Using HP VEE

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
Edition 3 - September 1993

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

Contents

1

Using HP VEE Elements

Using HP VEE Elements

What is HP VEE?

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

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

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

The HP VEE family includes several separate products:

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

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

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

About This Manual

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

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

Starting HP VEE

To start HP VEE on a UNIX system, 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.

To start HP VEE on a Microsoft® Windows system, double click on the HP VEE icon in the HP VEE for Windows application window.

You can specify how you want HP VEE to run by using command line options. For HP VEE for Windows, use the Program Manager `Run` ... menu pick under the `File` menu to specify command line options.

- *filename*

  HP VEE is started and the program 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, located in *directory* instead of the default HP VEE installation directory.

- `-r` *filename*

  The `-r` option starts HP VEE and runs the program specified by *filename*. When the program has completed, 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* (UNIX only)

  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* (UNIX only)

  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` (UNIX only)

  Shows the command line options.

- `-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* (UNIX only)

  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 (UNIX)" in Appendix A in the *HP VEE Advanced Programming Techniques* manual.

- `-prname` *name* (UNIX only)

  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 (UNIX)" in Appendix A in the *HP VEE Advanced Programming Techniques* manual for further information.

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. For HP VEE for Windows, this directory will generally be **C:\VEE_USER**.

When you **Merge** or **Save Objects**, the default directory is **/usr/lib/veeengine/lib/** or **/usr/lib/veetest/lib/** or **C:\VEE\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, **/**).

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.

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. For information about trapping errors with an error output pin, refer to "Trapping Errors" in Chapter 2.

If you don't get a caution or error message, but your program doesn't work as you expected, refer to "Troubleshooting Problems" in the *HP VEE Advanced Programming Techniques* manual.

Accessing Example and Library Programs

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

Examples

For your convenience, many example programs are provided with HP VEE. On installation, these examples are stored in subdirectories under `/usr/lib/veeengine/examples/` or `/usr/lib/veetest/examples/` or `C:\VEE\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 on UNIX systems. If you modify an example program, you'll have to save it in a different directory, such as in your `$HOME` directory.

Refer to "Example Programs and Library Objects," in the *HP VEE Advanced Programming Techniques* manual for further information.

Libraries

HP VEE also contains library programs that you can incorporate in your programs. They are stored in `/usr/lib/veeengine/lib/`, `/usr/lib/veetest/lib/` or `C:\VEE\LIB`. The `lib` directory is read-only on UNIX systems. If you modify a library program, 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 programs. This directory is writable by everyone so you can use files you and others saved in `contrib/`.

Numeric Data Entry Fields

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

Table 1-1. ISO Abbreviations

Abbreviation	Suffixes	Multiple
X	exa	10^{18}
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}
a	atto	10^{-18}

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**. (To add a terminal, **Show Terminals** must be active and the pointer must be over the terminal area of the object.)
CTRL-D	Object menu ⟹ **Terminals ⟹ Delete Input** or Object menu ⟹ **Terminals ⟹ Delete Output**. (To delete a terminal, **Show Terminals** must be active and the pointer must be over the terminal to be deleted. Otherwise, CTRL-D will delete the object itself.)
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 (except when the pointer is over a terminal, as described above.)

CTRL-C	Pauses a running program or cancels an edit
Shift-Print	Prints screen with the current options specified in `Printer Config` (UNIX only)
Print Screen	Copies screen to clipboard. (MS-Windows only)
Arrow Keys (▲, ▼, ◀, ▶)	Moves the pointer (UNIX only)
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 program to the upper left corner of the work area
Shift-↖	Moves the lower right corner of the program 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.

Understanding How Programs Run

With HP VEE, you build a program to solve your engineering problem. A program looks like a block diagram. When you run the program, 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.

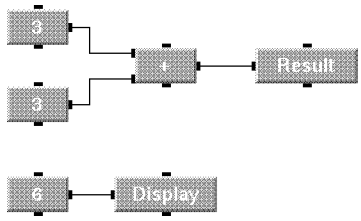


Figure 1-1. Two Parallel Threads

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.

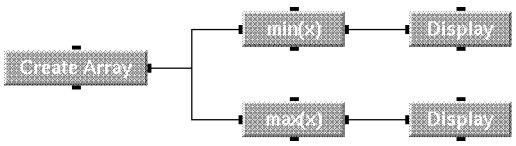


Figure 1-2. Two Subthreads

Running Programs

You run a program 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 program by adding a `Start` object to the thread and then pressing `Start`.

After you start to run your program (by pressing `Run` or `Start`), HP VEE initializes your program 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 "Understanding PreRun".

Stopping Programs

There are three reasons why a program stops.

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

When a program 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 program communicates with other gcs, you need to monitor their execution. They do not necessarily stop when the program stops.

Understanding PreRun

After you start a program but before the objects start to operate, HP VEE PreRuns your program. 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 program 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 program 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 program (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 program is run and is done for each `UserObject` every time it operates. For more information about contexts, refer to Chapter 4.

Understanding Propagation

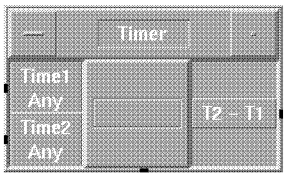
Propagation is the general flow of execution through your program. The propagation guidelines define the order in which objects operate.

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 1-3, 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 1-3. Object Operation

Basic Propagation Order

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

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

> **NOTE**
>
> **Start** is not required *unless* your program 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 1-4, 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.

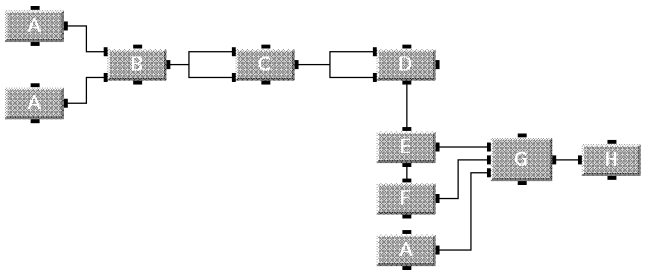


Figure 1-4. Example of Basic Propagation

The program shown in Figure 1-4 is saved in `manual01.vee` in your `examples` directory.

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 1-5, the order of operation is `A`, `B`, `C`, `D`, `E`, `F`, and `G`.

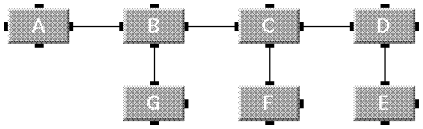


Figure 1-5. Propagation Through Data and Sequence Pins

The program shown in Figure 1-5 is saved in `manual02.vee` in your `examples` directory.

When you have parallel subthreads (subthreads that begin at the same data output), they operate in no particular order to each other. In fact, the order of propagation is affected by the order in which the objects were wired and whether the program has been saved to a file. In Figure 1-6, 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.

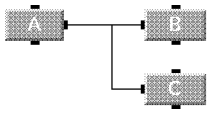


Figure 1-6. Propagation of Parallel Subthreads

Understanding Pins

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

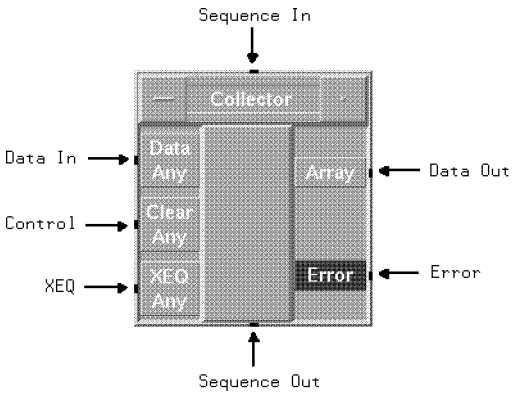


Figure 1-7. Pins on an Object

- **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 program 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) An error pin is an output pin that traps the error condition 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 2. For information about generating your own error codes within a `UserObject`, refer to "Raise Error" in Chapter 3.

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 program 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 program 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 1-8, the objects on both threads operate in parallel.

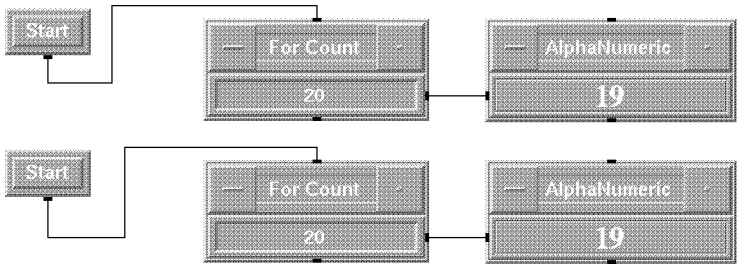


Figure 1-8. Running Multiple Threads

The program shown in Figure 1-8 is saved in `manual03.vee` in your `examples` directory.

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:

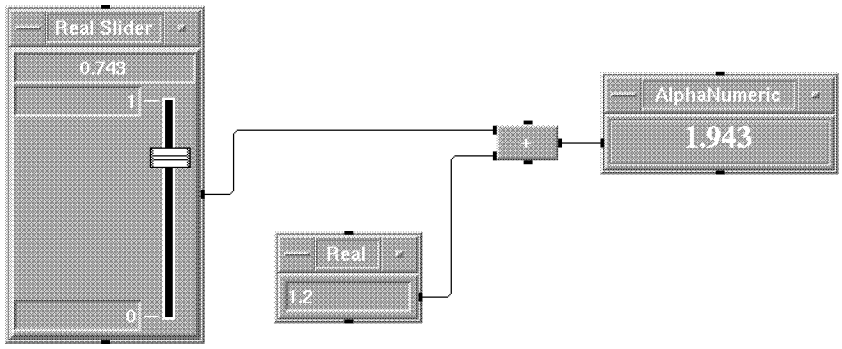


Figure 1-9. Using Auto Execute

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.*

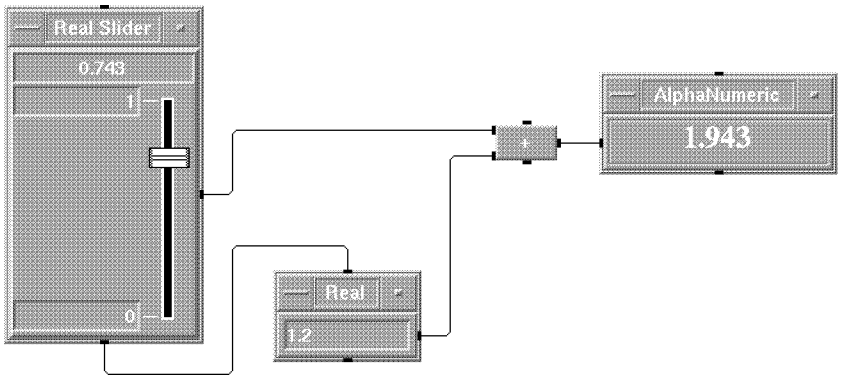


Figure 1-10. Using Sequence Pins with Auto Execute

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 program 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 1-11, the feedback loop is necessary to perform the recursive calculation: $x=(x)+2$.

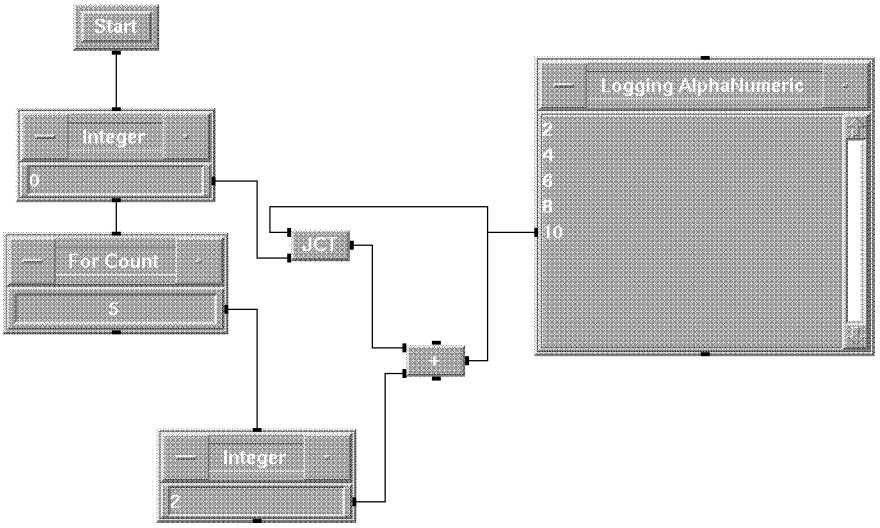


Figure 1-11. Example of Data Feedback

The program shown in Figure 1-11 is saved in `manual04.vee` in your `examples` directory. 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 1-12 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 program runs.

