until EOF is encountered resulting in a two dimensional array with ten columns. The number of rows is dependent on the amount of data in the file. The total number of data elements read must be evenly divisible by the product of the known dimension sizes, in this example: 10. If this criteria is not met, an error will occur.

Number of Characters Per READ. These READ TEXT formats support a choice between DEFAULT NUM CHARS and MAX NUM CHARS:

STRING
INTEGER
OCTAL
HEX
REAL

This section discusses the effects of DEFAULT NUM CHARS and MAX NUM CHARS on these formats.

The basic difference between DEFAULT NUM CHARS and MAX NUM CHARS is this:

- DEFAULT NUM CHARS causes HP VEE to read and ignore most characters that do not appear to be part of the number or string it expects.
- MAX NUM CHARS allows you to read *up to* the specified number of 8-bit characters in an attempt to build the type of number or string specified. HP VEE stops reading characters as soon as the READ is satisfied. All characters are read and HP VEE attempts to convert them to the data type specified in the transaction.

If you specify DEFAULT NUM CHARS, the transaction reads as many characters as it requires to fill each variable. Characters that are not meaningful to the specified data type are read and ignored.

If you specify MAX NUM CHARS, HP VEE makes no attempt to sort out characters that are not meaningful to the data type specified. If non-meaningful characters are encountered, they are read and may later generate an error.

In either case, newline and end-of-file are recognized as terminators for strings or numbers. For numeric formats, white space encountered before any significant characters (digits) is read and ignored; after reading significant characters, white space or other non-numeric characters terminate the current READ. These are the general behaviors; read the examples that follow for additional detail.

Consider this example that distinguishes between the behaviors of DEFAULT NUM CHARS and MAX NUM CHARS using INTEGER format. Assume that you are trying to read a file containing this data:

```
bird dog cat 12345 horse
```

It is impossible to extract the integer 12345 from this data with a READ TEXT INTEGER transaction using MAX NUM CHARS no matter how many characters are read. This is because the characters bird dog cat are always read before the digits, they cannot be converted to an Integer, and this generates an error.

DEFAULT NUM CHARS will extract the integer 12345 by reading and ignoring bird dog cat and treating the white space following 5 as a delimiter.

Effects of Quoted Strings. The presence of quoted strings affects the behavior of READ TEXT STRING and READ TEXT TOKEN. In this discussion, a quoted string means a set of characters beginning and ending with a double quote character and no embedded (non-escaped) double quote characters. The double quote character is ASCII 34 decimal. The presence of double quotes affects the way that these READ transactions group characters into strings and tokens, and how embedded control and escape characters are handled.

In this discussion, the terms **control character** and **escape character** have specific meaning. A control character is a single byte of data corresponding to one of the ASCII characters 0-31 decimal. For example, linefeed is ASCII 10 decimal and the symbol <LF> denotes linefeed character in this discussion. The string \n is a human-readable escape character representing linefeed that is recognized by HP VEE.

Outside of double quoted strings, control characters other than linefeed are read and ignored 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, all enclosed characters (including control characters) are read and stored in the input variable. Embedded linefeeds are read and treated like any other character; they do not terminate the current READ. Escape character sequences are read and translated to their single-character counterpart.

Grouping effects are best explained by using an example. For the discussion in the rest of this section, the data being read is a file with the contents shown in Figure E-28.

```
"This is in quotes."  This is not.
```

Figure E-28. Quoted and Non-Quoted Data

Assume that you read the file shown in Figure E-28 using `From File` with these transactions:

```
READ TEXT x STR
READ TEXT y STR
```

After reading the file, the results are:

```
x = This is in quotes.
y = This is not.
```

Note that the double quotes are interpreted as delimiters and do not appear in the input variable.

Now assume that you read the file shown in Figure E-28 using `From File` with these transactions:

```
READ TEXT x STR MAXFW:4
READ TEXT y STR
```

After reading the file, the results are:

```
x = This
y = This is not.
```

Here the double quotes are still acting a delimiters; the first transaction reads from double quote to double quote and assigns the first four characters to `x`. This leaves the file's read pointer positioned before the second occurrence of `This`. The second transaction reads the same string as before.

Next, assume that you read the file shown in Figure E-28 using `From File` with these transactions:

```
READ TEXT x TOKEN
READ TEXT y STR
```

Now after reading the file, the results are:

```
x = This is in quotes.
y = This is not.
```

Here the double quotes effectively make the entire first sentence into a single token. Even though default `TOKEN` delimiter is white space, the entire quoted string is treated as a single token. In addition, `TOKEN` reads and discards the double quote characters.

CHAR Format

`READ TEXT CHAR` transactions are of this form:

`READ TEXT` *VarList* `CHAR:`*NumChar* `ARRAY:`*NumStr*

VarList is a single Text variable or a comma-separated list of Text variables.

NumChar specifies the number of 8-bit characters that must read to fill each element of each variable in *VarList*.

NumStr is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

`CHAR` format is useful when you wish to simply read one character at a time, or when you need to read *every* character without ignoring any incoming data.

This transaction reads two two-dimensional Text arrays; each element in each array contains two characters.

```
READ TEXT X,Y CHAR:2 ARRAY:2,2
```

If a file read by the previous transaction contains these characters:

```
<space>ABCDEFG"AB"<LF>'CD
```

then the variables `X` and `Y` contain these values after the `READ`:

```
X [0 0] = <space>A
X [0 1] = BC
X [1 0] = DE
X [1 1] = FG

Y [0 0] = "A
Y [0 1] = B"
Y [1 0] = <LF>'
Y [1 1] = CD
```

The symbol `<space>` means the single character, space (ASCII 32 decimal). The symbol `<LF>` means the single character, linefeed (ASCII 10 decimal). Note that space, linefeed, and double quotes are read without any special consideration or interpretation.

TOKEN Format

`READ TEXT TOKEN` transactions are of this form:

`READ TEXT` *VarList* `TOKEN` *Delimiter* `ARRAY:` *NumElements*

VarList is a single Text variable or a comma-separated list of Text variables.

Delimiter specifies the combinations of characters that terminate (delimit) each token.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

TOKEN format allows you to define the delimiter (boundary) for tokens using one of these choices for *Delimiter*:

- SPACE DELIM
- INCLUDE CHARS
- EXCLUDE CHARS

The following discussion of delimiters explains how the choice of delimiters affects reading a file with these contents:

```
A phrase.
"A phrase."
Tab follows     .
XOXXOOXXXOOOXXXX
XAXXBCXXXDEF
```

Figure E-29. Data for READ TOKEN

The file contains only the letter O, not the digit zero.

Note that there is an invisible linefeed character at the end of each of the first four lines of the file in Figure E-29. The figure shows the file as it would appear in a text editor like vi.

SPACE DELIM. If you use SPACE DELIM, tokens are terminated by any white space. White space includes spaces, tabs, newline, and end-of-file. This corresponds roughly to words in written English. Using SPACE DELIM, you could read a file containing a paragraph of prose and separate out individual words.

Note that double quoted strings receive special treatment. Double quoted strings are read as a single token and the double quotes are stripped away. Control characters (ASCII 0-31 decimal) embedded in double-quoted strings are returned in the output variable. Escape characters (such as \n) embedded in double-quoted strings are converted into their equivalent control characters. This special treatment of double-quoted strings applies only to SPACE DELIM transactions; INCLUDE CHARS and EXCLUDE CHARS treat double quotes, escapes, and control characters the same as any other character.

If you read the data shown in Figure E-29 using SPACE DELIM with this transaction:

```
READ TEXT a TOKEN ARRAY:8
```

then the variable a contains these values:

```
a[0] = A
a[1] = phrase.
a[2] = A phrase.
a[3] = Tab
a[4] = follows
a[5] = .
a[6] = XOXXOOXXXOOOXXXX
a[7] = XAXXBCXXXDEF
```

INCLUDE CHARS. If you use INCLUDE CHARS, you can specify a list of characters to be "included" in tokens returned by the READ. These specified characters will be the *only* characters returned in any token. Any character other than the specified INCLUDE characters terminates the current token. The terminating characters *are not* included in the token and are stripped away.

