HP VEE Advanced Programming Techniques

HEWLETT PACKARD

HP Part No. E2110-90051
Printed in USA March 1997

Edition 3

FINAL TRIM SIZE : 7.0 in x 8.5 in

Notice

Warranty Information

Restricted Rights Legend

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Printing History

Edition 1 - September 1993
Edition 2 - January 1995
Edition 3 - March 1997

FINAL TRIM SIZE : 7.0 in x 8.5 in

Conventions Used in this Manual

This manual uses the following typographical conventions:

Example	Represents
Getting Started with 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, buttons, or text you need 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`	The "⟹" is used in a shorthand notation to show the location of HP VEE features in the menu. For example, "`File` ⟹ `Open`" means to select the `File` menu and then select `Open`.
`Zoom In | Out 2x | Out 5x`	Choices in computer font, separated with a bar (|), indicates that you should choose one of the options.
Return	The keycap font graphically represents a key on the computer's keyboard.
Press CTRL+O	Represents a combination of keys on the keyboard that you should press at the same time.

Contents

Figures

Tables

FINAL TRIM SIZE : 7.0 in x 8.5 in

1

Introduction

About This Manual

This manual gives detailed information about using advanced features of HP VEE. This manual is meant to be used as needed, rather than read from beginning to end.

Note Throughout this manual, references to HP VEE apply to both HP VEE for HP-UX and HP VEE for Windows except where noted otherwise.

HP VEE Example Programs

HP VEE includes many examples programs to help you understand HP VEE. HP VEE also includes a library of objects that you can `Merge` into your programs. The example programs and library objects are installed as part of the normal HP VEE installation process.

Using the Examples

The examples referenced from the manuals are included in the `examples/manual` directory (with file names like `manual01.vee`, etc). Other examples, not referenced in any of the manuals, are available to illustrate specific HP VEE concepts, or to illustrate solutions to engineering problems. To help you find the example you want, the `examples` directory is divided into several subdirectories.

Running Examples

You load and run example programs using the `Help` menu. First, click on `Help` ⟹ `Open Example` on the menu bar. This presents a list of subdirectories which group similar examples together. Double-click on the desired subdirectory to see the list of available example programs in that group. Scroll through the list until you find the desired example. Click on the example name, then click on `OK` to open the program. You will be prompted to save the any existing program in the work area. To run the program, press the `Run` button on the tool bar.

Example Directories

You can also use `File` ⟹ `Open` to load HP VEE examples. For HP VEE for HP-UX, running on HP-UX 9.x the examples are installed in subdirectories under:

```
/usr/lib/veetest/examples/
```

For 10.x, the directory is:

```
/opt/veetest/examples/
```

For HP VEE for Windows, the directory is:

```
C:\Program Files\Hewlett-Packard\VEE 4.0\examples\
```

Using Library Objects

The object library provides objects that you can merge into your own program. Just select `Merge` from the `File` menu and a list box will appear for the appropriate library directory:

```
/usr/lib/veetest/lib/

- or -

/opt/veetest/lib/

- or -

C:\Program Files\Hewlett-Packard\VEE 4.0\lib\
```

Most of the library objects are UserObjects that encapsulate individual objects. You can create your own UserObjects for the library, but you must save them in the `contrib` subdirectory (HP-UX only):

```
/usr/lib/veetest/lib/contrib/

- or -

/opt/veetest/lib/contrib/
```

(You can't write to the `lib` directory unless you are logged on as "root" on HP-UX platforms.)

The `contrib` subdirectory is empty at installation — it provides a place for your own library of "contributed" objects.

There is another subdirectory under `lib`, named `conversions`: This subdirectory contains formula objects that you can `Merge` into your program. Each of these objects performs a useful conversion function such as degrees to radians.

The files are located in:

```
/usr/lib/veetest/lib/conversions/

- or -

/opt/veetest/lib/conversions/

- or -

C:\Program Files\Hewlett-Packard\VEE 4.0\lib\convert\
```

2

Variables

Using Variables

There are two types of variables in HP VEE, undeclared and declared.

Both types of variables can contain any data type, including complex data types such as waveforms and records. They can also be any data shape, including scalars and arrays.

About Undeclared Variables

Undeclared variables are the easiest to use, but execute slower and dot not allow scoping. Undeclared variables include the following:

- Global variables that can be used anywhere in the program. They are created with the `Set Variable` object. They are deleted before the program is run if the `Delete Variables at PreRun` property is set. Global variables must be created before they can be accessed via the `Get Variable` object or used in expressions, or else your program will generate an error.

  Undeclared global variables are useful if you don't know what data type or shape your values will be. Also if the values may change the type or shape, use an undeclared global variable. If you want a scoped variable (i.e. local), then use declared variables (see "About Declared Variables").
- Temporary variables that are used only in `Formula` objects. For example, to swap the values input in the terminals `a` and `b`, use the temporary variable `tmp`. The expression would look like: `tmp=a, a=b, b=tmp`. For more information about temporary variables, refer to `Assignment` in `Math Functions and Operators, under Reference` in HP VEE online help.
- Terminal names that are used as variables within objects (such as in transaction or `Formula` objects).

About Declared Variables

Declared variables are defined before they are used. They have the additional feature of scoping, which allows HP VEE to run faster because the data type and shape are known before run time. However, if you attempt to set a declared variable with values that are different than the data type or shape of the values set in the declaration, the program will error.

To declare a variable, use the `Data` $\Longrightarrow$ `Declare Variable` object. When placed in a context, it declares the variable before any of the other objects execute. When the variable has been declared, it has no value until it is set via a `Set Variable` or a `Formula` object.

The scope of a declared variable must be specified in the `Declare Variable` object. The scopings are as follows:

- `Global` - The variable can be used anywhere in the program.
- `Local to Context` - The variable can only be used in a single UserObject or UserFunction, or in Main. This variable can only be used in the context that the `Declare Variable` object is in. The variable can not be used in UserObjects or UserFunctions nested inside the context.
- `Local to Library` - The variable can only be used within the library of UserFunctions where the `Declare Variable` object is used. `Declare Variable` must be located in one of the UserFunctions.

You cannot define multiple variables with the same name and scope. If this happens, you will get an error.

About Naming

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

To retrieve the value of the variable, you must use the name that you specified when the variable was declared or set.

Naming Precedence

The question of precedence comes up when you have named a variable the same name as another variable. The order of precedence (from highest to lowest) is:

1. Input terminal name (such as in a `Formula` or a transaction object)
2. Temporary variable (as in a `Formula` object)
3. `Local to Context` declared variable
4. `Local to Library` declared variable
5. `Global` declared variable
6. Global undeclared variable

In other words, if you have two variables with the same name in an object, there is a conflict. The variable that has the highest precedence is used.

Setting Initial Values

You must have set initial values before accessing any variables or HP VEE generates an error.

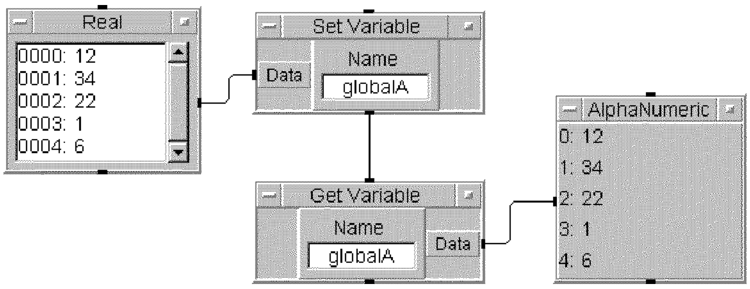


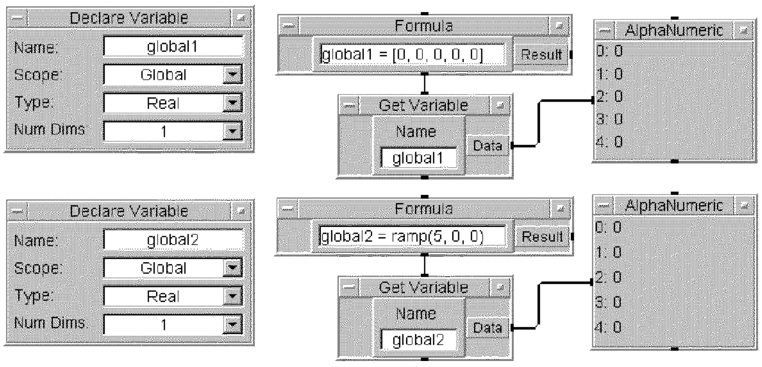
Figure 2-1. A Simple Variable Example

The `Set Variable` must set the global variable *before* the `Get Variable` attempts to retrieve it. To ensure this, the sequence output pin of the `Set Variable` object is connected to the sequence input pin of the `Get Variable` object. If this is not done, the `Get Variable` may try to access a non-existent global variable, and an error will occur.

If the property `Delete Variables at PreRun` is not set, you may not receive an error and may receive old data instead.

When declared variables are created, they are not initialized and must have a value set in them before they are accessed via the `Get Variable` object or used in expressions, or else your program will generate an error. You set values via the `Set Variable` object or by using the `Formula` object.

If the variable is an array or a record, when using the `Formula` object, you must set the values of the entire array or record before trying to access any of the elements. The following example shows two different ways to initialize values from a `Formula` object.



Name: global1
Scope: Global
Type: Real
Num Dims: 1

```
global1 = [0, 0, 0, 0, 0]
```

Name: global1

```
0: 0
1: 0
2: 0
3: 0
4: 0
```

Name: global2
Scope: Global
Type: Real
Num Dims: 1

```
global2 = ramp(5, 0, 0)
```

Name: global2

```
0: 0
1: 0
2: 0
3: 0
4: 0
```

Figure 2-2. Setting Array Values

Accessing Variable Values

Once you have named a variable, you can access its value as many times as you want in your program. You can use several methods to retrieve the variable value. In the following example, the value stored in the global variable `globalA` is retrieved once with a `Get Variable` object, a second time by including the name `globalA` in an expression in a `Formula` object, and a third time by including the name `globalA` in a transaction in a `To File` object:

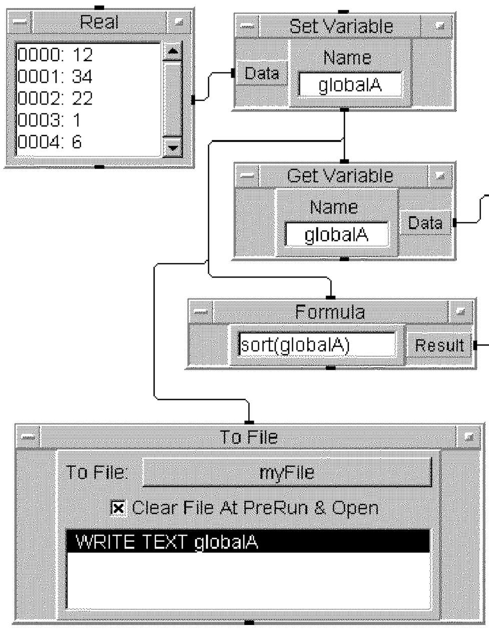


Figure 2-3. Accessing a Variable Multiple Ways

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

Deleting Variables

To improve memory usage, use the `Delete Variable` object to free up memory space when a variable is no longer needed. When undeclared variables are deleted, their values and definitions are both deleted. When declared variables are deleted, the values are reset to uninitialized values, but the definition remains.

When you set `Delete Variable` to `By Name`, the closest variable of the specified name is deleted. The closest variable is defined by the precedence listed in "Naming Precedence" section.

When you set `Delete Variable` to `All`, all declared and undeclared variables in all scopings are affected, even the variables that are in imported libraries. Declared variables are uninitialized and undeclared variables are deleted (as described previously).

To delete all variables before each execution of the program, select `File` ⟹ `Default Preferences` and click on the check box `Delete Variables at PreRun`. If this check box is not selected, the values of all variables will remain and the declarations of declared variables will not reinitialize the values

Using Variables in Libraries

Because only UserFunctions are loaded when the library is imported, when you use `Declare Variable` objects, you must put them in one of the UserFunctions, not in the `Main` window of the library.

When a variable is scoped as a `Global`, it is only used in the local program. It can not be used in any Remote Function that is called.

When a library is imported, all variables declared (via `Declare Variable` objects) in the imported UserFunction are defined at that time for the scope specified. For example, if the variable is scoped as a `Global`, it can be accessed from any part of the program, until the library is deleted. When a library is deleted, all variables declared in its UserFunctions are deleted as well.

3

Using Records and DataSets

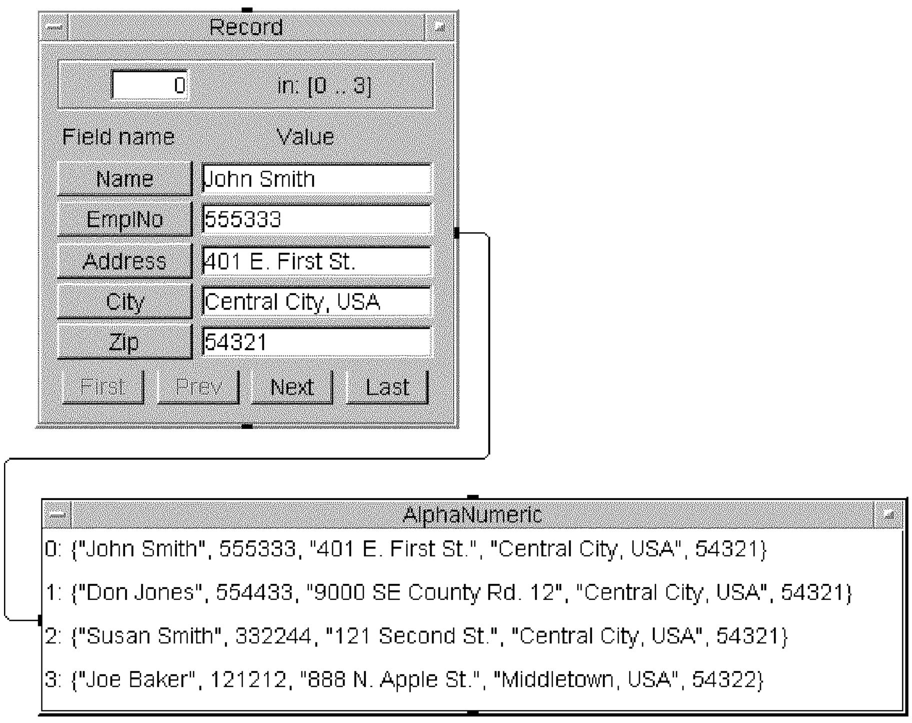
This chapter introduces two concepts: the Record data type and the DataSet, which is a collection of Record containers saved into a file for later retrieval. There are several HP VEE objects that allow you to create and manipulate records, including: `Record`, `Build Record`, `UnBuild Record`, `Merge Record`, `SubRecord`, `Set Field`, and `Get Field`. All of these objects are located under the `Data` menu.

The `To DataSet` and `From DataSet` objects allow you to store and retrieve records to and from DataSets; they are located under the `I/O` menu.

Record Containers

A container of the Record data type has named fields which represent data. You can have as many named fields as you like in a record. Each field can contain another record, a scalar, or an array. Let's look at a simple record, created with the `Record` object.

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



```
John Smith
555333
401 E. First St.
Central City, USA
54321
```

```
0: {"John Smith", 555333, "401 E. First St.", "Central City, USA", 54321}
1: {"Don Jones", 554433, "9000 SE County Rd. 12", "Central City, USA", 54321}
2: {"Susan Smith", 332244, "121 Second St.", "Central City, USA", 54321}
3: {"Joe Baker", 121212, "888 N. Apple St.", "Middletown, USA", 54322}
```

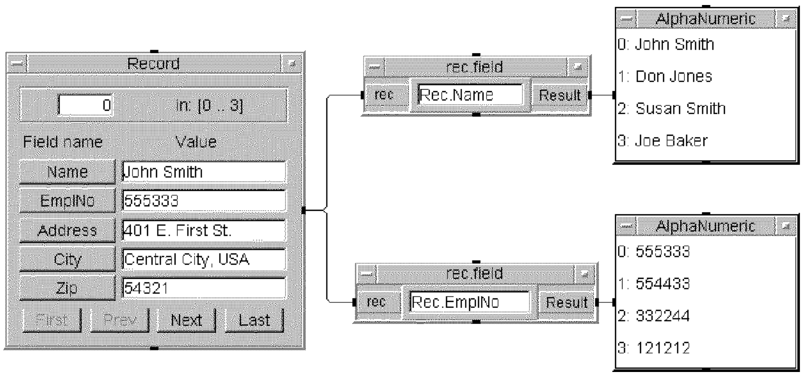
Figure 3-1. A Simple Record Container

When the program is run, the entire record is output on the `Record` output pin. The `AlphaNumeric` display shows the entire record, with four array elements (0 through 3), each consisting of five record fields enclosed in braces ("`{}`").

Accessing Records

The following examples show how to access a record and extract individual fields.

Use the `Get Field` object to extract an individual field from the record. `Get Field` is located under `Data` ⟹ `Access Record`. In the following example `Get Field` objects are used to extract the entire `Name` and `EmplNo` fields: Note that the `Get Field` object is really just a `Formula` object titled `rec.field`.



Field name	Value
Name	John Smith
EmplNo	555333
Address	401 E. First St.
City	Central City, USA
Zip	54321

Rec.Name

```
0: John Smith
1: Don Jones
2: Susan Smith
3: Joe Baker
```

Rec.EmplNo

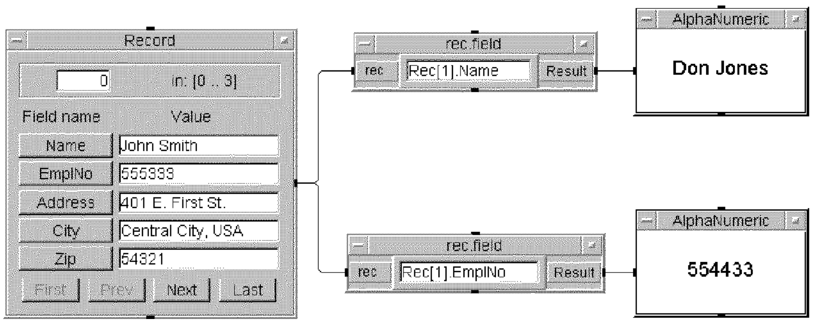
```
0: 555333
1: 554433
2: 332244
3: 121212
```

Figure 3-2. Retrieving Record Fields with Get Field

Use the "dot" syntax to access individual fields, for example: `Rec.Name` and `Rec.EmplNo`. This syntax is described in detail in `Mathematically Processing Data` ⟹ `General Concepts` in `How Do I` in HP VEE online Help.

`Rec.Name` means "get the `Name` field from the record on the `Rec` input pin." This syntax can be used in an expression in a `Formula` object, or in any other expression that is evaluated at run time. For example, you could use this syntax in a transaction in the `To String` object.

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



Field name	Value
Name	John Smith
EmplNo	555333
Address	401 E. First St.
City	Central City, USA
Zip	54321

```
Rec[1].Name
Rec[1].EmplNo
```

```
Don Jones
```

```
554433
```

Figure 3-3. Using Array Syntax in Get Field

FINAL TRIM SIZE : 7.0 in x 8.5 in

To retrieve several or all fields from a record use the `UnBuild Record` object, as shown in the next example:

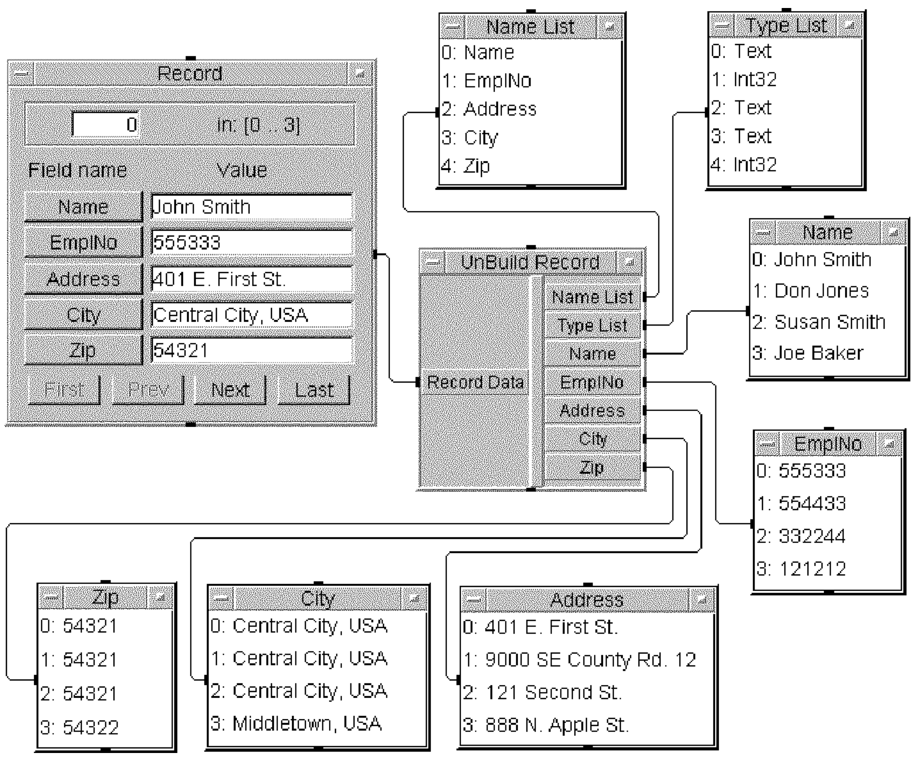


Figure 3-4. Retrieving Record Fields with UnBuild Record

The `UnBuild Record` object not only allows you to add outputs for every field in the record, but provides `Name List` and `Type List` outputs. These outputs list the name and type of each field in the record.

The program is saved in the file `manual38.vee` in your `examples` directory.

Note

Data cannot be automatically converted to and from the Record data type. For example, to send Record data into a Real input terminal, you must extract the field from the Record with the `Unbuild Record` object, or use `Get Field` with the `Rec.A` syntax as described previously.

Programmatically Building Records

The Record object is useful to create and edit simple records, however it is cumbersome to create large records. You also may want to create a record from existing data. In such cases, you use Build Record to build a record.

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

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

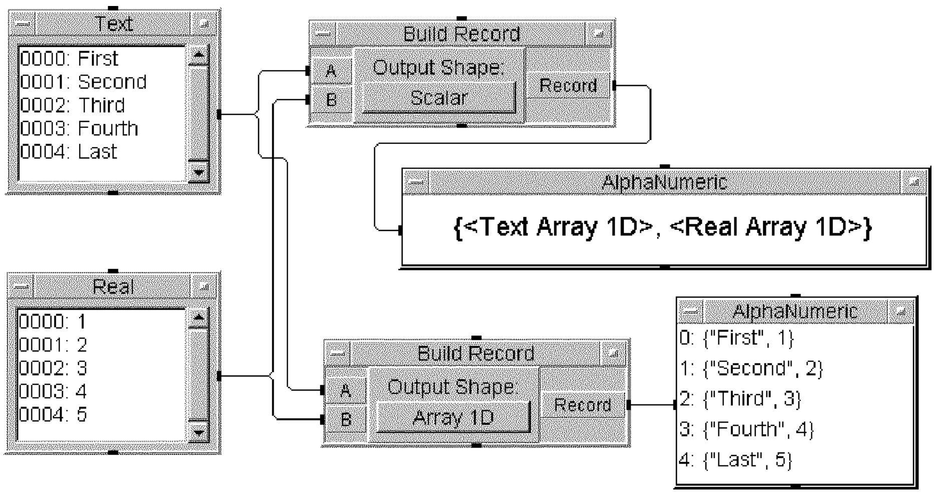


Figure 3-5. The Effect of Output Shape in Build Record

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

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

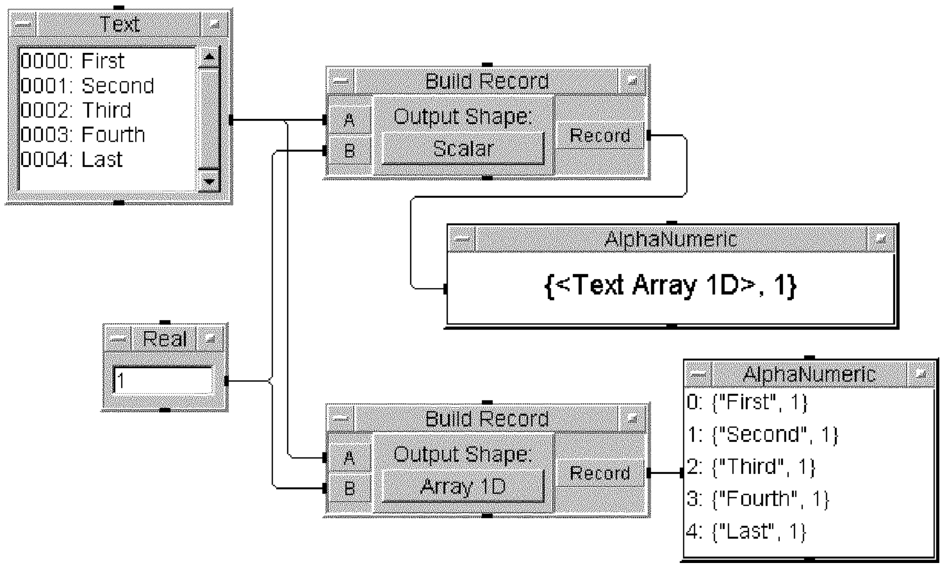


Figure 3-6. Mixing Scalar and Array Input Data

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

Editing Record Fields

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

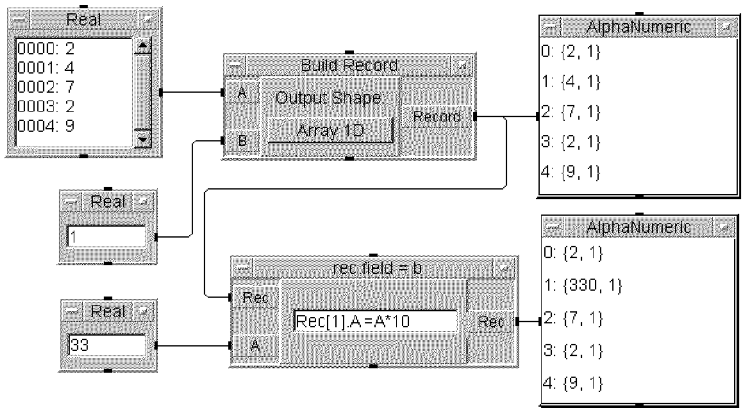


Figure 3-7. Using Set Field to Edit a Record

In this example, a five element record array is built with `Build Record`. The `Set Field` object (titled `rec.field = b`) specifies that the field `Rec[1].A` (the `A` field of record element `1`) is to be assigned the value `A*10`. Note that there is potential for confusion here. In the left-hand expression, the `A` in `Rec[1].A` refers to the `A` field of the record, however, in the right-hand expression, the `A` in `A*10` refers to the value at the `A` input of the `Set Field` object.

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

Note that `Set Field` is a `Formula` object, see `Assignment` in `Math Functions and Operators`, under `Reference` in HP VEE online help for more information.

Using DataSets

As we have seen, HP VEE data (including waveforms) can be built into records and later retrieved. But you can also store records into a file, called a DataSet.

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

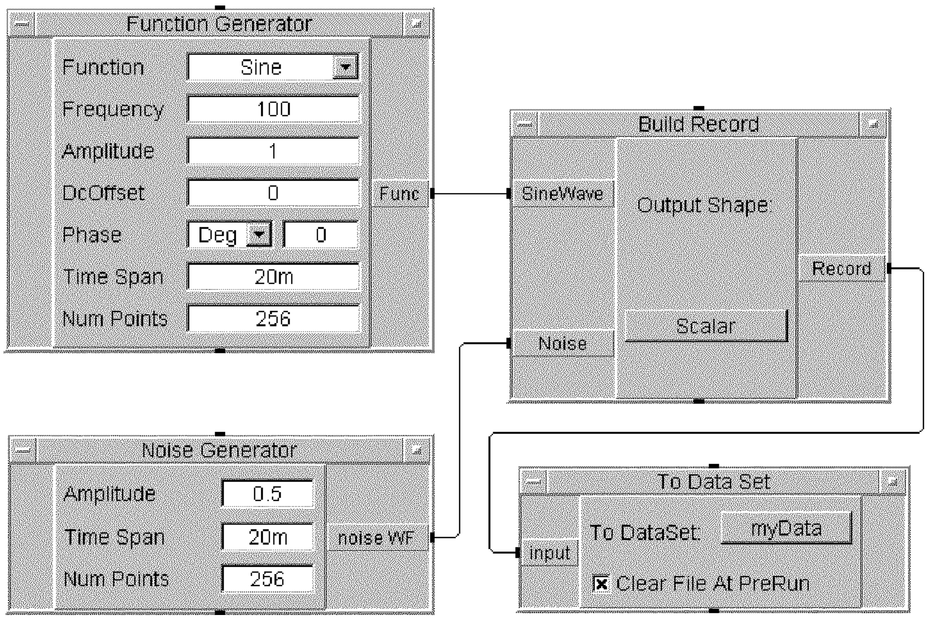


Figure 3-8. Using To DataSet to Save a Record

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

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

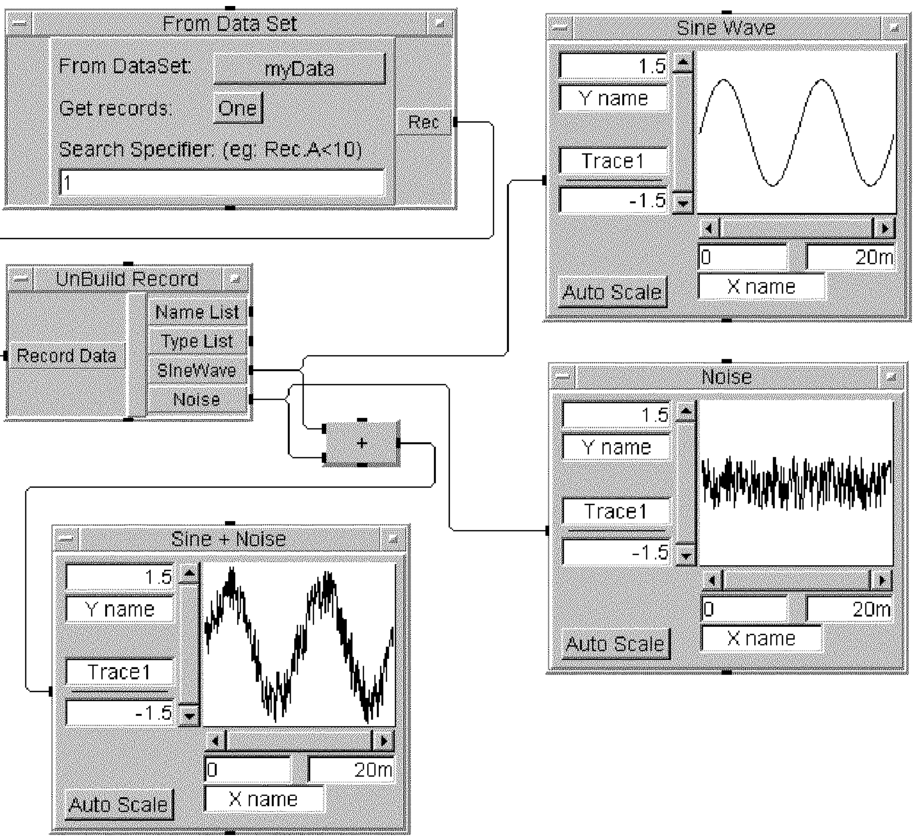


Figure 3-9. Using From DataSet to Retrieve a Record

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

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

4

Using User-Defined Libraries

HP VEE supports three kinds of user-defined functions:

- **UserFunctions**
- **Compiled Functions**
- **Remote Functions**

The methods for creating each type of user-defined function, and for using it into the HP VEE program, are similar. All of these functions are called using the `Call` object, or from certain expressions such as in `Sequencer` or `Formula` objects.

You can use any of the three kinds of user-defined functions in a library. To use a library of functions, generally follow the these steps:

1. Import the library.

   Use the `Device` ⟹ `Import Library` object. Select the `Library Type` (`UserFunction`, `Compiled Function`, or `Remote Function`) and fill in the appropriate fields. Specific information about these fields is explained in the associated section in this chapter.

2. Call one or more functions that are contained in the library.

   Use the `Call`, `Formula`, or `Sequencer` objects from the `Device` menu. You can also use other objects that expressions at run time, such as `If/Then/Else` or `To File`. If you want to have multiple values returned from the function, you must use a `Call` object.

3. Delete the library.

   If memory management or program execution speed is a concern, use the `Device` ⟹ `Delete Library` object to programmatically free the library from memory. Otherwise the libraries are automatically deleted when the program ends.

Specific information about using the different kinds of libraries is listed below.

About UserFunctions

A UserFunction is a user-defined function selected from the `Device` menu. It can also be created from an existing UserObject.

The advantage of creating a UserFunction over using a UserObject is that you can call a single UserFunction several times in your program. Thus, there is only one UserFunction to edit and maintain, rather than several instances of a UserObject. A UserFunction can be created and called locally within an HP VEE program, or it can be saved in a library and imported into a program with the `Import Library` object.

UserFunctions, when executed in compiled mode, will time-slice when called from `Call`, `Formula`, `If/Then/Else`, or `Sequencer` objects (only from the `Function` field). UserFunctions will not time-slice when called from a `To File`, `To String`, or similar object, or if the formula is supplied via a control pin.

For information about creating, editing, and calling a UserFunction, refer to `How Do I` in HP VEE online Help.

Converting Between UserObjects and UserFunctions

To convert a UserObject into a UserFunction, select `Make UserFunction` from the UserObject's object menu. The UserObject window will be replaced by a UserFunction window with the same input and output terminals. The UserObject object is replaced by a UserFunction `Call` object.

To reconvert the UserFunction back into a UserObject, select `Make UserObject` from the object menu of the UserFunction window. Any calls to the UserFunction remain (you'll have to manually delete them), but the UserFunction is automatically converted into a UserObject.

Calling a UserFunction from an Expression

You don't need to use the `Call` object to call a UserFunction. In fact you can call a UserFunction from an expression in a `Formula` object, or from any expression evaluated at run time such as from a `Sequencer` object. The following program demonstrates several ways to call a UserFunction.

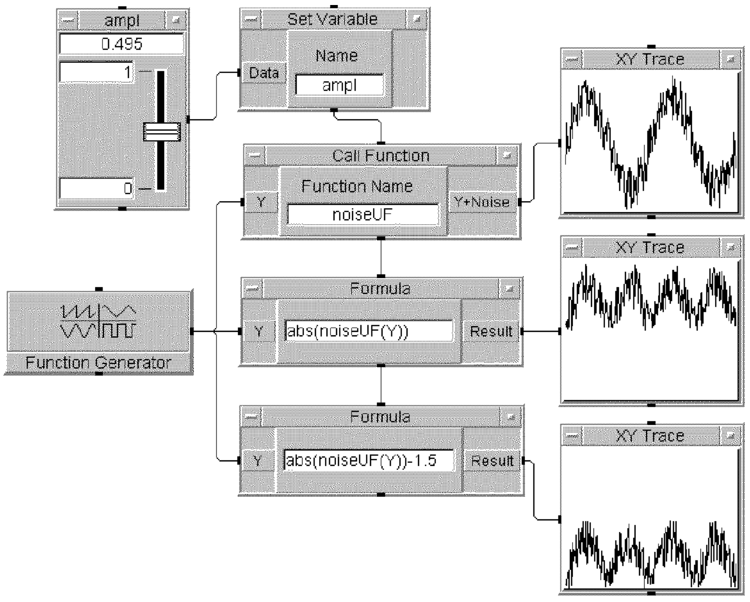


Figure 4-1. Calling a UserFunction from Expressions

In the program, the `Call` object calls the UserFunction `noiseUF` and returns a sine wave with an added noise component. The expression `abs(noiseUF(Y))` in the first `Formula` object returns the absolute value of the waveform returned by the UserFunction. Thus, the displayed noisy sine wave is rectified in the positive direction. The expression `abs(noiseUF(Y))-1.5` in the second `Formula` object also calls the UserFunction, but also adds a negative dc offset to the waveform. Note that the sequence pins are used to ensure correct propagation, because the UserFunction uses the global variable.

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

The ability to call a UserFunction from an expression is very useful — especially when you include such an expression in a transaction in the `Sequencer` object. Refer to Chapter 6 for more information about this topic.

Creating a UserFunction Library

So far we have looked at local UserFunctions, which are created and used within the same program. However, you can create a library of multiple UserFunctions which are stored externally, and later imported into a program.

To create a library of UserFunctions, create the individual UserFunctions in the empty HP VEE work area, and then save to a file. For example, to create a library of two UserFunctions, `myRand1` and `myRand2` (which add random numbers to an input value), you would start by creating two UserFunctions.

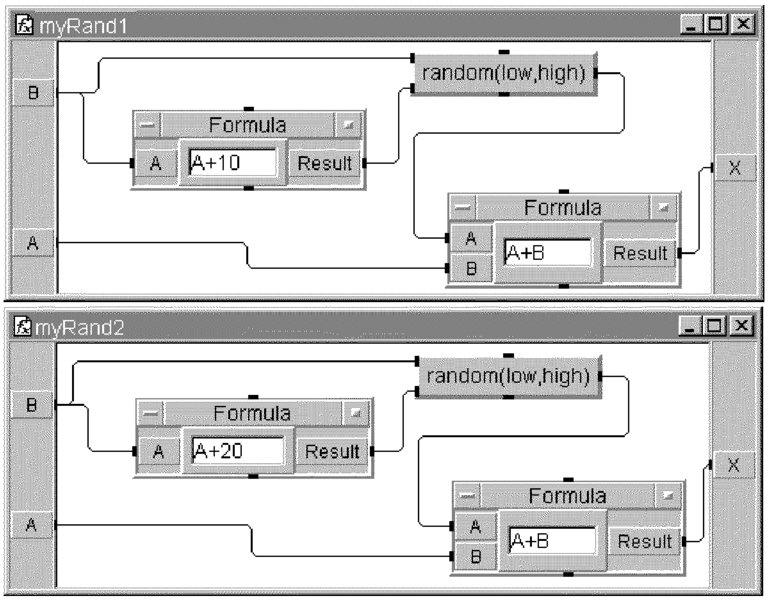


Figure 4-2. Creating UserFunctions for a Library

To create a UserFunction library, save the program containing the UserFunctions.

Note

Generally you want the program to contain only the UserFunctions, however if there are other objects in the program (e.g. in `Main`), they will be ignored when the library is imported.

Because only the UserFunctions are loaded when the library is imported, if you use `Declare Variable` objects, put them in one of the UserFunctions, not in the `Main` window of the library.

To import the UserFunction library into your program, use the `Import Library` object. The following program imports the library from the file `user_func_lib` and calls the UserFunctions `myRand1` and `myRand2`.