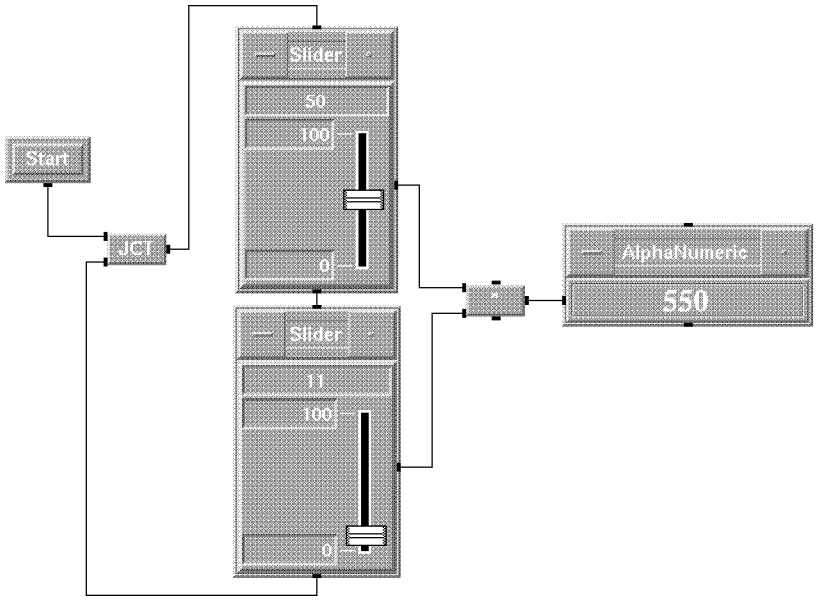


Figure 1-12. Example of Sequence Feedback

The program shown in Figure 1-12 is saved in `manual05.vee` in your `examples` directory.

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` object may be required to obtain correct propagation. Refer to "Iteration with Flow Branching" in the next chapter.

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. Remember that HP VEE-Engine does not support instrument I/O and therefore won't support the instrument I/O data types. 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

Note that these data types are not supported by HP VEE-Engine.

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 (±3.40282347E±38).

Instrument I/O Data Type Conversions

Note that these data type conversions are not supported by HP VEE-Engine.

On instrument I/O transactions involving integers, HP VEE 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 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 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 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 1-13. Note that the mappings are from the first element to the end of the last interval.

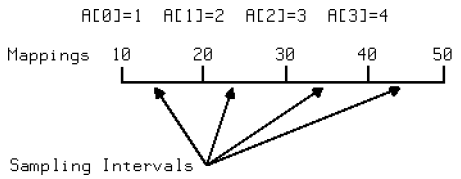


Figure 1-13. Array Mappings and Sampling Intervals

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 2.

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 "Using Records and DataSets" in the *HP VEE Advanced Programming Techniques* manual.

Table 1-2. 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.

Debugging Programs

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

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 program is running or paused. If the program 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 program is running.

- **Show Exec Flow** shows the object currently operating by surrounding it with a yellow highlight (default) when the program is running.

Using Breakpoints

Set breakpoints to stop the program 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 program.

When you run your program, 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 program 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 program 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

program), the program PreRuns. Press `Step` to operate each object. An arrow points to the object that will operate next.

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

When you want the program to continue running, press `Cont`. To pause the program, press `Stop` once. When the program 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 Programs With Others

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

- Include the `.veeio` file (UNIX) or `VEE.IO` file (MS-Windows) used.
- 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 program using the techniques described in "Documenting Your Programs".

If you are sharing a disk with others and want to share programs 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).

Changing 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 program 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 program uses: degrees, radians, or gradians. **Trig Mode** may also be set for each **UserObject**. This preference is saved with each program as well as in **.veerc** (or **VEE.RC** on PC's).
- **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 program as well as in **.veerc** (or **VEE.RC** on PC's).
- **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 program as well as in **.veerc** (or **VEE.RC** on PC's).
- **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** (UNIX only) 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** (UNIX only) 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** (or **VEE.RC on PC's) file in your $HOME** directory (typically your **/users** directory or **C:\VEE** directory on PC's).

For more information about customizing your HP VEE work sessions, refer to Appendix A in *HP VEE Advanced Programming Techniques*

Documenting Your Programs

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

- Give the program a meaningful symbolic name. You can name your program 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 program.
  - What this program does.
  - Who should use this program.
  - 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 "Configuring HP VEE" in the *HP VEE Advanced Programming Techniques* manual.

Archiving Programs

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

For an electronic archive, use the program 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 program, 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 program, 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 Configuring HP VEE in the *HP VEE Advanced Programming Techniques* manual.

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

2

How To Build HP VEE Programs

How To Build HP VEE Programs

This chapter explains the general tasks you'll do when building a program. For detailed information about the specific operation of an object, refer the *HP VEE Reference* manual.

Designing a Program

An important structured approach to building complex programs 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 program 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 programs if you take some time to design your program 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 program. 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 2-1 reads four values (separated by spaces or on different lines) from a file.

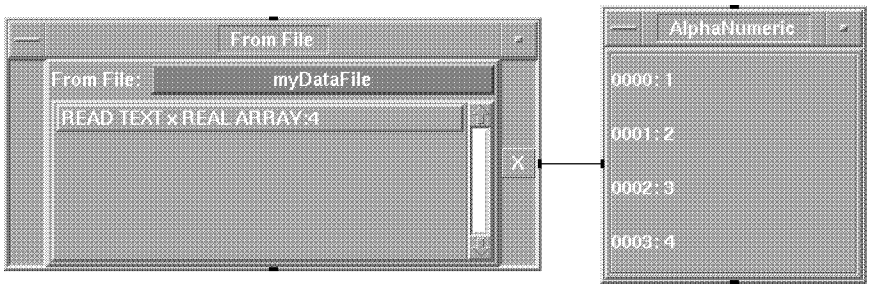


Figure 2-1. Reading Multiple Values From a File

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

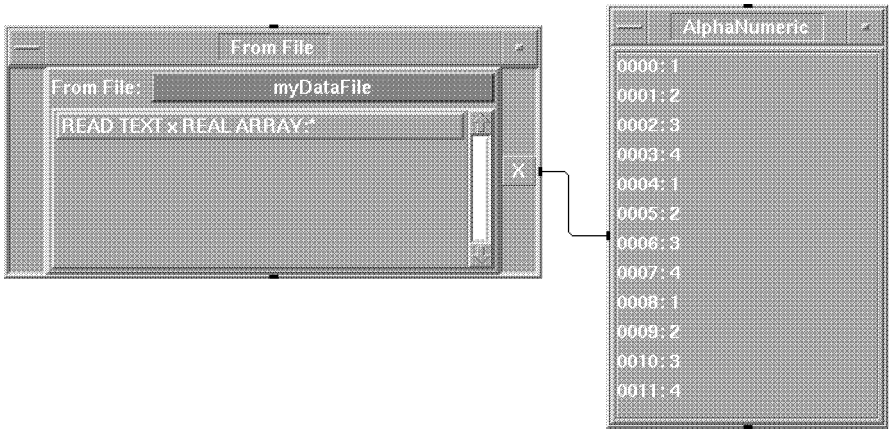


Figure 2-2. Using Read to End to Read Multiple Values

You can use the `From File` object in more complicated ways. For more information, refer to "Using Transaction I/O" in the *HP VEE Advanced Programming Techniques* manual.

To read multiple values when you don't know the number of values, click on the `SIZE:` field to get `To End:`. In Figure 2-2, 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 program. If the constant is not dependent on a sequence input, it will operate first. For more information about entry fields, refer to the *Getting Started with HP VEE for Windows* manual.

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 program 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 2-3 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.

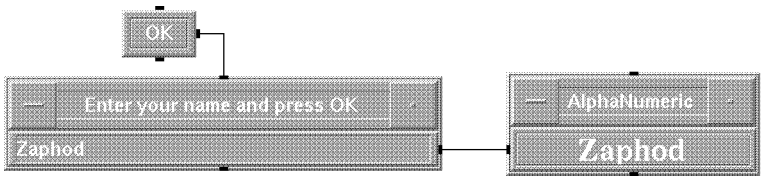


Figure 2-3. Getting User Input With a Constant Object

The program shown in Figure 2-3 is saved in `manual08.vee` in your `examples` directory.

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 2-4, 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.

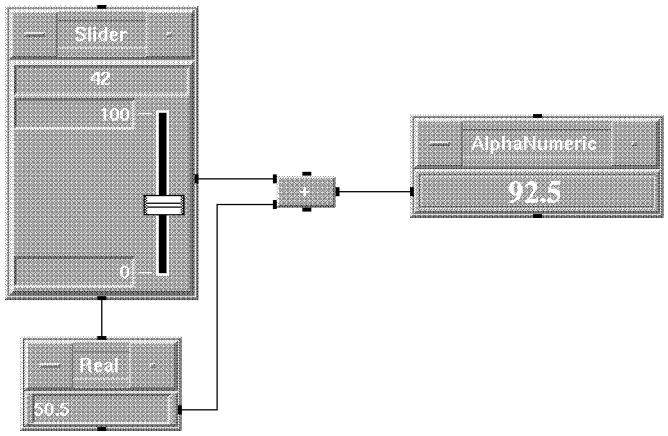


Figure 2-4. Example of Auto Execute

Refer to "Understanding Auto Execute" in Chapter 1 for information about how `Auto Execute` affects propagation. Refer to "Understanding Feedback" in Chapter 1 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 4.

Setting Values at Run-Time

Many objects allow you to set or clear values each time the program is run (at PreRun or Activate). For information about PreRun and Activate, refer to "Understanding PreRun" in Chapter 1.

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 2-5 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 program 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`.

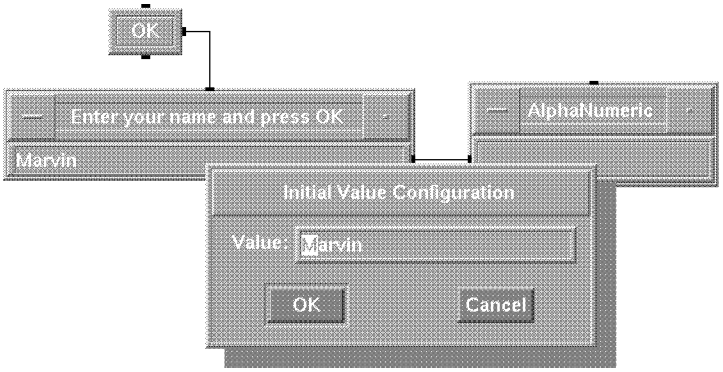


Figure 2-5. Initialize At PreRun

The program shown in Figure 2-5 is saved in `manual09.vee` in your `examples` directory.

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 program is running (not at a specific time such as PreRun or Activate).

The `Default Value` object (a `Text` constant) in Figure 2-6 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.

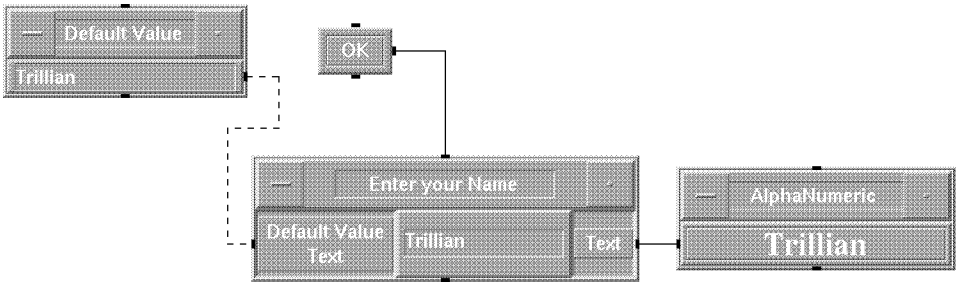


Figure 2-6. Resetting to a Default Value

The program shown in Figure 2-6 is saved in `manual10.vee` in your `examples` directory.