If HP VEE reads the data shown in Figure E-29 using INCLUDE CHARS with this transaction:

```
READ TEXT a TOKEN INCLUDE:"X" ARRAY:7
```

then the variable a contains these values:

```
a[0] = X
a[1] = XX
a[2] = XXX
a[3] = XXXX
a[4] = X
a[5] = XX
a[6] = XXX
```

If HP VEE reads the data shown in Figure E-29 using INCLUDE CHARS with this transaction:

```
READ TEXT a TOKEN INCLUDE:"OXZ" ARRAY:4
```

then the variable a contains these values:

```
a[0] = XOXXOOXXXOOOXXXX
a[1] = X
a[2] = XX
a[3] = XXX
```

Note that the first character in the INCLUDE list is the letter O, not the digit zero.

Assume that you are trying to read a file containing the data in Figure E-30.

```
111 222 333 444 555
```

Figure E-30. Data for READ TOKEN

If you try to read the file in Figure E-30 using this transaction:

```
READ TEXT x,y,z TOKEN INCLUDE:"1234567890"
```

then the Text variables `x`, `y`, and `z` will contain these values:

```
x = 111
y = 222
z = 333
```

Another way to do this is to specify an `ARRAY` greater than one and read data into an array. For example, if you read the data in Figure E-30 using this transaction:

```
READ TEXT x TOKEN INCLUDE:"1234567890" ARRAY:3
```

then the Text variable `x` contains these values:

```
x[0] = 111
x[1] = 222
x[2] = 333
```

EXCLUDE CHARS. If you use `EXCLUDE CHARS`, you can specify a list of characters, any one of which will terminate the current token. The terminating characters *are not* included in the token. they are read and discarded.

If you read the data shown in Figure E-29 using EXCLUDE with this transaction:

```
READ TEXT a TOKEN EXCLUDE:"X" ARRAY:8
```

then the variable a contains these values:

```
a[0] = A phrase.<LF>"A phrase."<LF>Tab follows     .<LF>
a[1] = 0
a[2] = 00
a[3] = 000
a[4] = <LF>
a[5] = A
a[6] = BC
a[7] = DEF
```

Assume the data shown in Figure E-31 is sent to HP VEE from an instrument.

```
++1.23++4.98++0.45++2.34++0.01++23.45++12.2++
```

Figure E-31. Data for READ TOKEN

If HP VEE reads the data in Figure E-31 with this transaction:

```
READ TEXT x TOKEN EXCLUDE:"+" ARRAY:7
```

then the variable x will contain these values:

```
x[0] = null string (empty)
x[1] = 1.23
x[2] = 4.98
x[3] = 0.45
x[4] = 2.34
x[5] = 0.01
x[6] = 23.45
```

Note that even though seven "numbers" were available, only six were read. At the end of this transaction, HP VEE has read seven tokens terminated by the +, including the first character which was terminated before it was filled with any data.

STRING Format

`READ TEXT STRING` transactions are of this form:

`READ TEXT` *VarList* `STR ARRAY:`*NumElements*
`-or-`
`READ TEXT` *VarList* `STR MAXFW:`*NumChars* `ARRAY:`*NumElements*

VarList is a single Text variable or a comma-separated list of Text variables.

NumChars specifies the maximum number of 8-bit characters that can be read in an attempt to build a string.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

This transaction reads all incoming characters and returns strings. Please refer to the section "Effects of Quoted Strings" earlier in this chapter for details about the effects of double quoted strings on `READ TEXT STRING`.

Effects of Control and Escape Characters. In this discussion, the terms **control character** and **escape character** have specific meaning. A control character is a single byte of data corresponding to one of the ASCII characters 0-31 decimal. For example, linefeed is ASCII 10 decimal and the symbol `<LF>` denotes linefeed character in this discussion. The string `\n` is a human-readable escape character representing linefeed that is recognized by HP VEE. HP VEE uses escape characters to represent control characters within quoted strings.

Control characters and escape characters are handled differently depending on whether or not they appear within double quoted strings.

Outside double quoted strings, control characters other than linefeed are read and discarded. Linefeed terminates the current string. Escape chracters, such as `\n`, are simply read as two individual characters (`\` and `n`).

Within double quoted strings, control characters and escape characters are read and included in the string returned by the `READ`. A linefeed within a double quoted string does *not* terminate the current string. Escape chracters, such as

\n, are interpreted as their single character equivalent (<LF>) and are included in the returned string as a control character.

Assume you wish to read the data in Figure E-32 using READ TEXT STRING transactions.

```
Simple string.
Random \n % $ * 'A'
"In quotes."
"In quotes
with control."
"In quotes\nwith escape."
```

Figure E-32. String Data

If you read the data in Figure E-32 using this transaction:

```
READ TEXT x STR ARRAY:5
```

then the variable x contains these values:

```
a[0] = Simple string.
a[1] = Random \n % $ * 'A'
a[2] = In quotes.
a[3] = In quotes<LF>with control.
a[4] = In quotes<LF>with escape.
```

If you read the same data in Figure E-32 using this transaction:

```
READ TEXT x STR MAXFW:16 ARRAY:5
```

then the variable x contains these values:

```
a[0] = Simple string.
a[1] = Random \n % $ *
a[2] = 'A'
a[3] = In quotes.
a[4] = In quotes<LF>with c
```

Note that the transaction terminates the current `READ` whenever 16 characters have been read (`a[1]`) or when a non-quoted `<LF>` (`a[2]`) is read. Double quoted strings are read from double quote to double quote and the first 16 delimited characters are returned (`a[4]`).

INTEGER Format

`READ TEXT INTEGER` transactions are of this form:

`READ TEXT` *VarList* `INT ARRAY:`*NumElements*

`-or-`

`READ TEXT` *VarList* `INT MAXFW:`*NumChars* `ARRAY:`*NumElements*

VarList is a single Integer variable or a comma-separated list of Integer variables.

NumChars specifies the maximum number of 8-bit characters that can be read in an attempt to build a number.

NumStr is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

`READ TEXT INTEGER` transactions interpret incoming characters as 32-bit, two's complement integers. The valid range for these integers is 2 147 483 647 to -2 147 483 648. Any numbers outside this range wrap around so there is never an overflow condition. For example, 2 147 483 648 is interpreted as -2 147 483 648. As it starts to build a number, HP VEE discards any leading characters that are not recognized as part of a number. Once HP VEE starts building a number, any character that is not recognized as part of a number terminates the `READ` for that number. These are the only combinations of characters that are recognized as part of an `INTEGER`:

Notation	**Characters Recognized**
Decimal	Valid characters are `+-0123456789`. Leading zeros are *not* interpreted as an octal prefix as they are in HP VEE data entry fields.
HP VEE hexadecimal	HP VEE interprets `0x` as a prefix for a hexadecimal number. Valid characters following the prefix are `0123456789aAbBcCdDeEfF`.
IEEE 488.2 binary	HP VEE interprets `#b` or `#B` as a prefix for a binary number. Valid characters following the prefix are `0` and `1`.
IEEE 488.2 octal	HP VEE interprets `#q` or `#Q` as a prefix for an octal number. Valid characters following the prefix are `01234567`.
IEEE 488.2 hexadecimal	HP VEE interprets `#h` or `#H` as a prefix for a hexadecimal number. Valid characters following the prefix are `0123456789aAbBcCdDeEfF`.

All of the following notations are interpreted as the Integer value 15 decimal:

```
15
+15
015
0xF
0xf
#b1111
#Q17
#hF
```

OCTAL Format

`READ TEXT OCTAL` transactions are of this form:

```
READ TEXT VarList OCT ARRAY:NumElements
-or-
READ TEXT VarList OCT MAXFW:NumChars ARRAY:NumElements
```

VarList is a single Integer variable or a comma-separated list of Integer variables.

NumChars specifies the number of 8-bit characters that can be read in an attempt to build a number.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

`READ TEXT OCTAL` transactions interpret incoming characters as octal digits representing 32-bit, two's complement integers. The valid range for these integers is 2 147 483 647 decimal to -2 147 483 648 decimal.