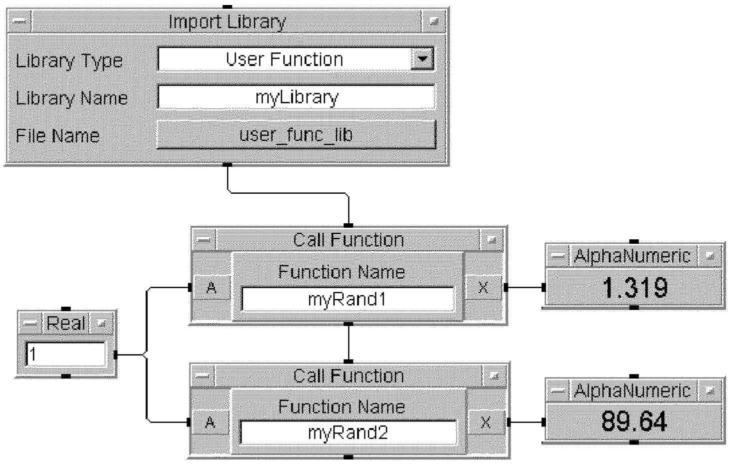


Figure 4-3. Importing a UserFunction Library

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

This program is simple, so it isn't necessary to delete the UserFunction library after it is used. However, in a large program with calls to large libraries, the ability to delete a library when you no longer need it, reduces the memory requirements of the program.

Note

You cannot edit the UserFunctions imported with `Device` ⟹ `Import Library`, but you can view their contents and set breakpoints for debugging. To view imported UserFunctions, use the `Program Explorer` or use `Edit` ⟹ `Edit UserFunction`.

You can *merge* a library of UserFunctions using `File` ⟹ `Merge Library`. Once the library is merged into your program, you can edit the individual UserFunctions with `Edit` ⟹ `Edit UserFunction`.

Differences Between Merging and Importing

You can bring existing, external UserFunctions into your program in two different ways.

- Importing UserFunctions
  - Can be done programmatically or manually.
  - Can be programmatically deleted after use (saving memory).
  - Allows you to use UserFunctions from a single source. For example a single set of common UserFunctions can be used by multiple HP VEE programs. None of the HP VEE programs can change the UserFunctions so the UserFunctions remain consistent, which simplifies maintenance.
  - Imported UserFunctions are not saved with the HP VEE program and therefore saves disk space and improves load time.
- Merging UserFunctions
  - Is done manually (via `File` ⟹ `Merge Library`).
  - Allows you to make the UserFunctions a part of your program and modify them as you need to.
  - Merged UserFunctions are saved with the HP VEE program.

About Compiled Functions

The second type of user-defined function is the Compiled Function, which is created by dynamically linking a library, written in C, C++, FORTRAN, or Pascal, into the HP VEE process. A library of compiled functions is called a shared library on UNIX and a dynamically linked library (DLL) on Microsoft Windows.

To use a Compiled Function, you:

1. Write the external program.
2. Create the DLL (Windows) or shared library (UNIX), and a definition file.
3. Import the library and call the function from HP VEE.
4. Delete the library from HP VEE's memory when you're done.

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

Basically, the methods for importing a Compiled Function library and for calling the function are very similar to what was discussed for UserFunction libraries. The `Import Library` object attaches the shared library to the HP VEE process and parses the definition file declarations. The definition file defines the type of data that is passed between the external library and HP VEE. This file will be discussed later in this section. The Compiled Function can then be called with the `Call` object, or from certain objects such as `Formula` and `Sequencer`. You'll find that creating a Compiled Function is considerably more difficult than creating a UserFunction. Once you have written a library of functions in C or another language, you need to create the shared library and definition file for the program to be linked.

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

Design Considerations for Compiled Functions

There are several reasons for using Compiled Functions in your HP VEE program. You can develop time-sensitive routines in another language and integrate them directly into your HP VEE program by using Compiled Functions. Also, you can use Compiled Functions as a means of providing security for proprietary routines. Because Compiled Functions do not timeslice (i.e. they execute until they are done without interruption) they are only useful for specific purposes that are not available in HP VEE.

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

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

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

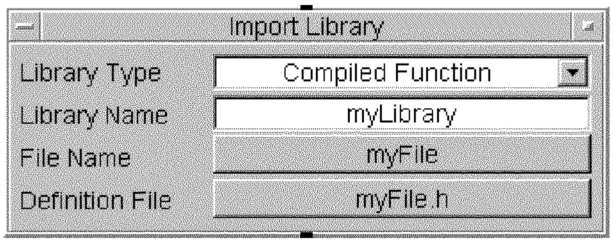
- HP VEE can not trap errors originating in the external routine. Because your external routine becomes part of the HP VEE process, any errors in that routine will propagate back to HP VEE, and a failure in the external routine may cause HP VEE to "hang" or otherwise fail. Thus, you need to be sure of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

- System I/O resources may become locked. Your external routine is responsible for timeout provisions, and so forth.
- If your external routine performs an invalid operation, such as overwriting memory beyond the end of an array or dereferencing a nil or bad pointer, this can cause HP VEE to exit or error with a General Protection Fault (MS Windows) or a segmentation violation (UNIX).

Importing and Calling a Compiled Function

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

The `Import Library` object was explained in the "UserFunctions" section at the beginning of this chapter. To import a Compiled Function library, select `Compiled Function` in the `Library Type` field. Just as for a UserFunction, the `Library Name` field attaches a name to identify the library within the program, and the `File Name` field specifies the file from which to import the library. In addition, there is a fourth field, which specifies the name of the `Definition File`:


Import Library

Library Type	Compiled Function
Library Name	myLibrary
File Name	myFile
Definition File	myFile.h

Figure 4-4. Using Import Library for Compiled Functions

The definition file defines the type of data that is passed between the external routine and HP VEE. It contains the prototypes for the functions.

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

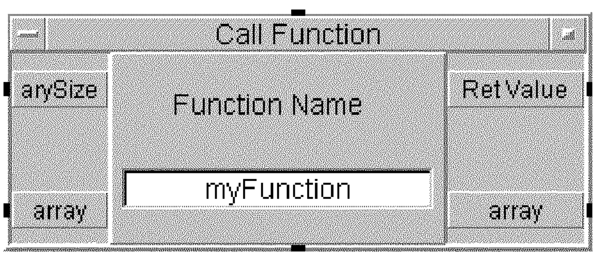


Figure 4-5. Using Call for Compiled Functions

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

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

Creating a Compiled Function (UNIX)

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

The Definition File

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

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

Where:

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

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

If you specify "pass by reference" for a parameter by preceding the parameter name with `*`, HP VEE will pass the address of the information to your function. If you specify "pass by value" for a parameter by leaving out the `*`, HP VEE will copy the value (rather than the address of the value) to your function. You'll want to pass the data by reference if your external routine changes that data for propagation back to HP VEE. *Also, all arrays must be passed by reference.*

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

HP VEE pushes 144 bytes on the stack. This means that it allows up to 36 parameters to be passed by reference to a Compiled Function. This would also imply that up to 36 long integer parameters, or up to 18 double-precision floating-point parameters, may be passed by value.

Note

For HP-UX, you must have the ANSI C compiler in order to generate the position independent code needed to build a shared library for a Compiled Function.

You may include comments in your definition file. HP VEE allows both "enclosed" comments and "to-end-of-line" comments. "Enclosed" comments use the delimiter sequence `/*comments*/`, where `/*` and `*/` mark the beginning and end of the comment, respectively.

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

Building a C Function

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

The following C function accepts a real array and adds 1 to each element in the array. The modified array is returned to HP VEE on the Array terminal, while the size of the array is returned on the Ret Value terminal. This function, once linked into HP VEE, becomes the Compiled Function called in the HP VEE program shown in Figure 4-6.

```
/*
  C code from manual49.c file
*/

#include <stdlib.h>

#ifdef WIN32
#  define DLLEXPORT __declspec(dllexport)
#else
#  define DLLEXPORT
#endif

/* The description will show up on the Program Explorer when you select
   "Show Description" from the object menu and the Function Selection
   dialog box in the small window on the bottom of the box.
*/
DLLEXPORT char myFunc_desc[] = "This function adds 1.0 to the array passed in";

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

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

 return(arraySize);
}
```

The definition file for this function is as follows:

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

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

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

You must include any header files on which the routine depends.

The library should link against any other system libraries needed to resolve the system functions it calls.

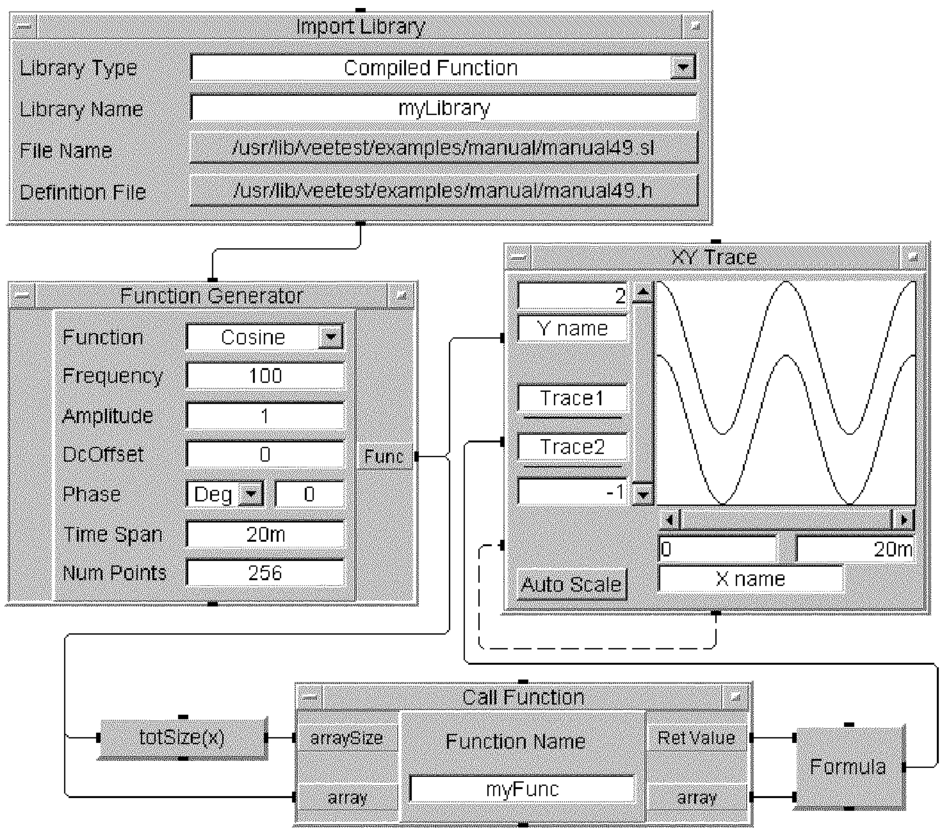
The example program uses the ANSI C function prototype. This isn't necessary, but it makes things a little easier to understand. The function prototype declares the data types that HP VEE should pass into the function. The array has been declared as a pointer variable. HP VEE will put the addresses of the information appearing on the `Call` data in terminals into this variable. The array size has been declared as a long integer. HP VEE will put the value (not the address) of the size of the array into this variable. The positions of both the data input terminals and the variable declarations are important. The addresses of the data items (or their values) supplied to the data input pins (from top to bottom) are placed in the variables in the function prototype from left to right.

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

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

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

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


Import Library
Library Type: Compiled Function
Library Name: myLibrary
File Name: /usr/lib/veetest/examples/manual/manual49.sl
Definition File: /usr/lib/veetest/examples/manual/manual49.h

Function Generator
Function: Cosine
Frequency: 100
Amplitude: 1
DcOffset: 0
Phase: Deg 0
Time Span: 20m
Num Points: 256

XY Trace
2
Y name
Trace1
Trace2
-1
0
20m
X name

Call Function
Function Name: myFunc

Figure 4-6. Program Calling a Compiled Function

The example in Figure 4-6 is saved in the file `manual49.vee` in your `examples` directory. The C file is saved as `manual49.c`, the definition file as `manual49.h`, and the shared library as `manual49.sl`.

Creating a Shared Library

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

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

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

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

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

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

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

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

Binding the Shared Library

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

You delete the shared library from HP VEE either by selecting `Delete Lib` from the `Import Library` object menu, or by including the `Delete Library` object in your program. Note, however, that you may have more than one library name pointing to the same shared library file. In this case, you use the `Delete Library` object to delete each library, but the shared library remains bound until the last library pointing to it is deleted. However, the `Delete Lib` selection in the `Import Library` object menu will unbind the shared library without regard to other `Import Library` objects.

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

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

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

Creating a Dynamic Linked Library (MS Windows)

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

Note

This section tells you how to call a DLL, not how to write a DLL.

HP VEE version 3.2 and greater only calls 32-bit DLLs, not 16-bit DLLs.

Creating the DLL

Create your DLL before writing your HP VEE program. Create your DLL as you would any other DLL except that only a subset of C types are allowed. (See "Creating the Definition File" below.)

If you are using Microsoft Visual C++ version 2.0 or greater, the function definition should be:

```
__declspec(dllexport) long myFunc (...);
```

This definition eliminates the need for a .DEF file to export the function from the DLL. Use the following command line to compile and link the DLL:

```
cl /DWIN32 $file.c /LD
```

/LD creates a DLL. Use /Zi to generate debug information. Note the MS linker links to the C multi-threaded Runtime Library by default. If you use functions like GetComputerName(), you need to link against Kernel32.lib. Then the compile/link line would look like:

```
cl /DWIN32 file.c /LD /link Kernel32.lib
```

Declaring DLL Functions. To work with HP VEE, DLL functions can be declared as __declspec(dllexport) using Mocrosoft C++ version 2.0 or greater. This application eliminates the need for a .DEF file. For example, a generic function could be created as follows:

```
__declspec(dllexport) long generic Func(long a) {return (a*2); }
```

If you are not using Microsoft Visual C++, then the .DEF file contains:

```
EXPORTS genericFunc
```

And the function definition looks like:

```
long generic Func(long a) {return(a*2);}
```

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

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

Where:

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

Examples.

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

Parameter Limitations

A DLL function pushes 144 bytes on the stack. This limits the number of parameters used by the function. Any combination of parameters may be used as long as the 144-byte limit is not exceeded. A long uses four bytes, a double uses eight bytes and a pointer uses four bytes. For example, a function could have 36 longs, or 18 doubles, or 20 pointers and 8 doubles.

The Import Library Object

Before you can use a `Call` object or `Formula` box to execute a DLL function you must import the function into the HP VEE environment via the `Import Library` object. On the `Import Library` object select `Compiled Function` under `Library Type`. Enter the correct definition file name using the `Definition`

File button. Finally, select the correct file using the File Name button. The Library Name button assigns a logical name to a set of functions and does not need to be changed.

The Call Object

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

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

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

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

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

The Delete Library Object

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

Using DLL Functions in Formula Objects

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

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

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

About Remote Functions

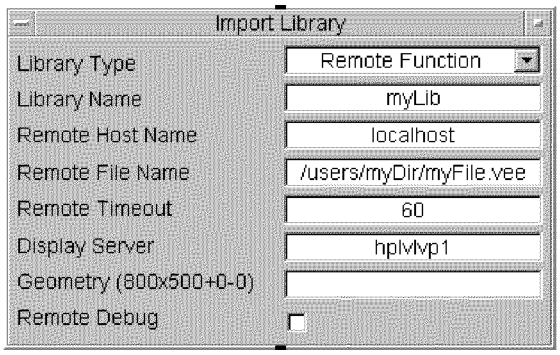
The third type of user-defined function is the Remote Function. A Remote Function is a UserFunction that runs in another HP VEE process on a remote host computer. Because Remote Functions are a special case of a UserFunction, refer to "About UserFunctions" for general information that applies to UserFunctions.

The Remote Function is called from the local HP VEE process over the LAN (Local Area Network). Just as for UserFunctions and Compiled Functions, you import a library of Remote Functions with the `Import Library` object.

Once one or more Remote Functions have been imported, they are called by either using the `Call` object, or by including function names in expressions. You include Remote Function calls in your program just as you would UserFunctions. However, there are some differences, and some networking technicalities, which are described in this section.

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

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



Library Type	Remote Function
Library Name	myLib
Remote Host Name	localhost
Remote File Name	/users/myDir/myFile.vee
Remote Timeout	60
Display Server	hplvlvp1
Geometry (800x500+0-0)	
Remote Debug	☐

Figure 4-7. Import Library for Remote Functions

The `Library Type` and `Library Name` fields function the same as for UserFunctions and Compiled Functions. The other fields are as follows:

- `Remote Host Name` - This is the name of the host on which the "service" HP VEE process is to run (the "remote host"). This name can be the common or symbolic name of the host (for example `myhost`). On the other hand, you can enter the IP address of the host in this field (for example `14.13.29.99`).
- `Remote File Name` - This is the name of the Remote Function library file. The `Remote File Name` is analogous to the `File Name` field for a UserFunction library. However, you must specify the *absolute* path to the file. Hence the path and file name can be rather long. You may want to have all users place remote function library files in a common place, for example: `/users/remfunc/` or `C:\USERS\REMFUNC`.

Note

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

- `Remote Timeout` - This field specifies a timeout period in seconds for communication with the HP VEE service. If the HP VEE service has not returned the expected results of a Remote Function within this time period, an error occurs.

- Display Server - Enter a resolvable host name or IP address. The host must have an X Server running and permissions must be set to have an X client display on the specified machine. If the service is instantiated on a MS Windows machine, the Display Server field must be the same as the Remote Host Name. On HP-UX, they can be different.
- Geometry - Enter the initial geometry for the window that contains the view of the remote HP VEE, in the standard geometry format. For example, 800x500+0-0.
- Remote Debug - When this check box is selected, all of the UserFunctions within the library will execute in debug mode (i.e., you will be able to perform debugging on them such as setting breakpoints and doing line probes). This setting works with UserFunctions whether or not they have panel views.

When the Import Library object is executed (either by selecting Load Lib from the object menu, or during normal program execution), a HP VEE server process is started on the remote host specified in the Host Name field. The client process and the server process are connected over the network, and are able to communicate. When a Call object in the client HP VEE calls a Remote Function, the arguments (the data input pins on the Call object) are sent over the network to the remote service, the Remote Function is executed, and the results are sent back to the Call object and output on its data output pins. If your program deletes the library of Remote Functions with the Delete Library object, the Remote Functions associated with the library are removed. You can load multiple libraries in a HP VEE server process, then delete each one as needed without canceling the service connection. The HP VEE server exists while the HP VEE client process continues to run.

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

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

1. The HP VEE service process will execute only Remote Functions that are contained in the Remote Function library named by Import Library.

2. Remote Functions have views associated with them. When you call a remote functions, you can have a HP VEE window appear if the UserFunction displays a panel view.
3. Global variables (declared and undeclared) are not shared between the processes.
4. Remote Functions will not time-slice when called.

UNIX Security, UIDs, and Names

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

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

Client host: `myhost`

User: `mike`

Password: `twoheads`

And the service host can be identified as follows:

Service host: `remhost`

User: `mike`

Password: `arebetter`

Directory: `/users/mike`

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

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

  or

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

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

Note

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

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

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

Resource Files

The VEE.IO or .veeio, and VEE.RC or .veerc files used by the HP VEE service process are those that belong to the user who invokes the process on the remote host. Thus, for the user `mike` in our previous example, the HP VEE service process will read the following files on host `remhost`:

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

(HP VEE only reads the VEE.IO or .veeio file. The VEE.RC or .veerc file is used for trig preferences only.)

Errors

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

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

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

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

Note

It is possible to write a Remote Function that will hang, such as an infinite loop. In this case, the Remote Function will time out with a fatal error message. The HP VEE client will attempt to remove the service, but will fail since the service will never respond. You then need to terminate the process on the remote machine. For example, in HP VEE for HP-UX you log onto the remote host and determine the process id with `ps` and terminate the process with `kill`.

About Callable VEE

In some cases you may want to build an application in another language, and still use HP VEE UserFunctions. Just as Remote Functions allow one HP VEE to access UserFunctions of another HP VEE, Callable VEE allows you to call UserFunctions from a C program, or any language that can access C routines.

Note that the server system needs to have HP VEE present and accessible to run the UserFunctions; they cannot be executed on their own. UserFunctions have to be organized into a library that HP VEE can load and execute.

The tools needed to support Callable VEE are provided with HP VEE:

- A C library, named `libvapi.a` is found in the `lib` subdirectory of the HP VEE installation. This library is to be linked to your C program.

  This library supports two Application Program Interfaces (APIs). One API (VEE RPC) sets up and controls the Remote Procedure Call (RPC) between the C program and HP VEE. The prototypes for the functions in this API are in `veeRPC.h` and perform the following actions:

  - Loading and unloading HP VEE servers.
  - Loading and unloading HP VEE libraries.
  - Listing UserFunctions in HP VEE libraries.
  - Calling and receiving data from UserFunctions.
  - Performing related status and housekeeping.

  The second API (VEE DATA) performs conversions between C and HP VEE data types. The prototypes for the functions in this API are in `veeData.h`.

- The HP VEE Service Manager, `veesm.exe` (`veesm` on HP-UX) is located with the other HP VEE executables in the HP VEE installation directory. It handles running the target HP VEE with its UserFunctions. and allows a remote client to bring up HP VEE as a server.

  On HP-UX systems, `veesm` is automatically run by the `inet` daemon process. On a PC, either run `veesm.exe` or put it into the Windows Startup Group so it is started when the PC is started.

There are example programs in the `escapes` directory that demonstrate Callable VEE. They are named `callVEE.c` and `callVEE.vee`.

About the VEE RPC API

The VEE RPC API handles setting up, maintaining, and closing the connection between the C client program and the HP VEE server.

The VEE RPC API's routines use one of three handles in their operation:

```
VRPC_SERVICE;  // Handle to a VEE server.
VRPC_LIBRARY;  // Handle to a VEE UserFunction library.
VRPC_FUNCTION; // Handle to a VEE UserFunction.
```

The API calls are organized as described in the following subsections.

Starting and Stopping a Server

The most essential API functions are the two that start and stop a HP VEE server. To load a HP VEE server use:

```
VRPC_SERVICE vrpcCreateService( char *hostName,
                                char *display,
                                char *geometry,
                                double aTimeoutInSeconds,
                                unsigned long flags );
```

This function starts an HP VEE server on the host given by `hostName`. The `hostName` can be in text form (for example, `mycomputer@lvld.hp.com`) or numeric form (`15.11.55.105`). The function returns a server handle. You get a NULL (effectively a zero) back if something goes wrong; you can then get the precise error information with the `veeGetErrorNumber()` and `veeGetErrorString()` functions as outlined in the next section.

The `display` argument specifies a remote display using a network address in text (`babylon:0.0`) or numeric form (`15.11.55.101:0.0`) on a networked X Windows system.

The `geometry` argument specifies the HP VEE window size and placement. For example `800x500+0+0` puts an 800x500 HP VEE window in the lower-left corner of the display.

The `aTimeoutInSeconds` argument gives the number of seconds to wait when starting the service. This value is used for all later calls in the session unless changed by `vrpcSetTimeout()`.

The `flags` argument is not normally used; you can, however, set it to the value `VEERPC_CREATE_NEW` to start a new copy of HP VEE on a server instead of using the one already started.

To stop a HP VEE server you use:

```
VRPC_SERVICE vrpcDeleteService( VRPC_SERVICE aService );
```

The only argument is the server handle obtained when you originally started the server. You get a NULL pointer back if all is OK, otherwise you get a non-NULL pointer.

Loading and Unloading a Library

Once you have started the server, you then need to load a library into the remote copy of HP VEE; this is done with:

```
VRPC_LIBRARY vrpcLoadLibrary( VRPC_SERVICE aService,
                              char *FunctionName );
```

This function accepts as arguments a server handle and the pathname of a library of UserFunctions specified by `FunctionName`; it returns a library handle. If it fails, you get a NULL back.

Once loaded, you can specify either normal or debugging execution mode for the library with:

```
void vrpcSetExecutionMode( VRPC_LIBRARY aLibrary,
                           unsigned long executionMode );
```

In this function, you specify the handle for the library and an `executionMode` flag, which can be set to `VRPC_DEBUG_EXECUTION` (which specifies single-stepping through the UserFunction on the target system) and then set back to the default `VRPC_NORMAL_EXECUTION`. This returns a 0 if successful, an error code if not.

You can similarly unload the library with:

```
VRPC_LIBRARY vrpcUnLoadLibrary( VRPC_LIBRARY aLibrary );
```

The only argument is the library handle.

Selecting UserFunctions

Now that you are connected to the server and have a library loaded, you need to get a handle to a UserFunction.

You get a function handle with:

```
VRPC_FUNCTION vrpcFindFunction( VRPC_LIBRARY aLibrary,
                                char *aFunctionName );
```

You specify the library handle and a string giving the UserFunction name as arguments, and get back the function handle, or a NULL if something goes wrong.

To get information on the function, use:

```
struct VRPC_FUNC_INFO*
           vrpcFunctionInfo( VRPC_FUNCTION aFunction );
```

This returns a data structure or a NULL if something goes wrong. The data structure is of the form:

```
typedef struct VRPC_FUNC_INFO
{
  char *functionName;              // Name of function.
  long numArguments;               // # of input pins on function.
  enum veeType *argumentTypes;     // List of argument types.
  veeShape *argumentShapes;        // List of argument shapes.
  long numResults;                 // # of output pins on function.
  enum veeType *resultTypes;       // List of output types.
  veeShape *resultShapes;          // List of output shapes.
};
```

If you get a NULL, the memory for this is taken up in your process space, so if you want to get rid of it you use:

```
struct VRPC_FUNC_INFO*
           vrpcFreeFunctionInfo( struct VRPC_FUNC_INFO *funcinfo );
```

You can determine what functions are in the library with:

```
      char** vrpcGetFunctionNames( VRPC_LIBRARY aLibrary,
                                   long *numberOfFunctions );
```

This accepts a library handle as an argument; it returns a pointer to an array of null-terminated strings giving the function names directly, and the `numberOfFunctions` in the library as a argument. You get a NULL pointer back if an error occurs. The string array exists in your process space.

Calling UserFunctions

Now you can call the UserFunction.

You call and receive in a single function using:

```
      VDC* vrpcCallAndReceive( VRPC_FUNCTION aFunction,
                               VDC *arguments );
```

This function blocks, waiting for the function to complete or until a timeout occurs. You specify a function handle and an input array of HP VEE Data Containers (VDCs). Handling VDCs is the function of the VEE DATA API and is covered in “About the VEE DATA API”.

Or to call a UserFunction in blocking mode, you can invoke:

```
long vrpcCall( VRPC_FUNCTION aFunction,
               VDC *arguments );
```

This function does not “block”, it returns immediately, whether it worked on not; it returns 0 if all is OK, and an error code if not.

Of course, since most UserFunctions will return sometime, you will want to get a value back, and for that you use:

```
VDC* vrpcReceive( VRPC_FUNCTION aFunction,
                  unsigned long waitMode );
```

You specify a function handle and a `waitMode` flag, which can have one of three values:

- `VRPC_NO_WAITING` The call returns immediately with or without results.
- `VRPC_WAIT_SLEEPING` Wait for data until timeout (server sleeps).
- `VRPC_WAIT_SPINNING` Wait for data until timeout (server busy).

If the function fails, a NULL is returned.

Other Functions

This section lists other utility functions in the VEE RPC API:

- This function allows you to change the timeout. You specify a server handle and the timeout in seconds. You get back a zero if all is OK, and an error code if not.

  ```
  long vrpcSetTimeout( VRPC_SERVICE aService,
                       double aTimeoutInSeconds );
  ```

- This function allows you to set the default C client behavior for receiving data:

  ```
  long vrpcSetBehavior( VRPC_SERVICE aService,
                        unsigned long flags );
  ```

  You specify a server handle and the flag, and get back 0 or an error code. The flags are as follows:

  `VRPC_WAIT_SLEEPING` Wait for data until timeout (client sleeps).
  `VRPC_WAIT_SPINNING` Wait for data until timeout (client busy).

  You can also OR in a flag, `VRPC_BUFFER_EXPAND`, to specify that the C client will allocate and retain larger buffers in response to increasing sizes of data returned from the server.

- You can query the revision number of the remote `veesm` with:

  ```
  long vrpcGetServerVersion( VRPC_SERVICE aService );
  ```

  You give this a server handle and get back either a revision code or a 0 (if you have an error).

Error Codes for the VEE RPC API

The following error codes are returned when a connection to the HP VEE server cannot be made:

Error Code	Meaning
850: `eUnknownHost`	The host name or IP address is unresolvable.
851: `eNoServiceManager`	`veesm` cannot be found on the server host.
861: `eServiceManagerTO`	The service manager timed-out.
863: `eServiceNotFound`	Unable to find the HP VEE service.
864: `eServiceNotStarted`	Unable to start the HP VEE service.
866: `eConnectRefused`	The connection to `veesm` or `inetd` was refused.
868: `eFailedSecurity`	Failed the security check on UNIX.

The following are fatal errors that occur after connection to a HP VEE server (the connection has been terminated):

Error Code	Meaning
852: `eHostDown`	The HP VEE server host is down.
853: `eConnectTimedOut`	The connection has timed out.
855: `eConnectBroken`	The connection has broken.

The following errors reflect an internal non-fatal state within the service:

Error Code	Meaning
865: `eSomeInternalError`	A non-fatal internal error occurred.
869: `eVeeServiceError`	There is an error within the UserFunction.
870: `eWouldBlock`	Returned for non-blocking RPC.
871: `eDebugTermination`	The user pressed stop during a debug session.

The following error is returned by a RPC function call:

Error Code	Meaning
851: `eInvalidArgument`	There is an invalid argument.

About the VEE DATA API

As shown in the previous section, performing a Call or Receive with a UserFunction requires handling data in the VEE Data Container (VDC) format, which is a set of data structures required by HP VEE for its internal operation. Communicating with HP VEE from your C program requires an ability to translate between VDCs and conventional C data types. The VEE DATA API provides this ability (and a few others).

Data Types, Shapes and Mappings

The fundamental VDC types are listed in the `veeData.h` header file as:

```
enum veeType
{
   VEE_TYPE_ANY=0,  // The default without constraints.
   VEE_NOT_DEFINED1,// Leave space.
   VEE_LONG,        // 32-bit signed integer (no 16-bit INTs in VEE).
   VEE_NOT_DEFINED2,// Leave space.
   VEE_DOUBLE,      // IEEE 754 64-bit floating-point number.
   VEE_COMPLEX,     // Complex number: 2 doubles in rectangular form.
   VEE_PCOMPLEX,    // Complex number: 2 doubles in polar form.
   VEE_STRING,      // 8-bit ASCII null-terminated string.
   VEE_NIL,         // Empty container returned by function call.
   VEE_NOT_DEFINED3,// Leave space.
   VEE_COORD,       // 2 or more doubles give XY or XYZ or ... data.
   VEE_ENUM,        // An ordered list of strings.
   VEE_RECORD,      // VEE record-structures data.
   VEE_NOT_DEFINED4,// Leave space.
   VEE_WAVEFORM,    // A 1D array of VEE_DOUBLE with a time mapping.
   VEE_SPECTRUM     // A 1D array of VEE_PCOMPLEX with a time mapping.
};
```

For convenience, the `veeData.h` file defines C data types for translation with HP VEE data types:

```
typedef short int16;
typedef long  int32;
typedef struct {double rval, ival;} veeComplex;
typedef struct {double mag, phase;} veePComplex;
typedef struct {double xval, yval;} vee2DCoord;
typedef struct {double xval, yval, zval;} vee3DCoord;
typedef void veeDataContainer;
typedef veeDataContainer* VDC;
```

The data types can also have a specified number of dimensions, or numDims, given by:

```
enum veeShape
{
   VEE_SHAPE_SCALAR,      // A single data element.
   VEE_SHAPE_ARRAY1D,     // A one-dimensional array.
   VEE_SHAPE_ARRAY2D,     // A two-dimensional array.
   VEE_SHAPE_ARRAY3D,     // A three-dimensional array.
   VEE_SHAPE_ARRAY,       // An array with from 4 to 10 dimensions.
   VEE_SHAPE_ANY          // Placeholder for undefined shape.
};
```

Arrays can be "mapped". Normally they aren't, but the VEE_WAVEFORM and VEE_SPECTRUM data types are mapped types where the array elements correspond to time intervals. Mappings are given by:

```
enum veeMapType
{
   VEE_MAPPING_NONE,      // No mapping.
   VEE_MAPPING_LINEAR,    // Linear mapping.
   VEE_MAPPING_LOG        // Log mapping.
};
```

Generally you don't need to worry about specifying mappings.

Scalar Data Handling

To create VDC scalars from C data, use the following functions:

```
VDC vdcCreateLongScalar( int32 aLong );

VDC vdcCreateDoubleScalar( double aReal );

VDC vdcCreateStringScalar( char *aString );

VDC vdcCreateComplexScalar( double realPart,
                            double imaginaryPart );

VDC vdcCreatePComplexScalar( double magnitude,
                             double phase );

VDC vdcCreate2DCoordScalar( double xval,
                            double yval );
```

```
VDC vdcCreate3DCoordScalar( double xval,
                            double yval,
                            double zval );

VDC vdcCreateCoordScalar( int16 aFieldCount,
                          double *values );
```

All these functions return a pointer to a VDC, or a NULL if they fail. There are, of course, no scalars of VEE_WAVEFORM or VEE_SPECTRUM types as they are always 1D arrays by definition.

You can change the values in the VDCs with another set of routines:

```
int32 vdcSetLongScalar( VDC aVD,
                        int32 aLong );

int32 vdcSetDoubleScalar( VDC aVD,
                          double aReal );

int32 vdcSetStringScalar( VDC aVD,
                          char *aStr );

int32 vdcSetComplexScalar( VDC aVD,
                           double realPart,
                           double imaginaryPart );

int32 vdcSetPComplexScalar( VDC aVD,
                            double magnitude,
                            double phase );

int32 vdcSet2DCoordScalar( VDC aVD,
                           double xval,
                           double yval );

int32 vdcSet3DCoordScalar( VDC aVD,
                           double xval,
                           double yval,
                           double zval );

int32 vdcSetCoordScalar( VDC aVD,
                         int16 aFieldCount,
                         double* values );
```

As described above, these functions return either 0 or an error code.

When you have created a scalar VDC or returned one from a function, you can get the C data type out of it with another set of routines:

```
int32 vdcGetLongScalarValue( VDC aVD,
                             int32 *aLong );

int32 vdcGetDoubleScalarValue( VDC aVD,
                               double *aReal );

char* vdcGetStringScalarValue( VDC aVD );

int32 vdcGetComplexScalarValue( VDC aVD,
                                veeComplex *aComplex );

int32 vdcGetPComplexScalarValue( VDC aVD,
                                 veePComplex *aPComplex );

int32 vdcGet2DCoordScalarValue( VDC aVD,
                                vee2DCoord *aCoord );

int32 vdcGet3DCoordScalarValue( VDC aVD,
                                vee3DCoord *aCoord );

double* vdcGetCoordScalarValue( VDC aVD,
                                int16 *aFieldCount );
```

In general, these functions take the data out of the first argument, a VDC, and put it into the second, with is a C variable (with some types as defined at the beginning of this section); they return 0 if no error and an error code if there is an error.

The exceptions are the `vdcGetStringScalarValue( )` function, which returns a string directly from the function (or a NULL string if something goes wrong), and the `vdcGetCoordScalarValue()` function, which returns a pointer to an array of N-dimensional coordinate data (with N returned as an argument).

Finally, you can interrogate coordinate types for their number of coordinate dimensions or set the coordinate dimensions to new values if desired:

```
int16 vdcNumCoordDims( VDC aVD );
int32 vdcCoordSetNumCoordDims( VDC, int16 );
```

Array Data Handling

These functions create array VDC of HP VEE types. The values you supply are copied into the VDC, the callers memory is never used. If an error occurs a null pointer is returned. You create VDC arrays with the following set of functions:

- This function returns a VDC of type `VEE_LONG` which is allocated to a size equal to the argument, `numPts`. The array of data pointed to by the argument, `values`, must be of the same specified size. The type of the argument, `int32`, is type defined to type `long` in `veeData.h`.

```
VDC vdcCreateLong1DArray( int32 numPts,
                          int32 *values );
```

- This function returns a VDC of type `VEE_STRING` which is allocated to a size equal to the argument, `numPts`. The argument, strings, points to an array of pointers which in turn point to null terminated strings. The number of strings in the array must equal the specified size. The type of the argument, `int32`, is type defined to type `long` in `veeData.h`.

```
VDC vdcCreateString1DArray( int32 numPts,
                            char **strings );
```

- This function returns a VDC of type `VEE_DOUBLE` which is allocated to a size equal to the argument, `numPts`. The argument, `values`, points to an array of data. The number of doubles in the array must equal the specified size. The type of the argument, `int32`, is type defined to type `long` in `veeData.h`.

```
VDC vdcCreateDouble1DArray( int32 numPts,
                            double *values );
```

- This function returns a VDC of type `VEE_COMPLEX` which is preallocated to a size equal to the argument, `numPts`. The type of the argument, `int32`, is type defined to type `long` in `veeData.h`. The argument, values, points to an array of structures of type `veeComplex`. This structure is defined in `veeData.h` as:

```
typedef struct {double rval, ival;} veeComplex;
```

```
VDC vdcCreateComplex1DArray( int32 numPts,
                             veeComplex *values );
```

- This function returns a VDC of type `VEE_PCOMPLEX` which is preallocated to a size equal to the argument, `numPts`. The type of the argument, `int32`, is type defined to type `long in` `veeData.h`. The argument, `values`, points to an array of structures of type `veePComplex`. This structure is defined in `veeData.h` as:

```
typedef struct {double mag, phase;} veePComplex;

  VDC vdcCreatePComplex1DArray( int32 numPts,
                                veePComplex *values );
```

- This function returns a VDC of type `VEE_COORD` which is preallocated to a size equal to the argument, `numPts`. The type of the argument, `int32`, is type defined to type `long` in `veeData.h`. The argument, `values`, points to an array of structures of type `vee2DCoord`. This structure is defined in `veeData.h` as:

```
typedef  struct {double xval, yval;} vee2DCoord;

   VDC vdcCreate2DCoord1DArray( int32 numPts,
                                vee2DCoord *values );
```

- This function returns a VDC of type `VEE_COORD` which is preallocated to a size equal to the argument, `numPts`. The type of the argument, `int32`, is type defined to type `long` in `veeData.h`. The argument, `values`, points to an array of structures of type `vee3DCoord`. This structure is defined in `veeData.h` as:

```
typedef  struct {double xval, yval, zval;} vee3DCoord;

  VDC vdcCreate3DCoord1DArray( int32 numPts,
                               vee3DCoord *values );
```

- This function returns a VDC of type `VEE_COORD` which is preallocated to a size equal to the argument, `numPts`. The argument, `aFieldCount`, is the number of fields in the coordinates. The type of the argument, `int32`, is type defined to type `long` in `veeData.h`. The argument, `values`, points to an array of type `double`. The length of this array must be equal to the product of `numPts` and `aFieldCount`.

```
VDC vdcCreateCoord1DArray( int32 numPts,
                           int16 aFieldCount,
                           double *values );
```

- This function returns a VDC of type VEE_WAVEFORM with a number of samples equal to the argument, numPts. The starting and ending times for the waveform are the arguments, from and thru. The argument, mapType, is of type VMT, defined in veeData.h; it declares what type of mapping is used. Refer to "Data Types, Shapes and Mappings" for more information. The array of doubles pointed to by the argument, data, must be equal in size to the argument, numPts. The type of the argument, int32, is type defined to type long in veeData.h.

```
    VDC vdcCreateWaveform( int32 numPts,
                           double from,
                           double thru,
                           VMT mapType,
                           double *data );
```

- This function returns a VDC of type VEE_SPECTRUM with a number of samples equal to the argument, numPts. The starting and ending frequencies for the spectrum are the arguments, from and thru. The argument, mapType, is of type VMT, defined in veeData.h; it declares what type of mapping is used. Refer to "Data Types, Shapes and Mappings" for more information. The array of type veePComplex pointed to by the argument, data, must be equal in size to the argument, numPts.Type. veePComplex is a structure defined in veeData.h:

```
typedef struct {double mag, phase;} veePComplex.
```

  The array of structures is copied. The type of the argument, int32, is type defined to type long in veeData.h.

```
    VDC vdcCreateSpectrum( int32 numPts,
                           double from,
                           double thru,
                           VMT mapType,
                           veePComplex *data);
```

In the functions listed above, you specify an array size, any additional data needed to represent the array (such as mapping data for VEE_WAVEFORM and VEE_SPECTRUM types), and the array data, and get back a VDC (or a NULL if something goes wrong).