Controlling Program Flow

HP VEE programs use constructs to control the flow of operation. These controls allow you to:

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

Starting

Normally you'll start your program 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 program, you can use `Start` objects to test individual threads. `Run` starts all threads in the program, 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 program. However, there is one exception. Any thread in your program 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 2-7, 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.

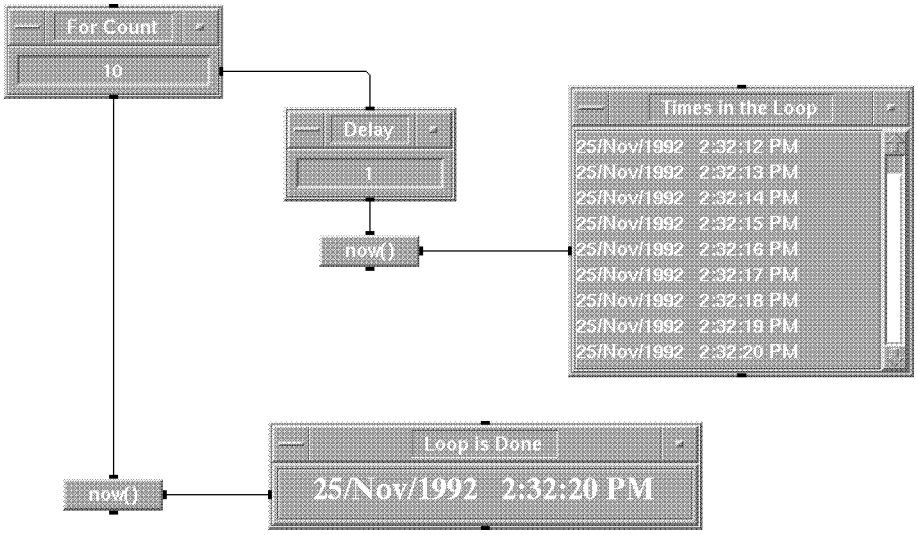


Figure 2-7. Iteration Example

The program shown in Figure 2-7 is saved in `manual11.vee` in your `examples` directory.

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 program 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 program stops.

If you want to pause the program momentarily, press the `Stop` button once. Use the `Cont` or `Step` button to continue execution. If you want your program 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 program 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.

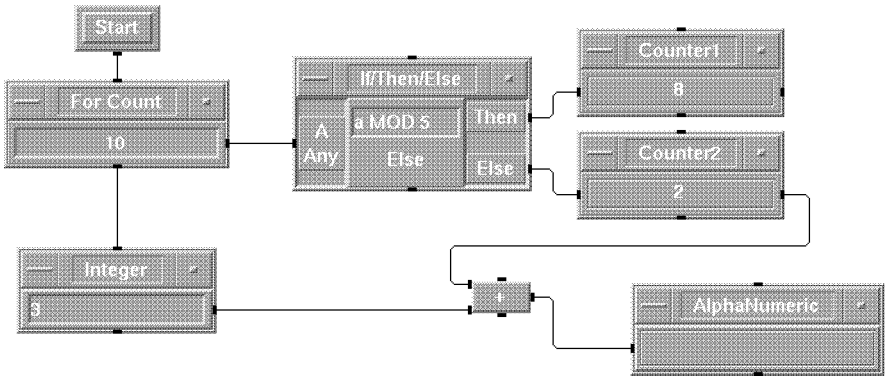


Figure 2-8. Iteration and Flow Branching

`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 iterations 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:

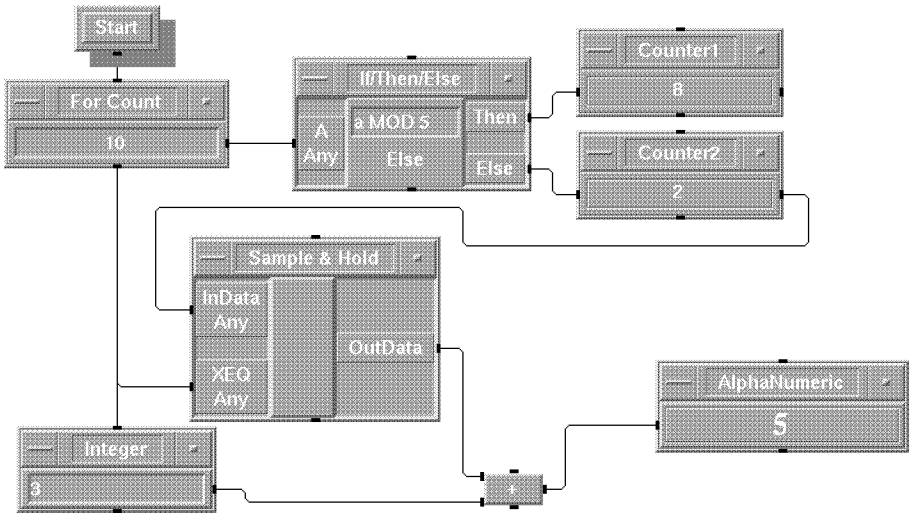


Figure 2-9. Using Sample & Hold

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 programs shown in Figure 2-8 and Figure 2-9 are included as examples "`manual12.vee`" and "`manual13.vee`", respectively. These examples are located in `manual12.vee` and `manual13.vee` in your `examples` directory.

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 program by using numeric entry fields such as those in **Constant** objects. Numeric entry fields on some objects support the use of arbitrary formulas. The formula is immediately evaluated; 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 program uses a `Set Global` device to define the global variable `numFiles`. Elsewhere in the program, 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 program, 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 program, 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 program.

NOTE

The syntax of the left-hand expression in the `Set Field` object is limited. Refer to `Set Field` in the *bookref* manual.

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 programs for dB conversions in the HP VEE `lib/conversions/` or `C:\VEE\LIB\CONVERT\` directory.

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 program, HP VEE normally stops running your program and displays an error dialog box that presents the error number and error message.

To trap the error so that the program 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 `UserObject`s, refer to Chapter 3.

Figure 2-10 shows how a "Divide by Zero" error is trapped and the iteration continues.

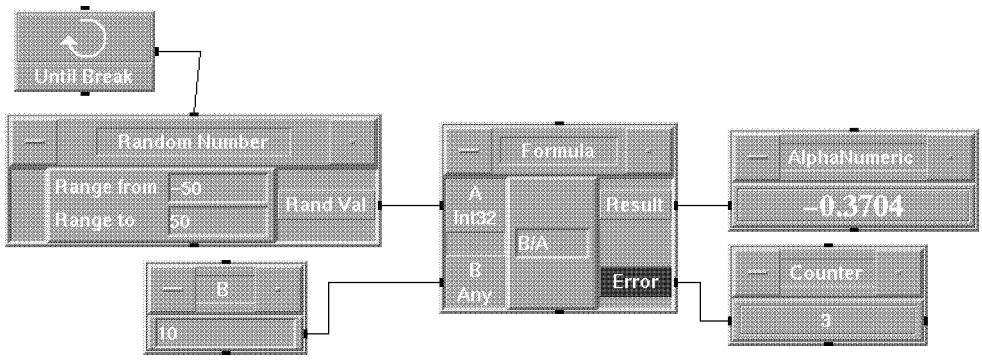


Figure 2-10. Trapping an Error

The program shown in Figure 2-10 is saved in `manual14.vee` in your `examples` directory.

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 3 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 1.

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

Table 2-1. 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 `xy_cntrl` in your `examples/lib` directory to get pre-built Records for these inputs. Or you can build your own Records. Refer to `xy_cntrl.vee` for an example of their use.

To display a family of curves (from one data input) while the program 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 program, select the `Next Curve` object menu feature before you run the program 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 program 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 "Using Transaction I/O" in the *HP VEE Advanced Programming Techniques* manual.

Exporting Graphics to a Report

You can use graphical program information such as the program itself, parts of the program, 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.

Printer Configuration (UNIX)

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.

Printer Configuration (MS-Windows)

To get a graphical image from HP VEE for Windows, open the MS-Windows **Control Panel** and double-click on the **Printers** icon. Press the Connect... button in the **Printers** dialog box. In the **Ports:** window select **FILE:** and then press the OK button. Verify that the **Default Printer** is selected to print to **on FILE:**. Then press the Close button. Now when you use the techniques described below MS-Windows will ask you which file to use when printing your output. The file is written in PCL format.

Graphic Output Techniques

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

- To get an image of the open view of an object when the program 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 program is running, operate the `Print Screen` object.
- To get an image when the program is not running, select `Print Screen`, `Print Objects`, or `Print All` from the `Edit` menu

NOTE

To manually capture graphics in a file from HP VEE for Windows press the Print Screen key. This places the screen graphics on the clipboard. You can paste this graphic in the Microsoft Windows `Paintbrush` program using the `Paste` menu selection from the `Edit` menu. Then save the file as a `.BMP` bitmap file. Many MS-DOS editors accept `.BMP` files for graphical input.

Output Formats (UNIX)

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

Exporting to Document Publishing Packages

Often, you may want to capture images of programs, 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 program. 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 program. 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 program. 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**.

Optimizing Your Program

Although the time to run a program 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 program 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 program 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 programs 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 2-11.

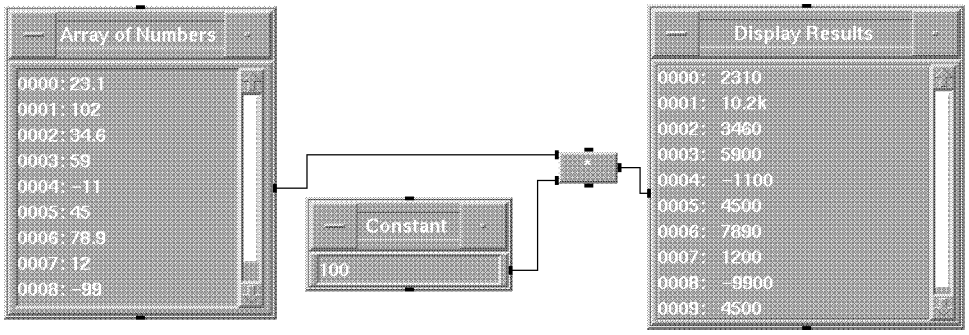


Figure 2-11. Example of a Parallel Operation

The program shown in Figure 2-11 is saved in `manual24.vee` in your `examples` directory.

Figure 2-12 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.

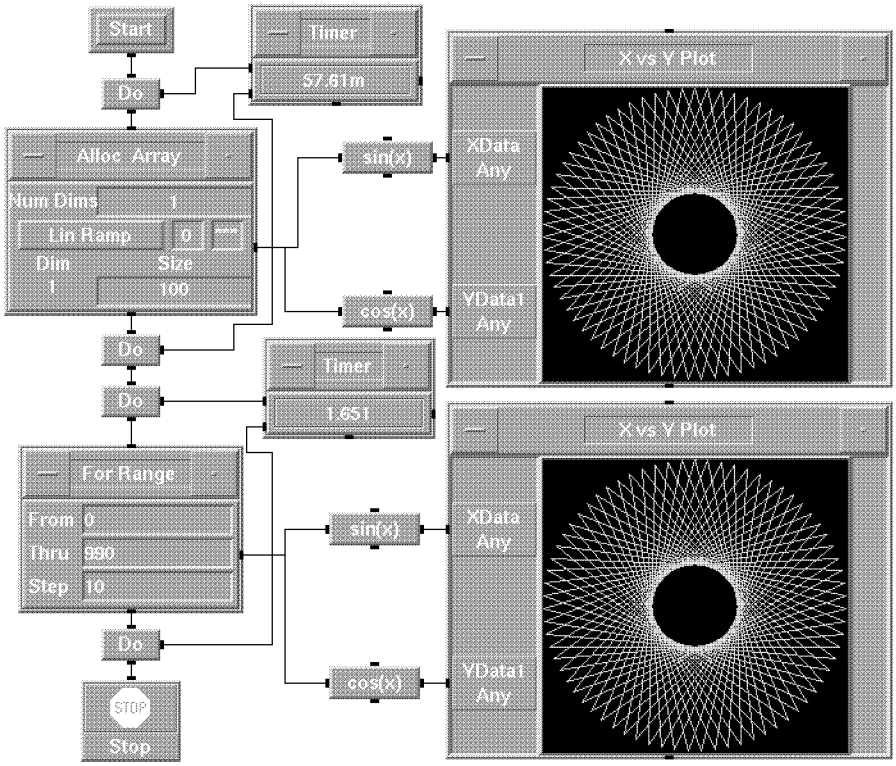


Figure 2-12. Another Parallel Operation Example

The program shown in Figure 2-12 is saved in `manual25.vee` in your `examples` directory.