If the transaction specifies a `MAX NUM CHARS` (`MAXFW`), the octal number read may contain more than 32 bits of data. For example, assume HP VEE reads the data in Figure E-33 using this transaction:

```
READ TEXT x OCT MAXFW:21
```

```
377237456214567243777
```

Figure E-33. Octal Data

HP VEE reads all the digits in Figure E-33, but uses only the last 11 digits (`14567243777`) to build a number for the value of `x`. This is because each digit corresponds to 3 bits and the octal number must be stored in an HP VEE Integer, which contains 32 bits. Eleven octal digits yield 33 bits; the most significant bit is dropped to fit the value in an HP VEE Integer. There is no possibility of overflow.

If the transaction specifies `DEFAULT NUM CHARS`, it will continue to read characters until it builds enough numbers to fill each variable in *VarList*. Linefeed characters will not terminate number building early. For example, this transaction:

```
READ TEXT x OCT ARRAY:4
```

interprets each line of data in Figure E-34 as the same set of four octal numbers.

```
0345 067 003<LF>0377<LF>
345 67 3 377<EOF>
345,67,3,377,45,67<EOF>
```

Figure E-34. Octal Data

The symbol `<LF>` represents the single character linefeed (ASCII 10 decimal). The symbol `<EOF>` represents the end-of-file condition.

HEX Format

`READ TEXT HEX` transactions are of this form:

`READ TEXT` *VarList* `HEX ARRAY:` *NumElements*
`-or-`
`READ TEXT` *VarList* `HEX MAXFW:` *NumChars* `ARRAY:` *NumElements*

VarList is a single Integer variable or a comma-separated list of Integer variables.

NumChars specifies the number of 8-bit characters that can be read in an attempt to build a number.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

READ TEXT HEX transactions interpret incoming characters as hexadecimal digits representing 32-bit, two's complement integers. The valid range for these integers is 2 147 483 647 decimal to -2 147 483 648 decimal.

If the transaction specifies a MAX NUM CHARS (MAXFW), the hexadecimal number read may contain more than 32 bits of data. For example, assume HP VEE reads the data in Figure E-35 using this transaction:

```
READ TEXT x HEX MAXFW:21
```

```
ad2469Ff725BCdef37964
```

Figure E-35. Hexadecimal Data

HP VEE reads all the digits in Figure E-35, but uses only the last 8 digits (def37964) to build a number for the value of x. This is because each digit corresponds to 4 bits and the hexadecimal number must be stored in an HP VEE Integer, which contains 32 bits. Eight hexadecimal digits yields exactly 32 bits. There is no possibility of overflow.

Assume HP VEE reads the same data in Figure E-35, but with a different MAX NUM CHARS, as in this transaction:

```
READ TEXT x HEX MAXFW:3 ARRAY:7
```

In this case, the transaction reads the same data and interprets it as seven Integers, each comprised of three hexadecimal digits.

If the transaction specifies DEFAULT NUM CHARS, it will continue to read characters until it builds enough numbers to fill each variable in *VarList*. Each number will read *exactly* 8 hexadecimal digits. Linefeed characters will not terminate number building early.

Assume HP VEE reads the same data in Figure E-35, but with DEFAULT NUM CHARS, as in this transaction:

```
READ TEXT x HEX ARRAY:2
```

In this case, the transaction reads the same data and interprets it as two Integers, each comprised of eight hexadecimal digits. The last five digits (37946) are not read.

REAL Format

READ TEXT REAL transactions are of this form:

```
READ TEXT VarList REAL ARRAY:NumElements
-or-
READ TEXT VarList REAL MAXFW:NumChars ARRAY:NumElements
```

VarList is a single Real variable or a comma-separated list of Real variables.

NumChars specifies the maximum number of 8-bit characters that can be read in an attempt to build a number.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the ARRAY keyword does not appear in the transaction. Note that ARRAY:1 is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

The decimal number read by this transaction is interpreted as an HP VEE Real which is a 64-bit IEEE 754 floating-point number. The range of these numbers is:

```
-1.797 693 134 862 315E+308
-2.225 073 858 507 202E-307
 0
 2.225 073 858 507 202E-307
 1.797 693 134 862 315E+308
```

If the transaction specifies a MAX NUM CHARS (MAXFW), the Real number read may contain more than 17 digits of data. For example, assume HP VEE reads the data in Figure E-36 using this transaction:

```
READ TEXT x REAL MAXFW:19
```

```
1.234567890123456789
```

Figure E-36. Real Data

HP VEE reads all the digits in Figure E-36, but uses only the 17 most-significant digits of the mantissa to build a number for the value of `x`. This is because each Real contains a 54-bit mantissa, which is equivalent to more than 16 but less than 17 decimal digits. As a result, `x` has the value `1.2345678901234567`. Text to Real conversions are not guaranteed to yield the same value to the least-significant digit. Comparisons of the two least-significant bits is unadvisable.

Assume HP VEE reads the same data in Figure E-36, but with a different `MAX NUM CHARS`, as in this transaction:

```
READ TEXT x REAL MAXFW:6 ARRAY:3
```

In this case, the transaction reads the same data and interprets it as 3 Real numbers, each comprised of six decimal characters. The last two characters are not read.

If the transaction specifies `DEFAULT NUM CHARS`, it will continue to read characters until it builds enough numbers to fill each variable in *VarList*. Each number will read at most 17 decimal digits. Linefeed characters, white space and other non-numeric characters will terminate number building before 17 digits have been read.

`READ TEXT REAL` transactions recognize most commonly used decimal notations for Real numbers including leading signs, decimal points, and signed exponents. The characters `+-.0123456789Ee` are recognized as valid parts of a Real number by *all* `READ TEXT REAL` transactions. If the transaction specifies `DEFAULT NUM CHARS`, the suffix characters shown in Table E-13 are also recognized. The suffix character must immediately follow the last digit of the number with no intervening white space.

Table E-13. Suffixes for REAL Numbers

Suffix	Multiplier
P	10^{15}
T	10^{12}
G	10^{9}
M	10^{6}
k or K	10^{3}
m	10^{-3}
u	10^{-6}
n	10^{-9}
p	10^{-12}
f	10^{-15}

The Text data in Figure E-37 represents six real numbers.

```
1001
+1001.
1001.0
1.001E3
+1.001E+03
1.001K
```

Figure E-37. Example of Real Notations

If HP VEE reads the data in Figure E-37 with this transaction:

```
READ TEXT x REAL ARRAY:6
```

then each element of the Real variable `x` contains the value `1001`.

If HP VEE reads the data in Figure E-37 with this transaction:

```
READ TEXT x REAL MAXFW:20 ARRAY:6
```

then the first five elements of the Real variable `x` contain the value `1001` and the sixth element contains the value `1.001`.

COMPLEX, PCOMPLEX, and COORD Formats

`COMPLEX`, `PCOMPLEX`, and `COORD` correspond to the HP VEE multi-field data types with the same names. The behavior of all three `READ` formats is very similar. The behaviors described in this section apply to all three formats except as noted.

Just as the HP VEE data types Complex, PComplex, and Coord are composed of multiple Real numbers, the `COMPLEX`, `PCOMPLEX`, and `COORD` formats are compound forms of the `REAL` format. Each constituent Real value of the mult-field data types is read using the same rules that apply to an individual `REAL` formatted value.

COMPLEX Format. `READ TEXT COMPLEX` transactions are of this form:

`READ TEXT` *VarList* `CPX ARRAY:` *NumElements*

Each `READ TEXT COMPLEX` transaction reads the equivalent of two `REAL` formatted numbers. The first number read is interpreted as the real part and the second number read is interpreted as the imaginary part.

PCOMPLEX Format. `READ TEXT PCOMPLEX` transactions are of this form:

`READ TEXT` *VarList* `PCX:` *PUnit* `ARRAY:` *NumElements*

PUnit specifies the units of angular measure in which the phase of the PComplex is measured.

Each `READ TEXT PCOMPLEX` transaction reads the equivalent of two `REAL` formatted numbers. The first number read is interpreted as the magnitude and the second number read is interpreted as the phase.

If any transaction reading COMPLEX, PCOMPLEX, or COORD formats encounters an opening parenthesis, it expects to find a closing parenthesis.