You can convert back from VDCs to C arrays with:

- This function returns a pointer to an array of type `int32`. The argument, `aVD`, must be of type, `VEE_LONG`, and be an array. The value returned in the pass-by-reference argument, `numPts`, is the length of the array.

  ```
      int32* vdcGetLong1DArray( VDC aVD,
                                int32 *numPts );
  ```

- This function returns a pointer to an array of type `double`. The argument, `aVD`, must be of type, `VEE_DOUBLE`. The value returned in the pass-by-reference argument, `numPts`, is the length of the array.

  ```
      double* vdcGetDouble1DArray( VDC aVD,
                                   int32 *numPts );
  ```

- This function returns a pointer to an array of pointers each pointing to a null terminated string. The argument, `aVD`, must be of type `VEE_STRING`. The value returned in the pass-by-reference argument, `numPts`, is the number of strings.

  ```
      char** vdcGetString1DArray( VDC aVD,
                                  int32 *numPts );
  ```

- This function returns a pointer to an array of structures of type, `veeComplex`. This structure is defined in `veeData.h` as:

  ```
   typedef struct {double rval, ival;} veeComplex;
  ```

  The argument, `aVD`, must be of type `VEE_COMPLEX`. The value returned in the pass-by-reference argument, `numPts`, is the length of the array.

  ```
      veeComplex* vdcGetComplex1DArray( VDC aVD,
                                        int32 *numPts );
  ```

- This function returns a pointer to an array of structures of type, `veePComplex`. This structure is defined in `veeData.h` as:

  ```
   typedef struct {double mag, phase;} veePComplex;
  ```

  The argument, `aVD`, must be of type `VEE_PCOMPLEX`. The value returned in the pass-by-reference argument, `numPts`, is the length of the array.

  ```
      veePComplex* vdcGetPComplex1DArray( VDC aVD,
                                          int32 *numPts );
  ```

- This function returns a pointer to an array of structures of type, vee2DCoord. This structure is defined in veeData.h as:

  ```
  typedef  struct {double xval, yval;} vee2DCoord;
  ```

  The argument, aVD, must be of type VEE_COORD. The value returned in the pass-by-reference argument, numPts, is the length of the array.

  ```
     vee2DCoord* vdcGet2DCoord1DArray( VDC aVD,
                                       int32 *numPts );
  ```

- This function returns a pointer to an array of structures of type, vee3DCoord. This structure is defined in veeData.h as:

  ```
  typedef  struct {double xval, yval, zval;} vee3DCoord;
  ```

  The argument, aVD, must be of type VEE_COORD. The value returned in the pass-by-reference argument, numPts, is the length of the array.

  ```
     vee3DCoord* vdcGet3DCoord1DArray( VDC aVD,
                                       int32 *numPts );
  ```

- This function returns a pointer to an array of type double. The argument, aVD, must be of type VEE_COORD. The value returned in the pass-by-reference argument, numPts, is the number of coordinate tuples in the array. The value returned in the pass-by-reference argument, aFieldCount, is the number of fields in each coordinate tuple. The length of the returned array is the product of numPts and aFieldCount.

  ```
     double* vdcGetCoord1DArray( VDC aVD,
                                 int32 *numPts,
                                 int16 *aFieldCount );
  ```

- This function returns a pointer to an array of type double. The argument, aVD, must be of type VEE_WAVEFORM. The pass-by-reference arguments numPts, from, thru, and mapType return, respectively, the length of the array, the start time, the end time, and the type of mapping.

  ```
     double* vdcGetWaveform( VDC aVD,
                             int32 *numPts,
                             double *from,
                             double *thru,
                             VMT *mapType );
  ```

- This function returns a pointer to an array of structures of type veePComplex. This structure is defined in veeData.h as:

  ```
   typedef struct {double mag, phase;} veePComplex;
  ```

The argument, aVD, must be of type VEE_WAVEFORM. The pass-by-reference arguments numPts, from, thru, and mapType return, respectively, the length of the array of structures, the starting frequency, the ending frequency, and the type of mapping.

```
veePComplex* vdcGetSpectrum( VDC aVD,
                             int32 *numPts,
                             double *from,
                             double *thru,
                             VMT *mapType );
```

These functions take a VDC, return a pointer to the array of data directly, and return the size of the array (or any other relevant information) as arguments.

Once the arrays are created, you can also check, interrogate, or manipulate the arrays with the following functions:

```
int32 vdcSetNumDims( VDC, int16 );

int16 vdcGetNumDims( VDC );

int32 vdcSetDimSizes( VDC, int32* );

int32 *vdcGetDimSizes( VDC );

int32 vdcCurNumElements( VDC );
```

Enum Types

HP VEE enumerated types, as noted, are ordered lists of strings; they are handled by the following routines:

- This function creates an empty VEE_ENUM structure with the given number of string-ordinal pairs. It returns a NULL VDC on error.

  ```
  VDC vdcCreateEnumScalar( int16 numberOfPairs );
  ```

- This function places an enumerated pair in the defined VEE ENUM structure, returns the updated structure, and returns 0 or an error code.

  ```
  int32 vdcEnumAddEnumPair( VDC aVD,
                            char* aString,
                            int32 aValue );
  ```

- This function deletes an enumerated pair as given by the ordinal value argument. It returns 0 or an error code.

```
int32 vdcEnumDeleteEnumPairWithOrdinal( VDC aVD,
                                        int32 anOrd );
```

- This function sets an ordinal value for use by other `vdcEnum` routines. It returns 0 or an error code.

```
int32 vdcSetEnumScalar( VDC aVD,
                        int32 anOrdinal );
```

- This function places a string in the VEE_ENUM structure with the ordinal value assigned by `vdcSetEnumScalar()`.

```
int32 vdcEnumDeleteEnumPairWithStr( VDC aVD,
                                    char* aString );
```

- This function returns the current ordinal number selection assigned by `vdcSetEnumScalar()`.

```
int32 vdcGetEnumOrdinal( VDC aVD );
```

- This function returns the string associated with the current ordinal number, or a NULL string if something goes wrong.

```
char* vdcGetEnumString( VDC aVD );
```

Mapping Functions

The VEE DATA API allows you to manipulate the mappings of arrays with the following functions:

```
int32 vdcAtDimPutLowerLimit( VDC aVD,
                             int16 aDim,
                             double aValue );
     // Specify mapping for lower limit.

int32 vdcAtDimPutUpperLimit( VDC aVD,
                             int16 aDim,
                             double aValue );
     // Specify mapping for upper limit.

int32 vdcAtDimPutRange( VDC aVD,
                        int16 aDim,
                        double lowerLimit,
                        double upperLimit );
     // Combines "vdcAtDimPutLowerLimit" & "vdcAtDimPutUpperLimit".

int32 vdcAtDimPutMapping( VDC aVD,
                          int16 aDim,
                          VMT aMapping);
     // Set the mapping between limits as defined above.

int32 vdcMakeMappingsSame( VDC VD1,
                           VDC VD2 );
     // Map two containers in the same way.

int32 vdcUnMap( VDC aVD );
     // Delete mapping information from container.
```

Other Functions

Other VEE DATA API functions include:

- Get the type of VDC. Return `VEE_NOTDEFINED1` on error.

```
enum veeType vdcType( VDC aVD );
```

- Make a copy of a VDC. Return NULL on error.

```
VDC vdcCopy( VDC oldVD );
```

- Destroy a container and release its memory. Return NULL on error.

```
VDC vdcFree( VDC aVD );
```

- Get error number/message of last error.

```
int16 veeGetErrorNumber( void );
char *veeGetErrorString( void );
```

About ActiveX Controls

HP VEE includes an ActiveX control that encapsulates the functionality of Callable VEE. Instead of a C program calling HP VEE UserFunctions, the UserFunctions are called from OLE-complient applications such as Visual Basic or Microsoft Excel. The control is located at `%SystemRoot%\system32\call.ocx` (Windows NT) or `Windows95\System\call.ocx` (Windows 95).

Use the control to:

- Explore the network domain, looking for the UserFunction to call.
- Get information about the UserFunction's input and output pins.
- Build the VEE Data Container (VDC) needed.
- Call the UserFunction.

Online help is available from the control. It explains the specific use of the control.

5

Using Transaction I/O

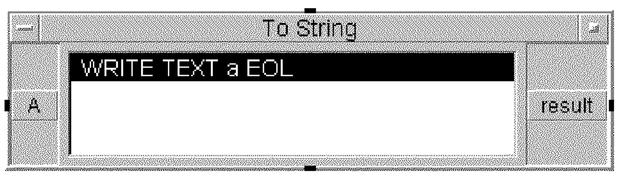
HP VEE for UNIX includes objects for communicating with files, printers, named pipes, and other processes, plus the ability to communicate with HP BASIC, and various hardware interfaces and the instruments connected to them.

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

All of these types of communication are controlled by I/O objects using transactions. This chapter explains the general concepts common to all objects using transactions and the details of how to use each type of object. For information on how to use transactions in instrument I/O, refer to *Controlling Instruments with HP VEE.*

Using Transactions

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



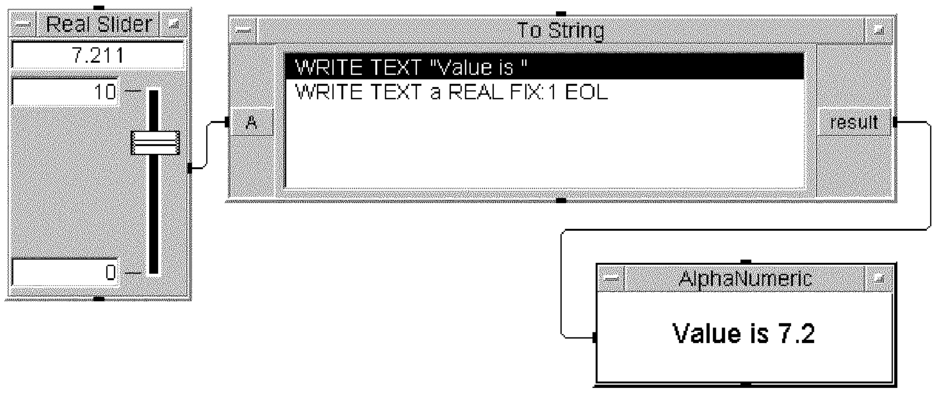
WRITE TEXT a EOL

Figure 5-1. Default Transaction in `To String`

The default transaction in `To String` is:

```
WRITE TEXT a EOL
```

Before exploring too many details, consider a simple program using the `To String` object to illustrate how transactions operate. The program in Figure 5-2 uses two transactions, one to write a string literal and one to write a number in fixed decimal format.



```
WRITE TEXT "Value is "
WRITE TEXT a REAL FIX:1 EOL
```

```
Value is 7.2
```

Figure 5-2. A Simple Program Using `To String`

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

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

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

Creating and Editing Transactions

Table 5-1. Editing Transactions With a Mouse

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

Table 5-2. Editing Transactions With the Keyboard

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

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

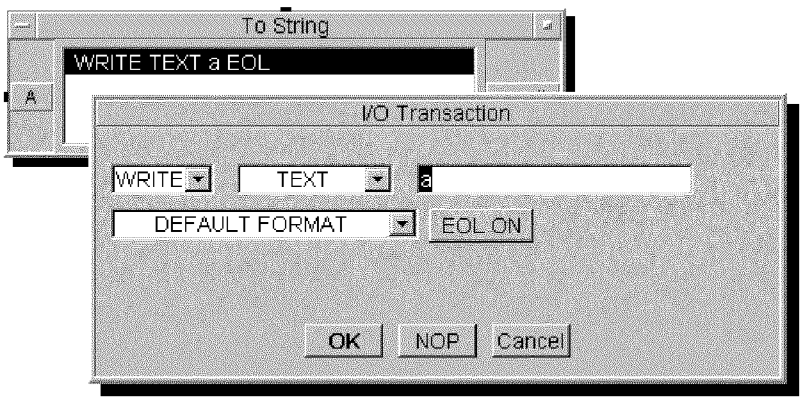


Figure 5-3. Editing the Default Transaction in `To String`

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

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

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

Editing the Data Field

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

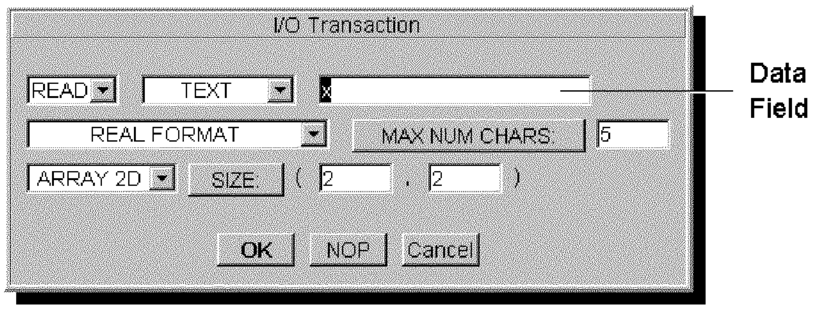


Figure 5-4. `READ` Transaction Using a Variable in the Data Field

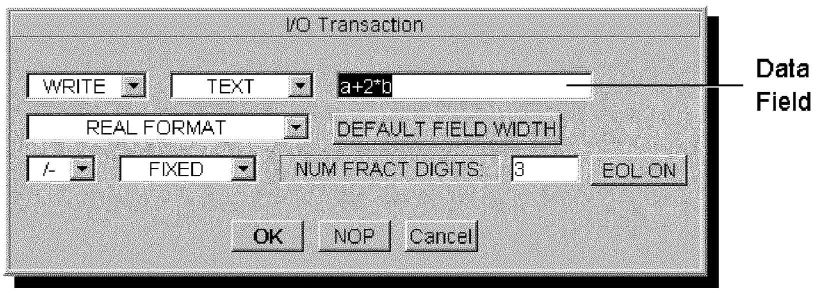


Figure 5-5. `WRITE` Transaction Using an Expression in the Data Field

You must type in the proper list of what you wish to read or write. Table 5-3 lists typical entries for the data field. Note that `WRITE` transactions allow you to specify an expression list (variables, constants, and operators), but `READ` allows only a variable list.

Table 5-3. Typical Data Field Entries

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

The expressions allowed in a `WRITE` data field are the same as those allowed in `Formula` objects. Note that you may include the escape characters shown in Table 5-4 in any field that accepts Text input in the form of a string delimited by double quotes.

Note

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

Table 5-4. Escape Characters

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

Adding Terminals

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

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

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

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

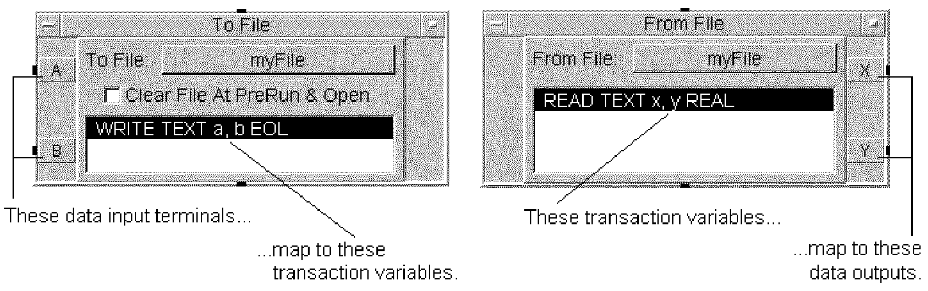

To File
To File: myFile
Clear File At PreRun & Open
WRITE TEXT a, b EOL
A
B
From File
From File: myFile
READ TEXT x, y REAL
X
Y
These data input terminals...
...map to these transaction variables.
These transaction variables...
...map to these data outputs.

Figure 5-6. Terminals Correspond to Variables

To edit the variable name of a terminal:

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

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

Reading Data

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

Transactions that Read a Specified Number of Data Elements

When you are editing a transaction, the last field in the transaction dialog box has the default value SCALAR. This specifies that the READ transaction is to read only one element. To change this, just click on the SCALAR field to reveal a list of available choices.

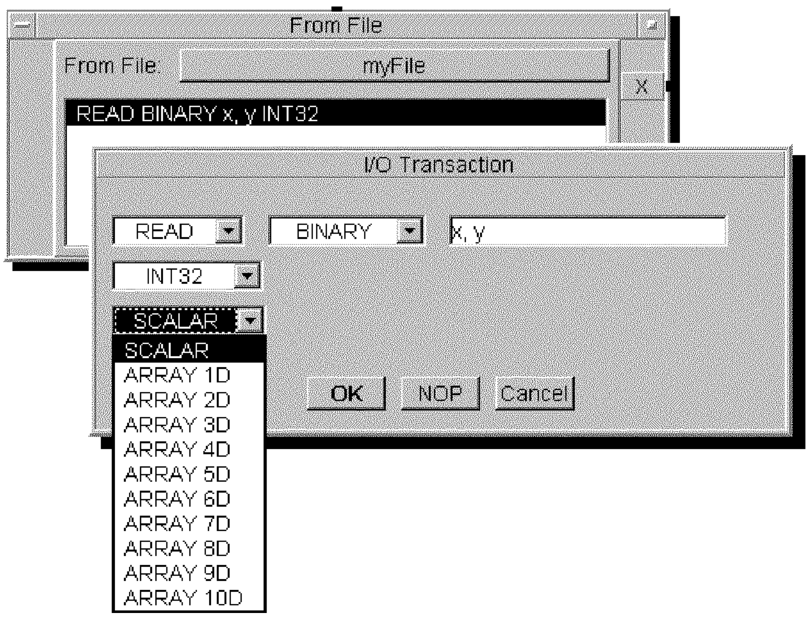
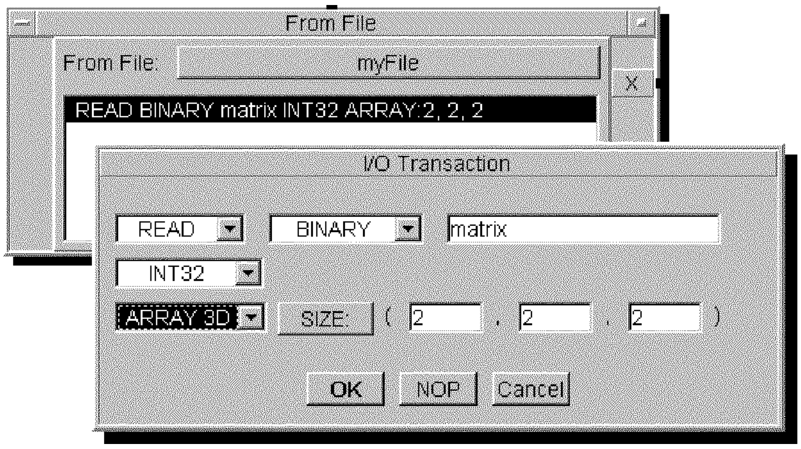


Figure 5-7. Select Read Dimension from List

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

When you click on a dimension in the list, the transaction dialog box will reconfigure itself with a fill-in field for each of the dimensions specified. Figure 5-8 shows the transaction dialog box configured to read a three-dimensional array of binary integers into the variable named `matrix`. Each of the three fields after `SIZE:` contains the number of integers for the corresponding dimension. (In this case, each dimension has two elements.)



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

matrix (2 , 2 , 2)

Figure 5-8. Transaction Dialog Box for Multi-Dimensional Read

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

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

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

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

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

```
READ BINARY x INT32
```

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

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

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

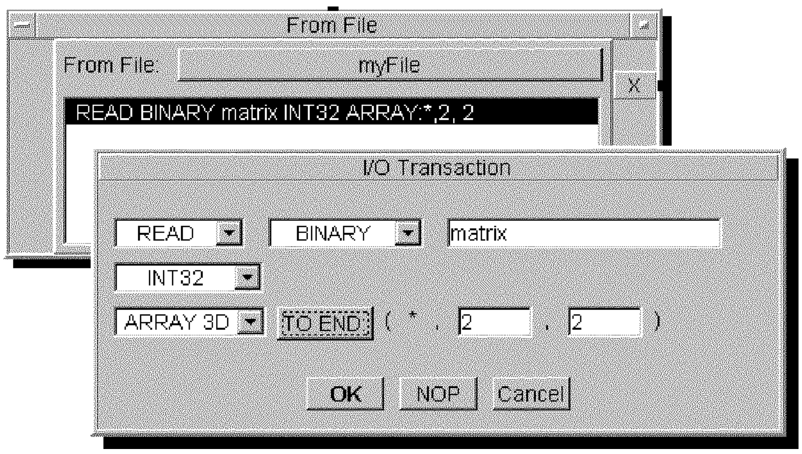
Read-To-End Transactions

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

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

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

Figure 5-9 shows the transaction dialog box of a `From File` object, reading a three dimensional array of binary integers, but configured for read-to-end:



READ BINARY matrix INT32 ARRAY:*,2, 2

Figure 5-9. Transaction Dialog Box for Multi-Dimensional Read-To-End

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

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

On the other hand, reading a multi-dimensional array to end is somewhat more complicated. In this case the number of elements must be supplied for each dimension, except the left-most or "outer" dimension. Figure 5-9 shows that this dimension has an (*) in place of a size in the transaction. This dimension size is unknown until the read-to-end is transaction complete.

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

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

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

Note

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

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

Non-Blocking Reads

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

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

- `To/From Named Pipe (UNIX)`

- `To/From Socket`
- `To/From HP BASIC/UX (UNIX)`
- `From StdIn (UNIX)`

Note

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

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

Figure 5-10 shows a program where `READ IOSTATUS DATAREADY` is used to detect data on the StdIn pipe.

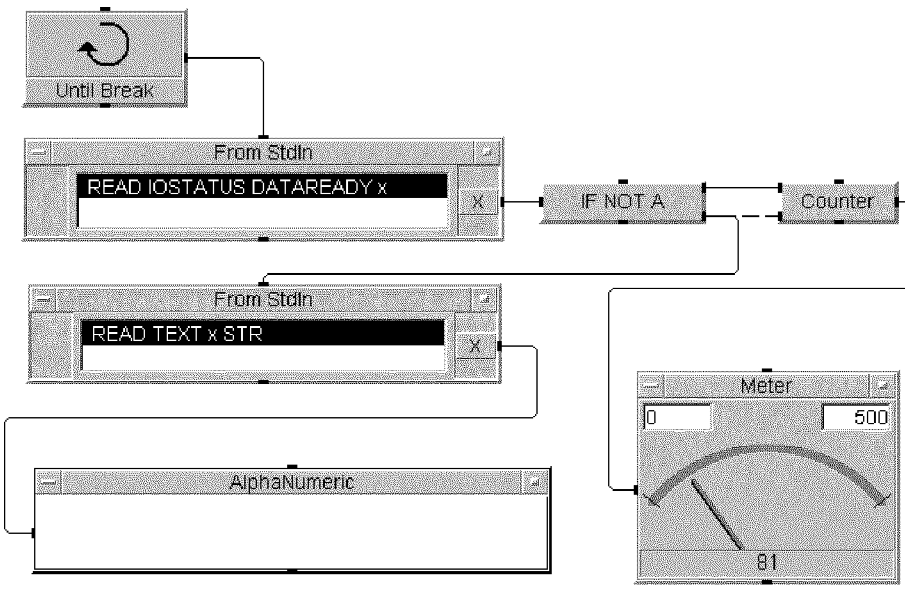


Figure 5-10. Using `READ IOSTATUS DATAREADY` for a Non-Blocking Read

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

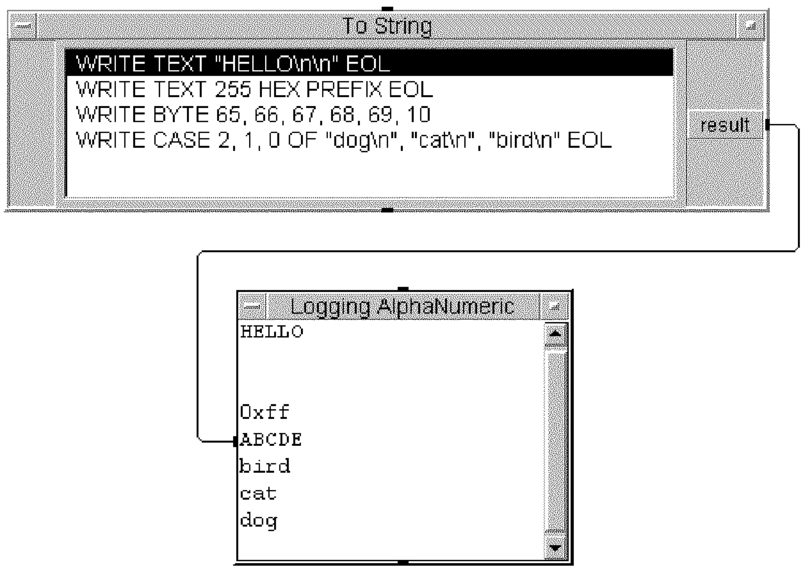
The program in Figure 5-10 shows the use of a READ IOSTATUS DATAREADY transaction in From StdIn. The transaction returns a zero (0) if no data is present on the stdin pipe. If data is present, a one (1) is returned. The If/Then/Else is used to test the returned value of the READ IOSTATUS DATAREADY transaction. If the result is 1, then the second From StdIn is allowed to execute, reading the data typed into the HP VEE start-up terminal window. If no data has been typed into the start-up terminal window (or a Return has not been typed), execution continues again at the start of the thread. Note the use of Until Break to iterate the thread so the From StdIn with the READ IOSTATUS DATAREADY transaction is continually tested.

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

Suggestions for Experimentation

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

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



```
WRITE TEXT "HELLO\n\n" EOL
WRITE TEXT 255 HEX PREFIX EOL
WRITE BYTE 65, 66, 67, 68, 69, 10
WRITE CASE 2, 1, 0 OF "dog\n", "cat\n", "bird\n" EOL
```

```
HELLO


0xff
ABCDE
bird
cat
dog
```

Figure 5-11. Experimenting with `To String`

Details About Transaction-Based Objects

Execution Rules

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

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

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

Object Configuration

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

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

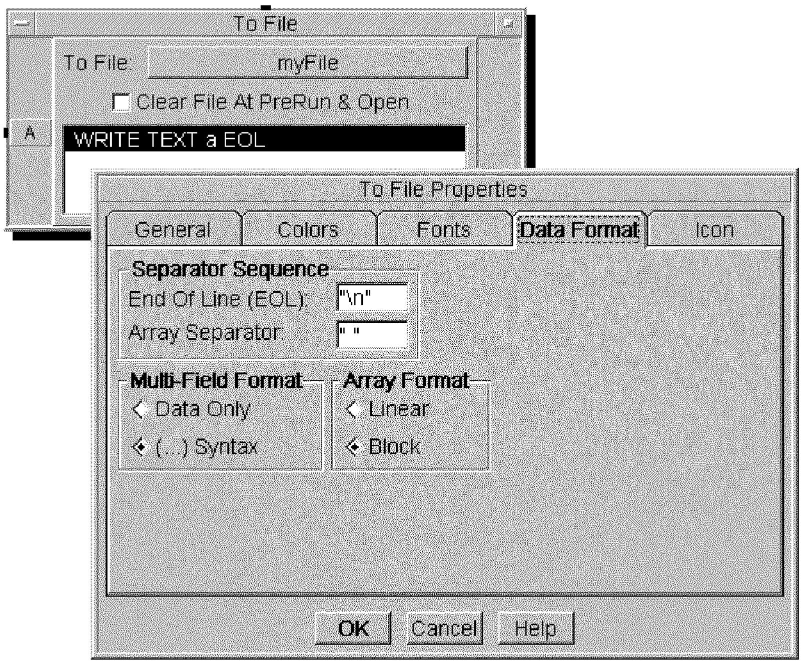
In most cases you do not need to be concerned about the `Properties` settings; the default values are generally suitable.

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

Note

`Direct I/O` objects behave differently than described above. `Direct I/O` objects include a `Show Config` feature in their object menu that allows you to view (but not edit) configuration settings. To edit the configuration of a `Direct I/O` object, you must use `I/O` ⟹ `Instrument Manager`. Refer to *Controlling Instruments with HP VEE* for more information on `Direct I/O`.

Clicking on `Properties` in the object menu of a transaction I/O object yields a `Properties` dialog box like the one in Figure 5-12.



To File: myFile

☐ Clear File At PreRun & Open

WRITE TEXT a EOL

Separator Sequence

End Of Line (EOL): "\n"

Array Separator: " "

Multi-Field Format: Data Only [✓ (...) Syntax]

Array Format: Linear [✓ Block]

Figure 5-12. The `Properties` **Dialog Box**

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

The sections that follow explain the fields in the `Data Format` tab in detail.

End Of Line (EOL)

The `End Of Line (EOL)` field specifies the characters that are sent at the end of `WRITE` transactions that use `EOL ON`. The entry in this field must be zero or more characters surrounded by double quotes. "Double quote" means ASCII 34 decimal. HP VEE recognizes any ASCII characters within `End Of Line (EOL)` including the escape characters shown previously in Table 5-4.

Array Separator

The `Array Separator` field specifies the character string used to separate elements of an array written by `WRITE TEXT` transactions. The entry in this field must be surrounded by double quotes. “Double quote” means ASCII 34 decimal. HP VEE recognizes any ASCII character as an `Array Separator` as well as the escape characters shown previously in Table 5-4.

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

Multi-Field Format

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

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

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

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

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

Array Format

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

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

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

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

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

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

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

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

```
WRITE TEXT A
```

an `End Of Line (EOL)` character will be output each time `x` or `y` changes value.

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

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

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

READ and WRITE Compatibility

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

Choosing the Correct Transaction

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

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

Table 5-5. Summary of Transaction-Based Objects

Object	Description

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

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

`Execute Program (UNIX)`	Spawns an executable file; writes to standard input and reads from standard output of the spawned process. Note that `Execute Program (PC)` is not transaction based.
`To Printer`	Writes text to the HP VEE text printer.

`To StdOut`
`To StdError`
`From StdIn`
	Writes data to HP VEE standard output. (A file on the PC)
	Writes data to HP VEE standard error. (A file on the PC)
	Reads data from HP VEE standard input. (A file on the PC)

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

`To/From Named Pipe (UNIX)`
`To/From HP BASIC/UX (UNIX)`
	Transmits data to and from named pipes to support interprocess communications.
	Transmits data to and from an HP BASIC/UX process via HP-UX named pipes.

`To/From DDE (PC)`	Dynamically exchanges data between programs running under Microsoft Windows.
`To/From Socket`	Uses interprocess communication to exchange data within networked computer systems.

Table 5-6. Summary of Transaction Types

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

Selecting the Correct Object and Transaction

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

For information about using transaction for instrument I/O, refer to *Controlling Instruments with HP VEE.*

Example of Selecting an Object and Transaction

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

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

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

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

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

Using To String and From String

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

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

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

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

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

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

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

Communicating With Files

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

Details About File Pointers

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

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

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

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

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

Read Pointers

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

Write Pointers

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

Note

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

Closing Files

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

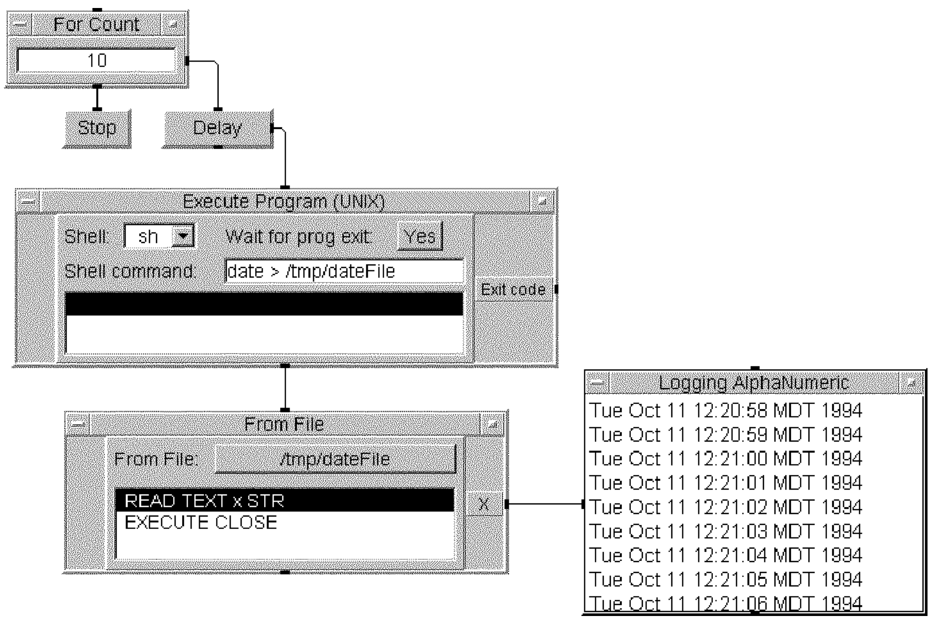
The UNIX operating system physically writes data buffered by the operating system to disk periodically, typically every 15-30 seconds. This buffered operation is part of the operating system; it is not unique to HP VEE.

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

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

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

Figure 5-13 shows an example of how to use `EXECUTE CLOSE`.



For Count: 10

Shell: sh   Wait for prog exit: Yes
Shell command: date > /tmp/dateFile

From File: /tmp/dateFile

```
READ TEXT x STR
EXECUTE CLOSE
```

```
Tue Oct 11 12:20:58 MDT 1994
Tue Oct 11 12:20:59 MDT 1994
Tue Oct 11 12:21:00 MDT 1994
Tue Oct 11 12:21:01 MDT 1994
Tue Oct 11 12:21:02 MDT 1994
Tue Oct 11 12:21:03 MDT 1994
Tue Oct 11 12:21:04 MDT 1994
Tue Oct 11 12:21:05 MDT 1994
Tue Oct 11 12:21:06 MDT 1994
```

Figure 5-13. Using the `EXECUTE CLOSE` Transaction

This program is saved in the file `manual48.vee` in your `examples` directory.

In Figure 5-13 `Execute Program` executes a shell command (`date`) that creates and writes the date and time to a file (`/tmp/dateFile`). Within the same thread, a `From File` reads the date from that file using a `READ TEXT x STR` transaction. The EXECUTE CLOSE transaction is necessary because the subthread is executed multiple times by For Count. Succeeding executions of Execute Program will overwrite the file. However, since From File only opens the file once, upon the second execution of From File the read pointer will be *stale* — it will no longer point to the file since Execute Program has *re-created* the file. An error will occur.

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

In the example of Figure 5-13, the error can be shown by using a NOP to "comment out" the EXECUTE CLOSE transaction. The error will state End of file or no data found. Removing the NOP will allow the program to run normally.

The EOF Data Output

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

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

Figure 5-14 illustrates a typical use of `EOF`. The file being read contains a list of `X-Y` data of unknown length. Here are typical contents of the file:

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

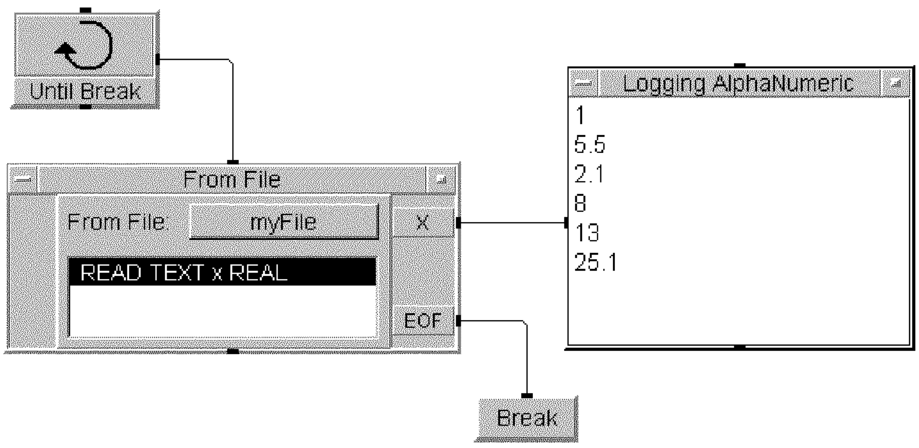


Figure 5-14. Typical Use of EOF to Read a File

Common Tasks for Importing Data

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

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

Importing X-Y Values

One very common problem is reading a text file containing an unknown number of X and Y values and plotting them. The program shown in Figure 5-15 solves this problem.

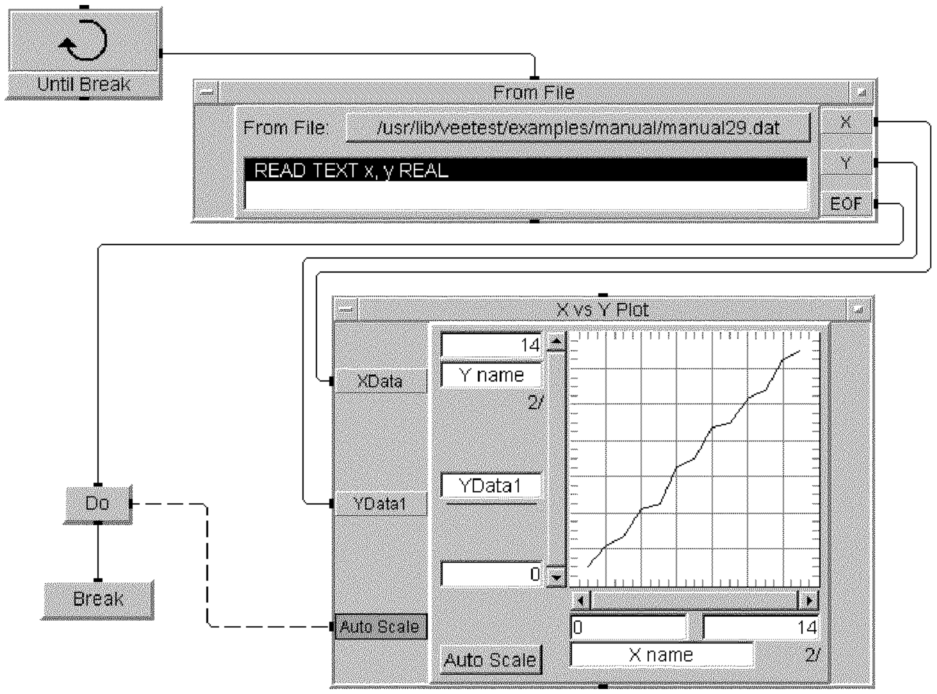


Figure 5-15. Importing XY Values

The program shown in Figure 5-15 is saved in the file `manual29.vee` in your `examples` directory.