Showing the Icon Instead of the Open View

Figure 2-13 shows that updating any display (including the open view of a `Counter`) takes time: you can increase the speed of your program by iconifying display objects.

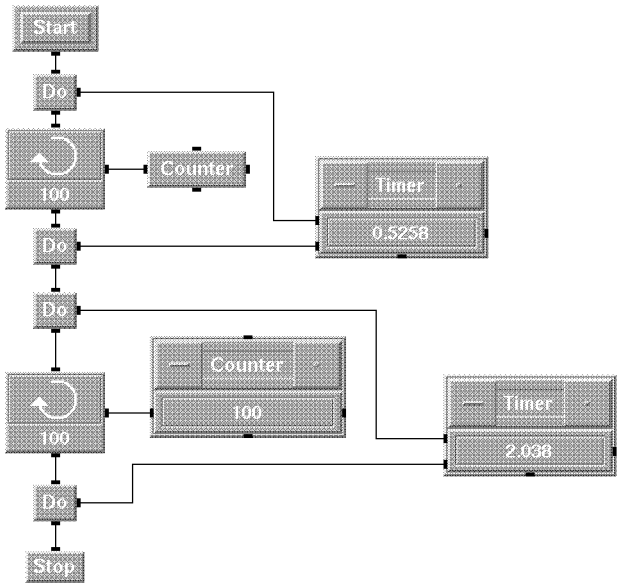


Figure 2-13. Increasing Speed with An Icon

The program shown in Figure 2-13 is saved in `manual26.vee` in your `examples` directory.

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 program becomes easier to maintain.

Figure 2-14 shows you two ways to complete the same operation. The second thread (on the bottom) is more compact than the first thread.

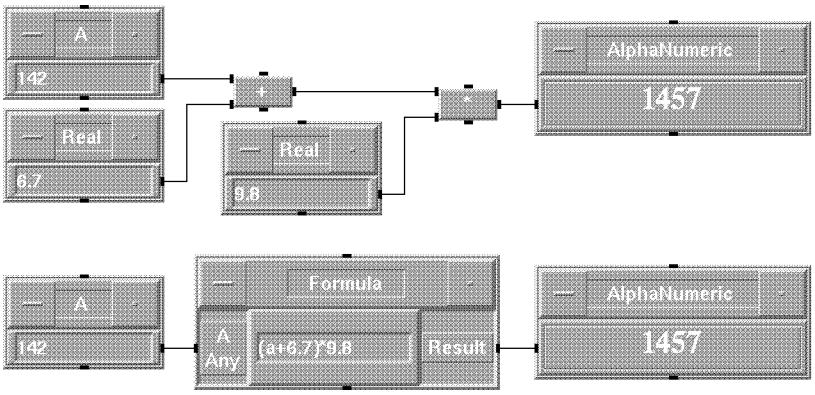


Figure 2-14. Compact Math Example

3

How To Create Your Own Objects and Functions

How To Create Your Own Objects and Functions

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 programs.

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

- Build user-defined objects that you can save in a library for later re-use.

  Once a `UserObject` is created and saved, you can `Merge` it in other programs 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 4.

Understanding UserObjects

A `UserObject` is an enclosed environment that provides a virtual HP VEE work area. Any program 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 program 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`.

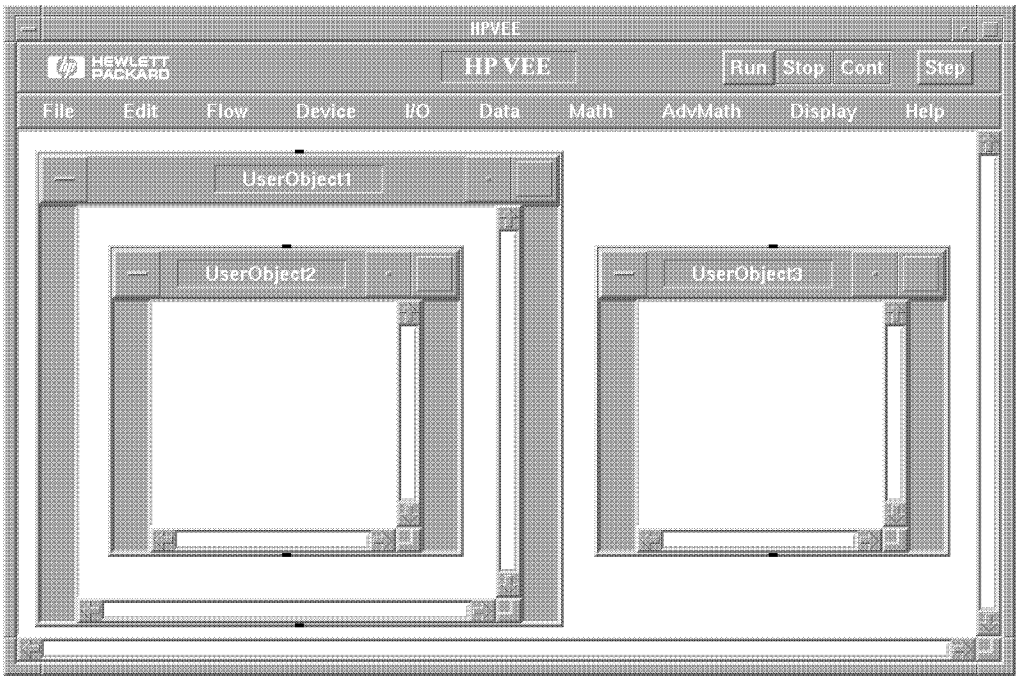


Figure 3-1. Four Different Contexts

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 program, 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 1. 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 `UserObject`s.
- 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 1 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 program. When the program 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 program. 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 program.

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 program

continues to run and you can process the error. For more information about trapping errors, refer to "Trapping Errors" in Chapter 2

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 3-2 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.

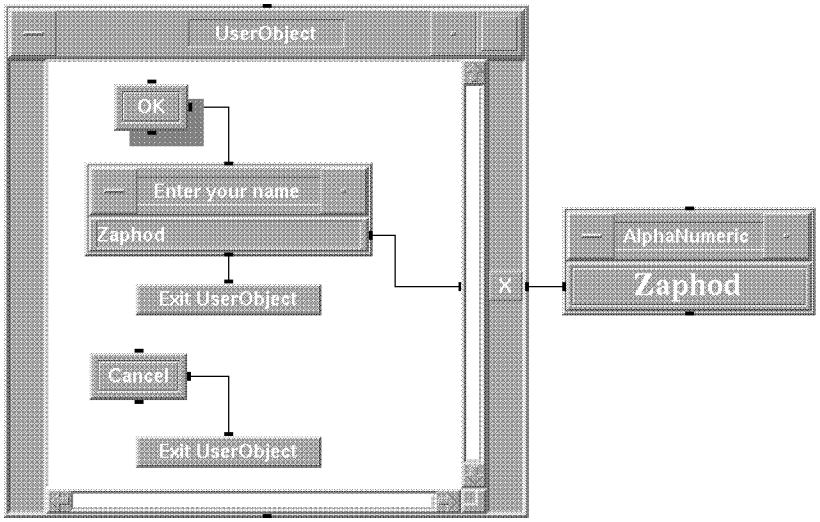


Figure 3-2. Example of `Exit UserObject`

The program shown in Figure 3-2 is saved in `manual18.vee` in your `examples` directory.

NOTE

In the above example, `Exit UserObject` is connected so that it will not operate until the data propagates to the User Object output data pin `X`. If the `Exit UserObject` were connected to the data output of the object labeled `Enter your name` the order of execution would not be determinant. The User Object might exit before passing the data to its output terminal. Therefore, to ensure that data is propagated before terminating, the `Exit UserObject` must be connected to the sequence out pin.

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 program stops running and an error dialog box is displayed. It contains the message and the error number.

To create a robust and usable program, 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 3-3 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.

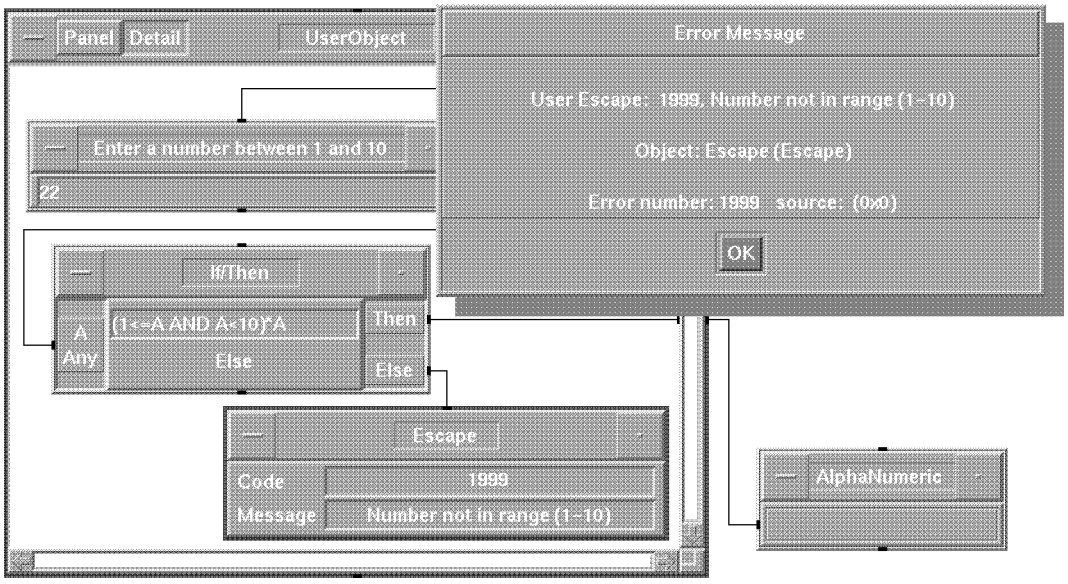


Figure 3-3. Using a `Raise Error`

The program shown in Figure 3-3 is saved in `manual19.vee` in your `examples` directory.

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 programs. 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 4.

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` or `C:\VEE\LIB`.

To retrieve functions from your library, select `Merge` from the `File` menu. `Merge` allows you to keep the existing program on the work area so you may add to it.

Occasionally, you may want to merge a library object or program 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`.

4

How To Build an Operator Interface

How To Build an Operator Interface

HP VEE provides a powerful tool to create an operator interface for your program with the panel view. The panel view uses the graphical interface provided by HP VEE while hiding the details of the program.

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 program. (This could be as little as the data input fields and a display).

- Protects the program from intervention by a user.

  From a panel view, a user does not see the details of a program. From a secured panel view, a user cannot change the program in any way.

- Allows you to create a complete application from your program.

  A panel view provides a user interface to the complex program, making an application that is easier to use and understand.

- Improves the performance of programs by decreasing run time.

  Updating object views takes time. Depending on the complexity of a program, running from the simpler panel view may increase performance.

Understanding Panel Views

A **panel view** is an alternate view of the program that provides a user interface to your program (you created your program 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 program 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 4-1 and Figure 4-2 show the detail view and the panel view of a program 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.