Assume you wish to read a file containing the data shown in Figure E-38.

```
(1.23 , 3.45  (6.78 , 9.01) (1.23 , 4.56)
```

Figure E-38. Data Containing Parentheses

If HP VEE reads the data in Figure E-38 with this transaction:

```
READ TEXT x,y CPX
```

then the variables x and y contain these Complex values:

```
x = (1.23 , 3.45)
y = (1.23 , 4.56)
```

Note that the transaction read past 6.78 and 9.01 to find the closing parenthesis. If parentheses had been omitted from the data entirely, y would have the value (6.78 , 9.01).

COORD Format. READ TEXT COORD transactions are of this form:

READ TEXT *VarList* COORD:*NumFields* ARRAY:*NumElements*

VarList is a single Coord variable or a comma-separated list of Coord variables.

NumFields is a single variable or expression that specifies the number of rectangular dimensions in each Coord value. This value must be 2 or more for the READ to execute without error.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the ARRAY keyword does not appear in the transaction. Note that ARRAY:1 is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

BINARY Encoding

`READ BINARY` transactions are of this form:

`READ BINARY` *VarList DataType* `ARRAY:` *NumElements*

VarList is a single variable or a comma-separated list of variables.

DataType is one of the following pre-defined formats corresponding to the HP VEE data type with the same name:

- `BYTE` - 8-bit byte
- `INT16` - 16-bit two's complement integer
- `INT32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating-point number
- `REAL64` - 64-bit IEEE 754 floating-point number
- `STRING` - null terminated string
- `COMPLEX` - equivalent to two `REAL`s
- `PCOMPLEX` -equivalent to two `REAL`s
- `COORD` - equivalent to two or more `REAL`s

Note

HP VEE stores and manipulates all integer values as the `INT32` data type, and all real numbers as the `Real` data type, also known as `REAL64`. Thus, the `INT16` and `REAL32` data types are provided for I/O only. HP VEE-Test performs the following data-type conversions for instrument I/O:

- On an input transaction `INT16` values from an instrument are *individually* converted to `INT32` values by HP VEE.
- On an input transaction `REAL32` values from an instrument are *individually* converted to `REAL64` values by HP VEE.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the first expression is an asterisk (`*`), the transaction will read data until an EOF is encountered. Read to end is supported only for `From File`, `From String`, `From StdIn`, `Execute Program`, `To/From Named Pipe`, and `To/From HP BASIC/UX` transactions. Only the first dimension can have an asterisk rather than a number. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

For example, the following transaction, reading from a file:

```
READ BINARY a REAL64 ARRAY:*,10
```

will read until EOF is encountered, resulting in a two dimensional array with 10 columns. The number of rows is dependent on the amount of data in the file. The total number of data elements read must be evenly divisible by the product of the known dimension sizes, in this example: 10.

`READ BINARY` transactions expect that incoming data is in *exactly* the same format that would be produced by an equivalent `WRITE BINARY` transaction.

`BINARY` encoded data has the advantage of being very compact, but it is not easily shared with non-HP VEE applications.

BINBLOCK Encoding

`READ BINBLOCK` transactions are of this form:

`READ BINBLOCK` *VarList DataType*

VarList is a single variable or a comma-separated list of variables.

DataType is one of these pre-defined HP VEE data types:

- `BYTE` - 8-bit byte
- `INT16` - 16-bit two's complement integer
- `INT32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating-point number
- `REAL64` - 64-bit IEEE 754 floating-point number
- `COMPLEX` - equivalent to two `REAL`s
- `PCOMPLEX` -equivalent to two `REAL`s
- `COORD` - equivalent to two or more `REAL`s

You do not need to specify any addtional information about the format of incoming data; the block header contains sufficient information. `READ BINBLOCK` can read any of the block formats previously described previously with `WRITE BINBLOCK` transactions.

This transaction reads two traces from an oscilloscope that formats its traces as IEEE 488.2 Definite Length Arbitrary Block Response Data:

```
READ BINBLOCK a,b REAL
```

CONTAINER Encoding

READ CONTAINER transactions are of the form:

READ CONTAINER *VarList*

VarList is a single variable or a comma-separated list of variables.

READ CONTAINER transactions reads data stored in the special special text representation written by WRITE CONTAINER transactions. No additional specifications, such as format, need to be specified with READ CONTAINER since that information is part of the container.

REGISTER Encoding

REGISTER encoding is supported by HP VEE-Test only.

READ REGISTER is used to read values from a VXI device's A16 memory.

READ REGISTER transactions are of this form:

READ REG: *SymbolicName ExpressionList* INCR ARRAY:*NumElements*
-or-
READ REG: *SymbolicName ExpressionList* ARRAY:*NumElements*

where:

SymbolicName is a name defined during configuration of a VXI device. The name refers to a specific address within a device's register space. Specific data types for READ REGISTER transactions are:

- BYTE - 8 bit byte
- WORD16 - 16-bit two's complement integer
- WORD32 - 32-bit two's complement integer
- REAL32 - 32-bit IEEE 754 floating point number

These data types are also specified during configuration of a VXI device and do not appear in the transaction.

ExpressionList is a single expression or a comma-separated list of expressions.

INCR specifies that array data is to be read from the register incrementally starting at the address specified by *SymbolicName*. The first element of the

array is read from the starting address, the second from that address plus an offset equal to the length in bytes of the data type, and so forth until all array elements have been read. If `INCR` is not specified in the transaction, the entire array is read from the single location specified by *SymbolicName*.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

MEMORY Encoding

`MEMORY` encoding is supported by HP VEE-Test only.

`READ MEMORY` is used to read values from a VXI device's A24 or A32 memory.

`READ MEMORY` transactions are of this form:

`READ MEM:` *SymbolicName ExpressionList* `INCR ARRAY:` *NumElements*
`-or-`
`READ MEM:` *SymbolicName ExpressionList* `ARRAY:` *NumElements*

where:

SymbolicName is a name defined during configuration of a VXI device. The name refers to a specific address within a device's extended memory. Specific data types for `READ MEMORY` transactions are:

- `BYTE` - 8 bit byte
- `WORD16` - 16-bit two's complement integer
- `WORD32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating point number

These data types are also specified during configuration of a VXI device and do not appear in the transaction.

ExpressionList is a single expression or a comma-separated list of expressions.

`INCR` specifies that array data is to be read from the memory location incrementally starting at the location specified by *SymbolicName*. The first

element of the array is read from the starting location, the second from that location plus an offset equal to the length in bytes of the data type, and so forth until all array elements have been read. If `INCR` is not specified in the transaction, the entire array is read from the single memory location specified by *SymbolicName*.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

IOSTATUS Encoding

`READ IOSTATUS` transactions are of this form:

```
READ IOSTATUS STS Bits VarList
-or-
READ IOSTATUS DATA READY VarList
```

VarList is a single Integer variable or a comma-separated list of Integer variables.

`READ IOSTATUS` transactions are used by `Direct I/O` for GPIO interfaces, `From StdIn`, `To/From Named Pipe`, and `To/From HP BASIC/UX`.

`READ IOSTATUS` transactions for GPIO reads the peripheral status bits available on the interface. The number of bits read is dependent on the model number of the interface. A single integer value is returned that is the weighted sum of all the status bits.

For example, the HP 98622A GPIO interface supports two peripheral status lines, STI0 and STI1. Table E-14 illustrates how to interpret the value of `x` in this transaction:

```
READ IOSTATUS STS Bits a
```

Table E-14. `IOSTATUS` Values

Value Read	STI1	STI0
0	0	0
1	0	1
2	1	0
3	1	1

`READ IOSTATUS` transactions read the instantaneous values of the status lines; the status line are not latched or buffered in any way.

`READ IOSTATUS` transactions for `To/From Named Pipe`, `To/From HP BASIC/UX` and `From StdIn` returns a Boolean YES (1) if there is data ready to read. If no data is present, a Boolean NO (0) is returned. The `READ IOSTATUS` transaction can be used to avoid a read that will block indefinitely.

EXECUTE Transactions

EXECUTE transactions send low-level commands to control the file, instrument, or interface associated with a particular object. EXECUTE is used to adjust file pointers, clear buffers, and provide low-level control of hardware interfaces. The various EXECUTE commands available are summarized in Table E-15.