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

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

Importing Waveforms

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

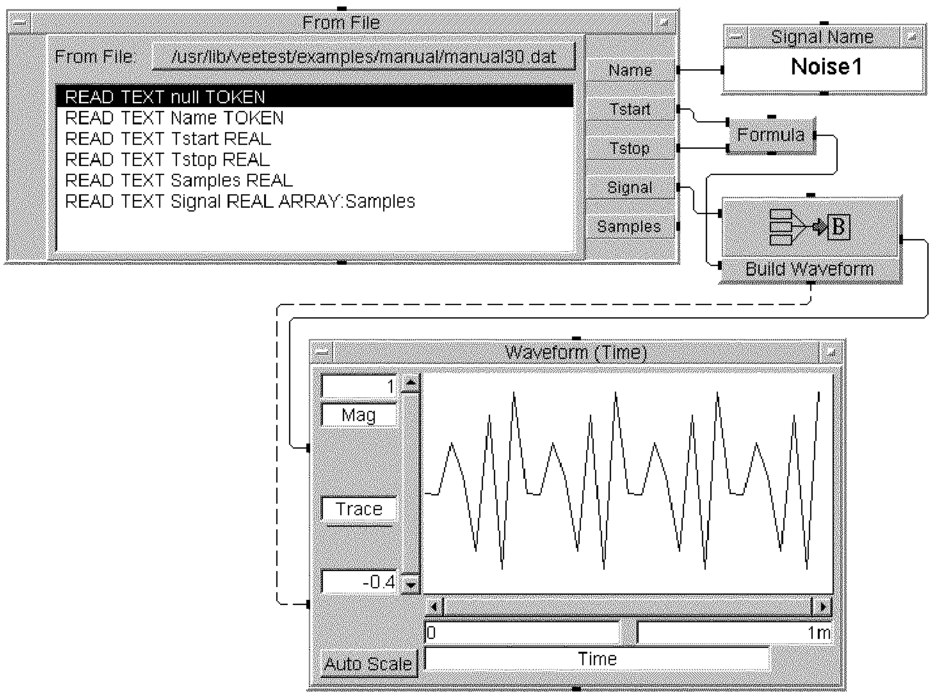
- Fixed-format file header. Waveform attributes are listed in fixed positions at the beginning of the file followed by a one-dimensional array of `Y` data.
- Variable-format file header. A variable number of attributes are listed at the beginning of the file followed by a one-dimensional array of `Y`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

Fixed-Format Header. Here is a portion of the data file read by the program in Figure 5-16:

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

              .243545
            .2345776
.
.
.
```

Since this is a fixed-format header, labels such as NAME and SAMPLES are irrelevant. The waveform attributes *always appear and are in the same position.* Figure 5-16 shows a program that reads the waveform data file.



```
READ TEXT null TOKEN
READ TEXT Name TOKEN
READ TEXT Tstart REAL
READ TEXT Tstop REAL
READ TEXT Samples REAL
READ TEXT Signal REAL ARRAY:Samples
```

Figure 5-16. Importing a Waveform File

The program shown in Figure 5-16 is saved in the file `manual30.vee` in your `examples` directory.

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

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

Variable-Format Header. Here is a portion of the data file read by the program in Figure 5-17:

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

<DATA>

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

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

To simplify this example, the program in Figure 5-17 finds only the data associated with `<DATA>`. In your own applications, you might need to search for several markers.

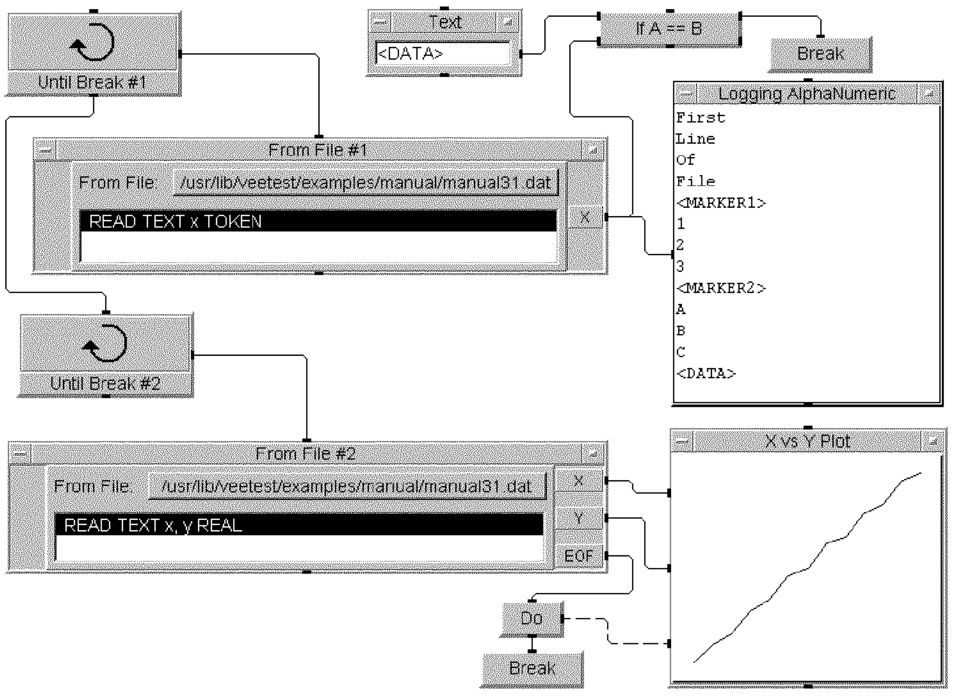


Figure 5-17. Importing a Waveform File

The program shown in Figure 5-17 is saved in the file `manual31.vee` in your `examples` directory.

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

Communicating With Programs (UNIX)

Program	Object(s)
Shell command	`Execute Program (UNIX)`

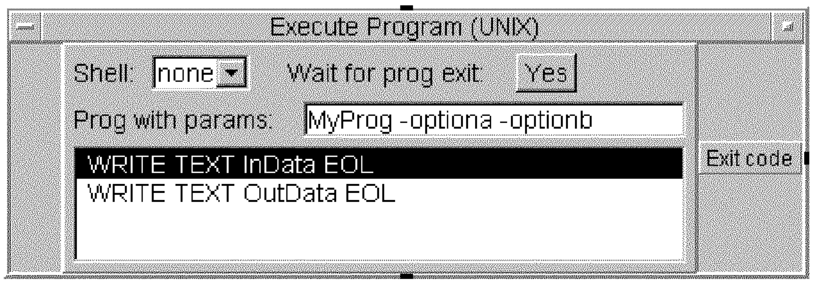
C program	`Execute Program (UNIX)`
	`To/From Named Pipe (UNIX)`
	`To/From Socket`

HP BASIC/UX	`Init HP BASIC/UX (UNIX)`
	`To/From HP BASIC/UX (UNIX)`

Execute Program (UNIX)

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

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



Shell: none Wait for prog exit: Yes
Prog with params: MyProg -optiona -optionb

```
WRITE TEXT InData EOL
WRITE TEXT OutData EOL
```

Figure 5-18. The `Execute Program (UNIX)` **Object**

Execute Program (UNIX) Fields

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

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

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

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

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

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

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

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

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

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

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

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

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

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

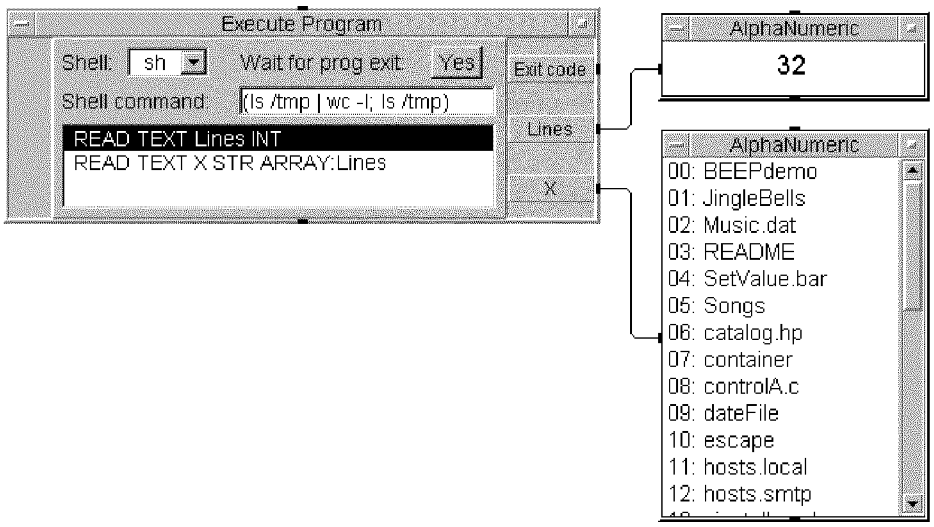
```
MyProg -optionA -optionB
```

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

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

Running a Shell Command

Execute Program (UNIX) can be used to run shell commands such as ls, mkdir, and rm. Figure 5-19 shows one method for obtaining a list of files in a directory using an HP VEE program.



Shell command: (ls /tmp | wc -l; ls /tmp)

```
READ TEXT Lines INT
READ TEXT X STR ARRAY:Lines
```

```
32
```

```
00: BEEPdemo
01: JingleBells
02: Music.dat
03: README
04: SetValue.bar
05: Songs
06: catalog.hp
07: container
08: controlA.c
09: dateFile
10: escape
11: hosts.local
12: hosts.smtp
```

Figure 5-19. Execute Program (UNIX) **Running a Shell Command**

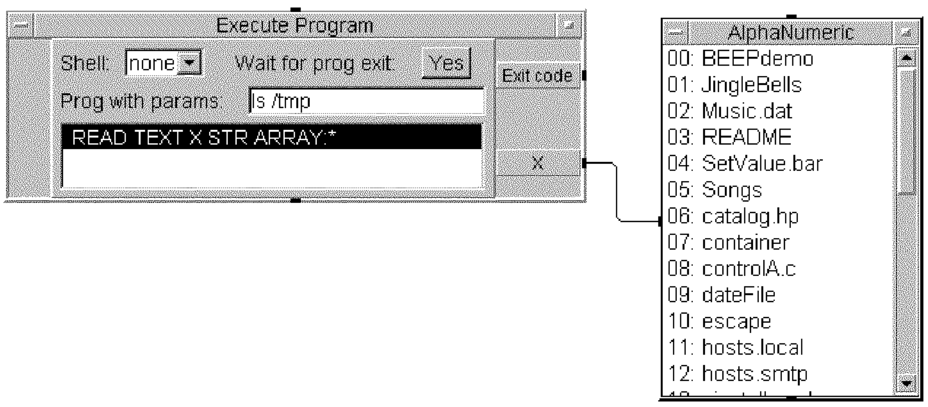
The program shown in Figure 5-19 is saved in the file manual32.vee in your examples directory.

In Figure 5-19, the Execute Program (UNIX) determines the number of file names in the /tmp directory by listing the names in a single column (ls -1) and piping this list to a line counting program (wc -l). Because the pipe is used, the command contained in the Prog with params field must be sent to a shell for interpretation. Thus, the Shell field is set to sh. The number of lines is read by the READ TEXT transaction and passed to the output terminal named Lines.

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

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

Figure 5-20 shows another method for obtaining a list of files in a directory using an HP VEE program.



```
READ TEXT X STR ARRAY:*
```

```
00: BEEPdemo
01: JingleBells
02: Music.dat
03: README
04: SetValue.bar
05: Songs
06: catalog.hp
07: container
08: controlA.c
09: dateFile
10: escape
11: hosts.local
12: hosts.smtp
```

Figure 5-20.
Execute Program (UNIX) **Running a Shell Command using Read-To-End**

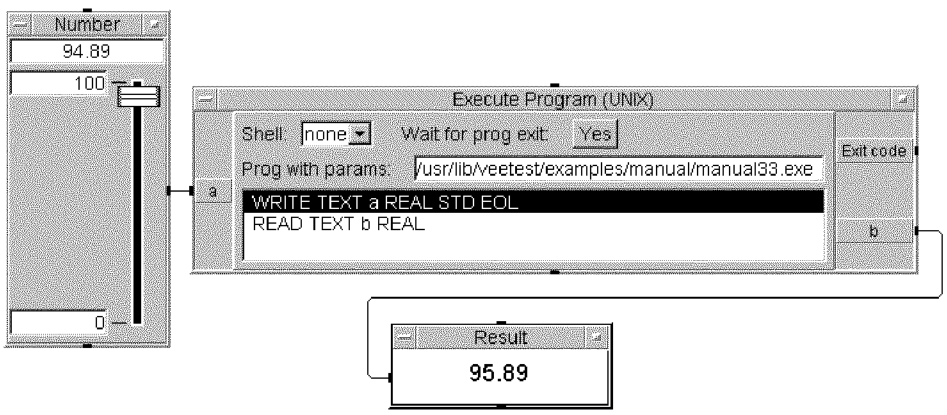
This program is saved in the file manual50.vee in your examples directory.

In Figure 5-20 the HP VEE program displays the contents of the /tmp directory in a simpler fashion than in Figure 5-19.

In Figure 5-20, Execute Program (UNIX) has in the Prog with params field the single shell command ls /tmp. There is no need to first obtain the number of files in the directory, as was done in the program in Figure 5-19, because the transaction READ TEXT x STR ARRAY:* uses the read-to-end feature discussed in "Reading Data". The shell command, when it is done executing, will close the pipe that Execute Program (UNIX) is using to read the list of files. This sends an end-of-file (EOF) which terminates the transaction.

Running a C Program

The program shown in Figure 5-21 illustrates one way to share data with a C program using `stdin` and `stdout` of the C program. In this case, the C program simply reads a real number from HP VEE, adds one to the number, and returns the incremented value.



```
WRITE TEXT a REAL STD EOL
READ TEXT b REAL
```

Figure 5-21. `Execute Program` **Running a C Program**

The program shown in Figure 5-21 is saved in the file `manual33.vee` in your `examples` directory.

Figure 5-22 contains a listing of the C program called by the HP VEE program in Figure 5-21.

The program listing in Figure 5-22 uses both `setbuf` and `fflush` to force data through `stdout` of the C program; in practice, either `setbuf` or `fflush` is sufficient. Using `setbuf(`*file*`,NULL)` turns off buffering for all output to *file*. Using `fflush(`*file*`)` flushes any already buffered data to *file*.

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

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

Figure 5-22. C Program Listing

To/From Named Pipe (UNIX)

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

All `To/From Named Pipe` objects contain the same default names for read and write pipes. Be certain that you correctly specify the names of the pipes you want to read or write. This can be a problem if you run HP VEE on a diskless workstation. You must be sure that the named pipes in your program are not being accessed by another user.

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

Hints for Using Named Pipes

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

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

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

or

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

or

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

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

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

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

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

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

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

Conversely, if HP VEE is the reading process, a `READ` transaction using the read-to-end feature will allow HP VEE to read an unknown amount of data from the named-pipe *if* the writing process performs a `close()` on the pipe, sending an `EOF`. Another way to avoid a read that will block indefinitely is to use the `READ IOSTATUS` transaction. See Appendix D for more information about using `READ IOSTATUS` transactions.

To/From Socket

The `To/From Socket` object is for *advanced users* who wish to implement interprocess communication for systems integration. Using sockets is not a task for casual users; sockets have some complex behaviors.

Sockets let you implement interprocess communication (IPC) to allow programs to treat the LAN as a file descriptor. IPC implies that there are two sockets involved between two or more processes on two different computers. Instead of a simple open()/close() interface as used in the `To/From Named Pipe` object, sockets use an exported address and an initial caller/receiver strategy, referred to as a connection-oriented protocol.

In a connection-oriented protocol, also known as a client/server arrangement, the server must obtain a socket, then *bind* an address known as the port number to the socket. After binding a port number, the server waits in a blocked state to accept a *connection* request. To call for a connection, the client must obtain a socket, then use two elements of the server's identity. The elements include the particular port number the server bound to its socket, and the server's host name or IP address. If the server's host name cannot be resolved into an IP address, the client *must* use the IP address specifically. After the server accepts the client's connection request, the connection is established and normal I/O activities can begin.

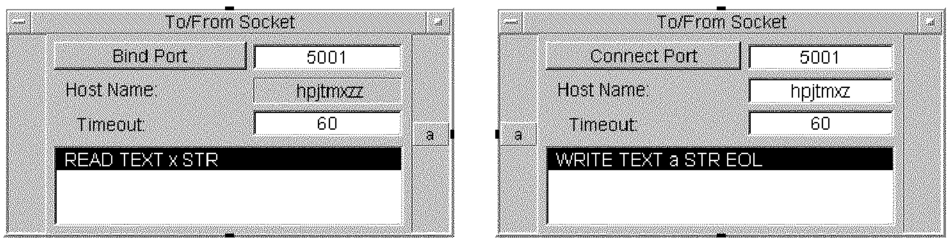


Figure 5-23. The `To/From Socket` **Object**

To/From Socket Fields

The `To/From Socket` object contains fields that let you do the following:

- Connect to a bound socket on a remote computer.
- Bind a socket on the computer on which HP VEE is running and wait for a connection to occur.

Of the four available fields, values of the following three fields can be input as control pins to the object:

- Connect/Bind Port Mode
- Host name
- Timeout

The following sections explain the fields visible in the `To/From Socket` open view.

Connect/Bind Port Mode. Connect/Bind Port Mode comprises two fields, the mode button and the text field. The mode button toggles between `Bind Port` and `Connect Port`. The text field lets you enter the port number. Allowed port numbers are integers from 1024 through 65535. Numbers from 0 through 1023 are reserved and will cause a run-time error if you use them. Port numbers above 5000 are commonly called transient, and are the range of numbers you should use.

Table 5-7.
Range of Integers Allowed for Socket Port Numbers

Number Range	Reserved for ...
0—1023	operating system
1024—5000	commercial or global application[1]
5001—65535	internal or closed distributed applications

1 Usually involves a registration process.

Host Name. If the mode is set to `Bind Port`, this field displays the name of host computer on which HP VEE is running. You cannot change this field to the host name of a remote computer, because it is not possible to bind a port number to a socket on a remote computer.

If the mode is set to `Connect Port`, you are allowed to edit this field. Enter the host name or IP address of the remote computer to which you want to connect. You must know the host name and it must be resolvable to the IP address. If a host name table is not available on the network to translate the host name to an IP address, you must enter the specific address, such as 15.11.29.103.

Timeout. `Timeout` lets you enter an integer value that represents the timeout period in seconds for all `READ` and `WRITE` transactions. This timeout period is also in effect for the initial connection when the `To/From Socket` object is set either in the `Bind Port` mode waiting for a connection to occur, or in the `Connect Port` mode waiting for a connection to be accepted.

Transactions. The `To/From Socket` object uses the same normal I/O transactions used by the `To/From Named Pipe` object. READ and WRITE transactions support all data types. See Appendix D for detailed information about transactions.

Data Organization

All binary data is placed on the LAN in network-byte order. This corresponds to Most Significant Byte (MSB) or Big Endian ordering. Binary transactions will swap bytes on READs and WRITEs, if necessary. This implies that any other process that HP VEE is connected to will need to conform to this standard. In the previous example, the server process could have been little endian ordered while the client could be big endian ordered. The byte swapping done by HP VEE is invisible.

Object Execution

A To/From Socket object set to bind a socket at a port number will use the timeout period waiting for a connection to occur. All concurrent threads in HP VEE will not execute during this period. The timeout value can be set to zero which disables timeouts, potentially making the period waiting for a connection infinitely long. Any timeout violation causes an error, and halts HP VEE execution.

Once a connection has been established the devices perform the transactions contained in the transaction list. All READ operations will block for the timeout period waiting for the amount and type of data specified in the transaction. To avoid potential blocked threads, use the READ IOSTATUS transaction to detect when data is available on the socket.

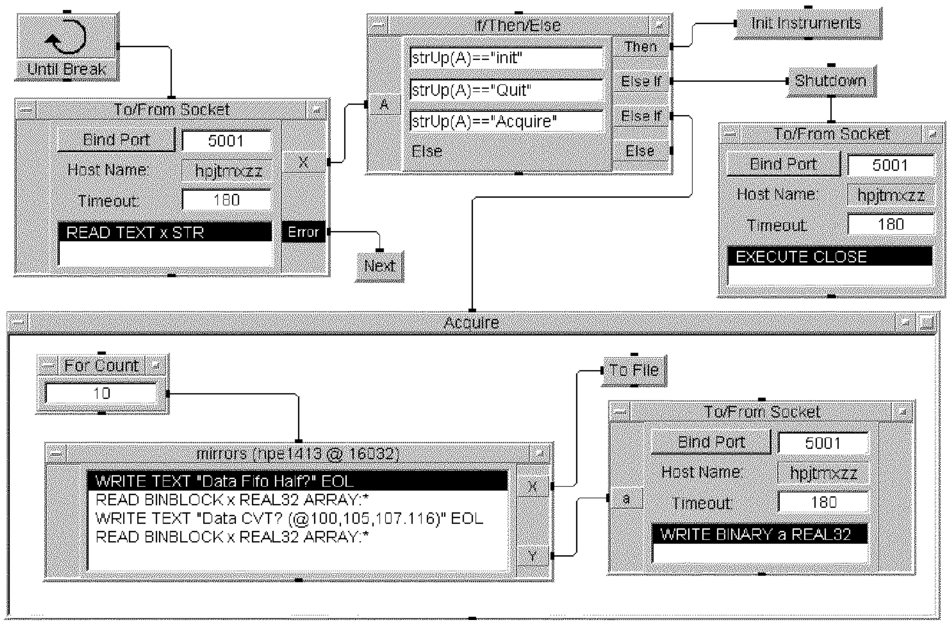
To specifically terminate a connection, use the EXECUTE CLOSE transaction. All socket connections established in a HP VEE program are broken when a program stops executing. Whichever way connections are broken, the server and client objects must repeat the bind-accept and connect-to protocols to re-establish connections. EXECUTE CLOSE should be used as a mutually agreed-upon termination method, and not merely an expedient way to flush data from a socket.

Multiple To/From Socket objects will share sockets. All objects that are binding an identical port number will share the same socket. All objects that are configured with identical port numbers and host names to attempt connection to the same bound socket will share the same socket. The overhead of establishing the connection is incurred in the first execution of one of the commonly configured objects.

Example

The following figure shows a HP VEE program which uses the To/From Socket object to provide a separate server process for data acquisition using the HPE 1413B. This simple server can honor client requests to initialize instruments, acquire and write data to disk, and shutdown and quit. During the acquisition phase data is read from the Current Value Table in the A/D and sent to the client.

The first `To/From Socket` object to execute, connected to the `Until Break` object, will bind a socket to port number 5001 on the host computer named `hpjtmxzz` and wait 180 seconds for another process to connect to that socket. Note the use of an error pin to avoid a halt due to a timeout. In this case that object is just executed again and will wait another 180 seconds for a connection. After the connection has been made, the object will then block on the `READ` transaction waiting for the client to send a command. Again, if a timeout occurs on the `READ`, the object will execute again and block on the `READ` transaction.



```
READ TEXT x STR
```

```
strUp(A)=="init"
strUp(A)=="Quit"
strUp(A)=="Acquire"
Else
```

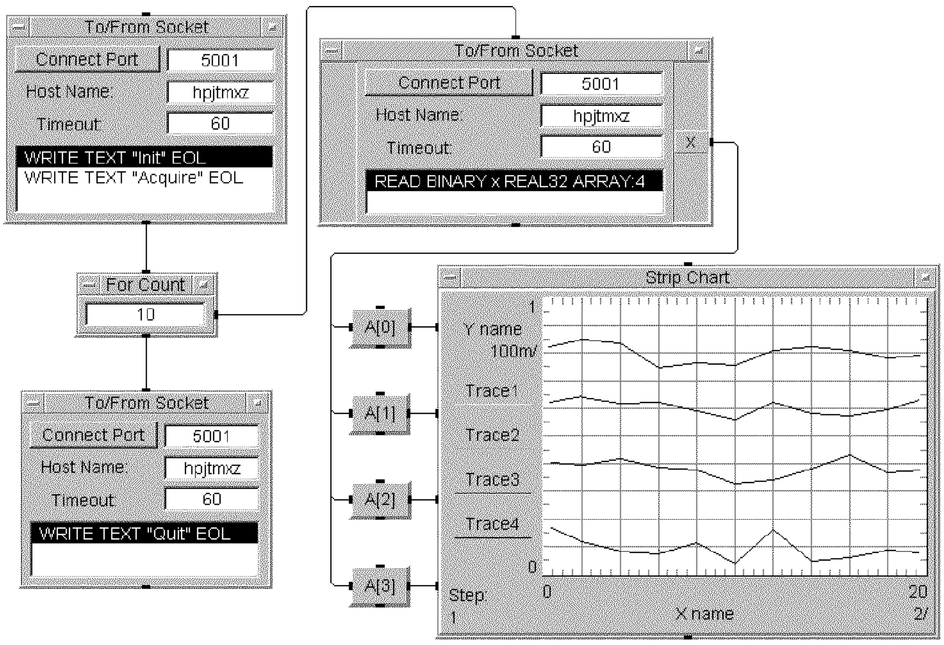
```
EXECUTE CLOSE
```

```
WRITE TEXT "Data Fifo Half?" EOL
READ BINBLOCK x REAL32 ARRAY:*
WRITE TEXT "Data CVT? (@100,105,107.116)" EOL
READ BINBLOCK x REAL32 ARRAY:*
```

```
WRITE BINARY a REAL32
```

Figure 5-24. `To/From Socket` **Binding Port for Server Process**

The following figure shows the client side of the service described previously. The first `To/From Socket` object to execute will wait, sleeping, for the attempted connection to occur. Note that unlike the server, any timeout error will cause the program to error and halt. The first object sends over the commands `Init` and `Acquire` then executes the loop to read the `CVT`.



```
WRITE TEXT "Init" EOL
WRITE TEXT "Acquire" EOL
```

```
READ BINARY x REAL32 ARRAY:4
```

```
WRITE TEXT "Quit" EOL
```

Figure 5-25. `To/From Socket` **Connecting Port for Client Process**

HP BASIC/UX Objects (HP-UX)

The `Init HP BASIC/UX` and `To/From HP BASIC/UX` objects are available in all versions of HP VEE, and work only in programs that run on HP 9000 Series 700 systems.

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

Init HP BASIC/UX

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

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

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

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

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

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

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

To/From HP BASIC/UX

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

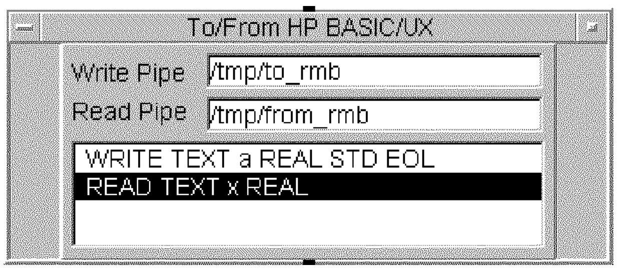
Type in the names of the pipes you wish to use in the `Read Pipe` and `Write Pipe` fields. Be certain that they match the names of the pipes used by your HP BASIC/UX program and that the read and write names are not inadvertently swapped relative to their use in the HP BASIC program. Use different pipes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

Examples Using To/From HP BASIC/UX

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

1. Start HP BASIC.
2. Run a specific HP BASIC program.
3. Send a single number to HP BASIC for analysis.
4. Retrieve the analyzed data.
5. Terminate HP BASIC.

Here are typical `To/From HP BASIC/UX` settings and the corresponding HP BASIC/UX program:



Write Pipe /tmp/to_rmb
Read Pipe /tmp/from_rmb

```
WRITE TEXT a REAL STD EOL
READ TEXT x REAL
```

Figure 5-26. `To/From HP BASIC/UX` **Settings**

Here is the HP BASIC/UX program:

```
100  ASSIGN @From_vee TO "/tmp/to_rmb"
110  ASSIGN @To_vee TO "/tmp/from_rmb"
120  ! Your analysis code here
130  ENTER  @From_vee;Vee_data
140  OUTPUT @To_vee;Rmb_data
150  END
```

To view an example program that solves this problem, open this the `manual34.vee` example.

Sharing Array Data. To share array data between HP VEE and HP BASIC using `TEXT` encoding, you must modify the default `Array Separator` in `To/From HP BASIC/UX`. To do this, click on `Properties` in the `To/From HP BASIC/UX` object menu and click on the `Data Format` tab in the `Properties` dialog box. Set the `Array Separator` field to `", "` (a comma followed by a blank).

Be sure that HP VEE and HP BASIC use the same size arrays.

Note that the order in which HP VEE and HP BASIC read and write array elements is compatible. If HP VEE and HP BASIC share an array using `READ` and `WRITE` transactions in `To/From HP BASIC/UX`, each element will have the same value in HP VEE as in HP BASIC.

To view an example program that shares arrays between HP VEE and HP BASIC, open the `manual35.vee` example.

Sharing Binary Data. It is possible to share numeric data between HP VEE and HP BASIC without converting the numbers to text. To do this, you must select `BINARY` encoding in the `To/From HP BASIC/UX` transactions and `FORMAT OFF` for the `ASSIGN` statements that reference the named pipes in HP BASIC.

There are only two cases where it is possible to share numeric data in binary form:

- HP VEE `BINARY REAL` is equivalent to HP BASIC `REAL`
- HP VEE `BINARY INT16` is equivalent to HP BASIC `INTEGER`

Communicating With Programs (PC)

Program	Object(s)
MS-DOS command	`Execute Program (PC)`

Windows Application	`Execute Program (PC)`
	`To/From DDE (PC)`
	`To/From Socket`

C program	`Execute Program (PC)`
	`Import Library`
	`Call Function`
	`Formula`

Execute Program (PC)

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

- Compiled C programs
- Any MS-DOS program (`*.EXE` or `*.COM` files)
- `.BAT` files
- MS-DOS system commands, such as `dir`

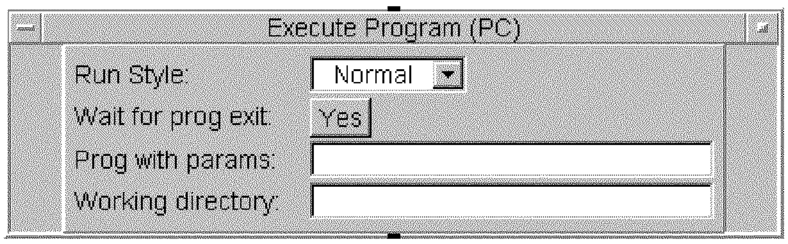

Execute Program (PC)
Run Style: Normal
Wait for prog exit: Yes
Prog with params:
Working directory:

Figure 5-27. The `Execute Program (PC)` **Object**

Execute Program (PC) Fields

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

Run Style. If the program you want to execute runs in a window, `Run Style` specifies the window style:

- `Normal` runs the program in a standard window.
- `Minimized` runs the program in a window minimized to an icon.
- `Maximized` runs the program in a window enlarged to its maximum size.

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

1. Execute the command specified in the `Execute Program (PC)` object.
2. Wait until the process terminates before activating any output pins of the `Execute Program (PC)` object.

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

1. Execute the command specified in the `Execute Program (PC)` object.
2. Activate any data output pins on the `Execute Program (PC)` object.

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

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

1. The name of an executable file and command line parameters.
2. A command that will be sent to MS-DOS for interpretation.

If you have included the appropriate path in the `PATH` variable in your `AUTOEXEC.BAT` file, you don't need to include the path in the `Prog with params` field. Here are examples of what you typically type into the `Prog with params` field:

To execute a MS-DOS command:

```
COMMAND.COM /C DIR *.DAT
```

To run a compiled C program:

```
MyProg -optionA -optionB
```

Working Directory. `Working directory` points to a directory where the program you want to execute can find files it needs. So, if you want to run the program `nmake` using the makefile in the directory `c:\progs\cprog1`:

In `Prog with params:`, enter `nmake`.

In `Working directory:`, enter `c:\progs\cprog1`.

Using Dynamic Data Exchange (DDE)

Dynamic Data Exchange (DDE) defines a message-based protocol for communication between Windows applications. This communication takes place between a DDE client and a DDE server. The DDE client requests the conversation with the DDE server. The client then requests data and services from the server application. The server responds by sending data or executing procedures.

A Windows application that supports DDE may act as either a client, a server or both. HP VEE for Windows provides only client capabilities. It implements DDE capabilities with the `To/From DDE` object.

The HP VEE for Windows `To/From DDE` object uses four types of transactions:

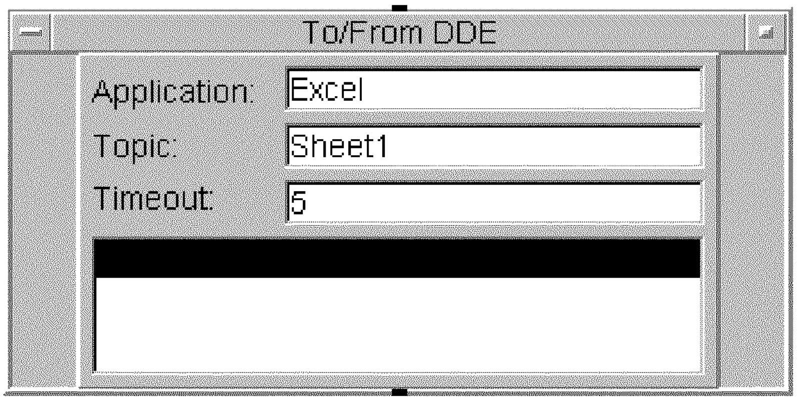
`READ(REQUEST)`	Reads Data from a DDE transfer.
`WRITE(POKE)`	Writes (pokes) Data to a DDE transfer.
`EXECUTE`	Sends a command to the DDE server that HP VEE for Windows is communicating with. The server then executes the command.
`WAIT`	Waits for the specified amount of time (in seconds).

Note that the `To/From DDE` object initiates and terminates DDE operations as part of its function. You do not need to explicitly perform the initiate and terminate functions.

Definitions

- Application - The DDE name for the application.
- Topic - An application-specific identifier of the kind of data. For example, a word processor's topic would be the document name.
- Item - An application-specific identifier for each piece of data. For example, a spreadsheet data item might be a cell location; a word processor data item might be a bookmark name.

To/From DDE Object



Application: Excel
Topic: Sheet1
Timeout: 5

Figure 5-28. The `To/From DDE` **Object**

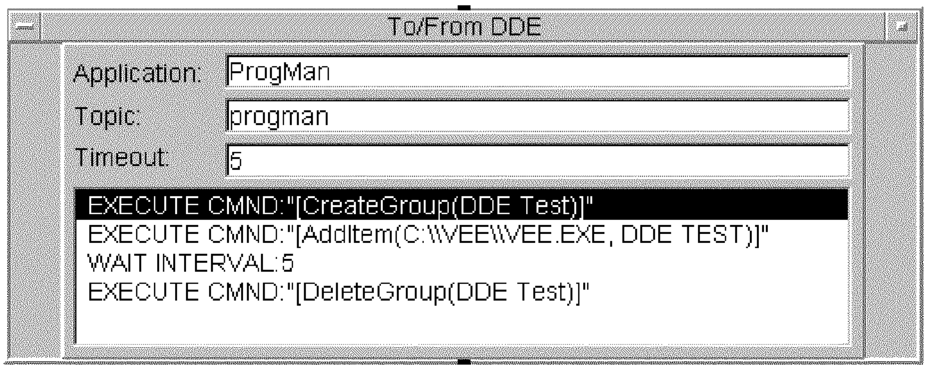
The `To/From DDE` object has three main fields. In the `Application` field enter the application name for the Windows application that you want to communicate with. Generally, this is the `.EXE` file name. See the manual for each specific application to determine its DDE application name.

The `Topic` field contains the Topic name for the application.

The `Timeout` field lets you specify the timeout period for HP VEE to wait if the application does not respond. The default value is five seconds.

The last field contains transactions to communicate with the other application. For `READ(REQUEST)` and `WRITE(POKE)` transactions, you must also fill in an `Item` name in the transaction.

For example, the following `To/From DDE` object, communicating with the MS Windows Program Manager, creates a program group, adds an item to the group, displays it for 5 seconds and then deletes the program group.


To/From DDE

Application: ProgMan

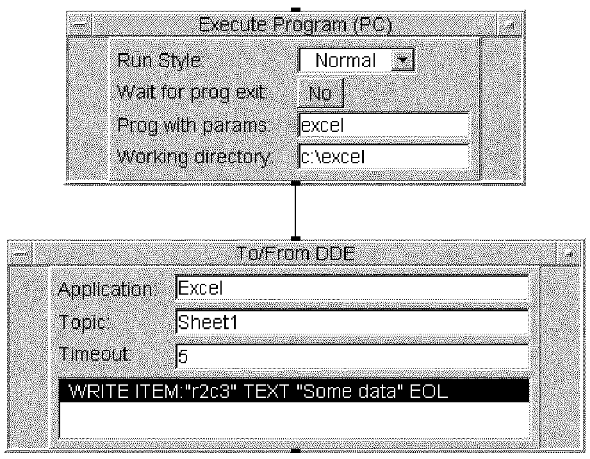
Topic: progman

Timeout: 5

```
EXECUTE CMND:"[CreateGroup(DDE Test)]"
EXECUTE CMND:"[AddItem(C:\\VEE\\VEE.EXE, DDE TEST)]"
WAIT INTERVAL:5
EXECUTE CMND:"[DeleteGroup(DDE Test)]"
```

Figure 5-29. The `To/From DDE` **Example**

Note that if the server DDE application is not currently running, HP VEE will attempt to start that application. This will only be successful if the application's executable file name is the same as the name in the application field. The executable file's directory must also be defined in your `PATH`. HP VEE will try to start the application for the amount of time entered in the `Timeout` field. Otherwise, use an `Execute Program (PC)` object before the `To/From DDE` object to run the application program, as illustrated in the following example.



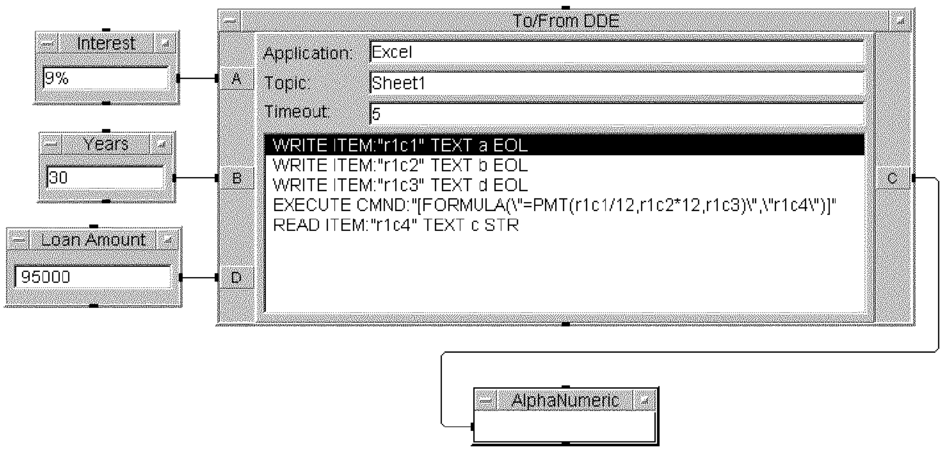
Run Style: Normal
Wait for prog exit: No
Prog with params: excel
Working directory: c:\excel

Application: Excel
Topic: Sheet1
Timeout: 5