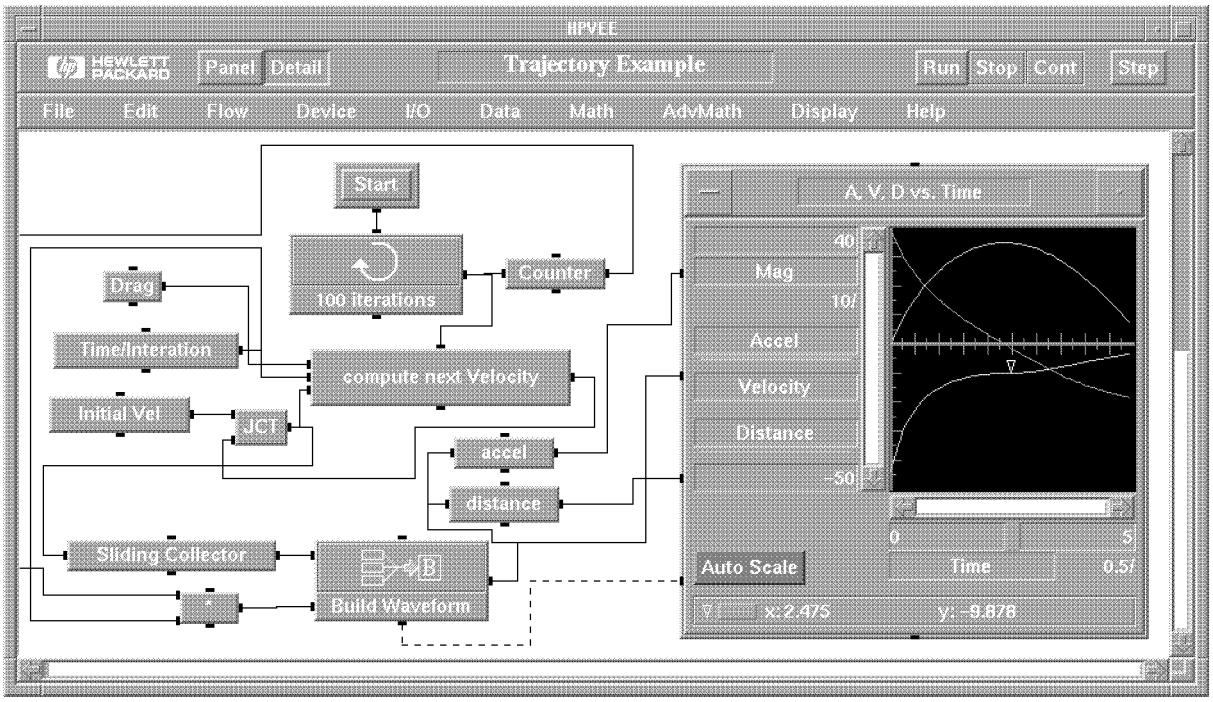


Figure 4-1. Detail View of Trajectory Example

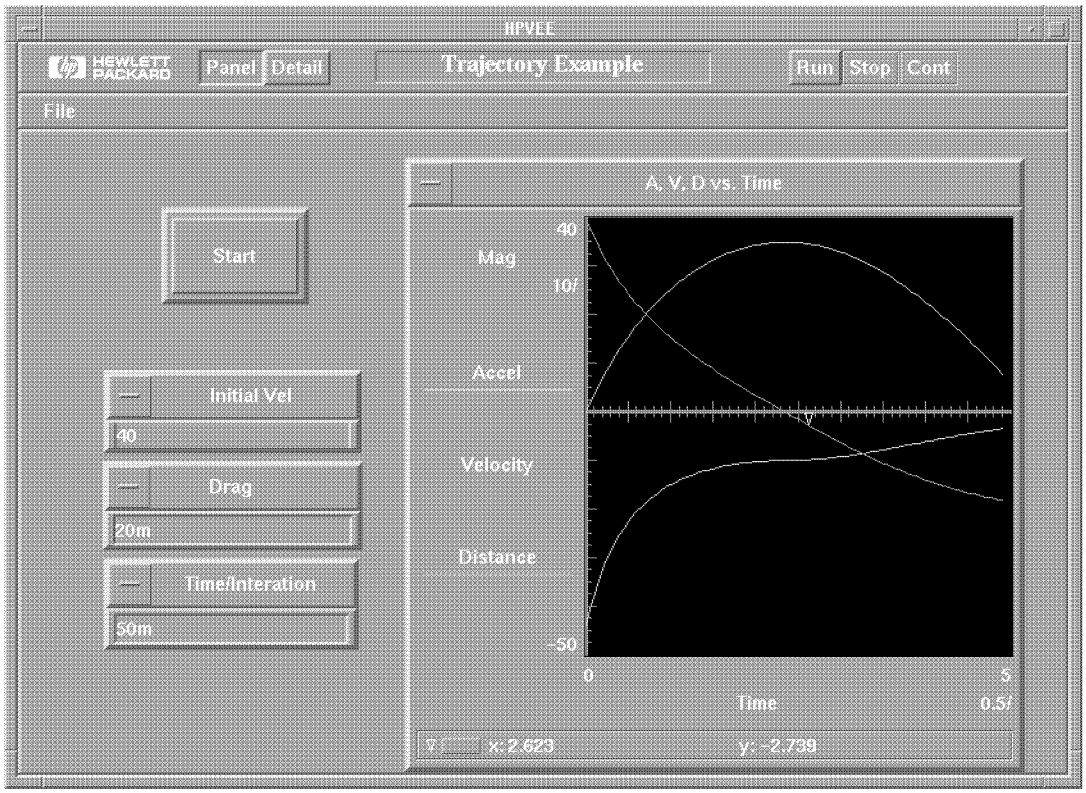


Figure 4-2. Panel View of Trajectory Example

The program shown in Figure 4-1 and Figure 4-2 is saved in `manual20.vee` in your `examples` directory.

Before You Start

Before you construct a panel view, the program 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 program first in the detail view, then create the panel view.

Creating Panel Views

After your program 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 4-3 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.

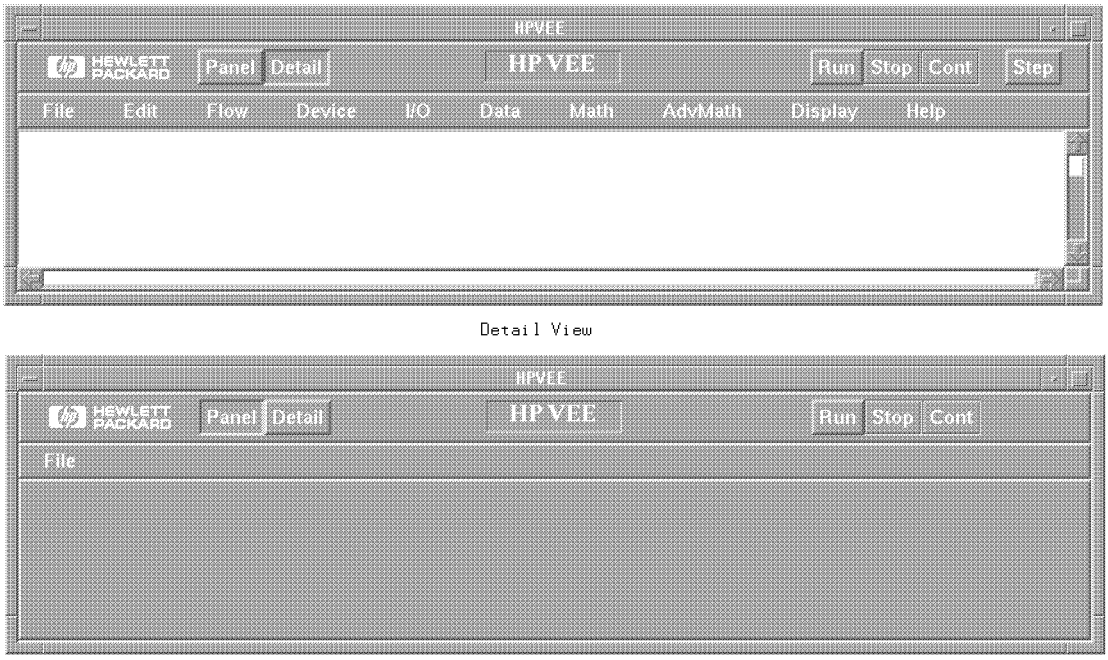


Figure 4-3. Differences Between Detail View and Panel View

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 program. 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 program. 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 program 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` ⟹ (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 program 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 4-4 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.

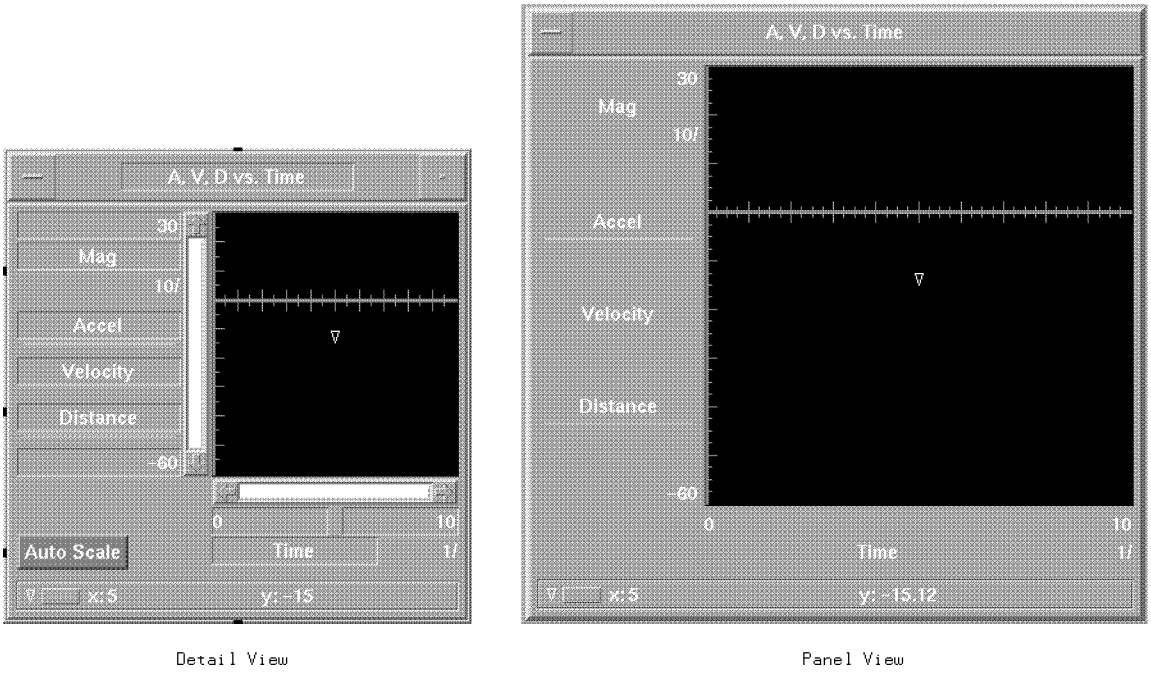


Figure 4-4. Panel View vs. Detail View of `X vs Y Plot`

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 program 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 program or `UserObject` can be secured. As mentioned previously, securing a program or `UserObject` prevents the user from altering the program and accessing the detail view. It may also enhance the program'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 program by selecting `Secure` from the `File` menu (on either the panel view or detail view). Before the program is secured, you'll be prompted to save the unsecured program. Note that this method secures the *entire* program, including any `UserObjects`.

After the program is secured, save it. Use a name that is different from the unsecured program name.

> **NOTE**
>
> The secured panel view and the unsecured program are unconnected. If you change the unsecured program, you'll have to `Secure` it again. Once a program is secured, it *cannot* be unsecured.

UserObject Panel View

To secure a `UserObject` without securing the rest of the program, 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 program 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 program 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 program still works. Many times the action of creating a `UserObject` will change the way your program operates. For more information about `UserObjects`, refer to Chapter 3.

`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 program. 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 4-5, after the user has read the message, he or she clicks `OK` and the pop-up disappears.