Table E-15. Summary of EXECUTE Commands

Commands	Description
To File, From File	
REWIND	Sets the read pointer (From File) or write pointer (To File) to the beginning of the file without changing the data in the file.
CLEAR	(To File only). Erases existing data in the file and sets the write pointer to the beginning of the file.
CLOSE	Explicitly closes the file. Useful when multiple processes are reading and writing the same file.
Interface Operations (HP VEE-Test only)	
CLEAR	For HP-IB clears all devices by sending DCL (Device Clear). For VXI, resets the interface and runs the resource manager
TRIGGER	For HP-IB triggers all devices addressed to listen by sending GET (Group Execute Trigger). For VXI triggers specified backplane trigger lines or external triggers on an embedded controller.
LOCAL	For HP-IB releases the REN (Remote Enable) line.
REMOTE	For HP-IB asserts the REN (Remote Enable) line.
LOCAL LOCKOUT	For HP-IB sends the LLO (Local Lockout) message. Any device in remote at the time LLO is sent will lock out front panel operation.
ABORT	Clears the HP-IB interface by asserting the IFC (Interface Clear) line.

Table E-15. Summary of `EXECUTE` Commands (continued)

Commands	Description
Direct I/O to HP-IB (HP VEE-Test only)	
`CLEAR`	Clears the HP-IB interface associated with the `Direct I/O` object by asserting the IFC (Interface Clear) line.
`TRIGGER`	Triggers the device at the address of a `Direct I/O` object by addressing it to listen and sending GET (Group Execute Trigger).
`LOCAL`	Places the device at the address of the `Direct I/O` object in the local state.
`REMOTE`	Places the device at the address of the `Direct I/O` object in the remote state.
`RESET`	Resets the HP-IB interface associated with the `Direct I/O` object. For HP series 300 computers, `RESET` conducts the following sequence: 1. Clears the REN (Remote Enable) line. 2. Pulses the IFC (Interace Clear) line. 3. Asserts the REN (Remote Enable) line.
Direct I/O to GPIO (HP VEE-Test only)	
`RESET`	Resets the GPIO interface associated with the `Direct I/O` object by pulsing the PRESET line (Peripheral Reset).
Direct I/O to message based VXI (HP VEE-Test only)	
`CLEAR`	Clears the VXI device associated with the `Direct I/O` object by sending the word-serial command Clear (0xffff).
`TRIGGER`	Triggers the VXI device associated with the `Direct I/O` object by sending the word-serial command Trigger (0xedff).
`LOCAL`	Places the VXI device associated with the `Direct I/O` object into local state by sending the word-serial command Clear Lock (0xefff).
`REMOTE`	Places the VXI device associated with the `Direct I/O` object into local state by sending the word-serial command Set Lock (0xeeff). in the remote state.

Table E-15. Summary of `EXECUTE` Commands (continued)

Commands	Description
Direct I/O to Serial Interfaces (HP VEE-Test only)	
`RESET`	Resets the serial interface associated with the `Direct I/O` object.
`BREAK`	Transmits a signal on the Data Out line of the serial interface associated with the `Direct I/O` object as follows: 1. A logical High for 400 milliseconds 2. A logical Low for 60 milliseconds
Execute Program, To/From Named Pipe, To/From HP BASIC/UX	
`CLOSE READ PIPE`	Closes the read named pipe associated with the (To/From) object or the stdin pipe associated with the (Execute Program).
`CLOSE WRITE PIPE`	Closes the write named pipe associated with the (To/From) object or the stdout pipe associated with the (Execute Program).

Details About HP-IB

The `EXECUTE` commands used by `Direct I/O` to HP-IB devices and `Interface Operations` are similar but different.

- `Direct I/O EXECUTE` commands address an instrument to receive the command.
- `Interface Operations EXECUTE` commands may affect multiple instruments already addressed to listen.

The following series of tables indicate the exact bus actions conducted by `Direct I/O` and `Interface Operations EXECUTE` transactions.

Table E-16. `EXECUTE ABORT` HP-IB Actions

Direct I/O	Interface Operations
Not applicable.	IFC ($\geq$ 100 μsec) REN $\overline{\text{ATN}}$

Table E-17. `EXECUTE CLEAR` HP-IB Actions

Direct I/O	Interface Operations
ATN	ATN
MTA	DCL
UNL	
LAG	
SDC	

Table E-18. `EXECUTE TRIGGER` HP-IB Actions

Direct I/O	Interface Operations
ATN	ATN
MTA	GET
UNL	
LAG	
GET	

Table E-19. `EXECUTE LOCAL` **HP-IB Actions**

Direct I/O	Interface Operations
ATN	$\overline{\text{REN}}$
MTA	$\overline{\text{ATN}}$
UNL	
LAG	
GTL	

Table E-20. `EXECUTE REMOTE` **HP-IB Actions**

Direct I/O	Interface Operations
REN	REN
ATN	$\overline{\text{ATN}}$
MTA	
UNL	
LAG	

Table E-21. `EXECUTE LOCAL LOCKOUT` **HP-IB Actions**

Direct I/O	Interface Operations
Not applicable.	ATN
	LLO

Details About VXI

The `EXECUTE` commands used by `Direct I/O` to VXI devices and `Interface Operations` are similar, but different.

- `Direct I/O EXECUTE` commands address a message based VXI device to receive a word-serial command.
- `Interface Operations EXECUTE` commands affect the VXI interface directly and may affect VXI devices within the interfaces servant area.

`EXECUTE TRIGGER` transactions for the `Interface Operations` object are of the form:

`EXECUTE TRIGGER` *TriggerType Expression TriggerMode*

TriggerType specifies which trigger group will be used by the `EXECUTE TRIGGER` transaction. The groups are:

- `TTL` - Specifies the eight TTL trigger lines on the VXI backplane.
- `ECL` - Specifies the four ECL trigger lines on the VXI backplane.
- `EXT` - Specifies the external triggers on a embedded VXI controller.

Expression evaluates to a single Integer variable that represents a bit pattern indicating which trigger lines for a particular *TriggerType* are to be triggered. A value of 5, represented in binary as `101`, indicates that TTL lines 0 and 2 are to be triggered. A value of 255 triggers all eight TTL lines.

TriggerMode indicates the way the trigger lines are to be asserted:

- `PULSE` - Lines are to be pulsed for a discreet time limit (*TriggerType* dependent).
- `ON` - Asserts the trigger lines and leaves them asserted.
- `OFF` - Removes the assertion from trigger lines that were asserted by a previous `ON` transaction.

The following series of tables indicate the exact bus actions conducted by `Direct I/O` and `Interface Operations` `EXECUTE` transactions.

Table E-22. `EXECUTE CLEAR` **VXI Actions**

Direct I/O	Interface Operations
Word-serial command Clear(0xffff)	Pulse SYSRESET line, rerun Resource Manager

Table E-23. `EXECUTE TRIGGER` **VXI Actions**

Direct I/O	Interface Operations
Word-serial command Trigger(0xedff)	Triggers either the TTL or ECL trigger lines in the backplane, or the external trigger(s) on the embedded VXI controller. You can specify which lines are to be triggered for each trigger type.

Table E-24. `EXECUTE LOCAL` **VXI Actions**

Direct I/O	Interface Operations
Word-serial command Set Lock(0xeeff)	Not applicable.

Table E-25. `EXECUTE REMOTE` **VXI Actions**

Direct I/O	Interface Operations
Word-serial command Clear Lock(0xefff)	Not applicable.

WAIT Transactions

There are four types of WAIT transactions:

- WAIT INTERVAL
- WAIT SPOLL (Direct I/O to HP-IB and message based VXI devices only)
- WAIT REGISTER (Direct I/O to VXI devices only)
- WAIT MEMORY (Direct I/O to VXI devices only)

WAIT INTERVAL transactions simply wait for the specified number of seconds before executing the next transaction listed in the open view of the object. For example, this transaction waits for 10 seconds:

```
WAIT INTERVAL:10
```

WAIT SPOLL transactions are of the form:

WAIT SPOLL *Expression Sense*

Expression is an expression that evaluates to an integer. The integer will be used as as a bit mask.

Sense is a field with two possible values.