```
WRITE ITEM:"r2c3" TEXT "Some data" EOL
```

Figure 5-30. Execute PC before To/From DDE

The following example shows the use of input and output terminals with a `To/From DDE` object.



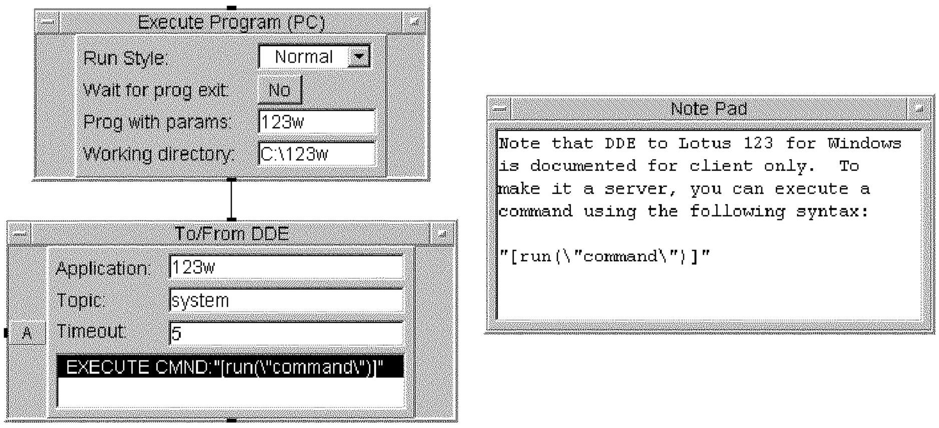
Application: Excel
Topic: Sheet1
Timeout: 5

```
WRITE ITEM:"r1c1" TEXT a EOL
WRITE ITEM:"r1c2" TEXT b EOL
WRITE ITEM:"r1c3" TEXT d EOL
EXECUTE CMND:"[FORMULA(\"=PMT(r1c1/12,r1c2*12,r1c3)\",\"r1c4\")]"
READ ITEM:"r1c4" TEXT c STR
```

Figure 5-31. I/O Terminals and To/From DDE

DDE Examples

The following figures are examples of how to communicate with various popular Windows software. Read the `Note Pad` in each example for important information regarding each example.



Execute Program (PC)

Run Style: Normal

Wait for prog exit: No

Prog with params: 123w

Working directory: C:\123w

To/From DDE

Application: 123w

Topic: system

Timeout: 5

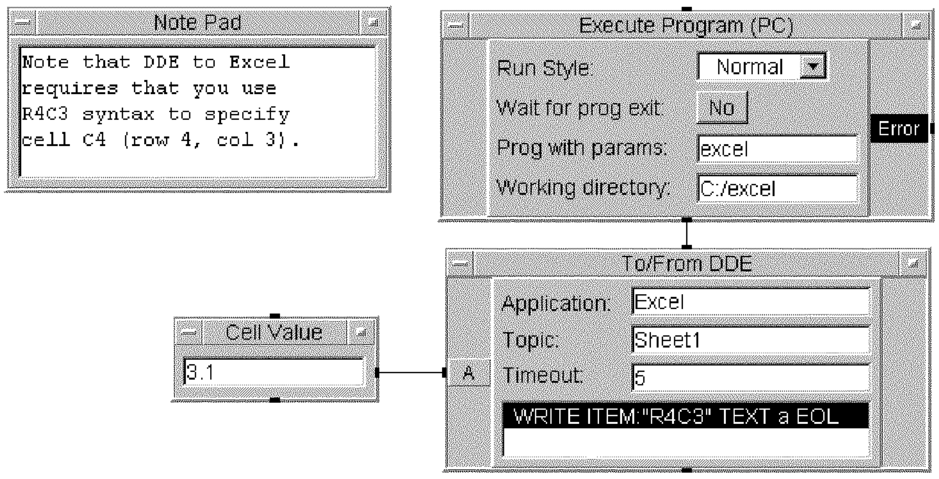
```
EXECUTE CMND:"[run(\"command\")]"
```

Note Pad

```
Note that DDE to Lotus 123 for Windows
is documented for client only.  To
make it a server, you can execute a
command using the following syntax:

"[run(\"command\")]"
```

Figure 5-32. Lotus 123 DDE Example



Note Pad

```
Note that DDE to Excel
requires that you use
R4C3 syntax to specify
cell C4 (row 4, col 3).
```

Execute Program (PC)

Run Style: Normal

Wait for prog exit: No

Prog with params: excel

Working directory: C:/excel

Cell Value

3.1

To/From DDE

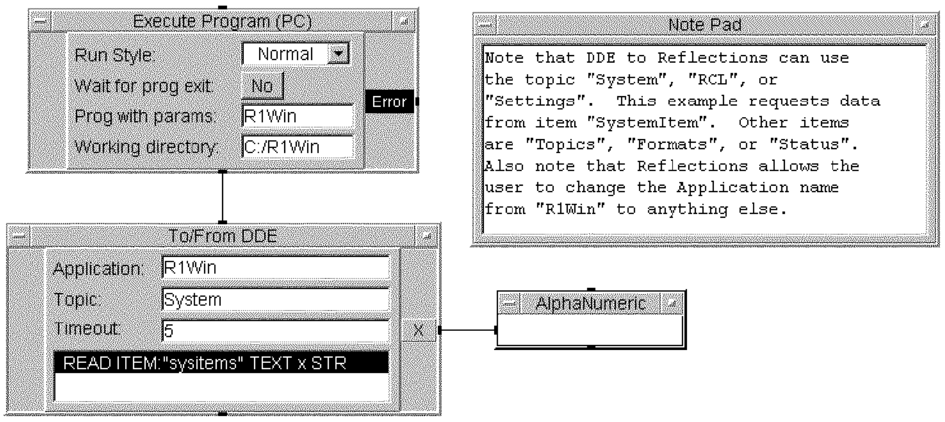
Application: Excel

Topic: Sheet1

Timeout: 5

```
WRITE ITEM:"R4C3" TEXT a EOL
```

Figure 5-33. Excel DDE Example

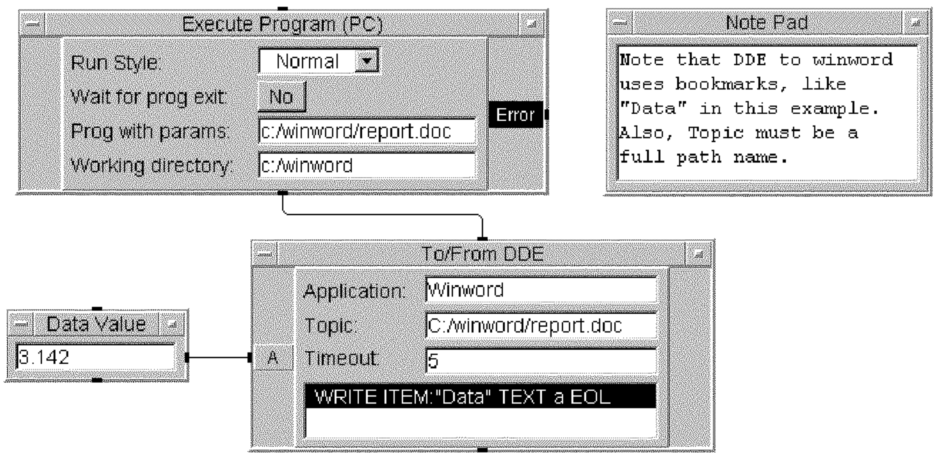


Run Style: Normal
Wait for prog exit: No
Prog with params: R1Win
Working directory: C:/R1Win

Application: R1Win
Topic: System
Timeout: 5
READ ITEM:"sysitems" TEXT x STR

```
Note that DDE to Reflections can use
the topic "System", "RCL", or
"Settings".  This example requests data
from item "SystemItem".  Other items
are "Topics", "Formats", or "Status".
Also note that Reflections allows the
user to change the Application name
from "R1Win" to anything else.
```

Figure 5-34. Reflections DDE Example



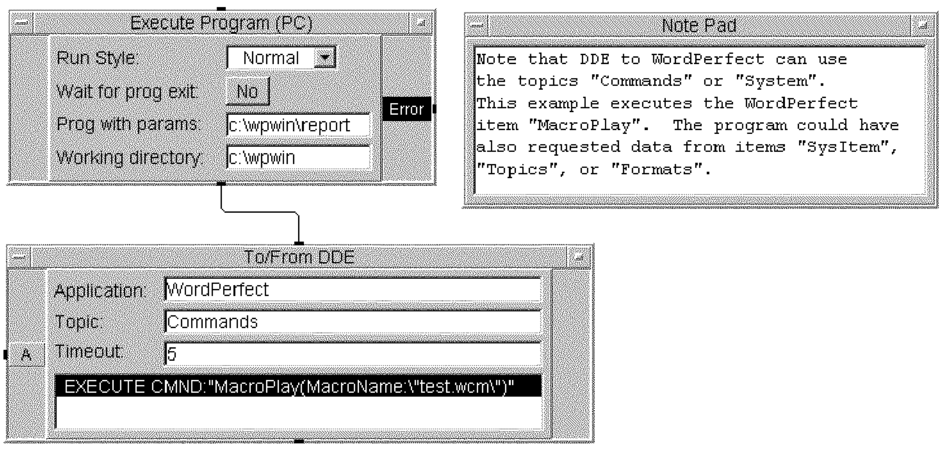
Run Style: Normal
Wait for prog exit: No
Prog with params: c:/winword/report.doc
Working directory: c:/winword

3.142

Application: Winword
Topic: C:/winword/report.doc
Timeout: 5
WRITE ITEM:"Data" TEXT a EOL

```
Note that DDE to winword
uses bookmarks, like
"Data" in this example.
Also, Topic must be a
full path name.
```

Figure 5-35. Word for Windows DDE Example



```
Note that DDE to WordPerfect can use
the topics "Commands" or "System".
This example executes the WordPerfect
item "MacroPlay".  The program could have
also requested data from items "SysItem",
"Topics", or "Formats".
```

```
EXECUTE CMND:"MacroPlay(MacroName:\"test.wcm\")"
```

Figure 5-36. WordPerfect DDE Example

Dynamic Linked Libraries (DLL)

For information on using DLLs see "Creating a Dynamic Linked Library (MS Windows)" in Chapter 4.

Related Reading

1. Haviland, Keith and Salama, Ben, *UNIX System Programming*. (Addison-Wesley Publishing Company, Menlo Park, California, 1987).

This book contains information of general interest to programmers using UNIX. In particular, it contains explanations of interprocess communications and pipes that are applicable to with `To/From Named Pipe`, `To/From Socket`, `To/From HP BASIC/UX`, and `Execute Program`.

For information on using transactions for instrument I/O, refer to *Controlling Instruments with HP VEE.*

6

Using the Sequencer Object

You'll need to understand several topics covered in this and other manuals in order to use the Sequencer object effectively. These topics include instrument I/O operations (*Controlling Instruments with HP VEE*), UserObjects (How Do I in HP VEE online Help), Records and DataSets (Chapter 3), and UserFunctions (Chapter 4). Also, for information on how to use a transaction, refer to "Using Transactions" in Chapter 5.

You can use the Sequencer object, found under the Device menu, to control the order of calling of a series of tests. The Sequencer object executes a series of sequence transactions. Each of these transactions evaluates an HP VEE expression, which may contain calls to UserFunctions, Compiled Functions, Remote Functions, or other HP VEE functions. After evaluating the HP VEE expression, the transaction compares the value returned by that expression against a test specification. Depending on whether the test passes or fails, the transaction then evaluates different expressions and selects the next transaction to be executed. Transactions may optionally log their results to the Log output pin, or to a UserFunction, Compiled Function, or Remote Function. Logging actions are specified in the Sequencer Properties dialog box on the Logging tab.

Sequence Transactions

The Sequencer object, in its open view, shows a list of sequence transactions. Each transaction is similar to the other types of transactions shown in Chapter 5. To see how the Sequencer uses transactions to execute expressions and call functions, let's look at a simple example.

In the following program there are two UserFunctions in the background: myRand1, which adds a random number from 0 to 1 to the value of its input, and myRand2, which adds a random number from 0 to 100 to its input. (Refer to Chapter 4 for further information on creating and using UserFunctions.)

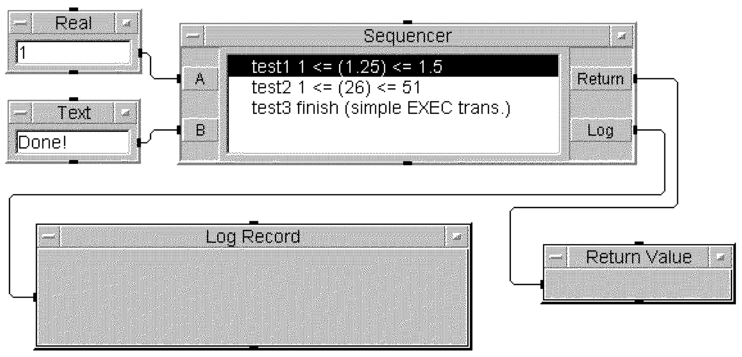
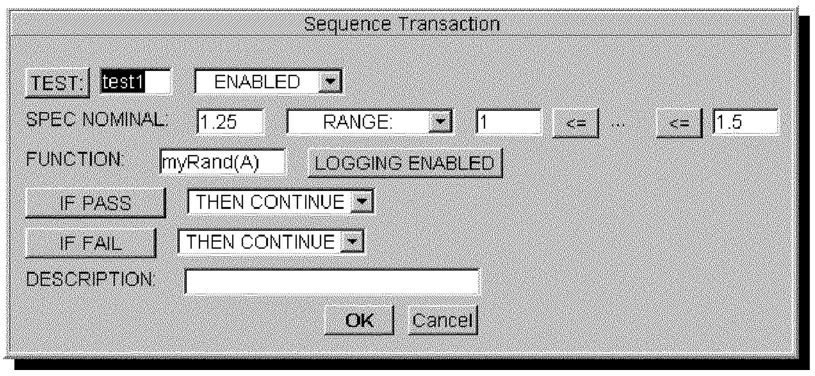


Figure 6-1. A Simple Sequencer Program

When you click on a transaction with the mouse, a dialog box "expands" the transaction so you can view and edit it. The following dialog box shows the first transaction, `test1`:



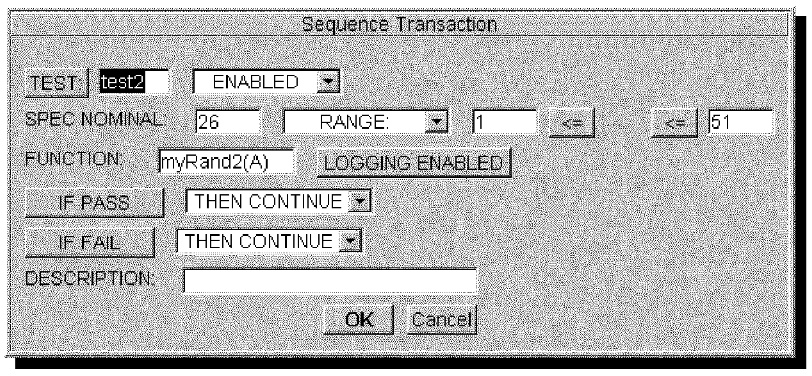
A sequence transaction can either be a TEST transaction or an EXEC transaction. In this transaction, the type is `TEST:`, the name field is `test1`, the nominal specification is `1.25`, a `RANGE:` specification is used, and the range is `1 <= ... <= 1.5`. Thus, only values from 1 to 1.5 will pass the test. The expression `myRand1(A)` calls the user function using the value on the `A` input terminal of the `Sequencer` as its input parameter. The transaction has logging enabled, so a local variable named `Test1` will be automatically created, which contains the log record of the results of this test. This log record will also be available as part of the Log output terminal. The IF PASS and IF FAIL conditions are both THEN CONTINUE. This means that, pass or fail, once test1 is done, the next transaction, test2, will be executed.

The DESCRIPTION field is simply a comment area for this test.

Note

For RANGE or LIMIT tests, the SPEC NOMINAL value is not used, except for "documentation" purposes. However, if you use tests based on TOLERANCE or %TOLERANCE values, the tolerance will be calculated relative to the SPEC NOMINAL value.

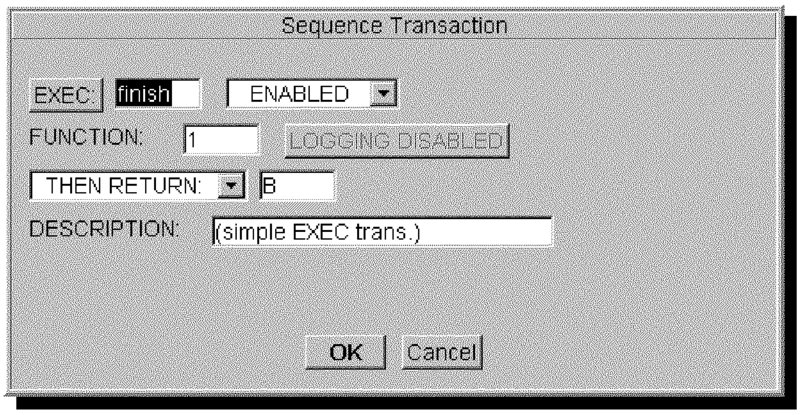
The second transaction, test2, is also a TEST transaction:



TEST: test2 ENABLED
SPEC NOMINAL: 26 RANGE: 1 <= ... <= 51
FUNCTION: myRand2(A) LOGGING ENABLED
IF PASS THEN CONTINUE
IF FAIL THEN CONTINUE
DESCRIPTION:

This second test is similar to the first. The UserFunction myRand2 is called with the expression myRand2(A) and the resulting value is tested to see if it is in the range 1 through 51, with a nominal specification of 26. Again, pass or fail, the Sequencer continues to the next transaction.

The third transaction is an EXEC transaction:


Sequence Transaction

EXEC: finish ENABLED

FUNCTION: 1 LOGGING DISABLED

THEN RETURN: B

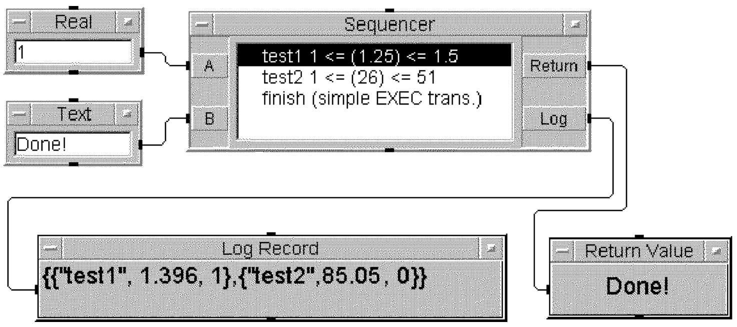
DESCRIPTION: (simple EXEC trans.)

An EXEC transaction, unlike a TEST transaction, performs no comparison of the function result to a specification or range. EXEC transactions are used to perform an action that does not require a pass/fail test. For example, an EXEC transaction could call a routine that sets up an external configuration before a TEST transaction is performed, or it could execute a power down procedure after a series of tests. (An EXEC transaction is a short cut for specifying an "always pass" test condition.)

In our example, the transaction named `finish` returns the value of `B` to the `Return` output terminal of the `Sequencer` object. Since no test is performed, logging does not occur for an EXEC transaction.

Note that you can use the `DESCRIPTION` field to briefly describe any transaction.

When you run the program, the three transactions are executed in sequence:



```
test1 1 <= (1.25) <= 1.5
test2 1 <= (26) <= 51
finish (simple EXEC trans.)
```

```
{{"test1", 1.396, 1},{"test2",85.05, 0}}
```

Figure 6-2. Running the Program

The logged test results are output on the Log output terminal and displayed. Note that the results are logged as the Record data type, in fact a record of records. In this case, `test1` has passed with a value of `1.396` and `test2` has failed with a value of `85.05`. The third transaction returns the value on the `B` input, which is the string `Done!`.

Let's look more closely at how logging works. Each transaction that has logging enabled creates a log record and attaches it to the transaction name. In our example, logging is enabled for the first two tests, so local variables named `Test1` and `Test2` contain the log records for those transactions. The fields contained in the log records are defined on the `Properties` dialog box. To access the logging configuration, click on `Properties` in the `Sequencer` object menu, then on the `Logging` tab. By default, log records contain `Name`, `Result`, and `Pass` fields.

The `Test1` and `Test2` local variable names can be used in any expression *within* the `Sequencer` to access the results of the current or a previously executed transaction. For example, `Test3` could have called a function with `Test1.Result` as a parameter to pass the result of the first test. Or `Test2.Pass` could be used as an expression, which would evaluate to `1` if `Test2` passed, or `0` if `Test2` failed.

There is one more local variable, `thisTest`, available to access the logging records. The value of `thisTest` is always the same as the logging record for the currently executing transaction. This allows you to write transaction expressions that can be used in many transactions without having to include the name of each transaction.

Now let's examine the data structure produced by the Log output terminal on the `Sequencer`, which is a record of records:

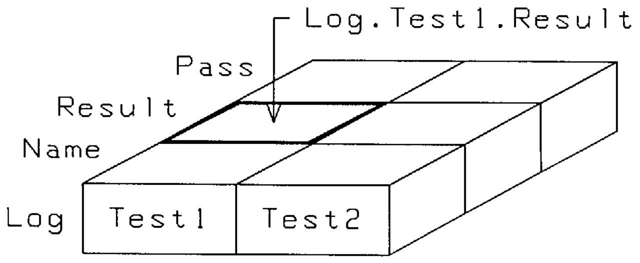


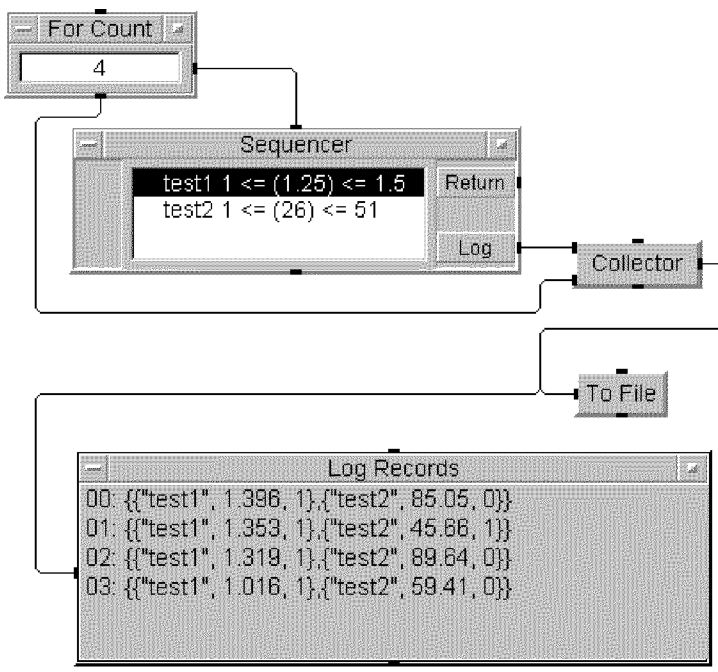
Figure 6-3. A Logged Record of Records

The record produced by the Log output pin contains a field for each transaction that has logging enabled — `Test1` and `Test2` in our example. Each of these fields is simply the log record for the specified transaction, containing the fields `Name`, `Result`, and `Pass`. This record of records is available on the Log output pin and can be used by other objects by using the record "dot" syntax. For example, the expression `Log.Test1.Result` would, in this case, return the value `1.396` (see Figure 6-2). Likewise, `Log.Test1.Name` would return `test1` and `Log.Test1.Pass` would return `1`.

Note that the data logged on the Log output pin is always the data from the *last* execution of each transaction. If you wish to log the results of *every* execution of each transaction, set `Logging Mode` to `Log Each Transaction To:` on the `Logging` tab of the `Sequencer Properties` dialog box. This option will call the specified function (or expression) at the completion of every transaction. This option can also be useful if you wish to log test results to a file or printer *as they happen*, rather than waiting until the `Sequencer` has completed. The local variable `thisTest` can be used as a parameter to the logging function to pass the log record of the transaction that has just completed.

Logging Test Results

Now let's look at a more practical example of logging test results, where an iterator causes the Sequencer to repeat the tests over and over, and to log the results:



```
00: {{"test1", 1.396, 1},{"test2", 85.05, 0}}
01: {{"test1", 1.353, 1},{"test2", 45.66, 1}}
02: {{"test1", 1.319, 1},{"test2", 89.64, 0}}
03: {{"test1", 1.016, 1},{"test2", 59.41, 0}}
```

Figure 6-4. A Simple Logging Example

In this example, the For Count object causes the Sequencer to execute its series of tests (test1 and test2 of the previous example) four times. For example, if four "widgets" are being tested on an assembly line, each execution of the Sequencer tests one widget. The resulting series of records from the Log output terminal is collected by the Collector and displayed as an array of records. Note, also, that you can use the To File object to output this array to a file using a WRITE CONTAINER I/O transaction or you could use a DataSet.

Conceptually, the output of the `Collector` in this example can be viewed as an array of records of records, as shown below:

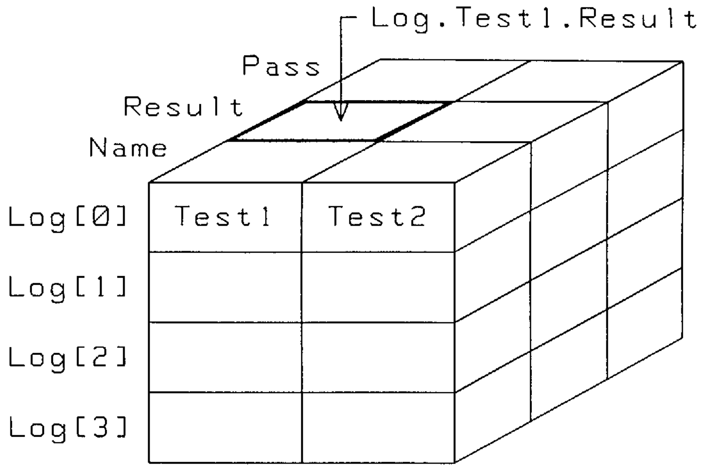


Figure 6-5. A Logged Array of Records of Records

Each array element (`Log[0]`, `Log[1]`, etc.) represents a single iteration of the sequencer, and is a record of records as shown in Figure 6-3. As mentioned before, the logged output is available for analysis in expressions. In this case, `Log[*].Test1.Result` is a "core sample" from the array. In fact, `Log[*].Test1.Result` would return an array of values (1.396, 1.353, 1.319, and 1.016 for the example results shown in Figure 6-4).

Note

The logged array is *not* a three-dimensional array, but is rather an array that consists of records of records. This is important because the individual fields of a record can be of differing data types. For example, while the `Name` field is Text, the `Result` field could be a Waveform, and so forth. Also, the `Test2.Result` field could be a Waveform, while the `Test1.Result` field is a Real value.

However, each individual field must be of a consistent data type throughout the array. For example, the field `Test1.Result` can't be a Real value for `Log[0]` and a Waveform for `Log[1]`.

Let's extend our example to 10 iterations of the `Sequencer`, and add some analysis of the logged data. In the following example, the expression `log[*].test1.result` in the `Formula` object returns a 10 element Real Array, which contains the results of `test1`. This array is then statistically analyzed by means of the `min(x)`, `max(x)`, `mean(x)`, and `sdev(x)` objects.

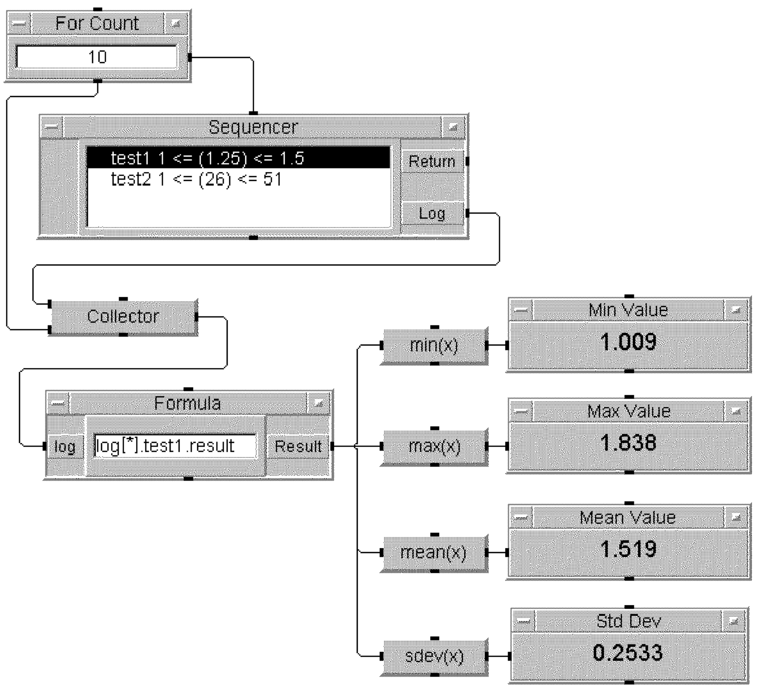


Figure 6-6. Analyzing the Logged Test Results

This example is saved in the file `manual44.vee` in your `examples` directory.

Logging to a DataSet

You can use a DataSet to store your logged test results. In the following program, the `Sequencer` object Log output terminal is connected to the `To DataSet` object.

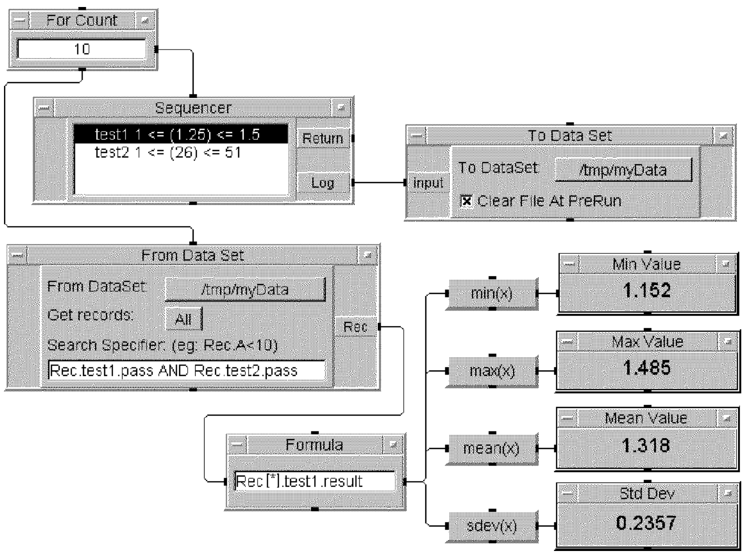


Figure 6-7. Logging to a DataSet

Once the `For Count` object is finished, it causes the `From DataSet` object to retrieve the stored DataSet (`myDataSet`). `From DataSet` is configured to retrieve `ALL` records from `myDataSet`, but to test each record against the condition `Rec.test1.pass AND Rec.test2.pass`. In other words, a particular record is retrieved only if *both* `test1` and `test2` passed for that record.

Of the retrieved records, if any, the expression `Rec[*].test1.result` returns all of the `test1.result` record fields, which are then statistically analyzed. (Note that this program will error if none of the records satisfy the expression `Rec.test1.pass AND Rec.test2.pass`.)

This example is saved in the file `manual45.vee` in your `examples` directory.

Some Restrictions in Logging Test Results

There are some situations where you must be careful in collecting Sequencer log records into an array of records. As explained in Chapter 3, to build an array of records, all of the array elements of a given field must be of the same type, shape, and size. For a record of records, as is generated by the Log output terminal of the Sequencer, the type, shape, and size of each field must match for sub-records as well.

For example, suppose you are collecting the logged results of several executions of a Sequencer, either by using the Collector to build an array (see Figure 6-6) or by sending the results to a DataSet (see Figure 6-7). In either case, if any of the logged values of a given transaction were to change type, shape, or size between executions of the Sequencer, an error will occur. The error will be generated by the Collector or To DataSet object because the array of records cannot be built.

This situation could easily occur if a transaction is not executed on every execution of the Sequencer; for example, if an ENABLED IF condition is specified. If the transaction is not executed, a log record will still be generated, but the NAME and DESCRIPTION fields will be empty strings and all the other fields will contain a Real scalar value of zero. If the same transaction, on a subsequent execution of the Sequencer, is executed and logs a result that is not a Real scalar, an error will occur. You might want to consider, in this situation, just writing each logged record out to a file in container format with To File, instead of using To DataSet.

An error could also occur if your tests return arrays of different sizes; for example, if the test returns an array of the failed data points. In this case, you might want to design the test so that it pads the array so as to always return the same size array.

A Practical Test Example

So far, we've just looked at how the `Sequencer` works, and how you might store, retrieve, and analyze the logged data. But normally, you'll want to use the `Sequencer` to control a series of "real world" tests. So let's look at a simple practical example.

In the old days, carbon resistors were manufactured by a rather imprecise process, and then tested, sorted, and marked. The trick was that the standard resistance values (for example, 220, 270, and 330 ohms) were chosen to overlap at the 10 percent tolerance. Thus, you didn't need to throw any resistors away. If a resistor was more than 10 percent greater than 220 ohms, it could be labeled as a 270 ohm resistor, and so forth.

So our problem is to construct a program in which the `Sequencer` calls a UserFunction, which returns a resistance value. The `Sequencer` will then run a series of tests to determine which nominal resistance value and percent tolerance the resistor satisfies. This is a "bin sort" problem. That is, the sequencer returns a result that identifies the bin in which to put the resistor.

One of the big advantages of using the `Sequencer` to call a UserFunction is that different UserFunctions can be substituted. For our problem, we'll just use a UserFunction (`simResist`) that returns a random resistance value in the expected range during development. You can easily substitute another UserFunction that executes instrument I/O and returns real resistance values once you've tested your solution.

The simplest solution to our problem is to use an extended series of sequence transactions, each testing the resistance value against a nominal value and tolerance.

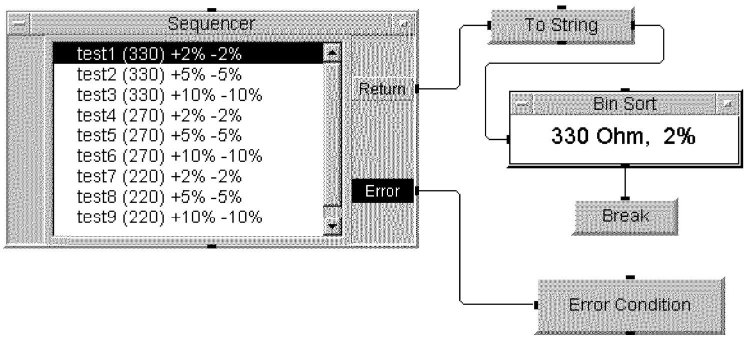
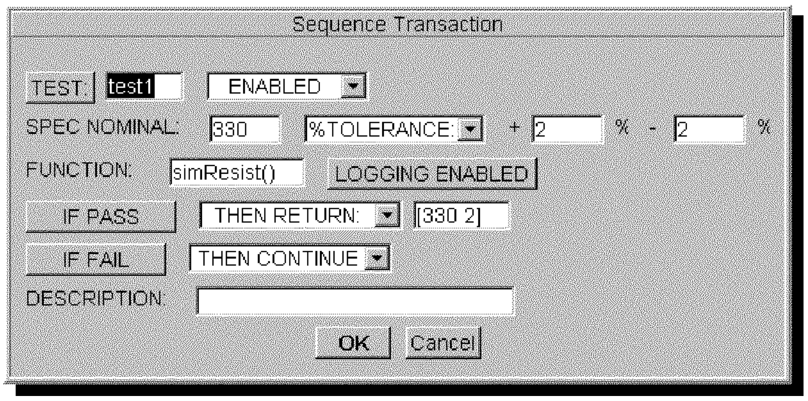


Figure 6-8. Simple Bin Sort Example

In this example, the first sequence transaction (`test1`) calls the UserFunction `simResist` with the expression `simResist()`. (This UserFunction requires no inputs.)


Sequence Transaction

TEST: test1 ENABLED

SPEC NOMINAL: 330 %TOLERANCE: + 2 % - 2 %

FUNCTION: simResist() LOGGING ENABLED

IF PASS THEN RETURN: [330 2]

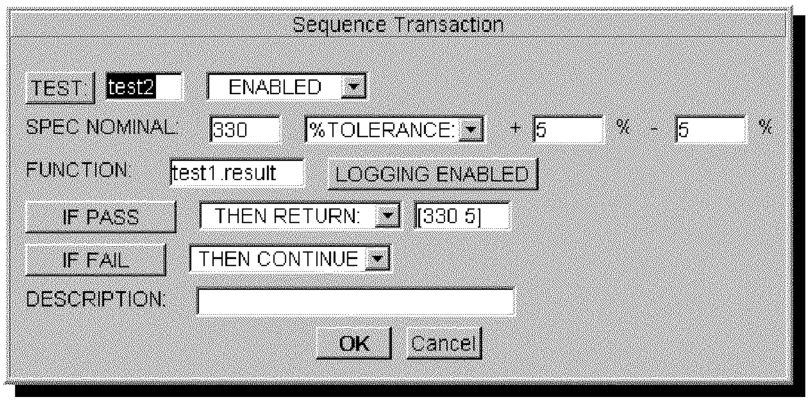
IF FAIL THEN CONTINUE

DESCRIPTION:

OK Cancel

Note that `test1` tests to see if the resistance value returned by `simResist` is within ±2 percent of the nominal value 330. If it is, the two-element Real array `[330 2]` is returned on the `Return` output terminal, and the `To String` object converts this value to the string `330 Ohm, 2%`. If the test fails, the `Sequencer` goes on to the next test.

The second transaction, `test2`, works just like the first except that instead of calling `simResist` again, the `FUNCTION` field contains the expression `test1.result`:


Sequence Transaction

TEST: test2 ENABLED

SPEC NOMINAL: 330 %TOLERANCE: + 5 % - 5 %

FUNCTION: test1.result LOGGING ENABLED

IF PASS THEN RETURN: [330 5]

IF FAIL THEN CONTINUE

DESCRIPTION:

Key Idea

Any transaction with logging enabled creates a "local" Record variable with the same name as the test. This record contains the fields specified for the logging record. Thus, for the transaction `test1`, the expression `test1.result` returns the value returned by the function called in `test1`.

There are two reasons for using the expression `test1.result` in our example. First, by using `test1.result` in transactions `test2` through `test9` we can ensure that each transaction uses the same function result, even if we later change `test1` to call a different function. More importantly in this example, each time you call the UserFunction, a *new* resistance value will be returned. Instead, we want to continue testing the original resistance value against successive nominal values and tolerances. So the transactions `test2` through `test9` all include the expression `test1.result` in the `FUNCTION` field. These transactions work like the first, returning the appropriate array (`[330 5]`, `[330 10]`, `[270 2]`, and so forth) if passed.

The first eight tests simply continue to the next test if failed. However, an indication is needed if *all* of the tests are failed. Thus, `test9` is configured `IF`

FAIL THEN ERROR. The Error output terminal causes the AlphaNumeric display entitled Error Condition to execute, displaying the text Out of Range.

Although this approach is simple, it is not very efficient. You would need to create quite a large number of sequence transactions to test several resistance values, with three tolerances in each case. Let's look at an improved version of our "bin sort" example.

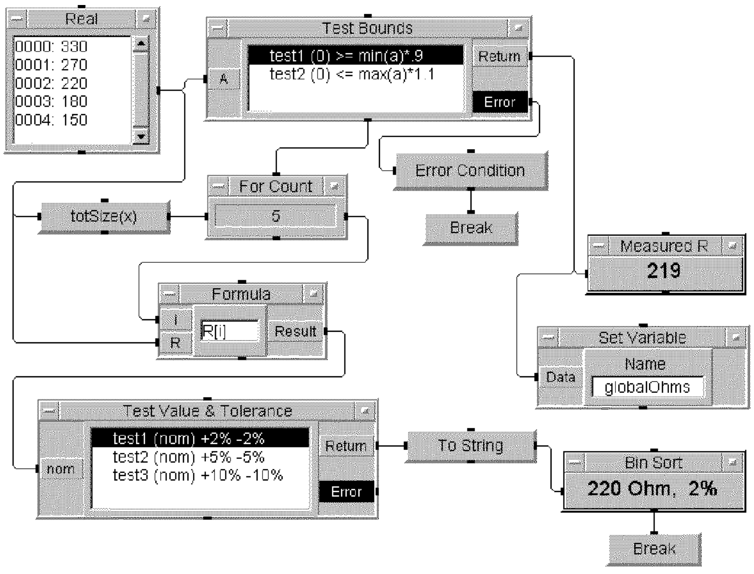


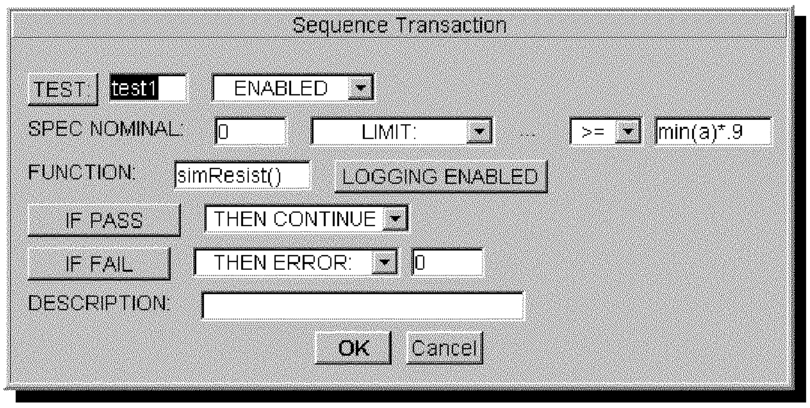
Figure 6-9. Improved Bin Sort Example

This example is saved in the file manual46.vee in your examples directory.

FINAL TRIM SIZE : 7.0 in x 8.5 in

You may want to load this program and explore how it works. Here are some key points:

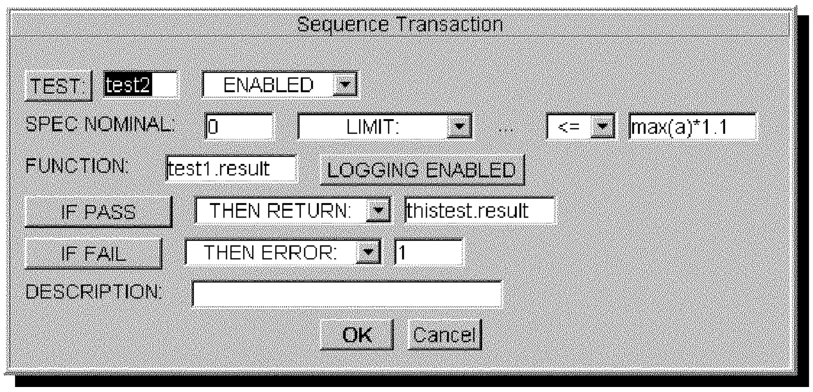
- This program uses two `Sequencer` objects. The first one (labeled `Test Bounds`) "re-uses" the tests in the second one (labeled `Test Value & Tolerance`).
- The `Real` array in the upper left corner of the program contains five elements, each representing a standard resistance value. However, the list of values is *extensible* in this example. Regardless of the number of array elements, the `TotSize(x)` function returns that number so that the `For Count` object will iterate the correct number of times. The expression `R[i]` in the `Formula` object takes care of the indexing.
- In the `Sequencer` named `Test Bounds`, the first transaction (`test1`) calls the UserFunction `simResist` with the expression `simResist()`:



TEST: test1	ENABLED			
SPEC NOMINAL: 0	LIMIT:	...	>=	min(a)*.9
FUNCTION: simResist()	LOGGING ENABLED			
IF PASS	THEN CONTINUE			
IF FAIL	THEN ERROR:	0		
DESCRIPTION:				

A simulated resistance test value is returned and tested to see if it is at least 90 percent of the lowest value (150 Ohms) in the array. (Note that any value field in a sequence transaction can contain an expression such as `min(a)*.9`.)

The second transaction (test2) tests to see if the value (test1.result) is less than or equal to 110 percent of the highest value (330 Ohms) in the array.

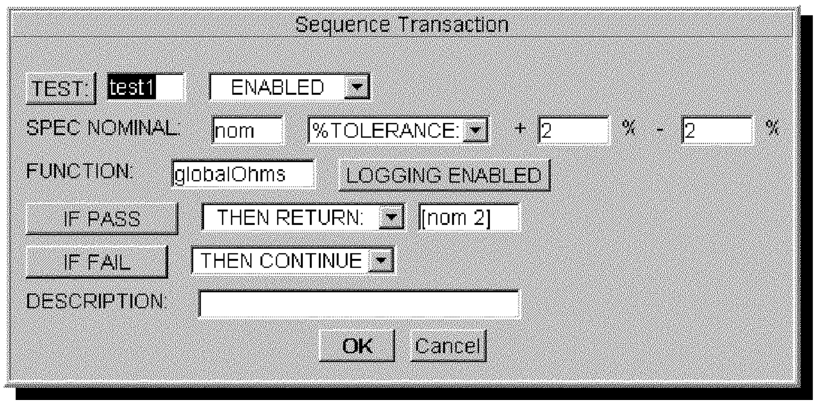

Sequence Transaction

TEST: test2 ENABLED
SPEC NOMINAL: 0 LIMIT: ... <= max(a)*1.1
FUNCTION: test1.result LOGGING ENABLED
IF PASS THEN RETURN: thistest.result
IF FAIL THEN ERROR: 1
DESCRIPTION:

If either test fails, an error occurs.

- If an error does occur, the UserObject named Error Condition uses a Triadic expression to ascertain whether to display Out of Range: LOW or Out of Range: HIGH. The UserObject is configured as Show Panel on Exec, so if either error condition occurs, a display "pops up" to show the error. You'll find that this happens once every few times you run the program because the UserFunction simResist returns random values in the range 100–400. (To continue, just press OK in the pop-up box.)
- The transaction test1 in the first Sequencer is the only transaction that calls the UserFunction simResist. (Instead, test2 includes the expression test1.result.) This is necessary in this case because we want to run multiple tests on just one resistance value. Otherwise, a new value would be returned every time the UserFunction was called. However, there is another reason. Since the UserFunction simResist is only called once, you can easily replace it with a call to a different UserFunction. The example (manual46.vee) contains a second UserFunction named measResist, which uses an HP Instrument Driver to call an HP 3478A Digital Voltmeter configured for resistance measurements. If you have an HP 3478A meter, just connect it to your HP-IB, change the FORMULA field in test1 to the expression measResist(), and run the program.
- Regardless of whether simulated or measured resistance values are taken, the Test Bounds return value is displayed, and is set as a global variable

(globalOhms). The three transactions in the Sequencer labeled Test Value & Tolerance each call this global variable using the expression globalOhms, for example:


Sequence Transaction

TEST: test1 ENABLED

SPEC NOMINAL: nom %TOLERANCE: + 2 % - 2 %

FUNCTION: globalOhms LOGGING ENABLED

IF PASS THEN RETURN: [nom 2]

IF FAIL THEN CONTINUE

DESCRIPTION:

If a test passes, the appropriate real array (e.g., [220 2]) is output. The To String object converts the data to a string (e.g., 220 Ohm, 2%). The Sequencer will be executed as many times as necessary until a Bin Sort result is found.

- Note that we are not using the Log output terminal in either Sequencer, so we've deleted it to speed up execution.
- If you want to see the flow of this program, try running it a few times with Show Exec Flow and Show Data Flow turned on.

For some further examples using the Sequencer, look in your examples directory.

7

Keys To Faster Programs

For general tips to increase the performance of your program, refer to `Improving the Performance of an HP VEE Program` under `How Do I` in HP VEE online Help.

If you developed programs on a previous version of HP VEE, refer to Appendix A for information on converting your program to use the compiler.

The following constructs will help you get the most speed benefit from the compiler (when the `Execution Mode` is set to `Compiled` in `File` ⟹ `Default Preferences`):

- Use the `Profiler` (located at `View` ⟹ `Profiler`) to categorize which routines are taking more time than you want them to. To run the `Profiler`:
  1. Click on `Start Profiling` and then run your program.
  2. When you have finished running your program, click on `Refresh` to see the results.
  3. Click on `Stop Profiling` to stop the profiler. Click on `Clear` to clear the current results displayed.
- Look at line colors. Lines are colored when HP VEE can determine the data type before execution. The more colored (non-black) lines, the faster the program will run.
- Because UserFunctions can be called from multiple places, HP VEE can't determine the input data types before the program runs. So to speed up UserFunctions, wherever possible add terminal constraints on their data input terminals.
- If you use global variables, use `Declare Variables` (located on the `Data` menu) when possible to declare the type and shape of your variables so HP VEE can infer types for them prior to execution. This technique also allow you to set the scope of your variables.

- A common programming practice is executing the `Autoscale` control input on graphical displays more often than necessary. If you can wait to execute `Autoscale` until after the display has finished updating, instead of after each point is plotted, your program will execute faster.
- If a display is showing the final output of a loop, but not tracking data generated for each iteration of the loop (for example, an `AlphaNumeric` object, not a `Logging AlphaNumeric`), don't have it execute every time in the loop. Hook the iterator's sequence output pin to the display's sequence input pin so the display only executes the last time.
- Once you know the program is running correctly, run the program with debugging features off. Use `File` ⟹ `Default Preferences` and select `Disable Debug Features` in the `Execution Mode` group.

  You can also use the `-r` option, or run HP VEE RunTime. Because no debug instructions are generated in those modes, your program will run a little faster. However, you will not be able to perform any debugging actions such as, pausing, stepping, `Breakpoints`, `Line Probe`, `Show Data Flow` and `Show Execution Flow`.

8

Troubleshooting Problems

This chapter explains common situations and recovery actions.

Table 8-1. Problems, Causes, and Solutions

Problem	Cause	Solution
When running a program created in previous versions in compiled mode, it doesn't operate when you think it should.		Refer to Appendix A for possible solutions.
Your `UserObject` doesn't operate when you think it should.	You might be crossing the context boundaries with asynchronous data (such as connecting to an `XEQ` pin on an object inside the `UserObject`).	Possible Solution 1: Move any asynchronous dependencies to outside the `UserObject`. Possible Solution 2: Enable `Show Execution Flow` or `Show Data Flow` to view the order of operation in your program.
You want to change the functionality of an object.		Use the object menu which includes features that let you add a control input terminal and edit properties.
You only get one value output from an iterator within a `UserObject`.	A `UserObject` only activates its outputs once.	Take the iterator out of the `UserObject`.
An iterator only operates once.	Your iteration subthread is connected to the sequence output pin, not the data output pin.	Start the iteration subthread from the data output pin.
`For Count` doesn't operate.	The value of `For Count` is `0` or negative.	Change the value; if you need a negative value, negate the output or use `For Range`.
`For Range` or `For Log Range` doesn't operate.	The sign of the step size is wrong. If `From` is less than `Thru`, `Step` must be positive. If `Thru` is less than `From`, `Step` must be negative.	Change `Step`.

Table 8-1. Problems, Causes, and Solutions (continued)

Problem	Cause	Solution
You get the UNIX message `sh:`*name* `- not found.`	You mistyped the name of the executable.	Retype `veetest`.
You get the UNIX message `Error: cannot open display`	Your DISPLAY environment variable is not set or is set to display on a machine for which permissions are not set up correctly.	Set (and export) your environment variable DISPLAY. Generally, this is set to *hostname:0.0*. To display on a remote machine, set up permissions with *xhost* on the remote machine.
HP VEE appears to hang—the pointer is an hourglass.	Possible Cause 1:HP VEE is rerouting lines because you have `Auto Line Routing` set on and you moved an object. Possible Cause 2: HP VEE is printing the screen or the program. Possible Cause 3: You just `Cut` a large object or a large number of objects. HP VEE is saving the objects to the `Paste` buffer.	Wait. If the pointer doesn't change back to the crosshairs within a few minutes, type CTRL-C (or whatever your `intr` setting is in the terminal window from which you started HP VEE), close the HP VEE window, or kill the HP VEE process.
You can't `Open` a program, `Cut` objects, or delete a line (the feature is grayed).	The program is still running.	Press `Stop` to stop the program, then try the action again.
You can't `Paste` (the feature is grayed).	The `Paste` buffer is empty.	`Cut`, `Copy`, or `Clone` the object(s) again.
You can't `Cut`, `Create UserObject`, or `Add to Panel` (the feature is grayed).	No objects are selected.	Select the objects and try the action again.
A `UserObject` only outputs the last data element generated.	`UserObjects` do not accumulate data in the output terminal buffer. It only holds the last data element received.	Use a `Collector` to gather all of the data generated into an array. Send this data to the output terminal.
You can't get out of line drawing mode.		Double-click or press Esc to end line drawing mode.

Table 8-1. Problems, Causes, and Solutions (continued)

Problem	Cause	Solution
You get a `Parse Error` object when you `Open` a program.	Replace the `Parse Error` object with a new object.	
Your characters are not appearing correctly.	You have a non-USASCII keyboard.	Refer to Appendix B for recovery information.
Your colors outside of HP VEE are changing (although when you're in HP VEE, the HP VEE colors look normal).	Your color map planes are all used.	Refer to Appendix B for recovery information.

A

Using the Compiler

When you have programs created in previous versions of HP VEE, you can run them in VEE 3 Compatibility mode and they will execute the same as before.

However, if you want to take advantage of the speed improvements, set the Execution Mode to Compiled in File ⟹ Default Preferences. Note there are some program execution differences in Compiled mode. This appendix presents these differences.

About The Compiler

It is not necessary to understand the information in this section to use the compiler. It explains the concepts behind the compiler for your information only.

The HP VEE compiler converts a HP VEE program into p-code, but there isn't any machine language or executable generated.

The compiler allows HP VEE to predict at compile time (instead of determining at run time) the order of execution of objects, determine what data types will be flowing on certain data lines, optimize code generation, and generate and execute the most optimal p-code for any given HP VEE object.

One of the goals was to maintain the great level of interactiveness associated with HP VEE, especially during development/debugging of programs. As such, the compilation of HP VEE programs takes place transparently right after you press the Run button. Stepping and breakpoints are also fully supported, as well as Show Execution Flow, Show Data Flow, and Line Probe.

Subsequent runs of the same unmodified program do not require recompilation. Also, when a program is modified, only the contexts needing recompiling are recompiled (much like an incremental compiler).

Most programs benefit from the use of the compiler, though the actual results vary. For example, a program using many levels of nested loops will probably see a greater speedup than one that does a lot of I/O or screen updates (e.g. displays).

In compiled mode, iterators and formulas gain the most execution speed benefit.

Compatibility Changes

HP VEE programs written with previous versions of HP VEE run *exactly* the same way as they used to when run in `VEE 3 Compatibility` mode. To ensure this, the interpreter is automatically enabled upon loading of older programs. It is possible that a program written with an previous version of HP VEE isn't going to run exactly the same way with the compiler. This could be due to specific programming techniques, the use of undocumented side-effects, or even slight changes in documented behavior.

Line Colors

In compiler mode, HP VEE assigns different colors to the data lines based on the type of data flowing through the line. Here are the default colors, along with the names of the color properties (changeable via `File` ⟹ `Default Preferences`):

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

If the data type is an array, HP VEE displays a wider line.

To increase speed, check your program for colored lines. The more non-black lines, the faster the program runs.

Compiling Existing Programs

To use the HP VEE compiler with older programs, change the `Execution Mode` checkbox on the `File` ⟹ `Default Preferences` dialog box to `Compiled`.

1. Open the old program, turn on compiler mode, and press `Step` (or `Run`). This will PreRun the program. If there are unsupported constructs like feedback without a `Junction`, intersecting loops, etc, HP VEE will error now. These constructs must be changed. The most common situation is feedback without `Junction` objects; simply insert a `Junction` fed by a `Constant` to initialize the value, refer to "Feedback Cycles" for more information.
2. Try running the program. Most everything will run the same way. The most common problem is not realizing part of your program relied on round-robin object order execution. Most of the time this will not matter. In a few cases, where one thread sets a global variable and another thread accesses it, programs may have "just worked" before and now may not if there is nothing to ensure the `Set Variable` executes before the `Get Variable` object.

Most of the time, the program will either error at PreRun or run normally.

There is the potential that the program will not work but also will not error in an obvious way, because of the way separate threads and parallel junctions execute in the compiler. Refer to "Program Changes" for the details on these changes.

Program Changes

Old program (written in previous versions) are automatically run in `VEE 3 Compatibility` mode.

New HP VEE programs are automatically compiled.

You can manually change the `Execution Mode` of a program at any time.

The following areas are where compatibility problems could arise: either when creating a new program to be compiled or when taking an existing program and executing it in compiler mode. The information about using a previous version of HP VEE is the same as when using interpreted mode or `VEE 3 Compatibility` mode.

Time-Slicing UserFunctions

In previous versions of HP VEE, UserFunctions did not time-slice with other parts of the program. In compiled mode, UserFunctions will time-slice when called from separate threads. Be sure to use sequence pins between `Call` objects when parallelism isn't desired.

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

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

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

UserObjects

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

Function Precedence

The precedence of functions called from the Formula object has changed to the following:

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

In previous versions of HP VEE, internal functions were last in precedence (this allowed you to override internal functions such as `totsize()` or `fft()` with your own).

Auto Execute and Start

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

OK Buttons and Wait for Input

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

1. Stepping: In previous versions, stepping over such an object would often result in the termination of the program. In compiler mode, stepping works without program termination.
2. CPU usage: In previous versions, executing such an object usually resulted in increased CPU usage. In compiler mode, the CPU stays in an idle state.

Collectors Without Data

In previous versions, hitting the `XEQ` pin of a `Collector` that has never been hit with data, outputs a nil container. In compiler mode, if the data type is known at compile time, you get a zero-element array of that data type. Otherwise, you get a zero-element array of type Integer.

This change allows the type inferencing to be more consistent, producing better p-code downstream from the `Collector` object.

Note that `totSize()` of a nil produces a one, while `totSize()` of a zero-element array produces a zero.

Sample & Hold Without Data

In previous versions, hitting the `XEQ` pin of a `Sample & Hold` object that has never been hit with data will yield a nil container. In compiler mode, the following error is generated (error number 937):

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

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

Timer Object

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

Feedback Cycles

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

```
A Junction is required inside of feedback cycles.
```

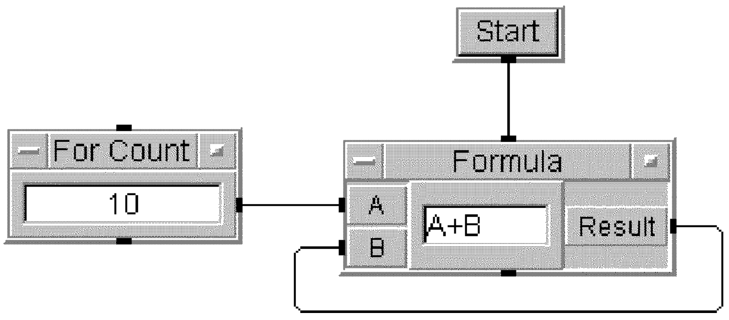


Figure A-1. Feedback in Previous Versions

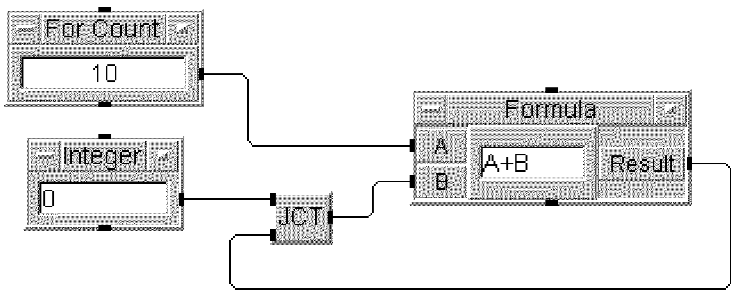


Figure A-2. Feedback in Compiled Mode

Note that the current version does not allow invalid connections, such as an object's data input pin connected to its data output pin.

Parallel Threads

In previous versions, independent threads would round-robin between each thread, meaning that one object will be executed in one thread, then an object in the other thread, etc. In compiler mode, this behavior is not guaranteed.

Loop Bounds

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

For example, if the `Step` value of a `For Range` object is changed via the data input pin while the loop runs, it is ignored in compiler mode. In previous versions, the step value would have been checked on every iteration.

UserObjects and Calls With XEQ Pins

In previous versions, you could have an `XEQ` pin on a UserObject or a `Call` object run the UserObject or UserFunction before all the data input pins were satisfied. In compiler mode, this is not allowed. `XEQ` pins on those objects will generate an error.

You can no longer add an `XEQ` pin to those objects.

OK Buttons With XEQ Pins

In previous versions, an `OK` object with an `XEQ` pin was only executed once, when either the `OK` button was pressed or when the `XEQ` pin was sent data.

In compiler mode, the `OK` button will execute every time the `XEQ` pin is sent data.

You can no longer add an `XEQ` pin to an `OK` object.

From File With EOF Pins

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

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

The following figure illustrates this situation. In previous versions, the data fed into A on the Formula would have remained valid even while another iteration of the loop executed. To get valid data fed into B on the Formula, the EOF pin (on the bottom) executes and then the Formula executes.

In compiled mode, the data fed into A is invalidated as soon as the next iteration of the loop begins. Because Formula does not get valid inputs on the same iteration of the loop, it never executes.

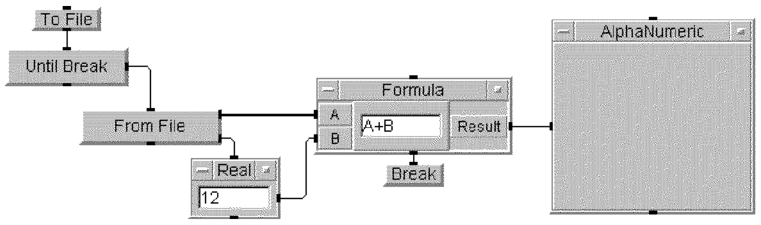


Figure A-3. EOF Differences

Parallel Junctions

In previous versions, if you had unconstrained objects that were connected in parallel to `Junction` objects, the order that you made the connections affected the execution order. In compiled mode, the order of connection does not matter.

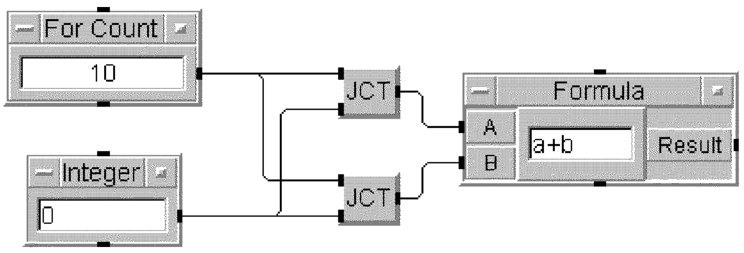


Figure A-4. Parallel Junctions

Intersecting Loops

In previous versions, you could intersect iteration objects. The execution order was undefined, but was affected by the order the connections were made. In compiler mode, only loops that intersect via a `Junction` object are allowed. Any other intersecting loops generate error 938.

`VEE was unable to compile this part of the program.`

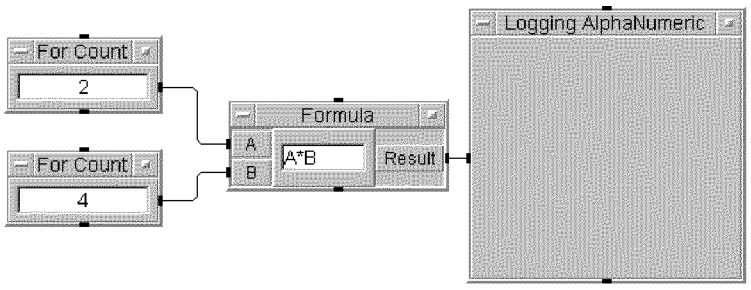


Figure A-5. Intersecting Loops

Intersecting Loops Via Junctions

In previous versions, the example shown below would execute the `Integer` first, and when the program encountered the `Break`, it would stop.

In compiled mode, the example below runs the `For Count` objects after the `Integer` objects because the `Break` does not stop the program.

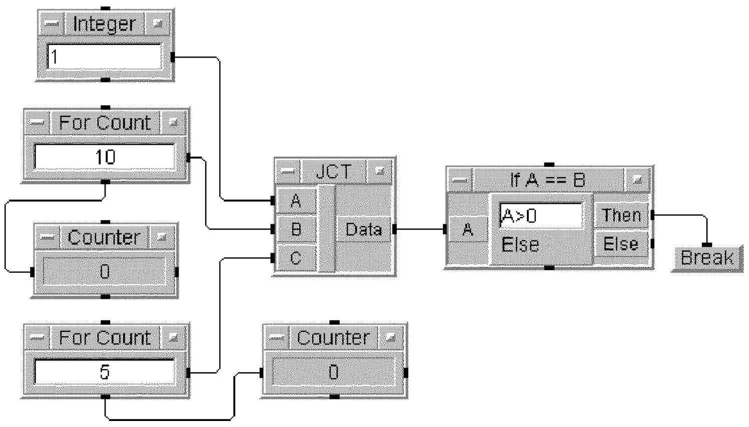


Figure A-6. Intersecting Loops Via Junctions

Open View Object Changes

In previous versions, you could change the data in open view fields while the program was running or paused. These changes would affect program behavior and the result was not guaranteed.

In compiler mode, many objects do not allow this type of modification when the program is running or paused (the input fields are grayed). Some examples of this are:

- `Formula` and `If/Then`
- `Collector`
- All Transaction objects' transactions
- `Get Mappings` and `Set Mappings`
- `Get Values` and `Set Values`
- `Constant`'s properties such as setting `Scalar` or `1D Array`, `Wait for Input`, or `Auto Execute`.

- Setting properties like `Clear at PreRun`
- UserObject and UserFunction `Trig Mode`
- `Dialog Boxes` properties

Adding or deleting input or output terminals on objects is grayed at run time (but not when paused). If this action is done at pause time, the program is stopped (this is what previous versions did).

B

Configuring HP VEE

This appendix explains how to configure and customize HP VEE for your environment by changing HP VEE options, and X11 options (in the UNIX environment) or Windows options (in the MS Windows environment). This appendix discusses the following topics:

- Color and font settings
- Changing X11 attributes (such as window size and placement)
- Changing MS Windows attributes
- Customizing your icon bitmaps
- Selecting a bitmap for a panel view
- Recovering from X11 color plane limitations
- Using non-USASCII keyboards and two-byte character sets
- Using HP-GL Plotters

Color and Font Settings

The HP VEE application contains default values for all color and font settings. You can change color and font settings (and many other properties) in the HP VEE `Default Preferences` dialog box (use `File` ⟹ `Default Preferences`). These properties are saved in the defaults file—`.veerc` in your UNIX $HOME directory, or `VEE.RC` in the `C:\Program Files\Hewlett-Packard\4.0` directory for MS Windows (or `%home%` if it is defined). For colors and fonts, only the settings you *change* are saved in this defaults file. See *Getting Started with HP VEE* and `Reference` in HP VEE online help for more information about changing colors and fonts in HP VEE.

Changing X11 Attributes (UNIX)

HP VEE provides an *app-defaults* file named `Vee` that you can use to customize several attributes of HP VEE. This file is in `/usr/lib/veetest/config/` for HP-UX 9.x. It is in `opt/veetest/config/` for HP-UX 10.x. In the same directory is the *app-defaults* file named `Helpview` which lets you customize the appearance of your Help windows. To use these files, you must install them into your X11 resources database.

The color and font settings that you *change* in HP VEE using `File` ⟹ `Default Preferences` are saved in the defaults file `$HOME/.veerc`.

If you are using `xrdb`, install the files by typing `xrdb -merge` *filename* (for each file) before starting HP VEE.

If you are not using `xrdb`, merge the files into your X11 resources file. Your X11 resources file is usually `.Xdefaults in your $HOME` directory, but may be in a file identified with the environment variable `$XENVIRONMENT`.

To change other X11 resources, you can change or add to your X11 resources file. For example, to change the default geometry of the HP VEE window so that it always started in the lower right corner of your screen and the window was sized to 640 by 480 pixels, you would add the following line to your X11 resources file (probably `.Xdefaults`): `Vee*geometry: =640x480-0-0`.

For more information about customizing an X11 environment, refer to *Beginner's Guide to the X Window System*.

Configuring HP VEE for Windows

HP VEE for Windows uses the Windows Registry to store HP VEE environment information.

The color and font settings that you change in HP VEE using `File` ⟹ `Default Preferences` are saved in the defaults file `VEE.RC` in your HP VEE installation directory if you do not have `%home%` defined. Otherwise `VEE.RC,VEE.IO`, and `V.INI` are saved in `%home%`.

General HP VEE Settings

The `Maximized` variable controls whether HP VEE for Windows starts up as a maximized window or not. The value `0` is for not maximized, `1` is for maximized.

The `Geometry` variable controls the initial size of the HP VEE for Windows window. For example:

```
Geometry=630x470
```

Customizing Icon Bitmaps

You can change the icon displayed on any iconized object to a bitmap or pixmap of your choice. HP VEE provides many files, or you can create your own. On UNIX platforms, HP VEE supports `.bmp` bitmap files, `.gif`, `.icn` icon files, and `.xwd` X11 bitmap files. HP VEE for Windows supports `.BMP` bitmap files, `.GIF`, and `.ICN` icon files. To select an object's icon, click on the object menu's `Properties` feature, then use the `Icon` tab on the `Properties` dialog box.

To create your own bitmaps for object icons, you can use any editor that outputs graphics formats that HP VEE supports. Examples of such editors include the `IconEditor` program on HP-UX, and the `Paint` program on MS Windows. You should specify `48x48` as the size for an icon. Larger icons use more space in the HP VEE program area, smaller icons are difficult to see. You can also use screen capture utilities such as X11 Window Dump (xwd) on UNIX, and `Print Screen` with `Paint` on MS Windows.

Selecting a Bitmap for a Panel View

You can select a bitmap to use as the background icon for a panel view. This applies to UserObjects and to HP VEE programs displayed in their panel views. Panel view icons must use the same formats HP VEE supports; `.bmp` bitmap files, `.gif`, and `.icn` icon files on all platforms; plus `.xwd` X11 bitmap files on UNIX. You can also use icons you create as described in the previous section.

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

If You See Colors Changing On Your Screen (UNIX)

Your workstation is equipped with a certain number of color planes (usually 1, 4, 6, or 8). X11 uses the information in these color planes to color your application's window. If you have more than one application running (each in its own window), and you notice the screen colors changing as you move from one application's window to another, then one of two things may be happening. Either all the applications, together, use more colors than your display has available, or one or more of the applications allocates its own private color map (for example, HP BASIC/UX).

HP VEE uses at least 39 colors (this varies depending on how you define the colors and which colors HP VEE actually uses while running), so you may experience this behavior when HP VEE is one of your applications. The symptoms are: when you are in the HP VEE window, the HP VEE colors will be correct for HP VEE, but may be wrong in other application's windows. When you move to another application's window, the colors will be correct for that application, but may be wrong for HP VEE. *This is typical X11 behavior—it is not a problem with HP VEE.*

This behavior does not affect the performance of HP VEE or any other application. However, if it bothers you, there are some things you can do to help, depending on the cause.

There are two causes of this behavior:

- You have requested more colors than your workstation can simultaneously display.
- One of the applications you are running controls a local color map.

Too Many Colors

Your workstation can display some number of colors at one time, based on the number of color planes for your display. This number is:

$$2^{number of color planes}$$

For example, if you have 4 color planes, you can use as many as 16 colors at a time on your display.

$$2^4 = 16$$

If you exceed this number, you may see the screen flashing as you change from one window to another.

If you exceed your total available colors, the first step in eliminating the "flashing" is to reduce your colors to be within the limits of your workstation. Some tips on reducing colors are:

- Remove any extra colors. If two applications can use the same color scheme, then customize them to do this.
- Use reduced-color color schemes in applications. For example, HP VEE allows customization of colors. Click on `File` ⟹ `Default Preferences`. In the `Default Preferences` dialog box, change your default colors to use only a few colors.
- Stop, or do not even start, any applications that you do not currently need. Often, each application uses its own color scheme. This can quickly increase your requested colors to exceed your color map limit. Once you have stopped other applications, you probably need to stop, then re-start, HP VEE before the behavior goes away.
- Reduce the number of colors allocated by the `xinitcolormap` command. Because these colors remain permanently in the color map, there is room for fewer temporary colors.

Some X11 window managers have a colormap focus directive (for example, `*colormapFocusPolicy`). The value to which this is set may also contribute to how colors are used on the screen. In particular, if you exceed the total number of colors you can simultaneously display, do not set this to be `explicit` or you may not see correct colors in your application's window.

Applications that Use a Local Color Map (UNIX)

Some applications use a local color map. This means that when you run this application, it saves the current color map and switches over to its own, local color map. When this happens you may see the "flashing" between windows.

One way to circumvent this is to pre-allocate the HP VEE colors using the `xinitcolormap` command. To do this, you create an ASCII file listing the colors you wish to pre-allocate. This file is described in the `man` page for `xinitcolormap`. Basically, though, the file cannot contain blank lines, must start with the colors `Black` and `White`, and the color format can be either pre-defined word colors or the actual RGB hex values, preceded by the symbol— `#`. For example, the following two examples contain black, white, and a shade of light gray:

```
Black
```

```
White
LightGray
```

Figure B-1. Color Map File Using Words

```
#000000
#ffffff
#a8a8a8
```

Figure B-2. Color Map File Using Hex Numbers

HP BASIC/UX is one application that uses a local color map and recommends that you pre-allocate the HP BASIC/UX colors at startup using the `xinitcolormap` command (refer to the `/usr/lib/rmb/newconfig/rgb.README` or `/opt/rmb/newconfig/rgb.README` file).

Because of this, if you will use HP VEE with HP BASIC/UX (or other applications that allocate colors in the same way HP BASIC/UX does), you need to also pre-allocate HP VEE colors at startup. If you do not, you may see the colors flash on the screen as you move from one window to another.

To do this:

1. Create a "colormap" file that contains all the different HP VEE colors you will use.
2. Change to your `$HOME` directory:

   ```
   cd $HOME
   ```

3. Concatenate the HP BASIC/UX and the HP VEE colormap files:

```
cat /usr/lib/rmb/newconfig/xrmbcolormap vee-colormapfile > .xveecolormap
```

   - or -