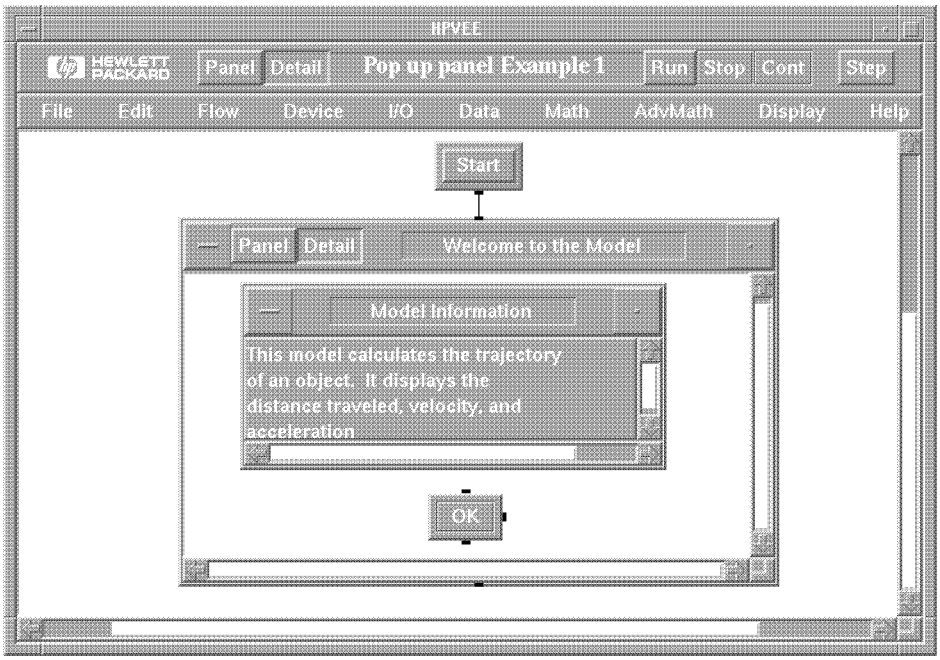


Figure 4-5. Informational Message (Detail View)

Figure 4-6 shows the panel after the user presses `Start`.

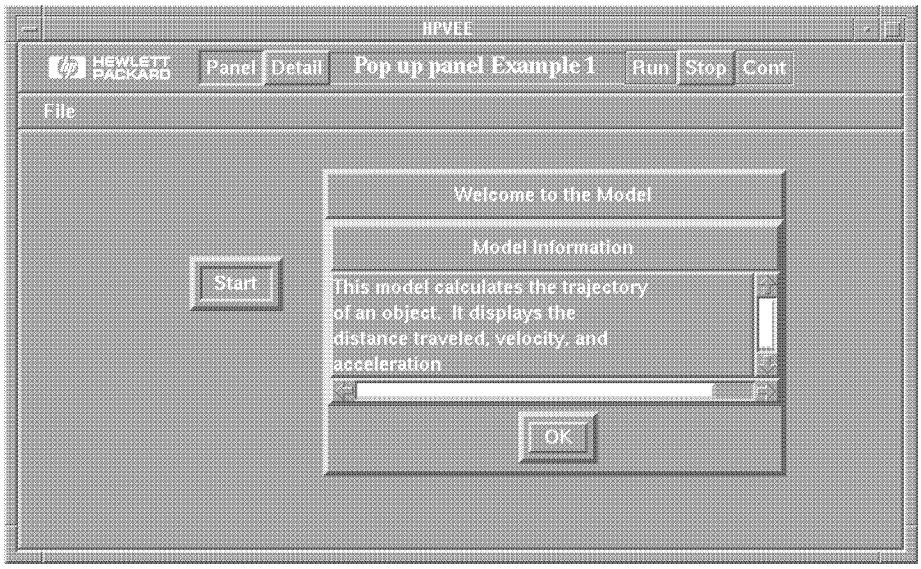


Figure 4-6. Informational Message (Panel View)

The program shown in Figure 4-5 and Figure 4-6 is saved in `manual21.vee` in your `examples` directory.

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 4-7, the display shows the data you select.

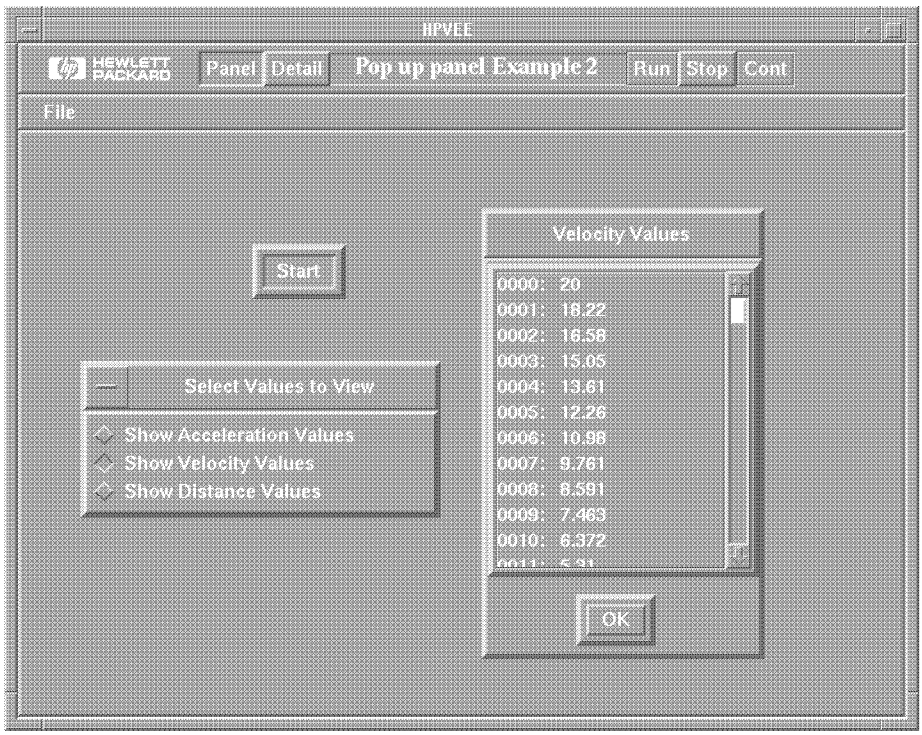


Figure 4-7. Overlaying Displays on a Panel View

The program shown in Figure 4-7 is saved in `manual22.vee` in your `examples` directory.

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 program in Figure 4-8 and Figure 4-9 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.

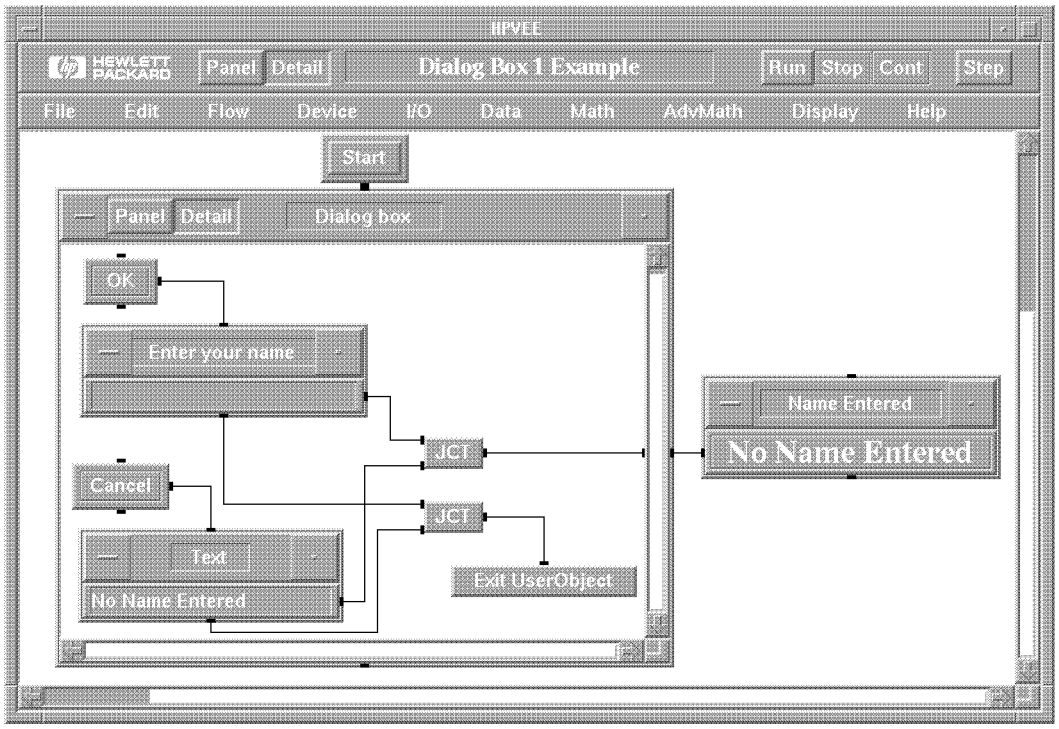


Figure 4-8. Dialog Box (Detail View)

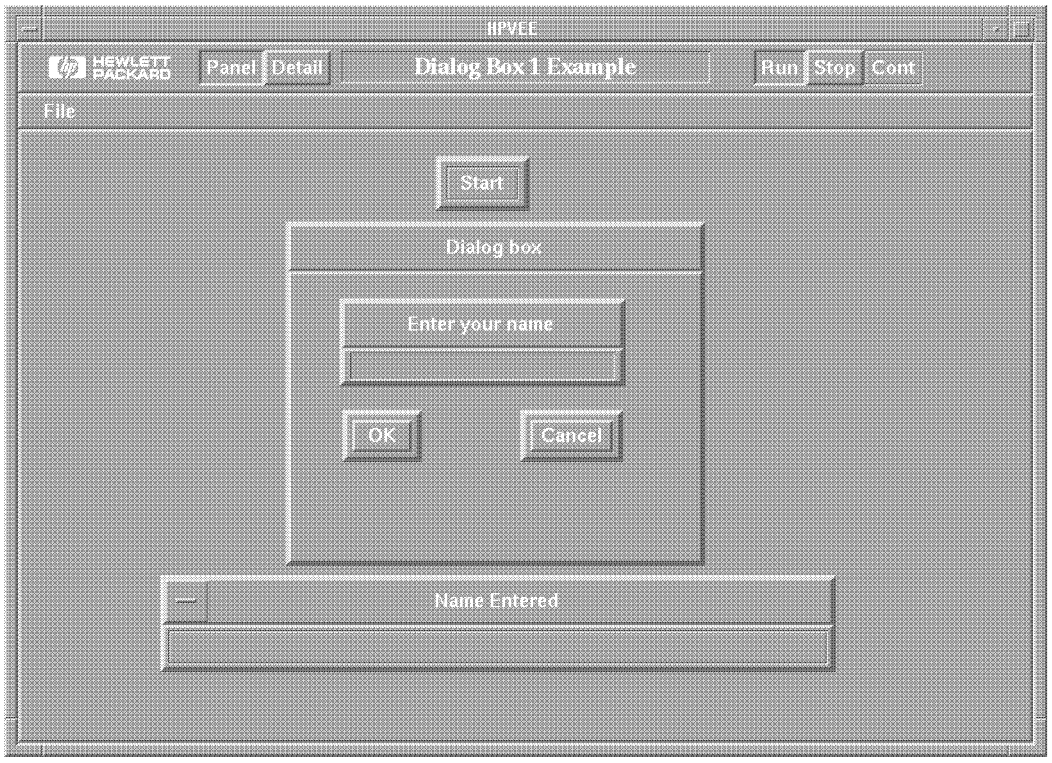


Figure 4-9. Dialog Box (Panel View)

The program shown in Figure 4-8 and Figure 4-9 is saved in `manual23.vee` in your `examples` directory.

NOTE

Some other commonly-used dialog boxes are in `/usr/lib/veeengine/concepts/` or `/usr/lib/veetest/concepts/` or `C:\VEE\EXAMPLES\CONCEPTS`.

5

Understanding Common Structures

Understanding Common Structures

This chapter shows some common structures that may be useful in your programs 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/` or `C:\VEE\EXAMPLES\CONCEPTS` directory.

Outputting Values from If/Then/Else

Figure 5-1 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 5-1 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.