- ANY SET
- ALL CLEAR

WAIT SPOLL transactions wait until the serial poll response byte of the associated instrument meets a specific condition. The serial poll response is tested by bitwise ANDing it with the specified mask and ORing the resulting bits into a single test bit. The transaction following WAIT SPOLL executes when one of the following conditions is met:

- The transaction specifies ANY (ANY SET) and the test bit is true (1).
- The transaction specifies CLEAR (ALL CLEAR) and the test bit is false (0).

The following transactions show how to use WAIT SPOLL:

```
WAIT SPOLL:256 ANY      Wait until any bit is set.
WAIT SPOLL:256 CLEAR    Wait until all are clear.
WAIT SPOLL:0x40 ANY     Wait until bit 6 is set.
WAIT SPOLL:0x40 CLEAR   Wait until bit 6 is clear.
```

WAIT REGISTER and WAIT MEMORY transactions are of the form:

```
WAIT REG:SymbolicName MASK:Expression Sense [Expression]
-or-
WAIT MEM:SymbolicName MASK:Expression Sense [Expression]
```

where:

SymbolicName is a name defined during configuration of a VXI device. The name refers to a specific address within a device's A16 or extended memory.

MASK:*Expression* is an expression that evaluates to an integer. The integer will be used as as a bit mask. The size in bytes of this mask value depends on the data type for which *SymbolicName* has been configured.

Sense is a field with three possible values.

- ANY SET
- ALL CLEAR
- EQUAL

[*Expression*] is an optional compare value that evaluates to an integer. The integer is used only when *Sense* is EQUAL.

WAIT REGISTER or MEMORY transactions wait until the value read from the register or memory location specified by *SymbolicName*s in the associated VXI device meets a certain condition. The value read is logically ANDed with the bit mask specified in MASK:*Expression*, resulting in a test value. The size of the test value is dependent on the data type configured for the specified register or memory location. The transaction following WAIT SPOLL executes when one of the following conditions is met:

- The transaction specifies ANY (ANY SET) and the test value has at least one bit true (1).
- The transaction specifies CLEAR (ALL CLEAR) and the test value has all bits false (0).
- The transaction specifies EQUAL and the test value is equal bit-for-bit with the compare value specified in [*Expression*].

SEND Transactions

`SEND` transactions are supported by HP VEE-Test only.

`SEND` transactions are of this form:

`SEND` *BusCmd*

BusCmd is one of the bus commands listed in Table E-26.

`SEND` transactions are used within `Interface Operations` objects to transmit low-level bus messages via an HP-IB interface. These messages are defined in detail in IEEE 488.1.

Table E-26. `SEND` Bus Commands

Command	Description
`COMMAND`	Sets ATN true and transmits the specified data bytes. ATN true indicates that the data represents a bus command.
`DATA`	Sets ATN false and transmits the specified data bytes. ATN false indicates that the data represents device-dependent information.
`TALK`	Addresses a device at the specified primary bus address (0-31) to talk.
`LISTEN`	Addresses a device at the specified primary bus address (0-31) to listen.
`SECONDARY`	Specifies a secondary bus address following a TALK or LISTEN command. Secondary addresses are typically used by cardcage instruments where the cardcage is at a primary address and each plug-in module is at a secondary address.
`UNLISTEN`	Forces all devices to stop listening; sends UNL.

Table E-26. `SEND` Bus Commands (continued)

Command	Description
`UNTALK`	Forces all devices to stop talking; sends UNT.
`MY LISTEN ADDR`	Addresses the computer running HP VEE to listen; sends MLA.
`MY TALK ADDR`	Addresses the computer running HP VEE to talk; sends MTA.
`MESSAGE`	Sends a multi-line bus message. Consult IEEE 488.1 for details. The multi-line messages supported by HP VEE-Test are: DCL Device Clear SDC Selected Device Clear GET Group Execute Trigger GTL Go To Local LLO Local Lockout SPE Serial Poll Enable SPD Serial Poll Disable TCT Take Control

Glossary

This Glossary contains the terms and definitions used to name or describe graphical objects and processes in the HP VEE software, as well as some hardware items related to installing and using HP VEE.

Activate

1. To send a container to a terminal. See also **Container** and **Terminal**.

2. The action that resets the context of a `UserObject` before it operates each time. See also **Context** and **PreRun**.

Application

A software program that completes work directly for the user. For example, HP VEE-Engine is a general purpose engineering application, and HP VEE-Test is a test and measurement application.

Array

A data shape that contains a systematic arrangement of data items in one or more dimensions. The data items are accessed via indexes. See also **Data Shape**.

Arrow

1. An arrow-shaped pointer. See **Pointer**.

2. A scroll arrow that is a part of a scroll bar and is used either to scroll a list box, or to pan the work area.

Asynchronous

A method of operating without a common signal to synchronize events; rather, the events occur at unspecified times. Control pins in HP VEE are asynchronous.

Auto Execute
An option on the object menus of the data constant objects. When `Auto Execute` is set, the object operates when its value is edited.

Bitmap
A bit pattern or picture. In HP VEE you can display a bitmap on an icon.

Buffer
An area where information is stored temporarily.

Button
1. A button on a mouse. See **Mouse Button**.

2. A graphical object in HP VEE that simulates a real-life pushbutton and appears to pop out from your screen. When a button is "pressed" in HP VEE, by clicking on it with a mouse, an action occurs.

Cascading Menu
A submenu on a pull-down or pop-up menu that provides additional selections to a menu selection (feature).

Checkbox
A recessed square box on HP VEE menus and dialog boxes that allows you to select a setting. To select a setting, click on the box and a checkmark appears in the box to indicate a selection has been made. To cancel the setting, simply click on the box again to remove the checkmark.

Click
To press and release a mouse button quickly. Clicking usually selects a menu feature or object in the HP VEE window. See also **Double-Click** and **Drag**.

Compiled Function
A user-defined function created by dynamically linking a program, written in a programming language such as C, into the HP VEE process. The user must create a shared library file and a definition file for the program to be linked. The `Import Library` object attaches the shared library to the HP VEE process and parses the definition file declarations. The Compiled Function can then be called with the `Call Function` object, or from certain expressions. See also **User Function** and **Remote Function**.

Component
A single instrument function or measurement value in an HP VEE-Test `State Driver` or `Component Driver`. For example, a voltmeter driver contains components that record the range, trigger source, and latest reading. See also **Component Driver**, **Driver Files**, **State**, and **State Driver**.

Component Driver
An instrument control object that reads and writes values to components you specifically select. Use `Component Drivers` to control an instrument using a driver by setting the values of only a few components at a time. See also **Component**, **Driver Files**, and **State Driver**.

Composite Data Type
A data type that has an associated shape. See also **Data Shape** and **Data Type**.

Configure
To arrange or modify software, hardware, or both in a computer system. In HP VEE, a menu selection with which you may change transaction array formats, the number of elements in a constant, and so forth.

Container
The package that is transmitted over lines and is processed by objects. Each container contains data, the data type, and the data shape. See also **Data Shape** and **Data Type**.

Context
A level of the work area that can contain other levels of work areas (such as nested `UserObjects`) but is independent of them. See also **UserObject**.

Control Pin
An asynchronous input pin that transmits data to the object without waiting for the object's other input pins to contain data. For example, control pins in HP VEE are commonly used to clear or autoscale a display.

Coupling
The interrelationship of certain functions in a test and measurement instrument. If in a state or component driver, functions A and B are coupled, changing the value of A may automatically change the value of B, even though you do not change B explicitly.

Crosshairs
A cross-shaped pointer in HP VEE that indicates that the software is waiting for your action. For example, when you see the crosshairs, you can select an object in the work area to perform some action on it, select a menu feature, or select any of the window controls (such as the scroll arrows, minimize or maximize buttons, and so forth).

Cursor
A white rectangular pointer in an entry field that shows where alphanumeric data will appear when you type information from the keyboard.

Data Field
The field within a transaction specification in which you specify either the expression to be written (WRITE transactions), or the variable to receive data that is read (READ transactions). See also **Transactions**.

Data Flow
The direction in which data moves in HP VEE. Data flows from left to right through objects and propagation has continued as far as it can in this way. Propagation continues through the sequence input and output pins.

Data Input Pin
A connection point on the left side of an object that permits data to flow into the object.

Data Output Pin
A connection point on the right side of an object that propagates data flow to the next object and passes the results of the first object's operation on to the next object.