```
cat /opt/rmb/newconfig/xrmbcolormap vee-colormapfile > .xveecolormap
```

   Note that the HP BASIC/UX colors must go first, because HP BASIC/UX assumes that they are the first 16 entries in the colormap. You can mix the word colors and the hex number colors in one file.

4. You must use the `xinitcolormap` command before you allocate any colors for other applications. This means that it should be placed near the beginning of your .x11start file.

   For example, if you use the `.x11start` file, your colors are in `$HOME/.xveecolormap`, and you have 55 colors listed in the file (16 from HP BASIC/UX + 39 from HP VEE), you would add the following line to `.x11start`:

   ```
   /usr/bin/X11/xinitcolormap -c 55 -f $HOME/.xveecolormap
   ```

   - or -

   ```
   /opt/X11/xinitcolormap -c 55 -f $HOME/.xveecolormap
   ```

5. Restart X11. To do this, stop the window manager by pressing the following three keys at the same time: Shift-CTRL-Break, or selecting `Reset` from your root menu (if it is configured for this choice), then type:

   ```
   x11start
   ```

Using Non-USASCII Keyboards (UNIX)

If you are using a non-USASCII keyboard, you need to modify the $LANG variable in your X11 environment. As an example, to use a German keyboard, use the command `export LANG=german.iso88591` in the Korn Shell. Once the LANG variable is set, use `File` ⟹ `Default Preferences` to change fonts.

Note

If you are accessing data that was created with the `Roman8` character set, you must translate any special characters (above ASCII 127) used.

Your terminal window may use `Roman8`; therefore `TEXT` written to stdout, file names (such as specified by `To File` and `From File`), and programs names must use ASCII characters 0-127 to match with those specified with HP VEE.

Using Two-Byte Character Sets (UNIX)

Two-byte characters (such as Kanji for Japanese) can be entered into any field in HP VEE where you can enter text. This includes all titles, text (string) constants, input/output pin names, and note pads. To use two-byte characters you must have the UNIX NLIO subsystem installed and initialized, and set the $LANG shell variable to your local language.

For Japanese, two character sets are available—`japanese` and `japanese.euc`. They are entered on the keyboard and encoded differently, and are different widths on the monitor screen. You may want to try each one, then pick the one you prefer. You can set the $LANG shell variable using the command `export LANG=japanese`.

When HP VEE starts running, if the $LANG shell variable is set correctly, HP VEE will initialize its fonts to two-byte fonts. The two-byte fonts can be replaced later by the default fonts saved in your `.veerc` file.

Refer to "Changing X11 Attributes" at the beginning of this appendix for further information.

Note

The Kanji app-defaults file specifies a two-byte "stroke" font. A stroke font consists of characters drawn as strokes, rather than as a raster image. The stroke font is required for output to most plotters, since most plotters do not directly support Kanji.

The following are some limitations to two-byte character support:

- When using the `Plot` command to send a graphical two-dimensional display (e.g., `XY Trace`) to a plotter or file, you must first specify that labels are to be output using the two-byte "stroke" font. To do this, go to the `Plotter Configuration` dialog box:

  `File` ⟹ `Default Preferences`, then select `Plotter Setup` on the `Printing` tab.

  The `Label Using` field gives you two choices: `Stroke Fonts` and `Plotter ROM`. You must select `Stroke Fonts` in order to output a display with two-byte characters. Otherwise, all two-byte characters in field labels will be encoded into HP-GL label commands as two-byte characters in the HP15 character set, which is not supported by most plotters.

  For further information, refer to the `Plotter Config` section in the `Reference` in HP VEE online help.

- When reading text that includes two-byte characters from a `Direct I/O` object, the two-byte character rules are not used when looking for the EOL string. Thus, an EOL character may be incorrectly found in the second byte of a two-byte character. (This is only a problem if an EOL character has an ASCII value greater than 32 decimal.)
- Date/Time parsing and formatting have not been globalized, and continue to only execute in English. To obtain a localized date string, use an `Execute Program` object with a program of "date" and a transaction of `READ TEXT x STR`.

Using HP-GL Plotters (UNIX)

HP VEE supports graphics output to plotters and files using HP-GL. Before you can send plots to a plotter (either local or networked) your system administrator must add the plotter as a spooled device on your system.

In addition to standard HP-GL plotters such as the HP 7475, the HP ColorPro (HP 7440), or the HP 7550, some printers can be used as plotters, such as the PaintJet XL, and the LaserJet III. The HP ColorPro plotter requires the Graphics Enhancement Cartridge in order to plot polar or Smith Chart graticules, or an Area-Fill line type. The PaintJet XL requires the HP-GL/2 Cartridge in order to make any plots. In order to make plots on the LaserJet III, at least two megabytes of optional memory expansion is required, and the Page Protection configuration option should be enabled. Plots of many vectors, especially with Polar or Smith chart graticules, may require even more optional memory in the LaserJet III. Any plot intended for a printer requires the plotter type to be set to HP-GL/2, which causes the proper HP-GL/2 setup sequence to be included with the plot information.

Any of the following graphical two-dimensional displays can be plotted to an HP-GL or HP-GL/2 plotter, or to a file:

- `XY Trace`
- `Strip Chart`
- `Complex Plane`
- `X vs Y Plot`
- `Polar Plot`
- `Waveform`
- `Magnitude Spectrum`
- `Phase Spectrum`

- Magnitude vs Phase

You can specify the appropriate default plotter configuration by selecting: File ⟹ Default Preferences. Then use the Printing tab in the Default Preferences dialog box; click on the Plotter Setup button to edit the Plotter Configuration dialog box.

To generate a plot directly from a display object, just select Plot on the display's object menu, specify the required parameters in the Plotter Configuration dialog box, and then press OK. You can also add Plot as a control input to generate plots programmatically. The entire view of the display object will be plotted, and scaled to fill the defined plotting area, while retaining the aspect ratio of the original display object. By re-sizing the display object, you can control the aspect ratio of the plotted image. By making the display object larger, you can reduce the relative size of the text and numeric labels around the plot.

For an explanation of the plotter configuration parameters in the Plotter Configuration dialog box, refer to the Default Preferences section in the Reference in HP VEE online help Also, refer to the reference sections for the appropriate two-dimensional display devices.

C

ASCII Table

This appendix contains reference tables of ASCII 7-bit codes.

ASCII 7-bit Codes

	Binary	Oct	Hex	Dec	HP-IB Msg
NUL	0000000	000	00	0	
SOH	0000001	001	01	1	GTL
STX	0000010	002	02	2	
ETX	0000011	003	03	3	
EOT	0000100	004	04	4	SDC
ENQ	0000101	005	05	5	PPC
ACK	0000110	006	06	6	
BEL	0000111	007	07	7	
BS	0001000	010	08	8	GET
HT	0001001	011	09	9	TCT
LF	0001010	012	0A	10	
VT	0001011	013	0B	11	
FF	0001100	014	0C	12	
CR	0001101	015	0D	13	
SO	0001110	016	0E	14	
SI	0001111	017	0F	15	
DLE	0010000	020	10	16	
DC1	0010001	021	11	17	LLO
DC2	0010010	022	12	18	
DC3	0010011	023	13	19	
DC4	0010100	024	14	20	DCL
NAK	0010101	025	15	21	PPU
SYN	0010110	026	16	22	
ETB	0010111	027	17	23	

ASCII 7-bit Codes (continued)

	Binary	Oct	Hex	Dec	HP-IB Msg
CAN	0011000	030	18	24	SPE
EM	0011001	031	19	25	SPD
SUB	0011010	032	1A	26	
ESC	0011011	033	1B	27	
FS	0011100	034	1C	28	
GS	0011101	035	1D	29	
RS	0011110	036	1E	30	
US	0011111	037	1F	31	
space	0100000	040	20	32	listen addr 0
!	0100001	041	21	33	listen addr 1
"	0100010	042	22	34	listen addr 2
#	0100011	043	23	35	listen addr 3
$	0100100	044	24	36	listen addr 4
%	0100101	045	25	37	listen addr 5
&	0100110	046	26	38	listen addr 6
'	0100111	047	27	39	listen addr 7
(	0101000	050	28	40	listen addr 8
)	0101001	051	29	41	listen addr 9
*	0101010	052	2A	42	listen addr 10
+	0101011	053	2B	43	listen addr 11
,	0101100	054	2C	44	listen addr 12
-	0101101	055	2D	45	listen addr 13
.	0101110	056	2E	46	listen addr 14
/	0101111	057	2F	47	listen addr 15

ASCII 7-bit Codes (continued)

	Binary	Oct	Hex	Dec	HP-IB Msg
0	0110000	060	30	48	listen addr 16
1	0110001	061	31	49	listen addr 17
2	0110010	062	32	50	listen addr 18
3	0110011	063	33	51	listen addr 19
4	0110100	064	34	52	listen addr 20
5	0110101	065	35	53	listen addr 21
6	0110110	066	36	54	listen addr 22
7	0110111	067	37	55	listen addr 23
8	0111000	070	38	56	listen addr 24
9	0111001	071	39	57	listen addr 25
:	0111010	072	3A	58	listen addr 26
;	0111011	073	3B	59	listen addr 27
<	0111100	074	3C	60	listen addr 28
=	0111101	075	3D	61	listen addr 29
>	0111110	076	3E	62	listen addr 30
?	0111111	077	3F	63	UNL
@	1000000	100	40	64	talk addr 0
A	1000001	101	41	65	talk addr 1
B	1000010	102	42	66	talk addr 2
C	1000011	103	43	67	talk addr 3
D	1000100	104	44	68	talk addr 4
E	1000101	105	45	69	talk addr 5
F	1000110	106	46	70	talk addr 6
G	1000111	107	47	71	talk addr 7

ASCII 7-bit Codes (continued)

	Binary	Oct	Hex	Dec	HP-IB Msg
H	1001000	110	48	72	talk addr 8
I	1001001	111	49	73	talk addr 9
J	1001010	112	4A	74	talk addr 10
K	1001011	113	4B	75	talk addr 11
L	1001100	114	4C	76	talk addr 12
M	1001101	115	4D	77	talk addr 13
N	1001110	116	4E	78	talk addr 14
O	1001111	117	4F	79	talk addr 15
P	1010000	120	50	80	talk addr 16
Q	1010001	121	51	81	talk addr 17
R	1010010	122	52	82	talk addr 18
S	1010011	123	53	83	talk addr 19
T	1010100	124	54	84	talk addr 20
U	1010101	125	55	85	talk addr 21
V	1010110	126	56	86	talk addr 22
W	1010111	127	57	87	talk addr 23
X	1011000	130	58	88	talk addr 24
Y	1011001	131	59	89	talk addr 25
Z	1011010	132	5A	90	talk addr 26
[	1011011	133	5B	91	talk addr 27
\	1011100	134	5C	92	talk addr 28
]	1011101	135	5D	93	talk addr 29
^	1011110	136	5E	94	talk addr 30
_	1011111	137	5F	95	UNT

ASCII 7-bit Codes (continued)

	Binary	Oct	Hex	Dec	HP-IB Msg
‘	1100000	140	60	96	secondary addr 0
a	1100001	141	61	97	secondary addr 1
b	1100010	142	62	98	secondary addr 2
c	1100011	143	63	99	secondary addr 3
d	1100100	144	64	100	secondary addr 4
e	1100101	145	65	101	secondary addr 5
f	1100110	146	66	102	secondary addr 6
g	1100111	147	67	103	secondary addr 7
h	1101000	150	68	104	secondary addr 8
i	1101001	151	69	105	secondary addr 9
j	1101010	152	6A	106	secondary addr 10
k	1101011	153	6B	107	secondary addr 11
l	1101100	154	6C	108	secondary addr 12
m	1101101	155	6D	109	secondary addr 13
n	1101110	156	6E	110	secondary addr 14
o	1101111	157	6F	111	secondary addr 15

ASCII 7-bit Codes (continued)

	Binary	Oct	Hex	Dec	HP-IB Msg
p	1110000	160	70	112	secondary addr 16
q	1110001	161	71	113	secondary addr 17
r	1110010	162	72	114	secondary addr 18
s	1110011	163	73	115	secondary addr 19
t	1110100	164	74	116	secondary addr 20
u	1110101	165	75	117	secondary addr 21
v	1110110	166	76	118	secondary addr 22
w	1110111	167	77	119	secondary addr 23
x	1111000	170	78	120	secondary addr 24
y	1111001	171	79	121	secondary addr 25
z	1111010	172	7A	122	secondary addr 26
{	1111011	173	7B	123	secondary addr 27
|	1111100	174	7C	124	secondary addr 28
}	1111101	175	7D	125	secondary addr 29
~	1111110	176	7E	126	secondary addr 30
[del]	1111111	177	7F	127	

D

I/O Transaction Reference

This appendix contains details about the behavior of all I/O transaction actions, encodings, and formats. For generaly information about using transactions for instrument I/O, refer to *Controlling Instruments with HP VEE.* This appendix is organized by the transaction actions summarized in Table D-1. For example, if you need detailed information about `TEXT` encoding, do this:

- Look in the `WRITE` section for details about `WRITE TEXT` transactions.
- Look in the `READ` section for details about `READ TEXT` transactions.

Table D-1. Summary of Transaction Types

Action	Description
`WRITE`	Writes data to the destination specified in the object.
`READ`	Reads data from the source specified in the object.
`EXECUTE`	Executes low-level commands to control the file, device, or interface associated with the object. `EXECUTE` is used to adjust file pointers, to close pipes and files, and to provide low-level control of devices and hardware interfaces.
`WAIT`	Waits for the specified number of seconds before executing the next transaction. For `Direct I/O` objects, `WAIT` can also wait for a specific serial poll response, or for specific values in accessible VXI device registers.
`SEND`	Sends IEEE 488-defined bus messages (bus commands and data) to an HP-IB interface.
`READ(REQUEST)`[1]	Reads DDE data from another application.
`WRITE(POKE)`[1]	Writes DDE data to another application.

1 HP VEE for Windows only.

Table D-2. Summary of I/O Transaction Objects

Objects	Supported Transactions				
	EXECUTE	WAIT	READ	WRITE	SEND
To File	X	X		X	
From File	X	X	X		
To Printer		X		X	
To String		X		X	
From String		X	X		
To StdOut		X		X	
From StdIn		X	X		
To StdErr		X		X	
Execute Program (UNIX)[1]	X	X	X	X	
To/From Named Pipe	X	X	X	X	
To/From Socket	X	X	X	X	
Direct I/O	X	X	X	X	
MultiDevice Direct I/O	X	X	X	X	
Interface Operations	X				X
To/From HP BASIC/UX[2]	X	X	X	X	
To/From DDE[3]	X	X	X	X	

1 Execute Program (PC) is not transaction based.

2 HP VEE for HP-UX only.

3 HP VEE for Windows only.

WRITE Transactions

This section is organized by the `WRITE` encodings summarized in Table D-3. Topics that apply to all `WRITE` encodings are summarized at the beginning of this section.

Path-Specific Behaviors

Some `WRITE` transactions behave differently depending on the I/O path of the destination. For example, `WRITE TEXT HEX` transactions format hexadecimal numbers differently depending on whether the destination is a UNIX file or an instrument. To distinguish these behaviors, this section uses the following terms:

Term	**Meaning**
UNIX paths	Any destination other than an instrument, such as a UNIX file, a string, the printer, or a UNIX pipe.
MS-DOS paths	Any destination other than an instrument, such as an MS-DOS file, a string, or the printer.
direct I/O paths	Any instrument accessed using `Direct I/O`.

The behaviors described in the following sections apply to all paths, except as specifically noted.

Table D-3. `WRITE` Encodings and Formats

Encodings	Formats

`TEXT`	`DEFAULT`
	`STRING`
	`QUOTED STRING`
	`INTEGER`
	`OCTAL`
	`HEX`
	`REAL`
	`COMPLEX`
	`PCOMPLEX`
	`COORD`
	`TIME STAMP`

`BYTE`	Not Applicable
`CASE`	Not Applicable

`BINARY`	`STRING`
	`BYTE`
	`INT16`
	`INT32`
	`REAL32`
	`REAL64`
	`COMPLEX`
	`PCOMPLEX`
	`COORD`

`BINBLOCK`	`BYTE`
	`INT16`
	`COMPLEX`
	`INT32`
	`PCOMPLEX`
	`REAL32`
	`REAL64`
	`COORD`

Table D-3. `WRITE` Encodings and Formats (continued)

Encodings	Formats
`CONTAINER`	Not Applicable
`STATE`[1]	Not Applicable

`REGISTER`[2]	`BYTE`
	`WORD16`
	`WORD32`
	`REAL32`

`MEMORY`[2]	`BYTE`
	`WORD16`
	`WORD32`
	`REAL32`

`IOCONTROL`[3]	Not Applicable

1 Direct I/O to HP-IB only.

2 Direct I/O to VXI only.

3 Direct I/O to GPIO only.

TEXT Encoding

`WRITE TEXT` transactions are of this form:

`WRITE TEXT` *ExpressionList [Format]*

ExpressionList is a single expression or a comma-separated list of expressions.

Format is an optional setting that specifies one of the formats listed in Table D-4.

Table D-4. Formats for `WRITE TEXT` Transactions

Format	Description
`DEFAULT`	HP VEE automatically determines an appropriate text representation based on the data type of the item being written.
`STRING`	Writes Text data without any conversion. Writes numeric data types as Text with maximum numeric precision.
`QUOTED STRING`	Writes data in the the same format as `STRING`, except the data is surrounded by double quotes (ASCII 34 decimal).
`INTEGER`	Writes data as a 32-bit two's complement integer in decimal form.
`OCTAL`	Writes data as a 32-bit two's complement integer in octal form.
`HEX`	Writes data as a 32-bit two's complement integer in hexadecimal form.
`REAL`	Writes data as a 64-bit floating point number in a variety of notations including fixed decimal and scientific notation.
`COMPLEX`	Writes a comma-separated pair of 64-bit floating point numbers that represent a complex number. The first number represents the real part and the second number represents the imaginary part.
`PCOMPLEX`	Writes a comma-separated pair of 64-bit floating point numbers that represent a complex number. The first number represents the magnitude and the second number represents the phase angle in the phase units specified in the transaction.
`COORD`	Writes a comma-separated series of 64-bit floating point numbers that represent a rectangular coordinate.
`TIME STAMP`	Converts a real number (for example, the output of the `now()` function) to a meaningful form and writes it in a variety of combinations of year, month, day, and time.

DEFAULT Format

`WRITE TEXT` (default) transactions are of this form:

`WRITE TEXT` *ExpressionList*

ExpressionList is a single expression or a comma-separated list of expressions.

The transaction converts each item in *ExpressionList* to a meaningful string and writes it. Consider the simple case of writing the scalar variable `X`:

```
WRITE TEXT X
```

Figure D-1. A `WRITE TEXT` Transaction

If `X` in Figure D-1 contains text, such as:

```
bird cat dog
```

then no conversion is performed and the transaction writes exactly 12 characters.

If `X` in Figure D-1 contains a scalar Integer, such as:

`8923` *the value o fX(decimal notation)*

then the numeric value is converted to text and HP VEE writes exactly four characters.

If `X` in Figure D-2 contains a scalar real value, such as:

```
1.2345678901234567
```

Figure D-2. Numeric Data

then each significant digit up to 16 significant digits is written. (The least significant digit is approximate because of the conversion between HP VEE's internal binary form and decimal notation).

For example, if you write the data in Figure D-2 using this transaction:

```
WRITE TEXT a EOL
```

then HP VEE writes this:

```
1.234567890123457
```

If the absolute value of the number is sufficiently large or small, exponential notation is used. The Reals that form the sub-elements of Coord, Complex, and PComplex behave the same way.

If `EOL ON` is specified for any `WRITE TEXT DEFAULT` transaction, the character specified in the `EOL Sequence` field for that object is written following the last character in *ExpressionList*.

STRING Format

`WRITE TEXT STRING` transactions are of this form:

`WRITE TEXT` *ExpressionList* `STR`

ExpressionList is a single expression or a comma-separated list of expressions.

`WRITE TEXT STRING` transactions behave basically the same as `WRITE TEXT` (default) transactions (one exception will be discussed). The significant difference is that `STRING` allows you to specify additional details about output formatting including field width, justification, and number of characters.

Field Width and Justification. If a transaction specifies `DEFAULT FIELD WIDTH`, only those characters resulting from the conversion of items within *ExpressionList* to Text are written.

If a transaction specifies `FIELD WIDTH:` *F*, then the converted Text is written right- or left-justified within a space *F* characters wide.

The transactions in Figure D-3 specify that all characters are to be written within a field of twenty characters with left justification.

```
WRITE TEXT X STR FW:20 LJ EOL
WRITE TEXT Y STR FW:20 LJ EOL
```

Figure D-3. Two `WRITE TEXT STRING` **Transactions**

If `X` and `Y` in Figure D-3 have these values:

`bird cat dog`	*the Text value o fX*
`12345678901234567`	*the Real value o fY*

then HP VEE writes this:

```
bird cat dog
12345678901234567
^                  ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the right of `dog` and to the right of the second 7 are spaces (ASCII 32 decimal).

If justification is changed to `RIGHT JUSTIFY`, then the transactions appear as shown in Figure D-4.

```
WRITE TEXT X STR FW:20 RJ EOL
WRITE TEXT Y STR FW:20 RJ EOL
```

Figure D-4. Two `WRITE TEXT STRING` Transactions

If `X` and `Y` in Figure D-4 have these values:

`bird cat dog`	*the Text value o fX*
`12345678901234567`	*the Real value o fY*

then HP VEE writes this:

```
        bird cat dog
   12345678901234567
^                  ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the left of `bird` and to the left of the first `1` are spaces (ASCII 32 decimal).

If the length of a string exceeds the specified field width, the entire string is written. The field width specification never truncates; only `MAX NUM CHARS` can truncate characters.

The transaction in Figure D-5 specifies that all characters are to be written in a field width of four characters with left justification.

```
WRITE TEXT X STR FW:4 LJ
```

Figure D-5. A `WRITE TEXT STRING` **Transaction**

If `X` in Figure D-5 has this value:

`bird cat dog`   *the Text value of X, 12 characters*

then HP VEE writes this:

`bird cat dog`   *all 12 characters*

Even though the specified field width is four characters, the transaction writes all twelve characters of the string.

Number of Characters. If you specify `ALL CHARS`, then all of the characters generated by the conversion of each item in *ExpressionList* are written. If you specify `MAX NUM CHARS:` *M*, then only the first *M* characters of each item in *ExpressionList* are written.

The transactions in Figure D-6 specify that a maximum of seven characters are written in each field, the field width is twenty characters, and field entries are left justified.

```
WRITE TEXT X STR:7 FW:20 LJ EOL
WRITE TEXT Y STR:7 FW:20 LJ EOL
```

Figure D-6. Two `WRITE TEXT STRING` **Transactions**

If X and Y in Figure D-3 have these values:

```
bird cat dog            the Text value of X
12345678901234567       the Real value of Y
```

then HP VEE writes this:

```
bird ca
1234567
^                        ^
```

Notice that the numeric value of Y is first converted to Text and characters are truncated. Numeric values are not rounded by MAX NUM CHARS.

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the right of bird and to the right of the first 1 are spaces (ASCII 32 decimal).

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

Note

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

QUOTED STRING Format

`WRITE TEXT QUOTED STRING` transactions are of this form:

`WRITE TEXT` *ExpressionList* `QSTR`

ExpressionList is a single expression or a comma-separated list of expressions.

In general, the behaviors previously discussed for the `STRING`

format apply to `QUOTED STRING` format. There are two differences between `STRING` and `QUOTED STRING`:

- For `QUOTED STRING`, a double quote (ASCII 34 decimal) is added to the beginning and the end of the string. Note that the double quotes are applied before any padding spaces are added to justify the string within the specified field width.
- Control characters (ASCII 0-31 decimal), escape characters (Table D-5), and the characters ' (ASCII 39 decimal) and " (ASCII 34 decimal) embedded inside a double-quoted string receive special treatment.

Field Width and Justification. If you specify `DEFAULT FIELD WIDTH`, only those characters resulting from the conversion of items within *ExpressionList* to Text and the surrounding double quotes are written.

If you specify `FIELD WIDTH:` *F*, then the converted Text and the surrounding quotes are written right or left justified within a space *F* characters wide.

The transactions in Figure D-7 specify that all characters are to be written as quoted strings in a field 20 characters wide with left justification.

```
WRITE TEXT X QSTR FW:20 LJ EOL
WRITE TEXT Y QSTR FW:20 LJ EOL
```

Figure D-7. Two `WRITE TEXT QUOTED STRING` **Transactions**

If X and Y in Figure D-7 have these values:

```
bird cat dog          the Text value of X
12345678901234567     the Real value of Y
```

then HP VEE writes this:

```
"bird cat dog"
"12345678901234567"
^                    ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the right of dog" and to the right of 7" are spaces (ASCII 32 decimal).

If justification is changed to RIGHT JUSTIFY, then the transactions appear as shown in Figure D-8.

```
WRITE TEXT X QSTR FW:20 RJ EOL
WRITE TEXT Y QSTR FW:20 RJ EOL
```

Figure D-8. Two WRITE TEXT QUOTED STRING **Transactions**

If X and Y in Figure D-8 have these values:

```
bird cat dog          the Text value of X
12345678901234567     the Real value of Y
```

then HP VEE writes this:

```
      "bird cat dog"
 "12345678901234567"
^                    ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the left of "bird and to the left of "1 are spaces (ASCII 32 decimal).

If the length of a string exceeds the specified field width, the entire string is output. The field width specification never truncates strings that are written; only `MAX NUM CHARS` can truncate characters.

The transactions in Figure D-9 that specifies that all characters are to be written within a field of four characters with left justification.

```
   WRITE TEXT X QSTR FW:4 LJ
```

Figure D-9. A `WRITE TEXT QUOTED STRING` Transaction

If `X` in Figure D-9 has this value:

`bird cat dog`   *the Text value of X, 12 characters*

then HP VEE writes this:

`"bird cat dog"`   *all 12 characters*

Number of Characters. If you specify `ALL CHARS`, then all of the characters generated by the conversion of each item in *ExpressionList* as well as the surrounding double quotes are written. If you specify `MAX NUM CHARS:` *M*, then only the first *M* characters of each item in *ExpressionList plus* the surrounding double quotes are written. In other words, a total of *M*+2 characters are written for each item in *ExpressionList*.

The transaction in Figure D-10 that specifies `MAX NUM CHARS:7` (field width 20, left justified).

```
   WRITE TEXT X QSTR:7 FW:20 LJ EOL
   WRITE TEXT Y QSTR:7 FW:20 LJ EOL
```

Figure D-10. Two `WRITE TEXT QUOTED STRING` Transactions

If X and Y in Figure D-10 have these values:

```
bird cat dog          the Text value of X
12345678901234567     the Real value of Y
```

then HP VEE writes this:

```
"bird ca"
"1234567"
^                    ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the right of `ca"` and to the right of `7"` are spaces (ASCII 32 decimal).

Embedded Control and Escape Characters. In this discussion, the terms **control character** and **escape character** have specific meaning. A control character is a single byte of data corresponding to one of the ASCII characters 0-31 decimal. For example, linefeed is ASCII 10 decimal and the symbol `<LF>` denotes linefeed character in this discussion. The string `\n` is a human-readable escape character representing linefeed that is recognized by HP VEE. HP VEE uses escape characters to represent control characters within quoted strings.

Table D-5. Escape Characters

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

Consider the effects of various embedded escape characters on the transaction in Figure D-11.

```
WRITE TEXT X QSTR EOL
```

Figure D-11. A `WRITE TEXT QUOTED STRING` **Transaction**

If `X` in Figure D-11 has this value:

```
bird\ncat dog
```

then HP VEE writes this to UNIX paths:

```
"bird\ncat dog"
```

For the same transaction and data, HP VEE writes this to direct I/O paths:

```
"bird<LF>cat dog"
```

Note that `<LF>` means the single character, linefeed (ASCII 10 decimal).

If `X` in Figure D-11 has this value:

```
bird \"cat\" dog
```

then HP VEE writes this to UNIX paths and Direct I/O paths for serial interfaces:

```
"bird \"cat\" dog"
```

For the same transaction and data, HP VEE writes this to direct I/O paths for HP-IB interfaces:

```
"bird ""cat"" dog"
```

This unique behavior for HP-IB interfaces is provided to support the requirements of IEEE 488.2.

INTEGER Format

`WRITE TEXT INTEGER` transactions are of this form:

`WRITE TEXT` *ExpressionList* `INT`

ExpressionList is a single expression or a comma-separated list of expressions.

The type of integer generated by this transaction is a 32-bit two's complement integer. The range of these integers is `2 147 483 647` to `-2 147 483 648`. The only characters written to represent these numbers are `+-0123456789`.

HP VEE attempts to convert each item in *ExpressionList* to the Int32 data type before converting it to Text for final formatting. HP VEE follows the usual conversion rules; refer to the `Data Type Conversions` appendix in the `Reference` in HP VEE online help for more details.

If a Real is written using `INTEGER` format:

- Real values outside the valid range of Int32 generate an error.
- Real values within the valid range of Int32 are converted by truncating the fractional portion of the Real.

Number of Digits. If you specify `DEFAULT NUM DIGITS`, the transaction writes only the digits required to express the value of the integer; leading zeros are not used.

If you specify `MIN NUM DIGITS:` *M*, the transaction pads the output with leading zeros as required to give a total of exactly *M* digits.

Consider the two transactions in Figure D-12 which differ only in their specification for the number of output digits.

```
WRITE TEXT X INT EOL      default number of digits
WRITE TEXT X INT:6 EOL    six digits
```

Figure D-12. Two `WRITE TEXT INTEGER` Transactions

If `X` in Figure D-12 has this value:

```
4567
```

then HP VEE writes this:

```
4567
004567
```

`MIN NUM DIGITS` never causes truncation of the output string. The transaction in Figure D-13 specifies the minimum number of digits to be 1.

```
WRITE TEXT X INT:1 EOL
```

Figure D-13. A `WRITE TEXT INTEGER` Transaction

If `X` in Figure D-13 has a value of:

```
12345678
```

then HP VEE writes this:

```
12345678    all eight digits
```

Sign Prefixes. You may optionally specify one of the sign prefixes listed in Table D-6 as part of a WRITE TEXT INT transaction.

Table D-6. Sign Prefixes

Prefix	Description
`/-`	Positive numbers are written with no prefix, neither a + nor a space. All negative numbers are written with a - prefix.
`+/-`	All positive numbers are written with a + prefix. All negative numbers are written with a - prefix.
`" "/-`	All positive numbers are written with a space (ASCII 32 decimal) prefix. All negative numbers are written with a - prefix.

Any prefixed signs do not count towards MIN NUM DIGITS. The transaction shown in Figure D-14 specifies explicit leading signs for positive and negative numbers.

```
WRITE TEXT X INT:6 SIGN:"+/-" EOL
WRITE TEXT Y INT:6 SIGN:"+/-" EOL
```

Figure D-14. Two WRITE TEXT INTEGER **Transactions**

If X and Y in Figure D-14 have values of:

```
123    the Integer value o fX
-123   the Integer value o fY
```

then HP VEE writes this:

```
+000123    six digits plus sign
-000123
```

OCTAL Format

`WRITE TEXT OCTAL` transactions are of this form:

```
WRITE TEXT ExpressionList OCT
```

ExpressionList is a single expression or a comma-separated list of expressions.

The type of integer written by this transaction is a 32-bit two's complement integer. The range of these integers is `2 147 483 647` to `-2 147 483 648`. The only characters written to represent these octal numbers are `01234567`. An optional prefix may be specified which may include other characters.

HP VEE attempts to convert any data written using `OCTAL` format to the Int32 data type before converting it to Text for final formatting. The usual HP VEE conversion rules are followed.

If a Real is written using `OCTAL` format:

- Real values outside the valid range of Int32 generate an error.
- Real values within the valid range of Int32 are converted by truncating the fractional portion of the Real.

Number of Digits. The behavior of `DEFAULT NUM DIGITS` and `MIN NUM DIGITS` is the same as described previously in the "Number of Digits" section for `WRITE TEXT INTEGER` transactions.

Octal Prefixes. You may specify one of the prefixes listed in Table D-7 as part of a `WRITE TEXT OCTAL` transaction.

Table D-7. Octal Prefixes

Prefix	Description
`NO PREFIX`	HP VEE writes each octal number without any prefix; only the digits `01234567` appear in the output.
`DEFAULT PREFIX`	For direct I/O paths, HP VEE prefixes each octal number with `#Q`. This supports the octal Non-Decimal Numeric data format defined by IEEE 488.2. For UNIX paths, HP VEE prefixes each octal number with a `0` (zero). If leading zeros are added to achieve the specified `MIN NUM DIGITS`, `DEFAULT PREFIX` will not add additional leading zeros.
`PREFIX:`*string*	HP VEE prefixes each octal number with the characters specified in *string*.

The transaction in Figure D-15 specifies the default prefix and six digits:

```
WRITE TEXT X OCT:6 PREFIX EOL
```

Figure D-15. A `WRITE TEXT OCTAL` Transaction

If `X` in Figure D-15 has this value:

`15` *the value 15 decimal*

then HP VEE writes this to direct I/O paths:

`#Q000017` *exactly six digits plus prefix*

Using the same transaction and data, HP VEE writes this to UNIX paths:

`000017` *exactly six digits*

The transaction in Figure D-16 specifies a custom prefix and ten digits:

```
WRITE TEXT X OCT:10 PREFIX:"oct>" EOL
```

Figure D-16. A `WRITE TEXT OCTAL` **Transaction**

If `X` in Figure D-16 has this value:

`15` *the Integer value 15 decimal*

then HP VEE writes this to UNIX paths and direct I/O paths:

```
oct>000017
```

Note that the prefix written by `DEFAULT PREFIX` depends on the destination, but the prefix written by `PREFIX:` *string* is independent of the destination.

HEX Format

`WRITE TEXT HEX` transactions are of this form:

`WRITE TEXT` *ExpressionList* `HEX`

The type of integer written by this transaction is a 32-bit two's complement integer. The range of these integers is 2 147 483 647 to -2 147 483 648.
The only characters written to represent these hexadecimal numbers are `0123456789abcdef`. An optional prefix may be specified that may include other characters.

The behavior of `WRITE TEXT HEX` is nearly identical to that of `WRITE TEXT OCTAL`. The only difference is the set of prefixes available and the behavior of `DEFAULT PREFIX`.

Hexadecimal Prefixes. You may specify one of the prefixes listed in Table D-8 as part of a `WRITE TEXT HEX` transaction.

Table D-8. Hexadecimal Prefixes

Prefix	Description
`NO PREFIX`	HP VEE writes each hexadecimal number without any prefix; only the digits `0123456789abcdef` appear in the output.
`DEFAULT PREFIX`	For direct I/O paths, HP VEE prefixes each hexadecimal number with `#H`. This supports the hexadecimal Non-Decimal Numeric data format defined by IEEE 488.2. For UNIX paths, HP VEE prefixes each hexadecimal number with `0x`.
`PREFIX:`*string*	HP VEE prefixes each hexadecimal number with the characters specified in *string*.

The transaction in Figure D-17 specifies the default prefix and six digits:

```
WRITE TEXT X HEX:6 PREFIX EOL
```

Figure D-17. A `WRITE TEXT HEX` Transaction

If `X` in Figure D-15 has this value:

`15` *the Integer value 15 decimal*

then HP VEE writes this to direct I/O paths:

`#H00000f` *exactly six digits plus prefix*

Using the same transaction and data, HP VEE this to UNIX paths:

`0x00000f` *exactly six digits plus prefix*

The transaction in Figure D-18 specifies a custom prefix and three digits:

```
WRITE TEXT X HEX:3 PREFIX:"hex>" EOL
```

Figure D-18. A `WRITE TEXT HEX` Transaction

If `X` in Figure D-18 has this value:

`15` *the Integer value 15 decimal*

then HP VEE writes this to UNIX paths and direct I/O paths:

`hex>00f` *exactly three digits plus prefix*

Note that the prefix written by `DEFAULT PREFIX` depends on the destination, but the prefix written by `PREFIX:` *string* is independent of the destination.

REAL Format

`WRITE TEXT REAL` transactions are of this form:

`WRITE TEXT` *ExpressionList* `REAL`

The type of Real number generated by this transaction is a 64-bit IEEE 754 floating-point number. The range of these numbers is:

```
-1.797 693 134 862 315E+308
-2.225 073 858 507 202E-307
 0
 2.225 073 858 507 202E-307
 1.797 693 134 862 315E+308
```

The only characters written to represent these numbers are `+-.0123456789E`.

Notations and Digits. You may optionally specify one of the notations in Table D-9 as part of a `WRITE TEXT REAL` transaction.

Table D-9. `REAL` Notations

Notation	Description
`STANDARD`	HP VEE automatically determines whether each Real value should be written in fixed-point notation (decimal points as required, no exponents) or in exponential notation. Non-significant zeros are never written.
`FIXED`	HP VEE writes each Real value as a fixed-point number. Numbers with fractional digits are automatically rounded to fit the number of fractional digits specified by `NUM FRACT DIGITS`. Trailing zero digits are added as required to give the specified number of fractional digits.
`SCIENTIFIC`	HP VEE writes each Real value using exponential notation. Each exponent includes an explicit sign (`+` or `-`) and the upper-case `E` is always used. Numbers with fractional digits are automatically rounded to fit the number of fractional digits specified by `NUM FRACT DIGITS`. Trailing zero digits are added as required to give the specified number of fractional digits.

The transactions in Figure D-19 specify `STANDARD` notation and four significant digits.

```
WRITE TEXT X REAL STD:4 EOL
WRITE TEXT Y REAL STD:4 EOL
WRITE TEXT Z REAL STD:4 EOL
```

Figure D-19. Three `WRITE TEXT REAL` Transactions

If X, Y, and Z in Figure D-19 have these values:

`1.23456E2`	*the Real value o fX*
`1.23456E09`	*the Real value o fY*
`1.23`	*the Real value o fZ*

then HP VEE writes this:

`123.5`	*mantissa rounded as required*
`1.235E+09`	*large numbers in exponential notation*
`1.23`	*never any trailing zeros*

The transactions in Figure D-20 specify `FIXED` notation and four fractional digits.

```
WRITE TEXT X REAL FIX:4 EOL
WRITE TEXT Y REAL FIX:4 EOL
WRITE TEXT Z REAL FIX:4 EOL
```

Figure D-20. Three `WRITE TEXT REAL` **Transactions**

If X, Y, and Z in Figure D-20 have these values:

`1.2345678E2`	*the Real value o fX*
`1.2345678E-09`	*the Real value o fY*
`1.23`	*the Real value o fZ*

then HP VEE writes this:

`123.4568`	*mantissa rounded as required*
`0.0000`	*small numbers round to zero*
`1.2300`	*trailing zeros added as required*

The transactions in Figure D-21 specify `SCIENTIFIC` notation and four fractional digits.

```
WRITE TEXT X REAL SCI:4 EOL
WRITE TEXT Y REAL SCI:4 EOL
WRITE TEXT Z REAL SCI:4 EOL
```

Figure D-21. Three `WRITE TEXT REAL` **Transactions**

If X, Y, and Z in Figure D-21 have these values:

`1.2345678E2`	*the Real value o fX*
`-1.2345678E-09`	*the Real value o fY*
`0`	*the Real value o fZ*

then HP VEE writes this:

`1.2346E+02`	*exponent is E plus two signed digits*
`-1.2346E-09`	*last digit rounded as required*
`0.0000E+00`	*trailing zeros padded as required*

COMPLEX, PCOMPLEX, and COORD Formats

`COMPLEX`, `PCOMPLEX`, and `COORD` correspond to the HP VEE multi-field data types with the same names. The behavior of all three formats is very similar. The behaviors described in this section apply to all three formats except as noted.

Just as the HP VEE data types Complex, PComplex, and Coord are composed of multiple Real numbers, the `COMPLEX`, `PCOMPLEX`, and `COORD` formats are essentially compound forms of the `REAL` format. Each constituent Real value of the multi-field data types is written with the same output rules that apply to an individual `REAL` formatted value.

The final output of transactions involving multi-field formats is affected by the `Multi-Field Format` setting for the object in question. `Multi-Field Format` is accessed via `I/O` $\Longrightarrow$ `Instrument Manager` for `Direct I/O` objects and via `Config` in the object menu for all other objects. The two possible settings for `Multi-Field Format` are:

- `Data Only`. This writes multi-field data formats as a list of comma-separated numbers *without* parentheses.
- `( ... ) Syntax`. This writes multi-field data formats as a list of comma-separated numbers grouped by parentheses.

Subsequent examples will illustrate these behaviors.

COMPLEX Format. `WRITE TEXT COMPLEX` transactions are of this form:

`WRITE TEXT` *ExpressionList* `CPX`

The transaction in Figure D-22 specifies a fixed-decimal notation, explicit leading signs, a field width of 10 characters, and right justification.

```
WRITE TEXT X CPX FIX:3 SIGN:"+/-" FW:10 RJ EOL
```

Figure D-22. A `WRITE TEXT COMPLEX` Transaction

If the `Multi-Field Format` is set to `( ... ) Syntax`, and `X` in Figure D-22 has this value:

`( -1.23456 , 9.8 )` *the Complex value of X*

then HP VEE writes this:

```
(    -1.235 ,     +9.800)
^         ^   ^        ^
```

If the `Multi-Field Format` is set to `Data Only` and `X` in Figure D-22 has the same value, then HP VEE writes this:

```
    -1.235,    +9.800
^        ^ ^        ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the left of + are spaces (ASCII 32 decimal).

Note that with `( ... ) Syntax`, a space-comma-space sequence separates the ten-character wide fields that contain the real and imaginary parts of the Complex number. With either `Multi-Field Format` there is a separate ten-character field for both the real and the imaginary part. Neither parentheses nor the separating comma and spaces are included in the field.

PCOMPLEX Format. `WRITE TEXT PCOMPLEX` transactions are of this form:

`WRITE TEXT` *ExpressionList* `PCX`

PCOMPLEX format allows you to specify the phase units for the polar complex number it writes. Note that phase units are independent of the units set by Trig Mode in Properties.

Table D-10. PCOMPLEX **Phase Units**

Unit	Description
DEG	Degrees
RAD	Radians
GRAD	Gradians

The first transaction in Figure D-23 specifies phase measurement in degrees, and the second transaction specifies phase measurement in radians.

```
WRITE TEXT X PCX:DEG STD EOL
WRITE TEXT X PCX:RAD STD EOL
```

Figure D-23. Two WRITE TEXT PCOMPLEX **Transactions**

If the Multi-Field Format is set to Data Only, and X in Figure D-23 has this value:

`(-1.23456 , @90)` *the PComplex value o fX, phase in degrees*

then HP VEE writes this:

```
1.23456,-90
1.23456,-1.570796326794897
```

The transaction in Figure D-24 specifies phase measurement in radians, fixed-decimal notation, three fractional digits, explicit leading signs, a field width of ten characters, and right justification.