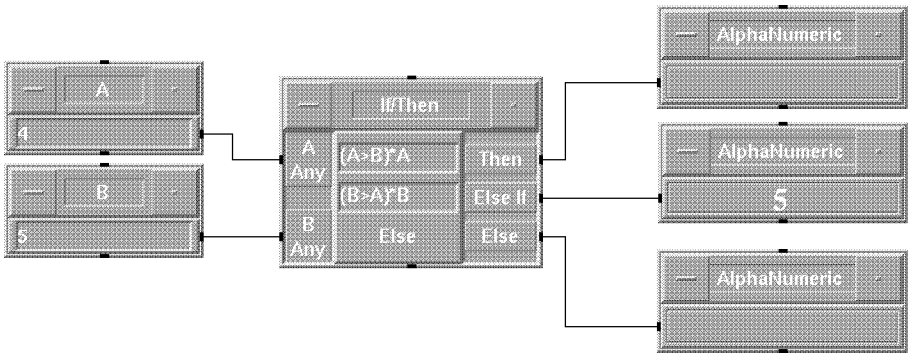


Figure 5-1. Getting a Value from If/Then/Else

Specifying Messages from Conditionals

Figure 5-2 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 5-2 works for both Scalar and Array data.

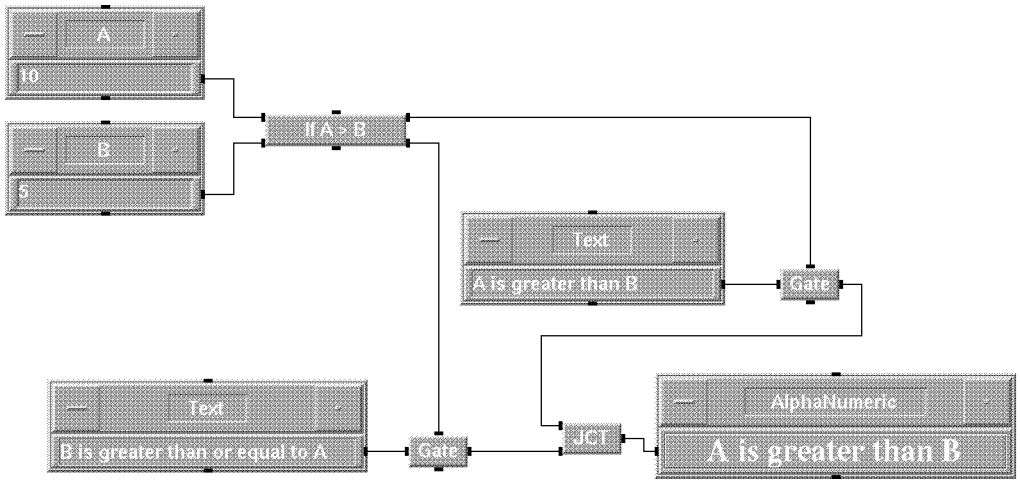


Figure 5-2. Specifying a Conditional Message

The program shown in Figure 5-2 is saved in `manual36.vee` in your `examples/concepts` directory.

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 5-3 shows two `Toggle` buttons. The first clears the display whenever the button is pushed; the second stops the program. Notice the use of `If/Then/Else` objects to check the `Toggle` output.

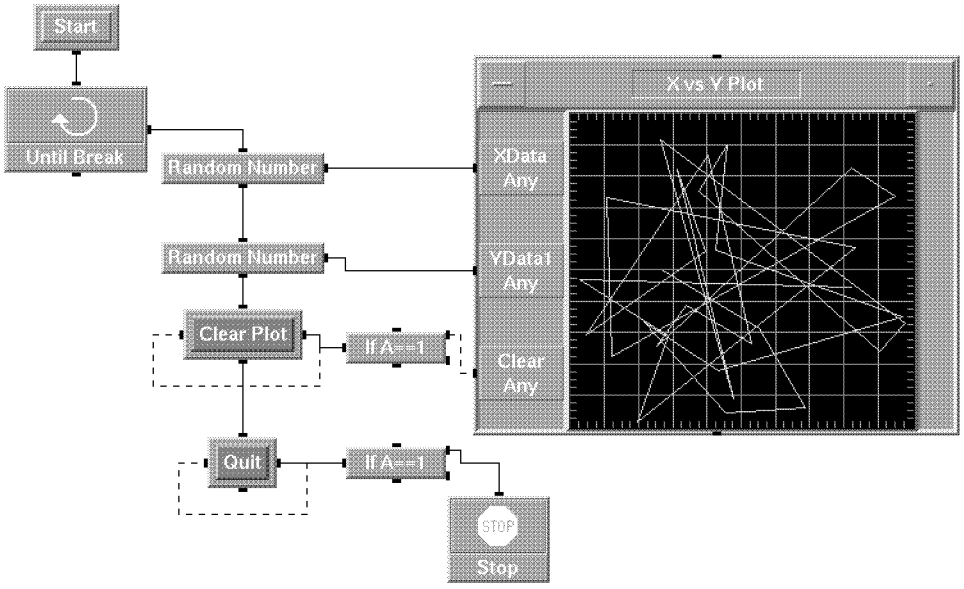


Figure 5-3. Using a `Toggle`

The program shown in Figure 5-3 is saved in `manual37.vee` in your `examples/concepts` directory.

Glossary

Glossary

This Glossary defines several terms used to name or describe HP VEE features.

Activate

1. To send a container to a terminal. See also "Container" and "Terminal."

2. The action that resets the context of a `UserObject` before it operates each time. See also "Context" and "PreRun."

Array

A data shape that contains a systematic arrangement of data items in one or more dimensions. The data items are accessed via indexes. See also "Data Shape."

Asynchronous

In asynchronous operation, a device operates without a common signal to synchronize events. Rather, the events occur at unspecified times. HP VEE control pins are asynchronous.

Auto Execute

An option on the object menus of the data constant objects. When `Auto Execute` is set, the object operates when its value is edited.

Bitmap

A bit pattern or picture. In HP VEE you can display a bitmap on an icon.

Buffer

An area in memory where information is stored temporarily.

Button

1. A button on a mouse.

2. A graphical object in HP VEE that simulates a real-life pushbutton and appears to pop out from your screen. When you "press" a button in HP VEE, by clicking on it with the mouse, an action occurs.

Cascading Menu

A sub-menu on a pull-down or pop-up menu that provides additional selections.

Checkbox
A recessed square box on HP VEE menus and dialog boxes that allows you to select a setting. To select a setting, click on the box and a checkmark appears in the box to indicate a selection has been made. To cancel the setting, simply click on the box again.

Click
To press and release a mouse button quickly. Clicking usually selects a menu feature or object in the HP VEE window. See also "Double-Click" and "Drag."

Compiled Function
A user-defined function created by dynamically linking a program, written in a programming language such as C, into the HP VEE process. For UNIX systems, the user must create a shared library file and a definition file for the program to be linked. For Windows, the user must create a DLL (Dynamically Linked Library) file and a definition file. The `Import Library` object attaches the shared library or DLL to the HP VEE process and parses the definition file declarations. The Compiled Function can then be called with the `Call Function` object, or from certain expressions. See also "User Function" and "Remote Function."

Component
A single instrument function or measurement value in an HP VEE instrument panel or component driver. For example, a voltmeter driver contains components that record the range, trigger source, and latest reading. See also "Component Driver," "Driver Files," "State," and "Instrument Panel."

Component Driver
An instrument control object that reads and writes values to components you specifically select. Use component drivers to control an instrument using a driver by setting the values of only a few components at a time. (Component drivers do not support coupling.)

Composite Data Type
A data type that has an associated shape. See also "Data Shape" and "Data Type."

Container
The package that is transmitted over lines and is processed by objects. Each container contains data, the data type, and the data shape.

Context
A level of the work area that can contain other levels of work areas (such as nested `UserObjects`), but is independent of them.

Control Pin
An asynchronous input pin that transmits data to the object without waiting for the object's other input pins to contain data. For example, control pins in HP VEE are commonly used to clear or autoscale a display.

Coupling
The inter-relationship of certain functions in an instrument. If, in an instrument panel, functions A and B are coupled, changing the value of A may automatically change the value of B, even though you do not change B explicitly.

Cursor
A pointer in an entry field that shows where alphanumeric data will appear when you type information from the keyboard.

Data Field
The field within a transaction specification in which you specify either the expression to be written (WRITE transactions), or the variable to receive data that is read (READ transactions). See also "Transactions."

Data Flow
The flow of data through and between HP VEE objects. Data flows from left to right through objects, but an object does not execute until it has data on all of its data input pins. Data is propagated from the data output pin of one object to the data input pin of the next object. Data flow is the chief factor that determines the execution of a HP VEE program.

Data Input Pin
A connection point on the left side of an object that permits data to flow into the object.

Data Output Pin
A connection point on the right side of an object that propagates data flow to the next object and passes the results of the first object's operation on to the next object.

DataSet
A collection of "Record" containers saved into a file for later retrieval. The `To DataSet` object collects Record data on its input and writes that data to a named file (the DataSet). The `From DataSet` object retrieves Record data from the named file (the DataSet) and outputs that data as Record containers on its `Rec` output pin. See also "Record."

Data Shape
A pre-defined structure that defines how data is grouped together (for example, an array).

Data Type
A pre-defined structure that determines how data is organized and treated by HP VEE (for example, Real or Complex).

DDE (Dynamic Data Exchange)
A communication mechanism that allows HP VEE for Windows to communicate with other Windows applications that support DDE. HP VEE can send data to, and receive data from, such applications. Also, HP VEE can execute commands in the other application. Examples of Windows applications that support DDE are Microsoft Excel and Microsoft Word for Windows.

Default
A value or action that HP VEE automatically selects.

Default Button
The button in a dialog box that is activated by default if Enter or Return is pressed, or the selection is double-clicked. The default button has a recessed border.

Demote
To convert from a data type that contains more information to one that contains less information. See also "Data Type" and "Promote."

Detail View
The view of an HP VEE program that shows all the objects and the lines between them.

Device
An instrument attached to or plugged into an HP-IB, RS-232, GPIO, or VXI interface. Specific HP VEE objects such as the `Direct I/O` object send and receive information to a device.

Device Driver
See "Interface Driver."

Dialog Box
A secondary window displayed when HP VEE requires information from you before it can continue. For example, a dialog box may contain a list of files from which you may choose.

Direct I/O Object
An instrument control object that allows HP VEE to directly control an instrument without using an HP Instrument Driver.

DLL (Dynamically Linked Library)
A collection of functions written in C that can be called from HP VEE for Windows. DLLs can be created by experienced C programmers using tools available from Microsoft and Borland. DLLs in the Windows environment are similar to shared libraries in the UNIX environment.

Double-Click
To press and release a mouse button twice in rapid succession. Double-clicking is usually a short-cut to selecting and performing an action. For example, double-clicking on a file name from `File` ⟹ `Open` will select the file and open it.

Drag
To press *and continue to hold down* a mouse button while moving the mouse. Dragging moves something (for example, an object or scroll slider).

Driver
Software that allows a computer to communicate with other software or hardware. See also "Component Driver," "Driver Files," "Interface Driver," and "Instrument Panel."

Driver Files
A set of files included with HP VEE that contains the information needed to create instrument panel and component driver objects for instrument control.

Entry Field
A field that is typically part of a dialog box or an editable object, and which is used for text entry. An entry field appears recessed. For

example, the open view of the `For Range` object has entry fields where you type values that specify the beginning, ending, and step values.

Error Message
Information that appears in an error dialog box, explaining that a problem has occurred.

Error Pin
A pin that traps any errors that occur in an object. Instead of getting an error message, the error number is output on the error pin. When an error is generated, the data output pins are not activated.

Execute
The action of a program, or parts of a program, running.

Execution Flow
The order in which objects operate. See also "Data Flow."

Expression
An equation in an entry field that can contain the input terminal names and any `Math` or `AdvMath` functions. An expression is evaluated at run-time. Expressions are allowed in `Formula`, `If/Then/Else`, `Get Values`, `Get Field`, `Set Field`, and `Sequencer` objects, and in I/O transaction objects.

Feature
An item on a menu that you select to cause a particular action to occur (for example, to open a file), or to get a particular object.