DataSet
A collection of **Record** containers saved into a file for later retrieval. The `To DataSet` object collects Record data on its input and writes that data to a named file (the DataSet). The `From DataSet` object retrieves Record data from the named file (the DataSet) and outputs that data as Record containers on its `Rec` output pin. See also **Record**.

Data Shape
A pre-defined structure that defines how data is grouped together. See also **Array**, **Container**, and **Data Type**.

Data Type
A pre-defined structure that determines how data is organized and treated by HP VEE and supports common engineering constructs. See also **Container** and **Data Shape**.

Default
A value or action that HP VEE automatically selects.

Default Button
The button in a dialog box who's action is performed by default if Return is pressed or the selection is double-clicked. The default button has a recessed border.

Demote
To convert from a data type that contains more information to one that contains less information. See also **Data Type** and **Promote**.

Detail View
A view of a model in HP VEE that shows all the objects and the lines between them. Compare with **Panel View**.

Device
An instrument attached to or plugged into an HP-IB, RS-232, GPIO or VXI interface. Specific HP VEE-Test objects such as the `Direct I/O` object send and receive information to a device.

Device Driver
See **Instrument Driver**.

Dialog Box
A secondary window displayed when HP VEE requires information from you before it can continue. For example, a dialog box may contain a list of files from which you must choose a file before HP VEE can perform a particular operation.

Directory
A collection of files. For example, your `/users` directory usually contains all the files you have created. See **$HOME** and **Startup Directory**.

Double-Click
To press and release a mouse button twice in rapid succession. Double-clicking is usually a short-cut to selecting and performing an action.

Drag
To press *and continue to hold down* a mouse button while moving the mouse. Dragging moves something (for example, an object or scroll slider) within the HP VEE window. A drag ends when you release the mouse button.

Driver
Software that allows a computer to communicate with other software or hardware more easily than with raw reads and writes. See also **Component Driver**, **Driver Files**, **Interface Driver**, and **State Driver**.

Driver Files
A set of files included with HP VEE-Test that contains the information needed to create `State Driver` and `Component Driver` objects for instrument control. These files (`.cid` files) are copied to your system's hard disk automatically when you install HP VEE-Test.

Edit
To make changes in a file or entry field containing text or data.

Entry Field
A field that is typically part of a dialog box or an editable object and is used for text entry. An entry field appears recessed. For example, the open view of the `For Range` object has entry fields where you type values that specify the beginning, ending, and step values.

Error Message
Information that appears in a special type of dialog box in the HP VEE window and explains that a problem has occurred.

Error Pin
A pin that traps any errors that occur in an object. Instead of getting an error message, the error number is output on the error pin. When an error is generated, the data output pins are not activated.

Execute
The action of a model, or parts of a model, running.

Execution Flow
The order in which objects operate. See also **Data Flow**.

Expression
An equation in an entry field that can contain the input terminal names and any `Math` or `AdvMath` functions. An expression is evaluated at run-time. Expressions are only allowed in the `Formula`, `If/Then/Else`, `Get Values`, and I/O transaction objects.

Feature
An item on a menu that you select to cause a particular action to occur (for example, to open a file), or to get a particular object.

Feedback
A continuous thread path of sequence and/or data lines that uses values from the previous execution to change values in the current execution.

File
A set of information (such as a model or data) that is stored in an area of computer storage.

Flow
See **Data Flow** and **Execution Flow**.

Function
The name and action of objects where the output is a function of the input. These objects are located under `Math` or `AdvMath` menus and may be used in the `Formula` object. For example `sqrt(x)` is a function; `+` is not.

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

with `Set Global` in the root context of the model, and can be accessed by name with `Get Global` or from certain expressions within the context of a `UserObject`. However, a local variable with the same name as the global variable takes precedence in an expression.

Grayed Feature
A menu feature that is not currently available for use. For example, `Move Object` is grayed when no objects are selected.

Highlight
1. The colored band or shadow around an object that provides a visual cue to the status of the object.

2. The change of color on a menu feature that indicates you are pointing to that feature.

Host
To begin a thread or subthread. For example, the subthread that is hosted by `For Count` is the subthread that iterates.

$HOME
Your home directory (usually `/users`).

HP-UX
Hewlett-Packard Company's enhanced version of the UNIX™ operating system.

Icon
The small, graphical representation of an HP VEE object, such as the representation of an instrument, a control, or a display. Compare with **Open View**.

Instrument Driver
See **Driver Files**, **Component Driver**, and **State Driver**.

Interface
HP-IB, RS-232, GPIO, and VXI are referred to as interfaces used for I/O. Specific HP VEE-Test objects, such as the `Interface Event` object can only send commands to an interface.

Interface Driver
Software that allows a computer to communicate with a hardware interface, such as HP-IB or RS-232. Also called *device driver* in the UNIX™ operating system, interface drivers are configured into the kernel of the operating system.

Interrupt
A signal that requires immediate attention that may suspend a process, such as the execution of a computer program. An interrupt is usually caused by an event external to that process; after the interrupt is serviced, the process may be resumed.

Label
The text area or name on an icon or button that identifies that object or button.

Library
A collection of often-used objects or small models grouped together for easy access.

Line
A link between two objects in HP VEE that transmits data containers to be processed. See also **Subthread** and **Thread**.

Log In
The process of typing in a valid user name and its associated password (if one exists) to gain access to a computer system.

Login
A valid name and password (if one exists) that you use to log in to a computer system.

Main Menu
The menus located in the HP VEE menu bar. The main menus may be opened by clicking or dragging on the menu titles in the menu bar.

Main Work Area
The area where you create a model. The main work area is the parent context of all other contexts.

Mapping
To associate a set of independent values with an array, when the array is a function of the values.

Maximize
To enlarge a window to its maximum size. You maximize a window by selecting the square button on the right side of the window's title bar. In HP VEE, the `UserObject` has a maximize button.

Menu
A collection of features that are presented in a list. See also **Cascading Menu**, **Main Menu**, **Object Menu**, **Pop-Up Menu**, and **Pull-Down Menu**.

Menu Bar
A rectangular bar at the top of the HP VEE window that contains titles of the pull-down, main menus from which you select features.

Menu Title
The name of a menu within the HP VEE menu bar. For example, `File` or `Edit`.

Minimize
1. To reduce an open view of an object to its smallest size—an icon.
2. To reduce an X11 window to its smallest size—an icon.

Model
In HP VEE, a set of objects connected with lines that simulates an engineering problem and, when run, provides a solution to that problem. You build, modify, and run models in HP VEE by selecting objects from menus and connecting them in the work area. A model can contain multiple threads. You load a model into the HP VEE work area with `Open`. A model includes both the detail and panel views and all related contexts.

Mouse
A pointing device that you move across a surface to move a pointer within the HP VEE window.

Mouse Button
One of the buttons on a mouse that you can click, double-click, or drag to

perform a particular action with the corresponding pointer in the HP VEE window.

Object
A graphical representation of an element in a model, such as an instrument, control, display, or mathematical operator. An object is placed on the work area and connected to other objects to create a model. Objects can be displayed as icons or as open views. See **Icon** and **Open View**.

Object Menu
The menu associated with an object that contains features that operate on the object such as moving, sizing, copying, deleting, and adding inputs to the object. It is accessed by clicking on the upper-left corner of an open view or clicking the right mouse button on any non-field area on the object.

Open
To start an action or begin working with a text, data, or graphics file. When you select `Open` from HP VEE, a model is loaded into the work area.

Open View
The representation of an HP VEE object that is more detailed than an icon. Within the open view, you can modify the operation of the object and change the object's title. Compare with **Icon**. See also **Object**.

Operate
The action of an object processing data and outputting a result. An object operates when its data and sequence input pins have been activated. See **Activate**.

Outline Box
A box that represents the outer edges of an object or set of objects and indicates where the object(s) will be placed in the work area.

Network
A group of computers and peripherals linked together to allow the sharing of data and work loads.

Palette
A set of colors and fonts that is supplied with HP VEE and used in your HP VEE environment.

Panel
Information displayed in the center of the object's open view. In a `UserObject`, the panel contains a work area. In a `For Count` object, the panel contains an entry field. Compare with **Panel View**.