Feedback
A continuous thread path of sequence and/or data lines that uses values from the previous execution to change values in the current execution.

Flow
See "Data Flow" and "Execution Flow."

Function
The name and action of objects where the output is a function of the input. These objects are located under `Math` or `AdvMath` menus and may be used in the `Formula` object. For example `sqrt(x)` is a function; `+` is not.

Global Variable
A named variable that is set globally, and which can be used by name in any context of an HP VEE program. For example, a global variable can be set with `Set Global` in the root context of the program, and can be accessed by name with `Get Global` or from certain expressions within the context of a `UserObject`. However, a local variable with the same name as the global variable takes precedence in an expression.

Grayed Feature
A menu feature that is displayed in gray rather than black, indicating that the feature is not active or not available.

Group Window
A group window in Microsoft Windows is a window that contains icons for a group of applications. Each icon starts an application in the group.

Highlight
1. The colored band or shadow around an object that provides a visual cue to the status of the object.

2. The change of color on a menu feature that indicates you are pointing to that feature.

Host
To begin a thread or subthread. For example, the subthread that is hosted by `For Count` is the subthread that iterates.

HP-UX
Hewlett-Packard Company's enhanced version of the UNIX operating system.

Hypertext
A system of linking topics so that you can jump to a related topic when you want more information. In online help systems, typically hypertext links are designated with underlined text. When you click on such text, related information is presented.

Icon
1. The small, graphical representation of an HP VEE object, such as the representation of an instrument, a control, or a display.

2. The small, graphical representation of a Microsoft Windows application within a group window. See "Group Window."

Instrument Driver
See "Driver Files," "Component Driver," and "Instrument Panel."

Instrument Panel
An instrument control object that forces all the function settings in the corresponding physical instrument to match the settings in the control panel displayed in the open view of the object.

Interface
HP-IB, RS-232, GPIO, and VXI are referred to as interfaces used for I/O. Specific HP VEE objects, such as the `Interface Event` object can only send commands to an interface.

Interface Driver
Software that allows a computer to communicate with a hardware interface, such as HP-IB or RS-232. Also called *device driver* in the UNIX operating system, interface drivers are configured into the kernel of the operating system.

Interrupt
A signal that requires immediate attention that may suspend a process, such as the execution of a computer program. An interrupt is usually caused by an event external to that process. After the interrupt is serviced, the process may be resumed.

Label
The text area or name on an icon or button that identifies that object or button.

Library
A collection of often-used objects or small programs grouped together for easy access.

Line
A link between two objects in HP VEE that transmits data containers to be processed. See also "Subthread" and "Thread."

Main Menu
The menus located in the HP VEE menu bar. The main menus may be opened by clicking or dragging on the menu titles in the menu bar.

Main Work Area
The area where you create a program. The main work area is the parent context of all other contexts.

Mapping
To associate a set of independent values with an array, when the array is a function of the values.

Maximize
To enlarge a window to its maximum size. In HP VEE, the `UserObject` has a maximize button.

Menu
A collection of features that are presented in a list. See also "Cascading Menu," "Main Menu," "Object Menu," "Pop-Up Menu," and "Pull-Down Menu."

Menu Bar
A rectangular bar at the top of the HP VEE window that contains titles of the pull-down, main menus from which you select features.

Menu Title
The name of a menu within the HP VEE menu bar. For example, `File` or `Edit`.

Minimize
1. To reduce an open view of an object to its smallest size—an icon.
2. To reduce a window to its smallest size—an icon.

Mouse
A pointing device that you move across a surface to move a pointer within the HP VEE window.

Mouse Button
One of the buttons on a mouse that you can click, double-click, or drag to perform a particular action with the corresponding pointer in the HP VEE window.

Network
A group of computers and peripherals linked together to allow the sharing of data and work loads.

Object
A graphical representation of an element in a program, such as an instrument, control, display, or mathematical operator. An object is placed in the work area and connected to other objects to create a program. Objects can be displayed as icons or as open views.

Object Menu
The menu associated with an object that contains features that operate on the object (for example, moving, sizing, copying, and deleting the object).

Open
To start an action or begin working with a text, data, or graphics file. When you select `Open` from HP VEE, a program is loaded into the work area.

Open View
The representation of an HP VEE object that is more detailed than an icon. Within the open view, you can modify the operation of the object and change the object's title.

Operate
The action of an object processing data and outputting a result. An object operates when its data and sequence input pins have been activated. See "Activate."

Outline Box
A box that represents the outer edges of an object or set of objects and indicates where the object(s) will be placed in the work area.

Palette
A set of colors and fonts that is supplied with HP VEE and used in your HP VEE environment.

Panel
Information displayed in the center of the object's open view. In a `UserObject`, the panel contains a work area. In a `For Count` object, the panel contains an entry field. Compare with "Panel View."

Panel View
The view of a program in HP VEE that shows only those objects needed for the user to run the program and view the resultant data. You can create a panel view to meet the needs of your users.

Pin
An external connection point on an object to which you can attach a line.

Pointer
The graphical image that maps to the movement of the mouse. A pointer allows you to make selections and provides you feedback on a particular process underway. HP VEE has pointers of different shapes that correspond to process modes, such as an arrow, crosshairs, and hourglass.

Pop-Up Menu
A menu that is raised by clicking the right mouse button. For example, you can raise the `Edit` menu by clicking the right mouse button in an empty area within the work area. Or you can raise the object menu by clicking the right mouse button on an inactive area of an object.

PostRun
The set of actions that are performed when the program is stopped.

PreRun
The set of actions that resets the program and checks for errors before the program starts to run.

Priority Thread
A priority thread executes to completion blocking all other parallel threads from executing. Certain of the I/O objects for devices and interfaces will host a priority thread.

Program
In HP VEE, a graphical program that consists of a set of objects connected with lines. The program typically represents a solution to an engineering problem.

Promote
To convert from a data type that contains less information to one that contains more information. See also "Data Type" and "Demote."

Propagation
The rules that objects and programs follow when they operate or run. See also "Data Flow."

Pterodactyl
Any of various extinct flying reptiles of the order Pterosauria of the Jurassic and Cretaceous periods. Pterodactyl are characterized by wings

consisting of a flap of skin supported by the very long fourth digit on each front leg.

Pull-Down Menu
A menu that is pulled down from the menu bar when you position the pointer over a menu title and click the left mouse button.

Radio Button
A diamond-shaped button in HP VEE dialog boxes that allows you to select a setting that is mutually exclusive with other radio buttons in that dialog box. To select a setting, click on the radio button. To remove the setting, click on another radio button in the same dialog box.

Record
A data type that has named data fields which can contain multiple values. Each field can contain another Record container, a Scalar, or an Array. The Record data type has the highest precedence of all HP VEE data types. However, data cannot be converted to and from the Record data type through the automatic promotion/demotion process. Records must be built/unbuilt using the using `Build Record` and `UnBuild Record` objects.

Remote Function
A User Function running on a remote host computer, which is callable from the local host. (Remote functions are supported only on UNIX systems.) The `Import Library` object starts the process on the remote host and loads the Remote File into the HP VEE process on the local host. You can then call the Remote Function with the `Call Function` object, or from certain expressions. See also "User Function" and "Compiled Function."

Resource Manager
A program which exists on VXI controllers that runs at start-up and after a VXI system reset. This program initializes and manages the instruments in a VXI card cage.

Restore
To return a minimized window or an icon to its full size as a window or open view by double-clicking on it.

Run
To start the objects on a program or thread operating.

Save
To write a file to a storage device, such as a hard disk, for safekeeping.

Scalar
A data shape that contains a single value. See also "Data Shape."

Schema
The structure or framework used to define a data record. This includes each field's name, type, shape (and dimension sizes), and mapping.

Screen Dump
A graphical printout of a window or part of a window.

Scroll
The act of using a scroll bar either to move through a list of data files or other choices in a dialog box, or to pan the work area.

Scroll Arrow
An arrow that, when clicked on, moves you through a list of data files or other choices in a dialog box, or moves the work area.

Scroll Slider
A rectangular bar that, when dragged, moves you through a list of data files or other choices in a dialog box, or moves the work area.

Select
To choose an object, an action to be performed, or a menu item. Usually you select by clicking with your mouse.

Select Code
A number used to identify the logical address of a hardware interface. For example, the factory default select code for most HP-IB interfaces is 7.

Selection
1. A menu selection (feature).
2. An object or action you have selected in the HP VEE window.

Sequence Input Pin
The top pin of an object. When connected, execution of the object is held off until the pin receives a container.

Sequence Output Pin
The bottom pin of an object. When connected, this output pin is activated when the object and all data propagation from that object finishes executing.

Sequencer
An object that controls execution flow through a series of sequence transactions, each of which may call a "User Function," "Compiled Function," or "Remote Function." The sequencer is normally used to perform a series of tests by specifying a series of sequence transactions.

Shared Library
A collection of functions, written in a programming language such as C, that can be called from HP VEE running on a UNIX system. Shared libraries can be created by experienced programmers. Shared libraries in the UNIX environment are similar to DLLs in the Windows environment.

Shell
In a UNIX system, the program that interfaces between the user and the operating system.

Shell Prompt
In a UNIX system, the character or characters that denote the place where you type commands while at the operating system shell level. The prompt you see displayed depends upon the type of shell you are running, such as a **#** prompt for the Bourne shell.

Sleep
An object sleeps during execution when it is waiting for an operation or time interval to complete, or for an event to occur. A sleeping object will allow other parallel threads to run concurrently. Once the event, time interval, or operation occurs, the object will execute, allowing execution to continue.

Startup Directory
The directory from which you start HP VEE on a UNIX system. This directory determines the default paths for most file actions including `Save` and `Open`.

State
A particular set of values for all of the components related to an HP VEE instrument panel, which represents the measurement state of an instrument. For example, a digital multimeter uses one state for

high-speed voltage readings and a different state for high-precision resistance measurements. See also "Instrument Panel."

Step
The action of operating one object at a time. An arrow points to the object that will operate next.

Terminal
The internal representation of a pin that displays information about the pin and the data container held by the pin. Double-click on the terminal to view the container information.

Thread
A set of objects connected by solid lines in an HP VEE program. A program with multiple threads can run all threads simultaneously.

Title Bar
The rectangular bar at the top of the open view of an object or window, which shows the title of the object or window.

Tool Bar
The rectangular bar at the top of the HP VEE window which provides the `Run`, `Stop`, `Cont`, and `Step` buttons to control HP VEE programs. The tool bar also displays the title of a program.

Transaction
The specifications for input and output (I/O) used by certain objects in HP VEE. These include the `To File`, `From File`, `Direct I/O`, and `Sequencer` objects. Transactions appear as English-like phrases listed in the open view of these objects.

User-Defined Function
HP VEE allows three types of user-defined functions: the "User Function," "Compiled Function," and "Remote Function."

User Function
A user-defined function created from a "UserObject" by executing `Make UserFunction`. The User Function exists in background, but provides the same functionality as the original UserObject. You can call a User Function with the `Call Function` object, or from certain expressions. A User Function can be created and called locally, or it can be saved in a library and imported into an HP VEE program with `Import Library`. See also "Compiled Function," "Remote Function," and "UserObject."

User Interface
The part of an application that permits a user and the application to communicate with each other to perform certain tasks. HP VEE uses a graphical user interface, which includes windows, menus, dialog boxes, and objects.

UserObject
An object that can encapsulate a group of objects to perform a particular purpose within a program. A UserObject allows you to use top-down design techniques when building a program, and to build user-defined objects that can be saved in a library and reused.

View
See "Detail View," "Icon," "Open View," and "Panel View."

Wait
See Sleep.

Window
A rectangular area on the screen that contains a particular application program, such as HP VEE.

Work Area
The area within the HP VEE window or the open view of a `UserObject` where you group objects together. When you `Open` a program, it is loaded into the main work area.

X Window System (X11)
An industry-standard windowing system used on UNIX computer systems.

X11 Resources
A file or set of files that define your X11 environment in a UNIX system.

XEQ Pin
A pin that forces the operation of the object, even if the data or sequence input pins have not been activated. See also "Control Pin," "Data Input Pin," and "Sequence Input Pin."