Panel View
The view of a model in HP VEE that shows only those objects needed for the user to run the model and view the resultant data. You create a panel view to meet the needs of your users. Compare with **Detail View** and **Panel**.

Pin
An external connection point on an object to which you can attach a line. See also **Control Pin**, **Data Input Pin**, **Data Output Pin**, **Error Pin**, **Sequence Input Pin**, **Sequence Output Pin**, **Terminal**, and **XEQ Pin**.

Pointer
The graphical image that maps to the movement of the mouse. A pointer allows you to make selections and provides you feedback on a particular process underway. HP VEE has pointers of different shapes that correspond to process modes, such as an arrow, crosshairs, and hourglass. See also **Arrow** and **Crosshairs**. Compare with **Cursor**.

Pop-Up Menu
A menu that provides no visual cue to its presence, but simply pops up when you perform a particular action. For example, the `Edit` menu in HP VEE pops up when you position the pointer in the work area and then click the right mouse button.

PostRun
The set of actions that are performed when the model is stopped.

PreRun
The set of actions that resets the model and checks for errors before the model starts to run.

Priority Thread
A priority thread executes to completion blocking all other parallel threads from executing. Certain of the I/O objects for devices and interfaces will host a priority thread.

Promote
To convert from a data type that contains less information to one that contains more information. See also **Data Type** and **Demote**.

Propagation
The rules that objects and models follow when they operate or run. See also **Data Flow** and **Execution Flow**.

Pterodactyl
Any of various extinct flying reptiles of the order Pterosauria of the Jurassic and Cretaceous periods. Pterodactl are characterized by wings consisting of a flap of skin supported by the very long fourth digit on each front leg.

Pull-Down Menu
A menu that is pulled down from the menu bar when you position the pointer over a menu title and click or drag the left mouse button.

Radio Button
A diamond-shaped button in HP VEE dialog boxes that allows you to select a setting that is mutually exclusive with other radio buttons in that dialog box. To select a setting, click on the radio button. To remove the setting, click on another radio button in the same dialog box.

Record
A data type that has named data fields which can contain multiple values. Each field can contain another Record container, a Scalar, or an Array. The Record data type has the highest precedence of all HP VEE data types. However, data cannot be converted to and from the Record data type through the automatic promotion/demotion process. Records must be built/unbuilt using the using `Build Record` and `UnBuild Record` objects.

Remote Function
A User Function running on a remote host computer, which is callable from the local host. The `Import Library` object starts the process on the remote host and loads the Remote File into the HP VEE process on the local host. You can then call the Remote Function with the `Call Function` object, or from certain expressions. See also **User Function** and **Compiled Function**.

Resource Manager
A program which exists on VXI controllers that runs at start-up and after a VXI system reset. This program initializes and manages the instruments in a VXI card cage.

Restore
To return a minimized window or an icon to its full size as a window or open view by double-clicking on it.

Run
To start the objects on a model or thread operating.

Save
To write a file to a storage device, such as a hard disk, for safekeeping.

Scalar
A data shape that contains a single value. See also **Data Shape**.

Schema
The structure or framework used to define a data record. This includes each field's name, type, shape (and dimension sizes) and mapping.

Screen Dump
A graphical printout of a window or part of a window.

Scroll
The act of using a scroll bar either to move through a list of data files or other choices in a dialog box or to pan the work area.

Scroll Arrow
An arrow that is part of a scroll bar and, when clicked on, moves you through a list of data files or other choices in a dialog box or pans the work area.

Scroll Bar
A graphical device used either to move through a list of data files or other choices in a dialog box or to pan the work area. A scroll bar consists of one or more scroll sliders and scroll arrows.

Scroll Slider
A rectangular bar that is part of a scroll bar and, when dragged, moves you through a list of data files or other choices in a dialog box or pans the work area.

Select
To choose an object, an action to be performed, or a menu item. Usually you select by clicking with your mouse.

Select Code
A number used to identify the logical address of a hardware interface. For example, the factory default select code for most HP-IB interfaces is 7.

Selection
1. A menu selection (feature).
2. An object or action you have selected in the HP VEE window.

Sequence Input Pin
The top pin of an object. When connected, this input pin must be activated before the object will operate.

Sequence Output Pin
The bottom pin of an object. When connected, this output pin is activated when the object and all data propagation from that object finishes executing.

Sequencer
An object that controls execution flow through a series of sequence transactions, each of which may call a **User Function**, **Compiled Function**, or **Remote Function**. The sequencer is normally used to perform a series of tests by specifying a series of sequence transactions.

Shell
The program that interfaces between the user and the operating system.

Shell Prompt
The character or characters that denote the place where you type commands while at the operating system shell level. The prompt you see displayed depends upon the type of shell you are running, such as a **#** prompt for the Bourne shell.

Sleep
An object sleeps during execution when it is waiting for an operation or time interval to complete or for an event to occur. A sleeping object will allow other parallel threads to run concurrently. Once the event, time interval, or operation occurs, the object will execute, allowing execution to continue.

Startup Directory
The directory from which you type `veeengine` or `veetest`. This directory determines the default paths for most file actions including `Save` and `Open`.

State
A particular set of values for all of the components related to an HP VEE-Test instrument driver which represents the measurement state of an instrument. For example, a digital multimeter uses one state for high-speed voltage readings and a different state for high-precision resistance measurements. See also **State Driver**.

State Driver
An instrument control object that forces all the function settings in the corresponding physical instrument to match the settings in the control panel displayed in the open view of the object. See also **Component Driver**, **Driver Files**, and **State**.

Step
The action of operating one object at a time. An arrow points to the object that will operate next.

Submenu
See **Cascading Menu**.

Subthread
A portion of a thread.

Synchronous
A method of execution that requires all events to occur before operation.

Terminal
The internal representation of a pin that displays information about the pin

and the data container held by the pin. Double-click on the terminal to view the container information.

Thread
A set of objects connected by solid lines in an HP VEE model. A model with multiple threads can run all threads simultaneously.

Title Bar
The rectangular bar at the top of the HP VEE window or the object's open view where the model's or object's name is shown.

Transactions
The specifications for input and output (I/O) used by certain objects in HP VEE, such as `To File` and `From File`, as well as by `Direct I/O` and `Sequencer`. Transactions appear as English-like phrases listed in the open view of I/O objects.

User-Defined Function
HP VEE allows three types of user-defined functions: the **User Function**, **Compiled Function**, and **Remote Function**. See also **Function**, **User Function**, **Compiled Function**, and **Remote Function**.

User Function
A user-defined function created from a **UserObject** by executing `Make UserFunction`. The User Function exists in background, but provides the same functionality as the original UserObject. You can call a User Function with the `Call Function` object, or from certain expressions. A User Function can be created and called locally, or it can be saved in a library and imported into an HP VEE model with `Import Library`. See also **Compiled Function**, **Remote Function**, and **UserObject**.

User Interface
The part of an application that permits a user and the application to communicate with each other to perform certain tasks. HP VEE uses a graphical user interface, which includes windows, menus, dialog boxes, and objects.

UserObject
An object that can encapsulate a group of objects that perform a particular function. A `UserObject` allows you to use top-down design techniques when

building a model and to build user-defined objects that can be saved in a library and reused. See also **Context**.

View
See **Detail View**, **Icon**, **Open View**, and **Panel View**.

Wait
See Sleep.

Window
A rectangular area on the screen that contains a particular application program, such as HP VEE or the shell.

Window Frame
The area surrounding a window that contains a resize border, window menu button, minimize and maximize buttons, and a title area.

Work Area
The area within the HP VEE window or the open view of a `UserObject` where you group objects together. When you `Open` a model, it is loaded into the main work area. The panel of a `UserObject` is a work area. See also **Panel**.

X Window System (X11)
An industry-standard windowing system used on UNIX™ computer systems. See also **X11 Resources**.

X11 Resources
A file or set of files that define your X11 environment.

XEQ Pin
A pin that forces the operation of the object, even if the data or sequence input pins have not been activated. See also **Control Pin**, **Data Input Pin**, and **Sequence Input Pin**.

Index

B

C

D

E

F

G

H

I

M

P

Q

R

S

T

U

X

Y