```
WRITE TEXT X PCX:RAD FIX:3 SIGN:"+/-" FW:10 RJ EOL
```

Figure D-24. A WRITE TEXT PCOMPLEX **Transaction**

If the Multi-Field Format is set to (...) Syntax, and X in Figure D-24 has this value:

```
(-1.23456 , @9.8)     the PComplex value of X, angle in radians
```

then HP VEE writes this:

```
(    +1.235 , @    +0.375)
 ^          ^   ^          ^
```

Note that HP VEE normalizes all PComplex numbers to yield a positive magnitude and a phase between $+\pi$ and $-\pi$.

If the Multi-Field Format is set to Data Only, and X in Figure D-24 has the same value, then HP VEE writes this:

```
    +1.235,    +0.375
^          ^ ^         ^
```

The caret characters (^) *are not* actually written by HP VEE; they are shown to help you visualize the field width. The characters to the left of - and to the left of + are spaces (ASCII 32 decimal).

COORD Format. WRITE TEXT COORD transactions are of this form:

WRITE TEXT *ExpressionList* COORD

COORD format has all the same behaviors of COMPLEX format. The only difference is that COORD may contain an arbitrary number of fields while COMPLEX has exactly two fields.

TIME STAMP Format

`WRITE TEXT TIME STAMP` transactions are of this form:

```
WRITE TEXT ExpressionList [DATE:DateSpec] [TIME:TimeSpec]
```

ExpressionList is a single expression or a comma-separated list of expressions.

DateSpec is one of the following pre-defined date and time combinations:

- `Date`
- `Time`
- `Date&Time`
- `Time&Date`
- `Delta Time`

If you specify a transaction that includes `Date`, you may also specify a *DateSpec* of `Weekday DD/Month/YYYY` or `DD/Month/YYYY`.

If you specify a transaction that includes `Time`, you may also specify a *TimeSpec*. *TimeSpec* is a combination of the following pre-defined time formats:

- `HH:MM` (hours and minutes)
- `HH:MM:SS` (hours, minutes, and seconds)
- `12 HOUR`
- `24 HOUR`

Each item in *ExpressionList* is converted to a Real and interpreted as a date and time. This Real number represents the number of seconds that have elapsed since midnight, January 1, AD 1 UTC. The most common source for this Real number is the output of a `Time Stamp` object. You use the `TIME STAMP` format to convert this Real number to a meaningful string that contains a human-readable date and/or time.

TIME STAMP supports a variety of notations for writing dates and times. If a Real variable contains this value:

```
62806574669.31164
```

then TIME STAMP can write it using any of these Time and Date notations:

Notation	**Result**
Date with Weekday DD/Month/YYYY	Thu 04/Apr/1991
Time with HH:MM:SS and 24 HOUR	15:44:29
Date&Time with Weekday DD/Month/YYYY, HH:MM:SS, and 24 HOUR	Thu 04/Apr/1991 15:44:29
Time&Date with HH:MM:SS, 24 HOUR, and Weekday DD/Month/YYYY	15:44:29 Thu 04/Apr/1991
Delta Time with HH:MM:SS	17446270:44:29
Date with Weekday DD/Month/YYYY	Thu 04/Apr/1991
Date with DD/Month/YYYY	04/Apr/1991
Time with HH:MM:SS and 24 HOUR	15:44:29
TIME with HH:MM and 24 HOUR	15:44
TIME with HH:MM:SS and 24 Hour	15:44:29
TIME with HH:MM:SS and 12 Hour	3:44:29 PM

BYTE Encoding

BYTE transactions are of this form:

WRITE BYTE *ExpressionList*

ExpressionList is a single expression or a comma-separated list of expressions.

HP VEE converts each item in *ExpressionList* to an Int16 (16-bit two's complement integer) and writes the least-significant 8-bits. This is a transaction for writing single characters to a device. Each expression in *ExpressionList* must be a scalar.

The transactions in Figure D-25 produce the output shown in Figure D-26.

```
WRITE BYTE 65,66,67
WRITE BYTE 65+1024,65+2048
```

Figure D-25. Two `WRITE BYTE` Transactions

```
ABCAA
```

Figure D-26. Character Data

CASE Encoding

`WRITE CASE` transactions are of this form:

`WRITE CASE` *ExpressionList1* `OF` *ExpressionList2*

ExpressionList is a single expression or a comma-separated list of expressions.

HP VEE converts each item in *ExpressionList1* to an integer and uses it as an index into *ExpressionList2*. The indexed item(s) in *ExpressionList2* are written in a string format that is the same as `WRITE TEXT` (default).

Note that the indexing of items in *ExpressionList2* is zero-based.

The transactions in Figure D-27 illustrate the behavior of `CASE` format.

```
WRITE CASE 2,1  OF "Str0","Str1","Str2"
WRITE CASE X  OF  1,1+A,3+A
```

Figure D-27. Two `WRITE CASE` Transactions

If the variables in Figure D-27 have these values:

`2`	*the Real value o fX*
`0.1`	*the Real value o fA*

then HP VEE writes this:

```
Str2Str1
3.1
```

BINARY Encoding

`WRITE BINARY` transactions are of this form:

`WRITE BINARY` *ExpressionList DataType*

ExpressionList is a single expression or a comma-separated list of expressions.

DataTypes is one of the following pre-defined HP VEE data types:

- `BYTE` - 8-bit byte
- `INT16` - 16-bit two's complement integer
- `INT32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating-point number
- `REAL64` - 64-bit IEEE 754 floating-point number
- `STRING` - null terminated string
- `COMPLEX` - equivalent to two `REAL`s
- `PCOMPLEX` -equivalent to two `REAL`s
- `COORD` - equivalent to two or more `REAL`s

Note

HP VEE stores and manipulates all integer values as the `INT32` data type, and all real numbers as the `Real` data type, also known as `REAL64`. Thus, the `INT16` and `REAL32` data types are provided for I/O only. HP VEE performs the following data-type conversions for instrument I/O:

- On an output transaction `INT32` values are individually converted to `INT16` values, which are output to the instrument. However, since the `INT16` data type has a range of -32768 to 32767, values outside this range will be truncated to 16 bits.
- On an output transaction `REAL64` values are individually converted to `REAL32` values, which are output to the instrument. However, since the `REAL32` data type has a smaller range than `REAL64` data type, values outside this range cannot be converted to `REAL32` and will result in an error.

BINARY encoded transactions convert each of the values specified in *ExpressionList* to the HP VEE data type specified by *DataType*. Each converted item is then written in the specified binary format. However, since the binary data written is a copy of the representation in computer memory, it is not easily shared by different computer architectures or hardware.

BINARY encoded data has the advantage of being very compact. READ BINARY transactions can read any corresponding WRITE BINARY data.

Note that BINARY encoding writes only the numeric portion of each data type. For example, the parentheses and comma that can be included when writing Complex and Coord data with TEXT encoding are never written with BINARY encoding. Similarly, when writing arrays, BINARY encoding does not write any Array Separators. WRITE BINARY transactions do allow you to specify EOL ON. There is rarely a need to write EOL with BINARY transactions because numeric data types are of fixed length and strings are null-terminated.

BINBLOCK Encoding

WRITE BINBLOCK transactions are of this form:

WRITE BINBLOCK *ExpressionList DataType*

ExpressionList is a single expression or a comma-separated list of expressions.

DataType is one of these pre-defined HP VEE data types:

- BYTE - 8-bit byte
- INT16 - 16-bit two's complement integer
- INT32 - 32-bit two's complement integer
- REAL32 - 32-bit IEEE 754 floating-point number
- REAL64 - 64-bit IEEE 754 floating-point number
- COMPLEX - equivalent to two REALs
- PCOMPLEX -equivalent to two REALs
- COORD - equivalent to two or more REALs

BINBLOCK writes *each item* in *ExpressionList* as a separate data block. The block header used depends on the type of object performing the WRITE and the object's configuration.

Non-HP-IB BINBLOCK

If the object is *not* `Direct I/O` to HP-IB, a `WRITE BINBLOCK` always writes an IEEE 488.2 Definite Length Arbitrary Block Response Data block. This data format is primarily used for communicating with HP-IB instruments using `Direct I/O`, although it is supported by other objects.

Each Definite Length Arbitrary Block is of the form:

```
#<Num_digits><Num_bytes><Data>
```

where:

`#` is literally the `#` character as shown.

`<Num_digits>` is an ASCII character that is a single digit (decimal notation) indicating the number of digits in `<Num_bytes>`.

`<Num_bytes>` is a list of ASCII characters that are digits (decimal notation) indicating the number of bytes that follow in `<Data>`.

`<Data>` is a sequence of arbitrary 8-bit data bytes.

HP-IB BINBLOCK

If the object is `Direct I/O` to HP-IB, the behavior of `WRITE BINBLOCK` transactions depends upon the `Direct I/O Configuration` settings for `Conformance` and `Binblock`; these settings are accessed via the `I/O` ⟹ `Instrument Manager` menu selection.

If `Conformance` is set to `IEEE 488.2`, then `WRITE BINBLOCK` *always* writes an IEEE 488.2 Definite Length Arbitrary Block Response Data block.

If `Conformance` is set to `IEEE 488`, then the type of header used depends on `Binblock`. `Binblock` may specify IEEE 728 `#A`, `#T`, or `#I` block headers. If `Binblock` is `None`, `WRITE BINBLOCK` writes an IEEE 488.2 Definite Length Arbitrary Block Response Data block.

IEEE 728 block headers are of the following forms:

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

where:

`#` is the character as shown.

`A`,`T`, `I` are the characters as shown.

`<Byte_Count>` consists of two bytes which together form a 16-bit unsigned integer that specifies the number of bytes that follow in `<Data>`. (HP VEE calculates this automatically.)

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

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

CONTAINER Encoding

`WRITE CONTAINER` transactions are of this form:

`WRITE CONTAINER` *ExpressionList*

ExpressionList is a single expression or a comma-separated list of expressions.

A `WRITE CONTAINER` transaction writes each item in *ExpressionList* using a special HP VEE text representation.

This representation retains all the HP VEE attributes associated with the data type written, such as shape, size, and name. Any `WRITE CONTAINER` data can be retrieved without any loss of information using `READ CONTAINER`.

For example, this transaction:

```
WRITE CONTAINER 1.2345
```

writes this:

```
(Real
 (data 1.2345)
)
```

STATE Encoding

`WRITE STATE` transactions are of the form:

```
    WRITE STATE [DownloadString]
```

DownloadString is an optional string that allows you to specify a download string if you have not previously specified one in the direct I/O configuration for the corresponding instrument. This explained in greater detail in the sections that follow.

`WRITE STATE` transactions are used by `Direct I/O` objects to download a learn string to an instrument. There is exactly one learn string associated with each instance of a `Direct I/O` object. This learn string is uploaded by clicking on `Upload` in the `Direct I/O` object menu. The learn string contains the null string before `Upload` is selected for the first time.

The behavior of `WRITE STATE` is affected by the `Direct I/O Configuration` settings for `Conformance` and `Download String`. These settings are accessed via the `I/O` ⟹ `Instrument Manager` menu selection. If `Conformance` is `IEEE 488`, the `WRITE STATE` transaction writes the `Download String` followed by the learn string. If `Conformance` is `IEEE 488.2`, the learn string is downloaded without any prefix as defined by IEEE 488.2. Please refer to *Controlling Instruments with HP VEE* for information about `WRITE STATE` transactions.

REGISTER Encoding

`WRITE REGISTER` is used to write values into a VXI device's A16 memory.

`WRITE REGISTER` transactions are of this form:

`WRITE REG:` *SymbolicName ExpressionList* `INCR`
`-or-`
`WRITE REG:` *SymbolicName ExpressionList*

where:

SymbolicName is a name defined during configuration of a VXI device. The name refers to a specific address within a device's register space. Specific data types for `WRITE REGISTER` transactions are:

- `BYTE` - 8 bit byte
- `WORD16` - 16-bit two's complement integer
- `WORD32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating point number

These data types are also specified during configuration of a VXI device and do not appear in the transaction.

ExpressionList is a single expression or a comma-separated list of expressions.

`INCR` specifies that array data is to be written incrementally starting at the register address specified by *SymbolicName*. The first element of the array is written at the starting address, the second at that address plus an offset equal to the length in bytes of the data type, and so forth until all array elements have been written. If `INCR` is not specified in the transaction, the entire array is written to the single location specified by *SymbolicName*.

MEMORY Encoding

`WRITE MEMORY` is used to write values into a VXI device's A24 or A32 memory.

`WRITE MEMORY` transactions are of this form:

WRITE MEM: *SymbolicName ExpressionList* INCR
-or-
WRITE MEM: *SymbolicName ExpressionList*

where:

SymbolicName is a name defined during configuration of a VXI device. The name refers to a specific address within a device's extended memory. Specific data types for `WRITE MEMORY` transactions are:

- `BYTE` - 8 bit byte
- `WORD16` - 16-bit two's complement integer
- `WORD32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating point number

These data types are also specified during configuration of a VXI device and do not appear in the transaction.

ExpressionList is a single expression or a comma-separated list of expressions.

`INCR` specifies that array data is to be written incrementally starting at the memory location specified by *SymbolicName*. The first element of the array is written at that location, the second at that location plus an offset equal to the length in bytes of the data type, and so forth until all array elements have been written. If `INCR` is not specified in the transaction, the entire array is written to the single memory location specified by *SymbolicName*.

IOCONTROL Encoding

`WRITE IOCONTROL` transactions are of this form:

WRITE IOCONTROL CTL *ExpressionList*
-or-
WRITE IOCONTROL PCTL *ExpressionList*

ExpressionList is a single expression or a comma-separated list of expressions.

`IOCONTROL` encoding is used only for `Direct I/O` to GPIO interfaces.

This transaction sets the control lines of a GPIO interface:

```
WRITE IOCONTROL CTL a
```

HP VEE converts the value of `a` to an Integer. The least *X* significant bits of the Integer value are mapped to the control lines of the interface, where *X* is the number of control lines.

For example, the HP 98622A GPIO interface uses two control lines, `CTL0` and `CTL1`.

Value Written	**CTL1**	**CTL0**
0	0	0
1	0	1
2	1	0
3	1	1

In the preceding table, `1` indicates that a control line is asserted, a `0` indicates that it is cleared.

This transaction controls the computer-driven handshake line of a GPIO interface:

```
WRITE IOCONTROL PCTL a
```

If the value of `a` is non-zero, the PCTL line is set. If the value is zero, no action is taken. PCTL is cleared automatically by the interface when the peripheral meets the handshake requirements.

READ Transactions

Table D-11. READ Encodings and Formats

Encodings	Formats
TEXT	CHAR TOKEN STRING QUOTED STRING INTEGER OCTAL HEX REAL COMPLEX PCOMPLEX COORD TIME STAMP
BINARY	STR BYTE INT16 INT32 REAL32 REAL64 COMPLEX PCOMPLEX COORD

Table D-11. READ Encodings and Formats (continued)

Encodings	Formats
BINBLOCK	BYTE INT16 INT32 REAL32 REAL64 COMPLEX PCOMPLEX COORD
CONTAINER	Not Applicable
IOSTATUS	Not Applicable
REGISTER[1]	BYTE WORD16 WORD32 REAL32
MEMORY[1]	BYTE WORD16 WORD32 REAL32

1 Direct I/O to VXI only.

TEXT Encoding

READ TEXT transactions are generally very easy to use. This is because they are able to read and discard what is irrelevant and selectively read what is important. This works well most of the time, but occasionally you must analyze very carefully what HP VEE considers to be irrelevant and what it considers to be important. This will rarely (if ever) be a problem if you are reading text files written by HP VEE, as long as you read them using the same format used to write them. Problems are most likely to occur when you are trying to import a file from another software application.

Table D-12 describes READ TEXT behavior in a general way only; be sure to read all the sections that follow to understand all the possible variations.

Table D-12. Formats for READ TEXT Transactions

Format	Description
CHAR	Reads *any* 8-bit character.
TOKEN	Reads a contiguous list of characters as a unit; this unit is called a token. Tokens are separated by specified delimiter characters (you specify the delimiters). For example, in normal written English, words are tokens and spaces are delimiters.
STRING	Reads a list of 8-bit characters as a unit. Most control characters are read and discarded. The end of the string is reached when the specified number of characters has been read, or when a newline character is encountered.
QSTRING	Reads a list of 8-bit characters that conform to the IEEE 488.2 arbitrary length string defined by a starting and ending double quote character (ASCII 34). Control characters are not discarded. Escaped characters are expanded to a corresponding control character. The end of the string is reached when the double quote character (ASCII 34) has been read.
INTEGER	Reads a list of characters and interprets them as a decimal or non-decimal representation of an integer. The only characters considered to be part of a decimal INTEGER are 0123456789-+. HP VEE recognizes the prefix 0x (hex) and all the Non-Decimal Numeric formats specified by IEEE 488.2: #H (hex), #Q (octal), #B (binary).
OCTAL	Reads a list of characters and interprets them as the octal representation of an integer. The characters considered to be part of an OCTAL are 01234567. HP VEE also recognizes the IEEE 488.2 Non-Decimal Numeric prefix #Q for octal numbers.

Table D-12. Formats for READ TEXT Transactions (continued)

Format	Description
HEX	Reads a list of characters and interprets them as the hexadecimal representation of an integer. The only characters considered to be part of a HEX are 0123456789abcdefABCDEF. The character combination 0x is the default prefix; it is not part of the number and is read and ignored. HP VEE also recognizes 0x and the IEEE 488.2 Non-Decimal Numeric prefix #H for hexadecimal numbers.
REAL	Reads a list of characters and interprets them as the decimal representation of a Real (floating-point) number. All common notations are recognized including leading signs, signed exponents, and decimal points. The characters recognized to be part of a REAL are 0123456789-+.Ee. HP VEE also recognizes certain characters as suffix multipliers for Real numbers (refer to Table D-13).
COMPLEX	Reads the equivalent of two REALs and interprets them as a complex number. The first number read is the real part and the second number read is the imaginary part.
PCOMPLEX	Reads the equivalent of two REALs and interprets them as a complex number in polar form. Some engineering disciplines refer to this as “phasor notation”. The first number read is considered to be the magnitude and the second is the angle. You may specify units of measure for phase in the transaction.
COORD	Reads the equivalent of two or more REALs and interprets them as rectangular coordinates.
TIME STAMP	Reads one of the specified HP VEE time stamp formats which represent the calendar date and/or time of day.

General Notes for READ TEXT

Read to End. The `READ TEXT` formats support a choice between reading a specified number of elements or reading until EOF is encountered. In a transaction, *NumElements* is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the first expression is an asterisk (*), the transaction will read data until an EOF is encountered. Read to end is supported only for `From File`, `From String`, `From StdIn`, `Execute Program`, `To/From Named Pipe`, `To/From Socket`, and `To/From HP BASIC/UX` transactions.

Only the first dimension can have an asterisk rather than a number.

For example, the following transaction, reading from a file:

```
READ TEXT a REAL ARRAY:*,10
```

will read until EOF is encountered resulting in a two dimensional array with ten columns. The number of rows is dependent on the amount of data in the file. The total number of data elements read must be evenly divisible by the product of the known dimension sizes, in this example: 10. If this criteria is not met, an error will occur.

Number of Characters Per READ. These `READ TEXT` formats support a choice between `DEFAULT NUM CHARS` and `MAX NUM CHARS`:

```
STRING
INTEGER
OCTAL
HEX
REAL
```

This section discusses the effects of `DEFAULT NUM CHARS` and `MAX NUM CHARS` on these formats.

The basic difference between `DEFAULT NUM CHARS` and `MAX NUM CHARS` is this:

- `DEFAULT NUM CHARS` causes HP VEE to read and ignore most characters that do not appear to be part of the number or string it expects.
- `MAX NUM CHARS` allows you to read *up to* the specified number of 8-bit characters in an attempt to build the type of number or string specified. HP VEE stops reading characters as soon as the `READ` is satisfied. All characters are read and HP VEE attempts to convert them to the data type specified in the transaction.

If you specify `DEFAULT NUM CHARS`, the transaction reads as many characters as it requires to fill each variable. Characters that are not meaningful to the specified data type are read and ignored.

If you specify `MAX NUM CHARS`, HP VEE makes no attempt to sort out characters that are not meaningful to the data type specified. If non-meaningful characters are encountered, they are read and may later generate an error.

In either case, newline and end-of-file are recognized as terminators for strings or numbers. For numeric formats, white space encountered before any significant characters (digits) is read and ignored; after reading significant characters, white space or other non-numeric characters terminate the current `READ`. These are the general behaviors; read the examples that follow for additional detail.

Consider this example that distinguishes between the behaviors of

`DEFAULT NUM CHARS` and `MAX NUM CHARS` using `INTEGER` format. Assume that you are trying to read a file containing this data:

```
bird dog cat 12345 horse
```

It is impossible to extract the integer `12345` from this data with a `READ TEXT INTEGER` transaction using `MAX NUM CHARS` no matter how many characters are read. This is because the characters `bird dog cat` are always read before the digits, they cannot be converted to an Integer, and this generates an error.

`DEFAULT NUM CHARS` will extract the integer `12345` by reading and ignoring `bird dog cat` and treating the white space following `5` as a delimiter.

Effects of Quoted Strings. The presence of quoted strings affects the behavior of `READ TEXT QSTR` and `READ TEXT TOKEN` for all I/O paths and `READ TEXT STRING` for instrument or interface I/O. In this discussion, a quoted string means a set of characters beginning and ending with a double quote character and no embedded (non-escaped) double quote characters. The double quote character is ASCII 34 decimal. The presence of double quotes affects the way that these `READ` transactions group characters into strings and tokens, and how embedded control and escape characters are handled.

In this discussion, the terms **control character** and **escape character** have specific meaning. A control character is a single byte of data corresponding to one of the ASCII characters 0-31 decimal. For example, linefeed is ASCII 10 decimal and the symbol `<LF>` denotes linefeed character in this discussion. The string `\n` is a human-readable escape character representing linefeed that is recognized by HP VEE.

The behavior of certain transactions when dealing with quoted strings is dependent on the particular I/O path. For all I/O paths except instrument I/O, READ TEXT QSTR treats quoted strings specially. For all I/O paths except instrument I/O, READ TEXT STRING does not recognize quoted strings. For instrument I/O there is no READ TEXT QSTR transaction. Instead, READ TEXT STRING recognizes quoted stings and deals with them accordingly. This is done since quoted strings have special meaning in the IEEE 488.2 specification. For all I/O paths including instruments, READ TEXT TOKEN treats quoted strings specially. In the following discussions, we will assume the I/O path to be file I/O.

When a string does not begin and end with double quotes, control characters other than linefeed are read and discarded by READ TEXT STRING transactions and by READ TEXT TOKEN transactions that specify SPACE DELIM. In both STRING and TOKEN transactions, linefeed terminates the READ. Escape character sequences, such as \n (newline) are simply read as the two characters \ and n.

Within double quoted strings, READ TEXT QSTR and READ TEXT TOKEN will read all enclosed characters (including control characters) store them in the input variable. Embedded linefeeds are read and treated like any other character; they do not terminate the current READ. Escape character sequences are read and translated to their single-character counterpart.

Grouping effects are best explained by using an example. For the discussion in the rest of this section, the data being read is a file with the contents shown in Figure D-28.

```
  "This is in quotes."  This is not.
```

Figure D-28. Quoted and Non-Quoted Data

Assume that you read the file shown in Figure D-28 using From File with these transactions:

```
READ TEXT x QSTR
READ TEXT y QSTR
```

After reading the file, the results are:

```
x = This is in quotes.
y = This is not.
```

Note that the double quotes are interpreted as delimiters and do not appear in the input variable.

Now assume that you read the file shown in Figure D-28 using `From File` with these transactions:

```
READ TEXT x QSTR MAXFW:4
READ TEXT y QSTR
```

After reading the file, the results are:

```
x = This
y = This is not.
```

Here the double quotes are still acting a delimiters; the first transaction reads from double quote to double quote and assigns the first four characters to `x`. This leaves the file's read pointer positioned before the second occurrence of `This`. The second transaction reads the same string as before.

Next, assume that you read the file shown in Figure D-28 using `From File` with these transactions:

```
READ TEXT x TOKEN
READ TEXT y QSTR
```

Now after reading the file, the results are:

```
x = This is in quotes.
y = This is not.
```

Here the double quotes effectively make the entire first sentence into a single token. Even though default `TOKEN` delimiter is white space, the entire quoted string is treated as a single token. In addition, `TOKEN` reads and discards the double quote characters.

CHAR Format

`READ TEXT CHAR` transactions are of this form:

```
READ TEXT VarList CHAR:NumChar ARRAY:NumStr
```

VarList is a single Text variable or a comma-separated list of Text variables.

NumChar specifies the number of 8-bit characters that must read to fill each element of each variable in *VarList*.

NumStr is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

`CHAR` format is useful when you wish to simply read one character at a time, or when you need to read *every* character without ignoring any incoming data.

This transaction reads two two-dimensional Text arrays; each element in each array contains two characters.

```
READ TEXT X,Y CHAR:2 ARRAY:2,2
```

If a file read by the previous transaction contains these characters:

```
<space>ABCDEFG"AB"<LF>'CD
```

then the variables `X` and `Y` contain these values after the `READ`:

```
X [0 0] = <space>A
X [0 1] = BC
X [1 0] = DE
X [1 1] = FG

Y [0 0] = "A
Y [0 1] = B"
Y [1 0] = <LF>'
Y [1 1] = CD
```

The symbol `<space>` means the single character, space (ASCII 32 decimal). The symbol `<LF>` means the single character, linefeed (ASCII 10 decimal). Note that space, linefeed, and double quotes are read without any special consideration or interpretation.

TOKEN Format

`READ TEXT TOKEN` transactions are of this form:

> `READ TEXT` *VarList* `TOKEN` *Delimiter*
> `ARRAY:`*NumElements*

VarList is a single Text variable or a comma-separated list of Text variables.

Delimiter specifies the combinations of characters that terminate (delimit) each token.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

`TOKEN` format allows you to define the delimiter (boundary) for tokens using one of these choices for *Delimiter*:

- `SPACE DELIM`
- `INCLUDE CHARS`
- `EXCLUDE CHARS`

The following discussion of delimiters explains how the choice of delimiters affects reading a file with these contents:

```
A phrase.
"A phrase."
Tab follows     .
XOXXOOXXXOOOXXXX
XAXXBCXXXDEF
```

Figure D-29. Data for `READ TOKEN`

The file contains only the letter `O`, not the digit zero.

Note that there is an invisible linefeed character at the end of each of the first four lines of the file in Figure D-29. The figure shows the file as it would appear in a text editor like `vi`.

SPACE DELIM. If you use `SPACE DELIM`, tokens are terminated by any white space. White space includes spaces, tabs, newline, and end-of-file. This corresponds roughly to words in written English. Using `SPACE DELIM`, you could read a file containing a paragraph of prose and separate out individual words.

Note that double quoted strings receive special treatment. Double quoted strings are read as a single token and the double quotes are stripped away. Control characters (ASCII 0-31 decimal) embedded in double-quoted strings are returned in the output variable. Escape characters (such as `\n`) embedded in double-quoted strings are converted into their equivalent control characters. This special treatment of double-quoted strings applies only to `SPACE DELIM` transactions; `INCLUDE CHARS` and `EXCLUDE CHARS` treat double quotes, escapes, and control characters the same as any other character.

If you read the data shown in Figure D-29 using `SPACE DELIM` with this transaction:

```
READ TEXT a TOKEN ARRAY:8
```

then the variable `a` contains these values:

```
a[0] = A
a[1] = phrase.
a[2] = A phrase.
a[3] = Tab
a[4] = follows
a[5] = .
a[6] = XOXXOOXXXOOOXXXX
a[7] = XAXXBCXXXDEF
```

INCLUDE CHARS. If you use `INCLUDE CHARS`, you can specify a list of characters to be "included" in tokens returned by the `READ`. These specified characters will be the *only* characters returned in any token. Any character other than the specified `INCLUDE` characters terminates the current token. The terminating characters *are not* included in the token and are stripped away.

If HP VEE reads the data shown in Figure D-29 using `INCLUDE CHARS` with this transaction:

```
READ TEXT a TOKEN INCLUDE:"X" ARRAY:7
```

then the variable `a` contains these values:

```
a[0] = X
a[1] = XX
a[2] = XXX
a[3] = XXXX
a[4] = X
a[5] = XX
a[6] = XXX
```

If HP VEE reads the data shown in Figure D-29 using `INCLUDE CHARS` with this transaction:

```
READ TEXT a TOKEN INCLUDE:"OXZ" ARRAY:4
```

then the variable `a` contains these values:

```
a[0] = XOXXOOXXXOOOXXXX
a[1] = X
a[2] = XX
a[3] = XXX
```

Note that the first character in the `INCLUDE` list is the letter `O`, not the digit zero.

Assume that you are trying to read a file containing the data in Figure D-30.

```
111 222 333 444 555
```

Figure D-30. Data for `READ TOKEN`

If you try to read the file in Figure D-30 using this transaction:

```
READ TEXT x,y,z TOKEN INCLUDE:"1234567890"
```

then the Text variables `x`, `y`, and `z` will contain these values:

```
x = 111
y = 222
z = 333
```

Another way to do this is to specify an ARRAY greater than one and read data into an array. For example, if you read the data in Figure D-30 using this transaction:

```
READ TEXT x TOKEN INCLUDE:"1234567890" ARRAY:3
```

then the Text variable x contains these values:

```
x[0] = 111
x[1] = 222
x[2] = 333
```

EXCLUDE CHARS. If you use EXCLUDE CHARS, you can specify a list of characters, any one of which will terminate the current token. The terminating characters *are not* included in the token. They are read and discarded.

If you read the data shown in Figure D-29 using EXCLUDE with this transaction:

```
READ TEXT a TOKEN EXCLUDE:"X" ARRAY:8
```

then the variable a contains these values:

```
a[0] = A phrase.<LF>"A phrase."<LF>Tab follows     .<LF>
a[1] = 0
a[2] = 00
a[3] = 000
a[4] = <LF>
a[5] = A
a[6] = BC
a[7] = DEF
```

Assume the data shown in Figure D-31 is sent to HP VEE from an instrument.

```
++1.23++4.98++0.45++2.34++0.01++23.45++12.2++
```

Figure D-31. Data for `READ TOKEN`

If HP VEE reads the data in Figure D-31 with this transaction:

```
READ TEXT x TOKEN EXCLUDE:"+" ARRAY:7
```

then the variable `x` will contain these values:

```
x[0] = null string (empty)
x[1] = 1.23
x[2] = 4.98
x[3] = 0.45
x[4] = 2.34
x[5] = 0.01
x[6] = 23.45
```

Note that even though seven "numbers" were available, only six were read. At the end of this transaction, HP VEE has read seven tokens terminated by the `+`, including the first character which was terminated before it was filled with any data.

STRING Format

`READ TEXT STRING` transactions are of this form:

`READ TEXT` *VarList* `STR ARRAY:`*NumElements*
`-or-`
`READ TEXT` *VarList* `STR MAXFW:`*NumChars* `ARRAY:`*NumElements*

VarList is a single Text variable or a comma-separated list of Text variables.

NumChars specifies the maximum number of 8-bit characters that can be read in an attempt to build a string.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

This transaction reads all incoming characters and returns strings. Leading spaces are deleted. The following discussion pertains to instrument I/O paths only, such as HP-IB or VXI. All other I/O paths, such as files or named-pipes, will not treat Quoted Strings specially. Please refer to the section "Effects of Quoted Strings" earlier in this chapter for details about the effects of double quoted strings on `READ TEXT STRING`.

Effects of Control and Escape Characters. In this discussion, the terms **control character** and **escape character** have specific meaning. A control character is a single byte of data corresponding to one of the ASCII characters 0-31 decimal. For example, linefeed is ASCII 10 decimal and the symbol `<LF>` denotes linefeed character in this discussion. The string `\n` is a human-readable escape character representing linefeed that is recognized by HP VEE. HP VEE uses escape characters to represent control characters within quoted strings.

Control characters and escape characters are handled differently depending on whether or not they appear within double quoted strings.

Outside double quoted strings, control characters other than linefeed are read and discarded. Linefeed terminates the current string. Escape characters, such as `\n`, are simply read as two individual characters (`\` and `n`).

Within double quoted strings, control characters and escape characters are read and included in the string returned by the `READ`. A linefeed within a double quoted string does *not* terminate the current string. Escape characters, such as `\n`, are interpreted as their single character equivalent (`<LF>`) and are included in the returned string as a control character.

Assume you wish to read the data in Figure D-32 using `READ TEXT STRING` transactions.

```
Simple string.
Random \n % $ * 'A'
"In quotes."
"In quotes
with control."
"In quotes\nwith escape."
```

Figure D-32. String Data

If you read the data in Figure D-32 using this transaction:

```
READ TEXT x STR ARRAY:5
```

then the variable x contains these values:

```
a[0] = Simple string.
a[1] = Random \n % $ * 'A'
a[2] = In quotes.
a[3] = In quotes<LF>with control.
a[4] = In quotes<LF>with escape.
```

If you read the same data in Figure D-32 using this transaction:

```
READ TEXT x STR MAXFW:16 ARRAY:5
```

then the variable x contains these values:

```
a[0] = Simple string.
a[1] = Random \n % $ *
a[2] = 'A'
a[3] = In quotes.
a[4] = In quotes<LF>with c
```

Note that the transaction terminates the current `READ` whenever 16 characters have been read (`a[1]`) or when a non-quoted `<LF>` (`a[2]`) is read. Double quoted strings are read from double quote to double quote and the first 16 delimited characters are returned (`a[4]`).

QUOTED STRING Format

`READ TEXT QUOTED STRING` transactions are of this form:

READ TEXT *VarList* QSTR ARRAY:*NumElements*
-or-
READ TEXT *VarList* QSTR MAXFW:*NumChars* ARRAY:*NumElements*

VarList is a single Text variable or a comma-separated list of Text variables.

NumChars specifies the maximum number of 8-bit characters that can be read in an attempt to build a string.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

This transaction reads all incoming characters and returns strings. The following discussion pertains to all non-instrument I/O paths. Instrument I/O paths do not implement the `READ TEXT QSTR` transaction. Please refer to the section "Effects of Quoted Strings" earlier in this chapter for details about the effects of double quoted strings on `READ TEXT STRING`.

Effects of Control and Escape Characters. In this discussion, the terms **control character** and **escape character** have specific meaning. A control character is a single byte of data corresponding to one of the ASCII characters 0-31 decimal. For example, linefeed is ASCII 10 decimal and the symbol `<LF>` denotes linefeed character in this discussion. The string `\n` is a human-readable escape character representing linefeed that is recognized by HP VEE. HP VEE uses escape characters to represent control characters within quoted strings.

Control characters and escape characters are handled differently depending on whether or not they appear within double quoted strings.

Outside double quoted strings, control characters other than linefeed are read and discarded. Linefeed terminates the current string. Escape characters, such as `\n`, are simply read as two individual characters (`\` and `n`).

Within double quoted strings, control characters and escape characters are read and included in the string returned by the `READ`. A linefeed within a double quoted string does *not* terminate the current string. Escape characters, such as `\n`, are interpreted as their single character equivalent (`<LF>`) and are included in the returned string as a control character.

Assume you wish to read the data in Figure D-33 using `READ TEXT QUOTED STRING` transactions.

```
Simple string.
Random \n % $ * ‘A’
"In quotes."
"In quotes
with control."
"In quotes\nwith escape."
```

Figure D-33. String Data

If you read the data in Figure D-33 using this transaction:

```
READ TEXT x QSTR ARRAY:5
```

then the variable x contains these values:

```
a[0] = Simple string.
a[1] = Random \n % $ * ‘A’
a[2] = In quotes.
a[3] = In quotes<LF>with control.
a[4] = In quotes<LF>with escape.
```

If you read the same data in Figure D-33 using this transaction:

```
READ TEXT x QSTR MAXFW:16 ARRAY:5
```

then the variable x contains these values:

```
a[0] = Simple string.
a[1] = Random \n % $ *
a[2] = ‘A’
a[3] = In quotes.
a[4] = In quotes<LF>with c
```

Note that the transaction terminates the current `READ` whenever 16 characters have been read (`a[1]`) or when a non-quoted `<LF>` (`a[2]`) is read. Double quoted strings are read from double quote to double quote and the first 16 delimited characters are returned (`a[4]`).

INTEGER Format

`READ TEXT INTEGER` transactions are of this form:

```
READ TEXT VarList INT ARRAY:NumElements
-or-
READ TEXT VarList INT MAXFW:NumChars
ARRAY:NumElements
```

VarList is a single Integer variable or a comma-separated list of Integer variables.

NumChars specifies the maximum number of 8-bit characters that can be read in an attempt to build a number.

NumStr is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

`READ TEXT INTEGER` transactions interpret incoming characters as 32-bit, two's complement integers. The valid range for these integers is 2 147 483 647 to -2 147 483 648. Any numbers outside this range wrap around so there is never an overflow condition. For example, 2 147 483 648 is interpreted as -2 147 483 648. As it starts to build a number, HP VEE discards any leading characters that are not recognized as part of a number. Once HP VEE starts building a number, any character that is not recognized as part of a number terminates the `READ` for that number. These are the only combinations of characters that are recognized as part of an `INTEGER`:

Notation	**Characters Recognized**
Decimal	Valid characters are `+-0123456789`. Leading zeros are *not* interpreted as an octal prefix as they are in HP VEE data entry fields.
HP VEE hexadecimal	HP VEE interprets `0x` as a prefix for a hexadecimal number. Valid characters following the prefix are `0123456789aAbBcCdDeEfF`.
IEEE 488.2 binary	HP VEE interprets `#b` or `#B` as a prefix for a binary number. Valid characters following the prefix are `0` and `1`.
IEEE 488.2 octal	HP VEE interprets `#q` or `#Q` as a prefix for an octal number. Valid characters following the prefix are `01234567`.
IEEE 488.2 hexadecimal	HP VEE interprets `#h` or `#H` as a prefix for a hexadecimal number. Valid characters following the prefix are `0123456789aAbBcCdDeEfF`.

All of the following notations are interpreted as the Integer value 15 decimal:

```
15
+15
015
0xF
0xf
#b1111
#Q17
#hF
```

OCTAL Format

`READ TEXT OCTAL` transactions are of this form:

```
READ TEXT VarList OCT ARRAY:NumElements
-or-
READ TEXT VarList OCT MAXFW:NumChars
ARRAY:NumElements
```

VarList is a single Integer variable or a comma-separated list of Integer variables.

NumChars specifies the number of 8-bit characters that can be read in an attempt to build a number.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

`READ TEXT OCTAL` transactions interpret incoming characters as octal digits representing 32-bit, two's complement integers. The valid range for these integers is 2 147 483 647 decimal to -2 147 483 648 decimal.

If the transaction specifies a `MAX NUM CHARS` (`MAXFW`), the octal number read may contain more than 32 bits of data. For example, assume HP VEE reads the data in Figure D-34 using this transaction:

```
READ TEXT x OCT MAXFW:21
```

```
377237456214567243777
```

Figure D-34. Octal Data

HP VEE reads all the digits in Figure D-34, but uses only the last 11 digits (14567243777) to build a number for the value of `x`. This is because each digit corresponds to 3 bits and the octal number must be stored in an HP VEE Integer, which contains 32 bits. Eleven octal digits yield 33 bits; the most significant bit is dropped to fit the value in an HP VEE Integer. There is no possibility of overflow.

If the transaction specifies `DEFAULT NUM CHARS`, it will continue to read characters until it builds enough numbers to fill each variable in *VarList*. Linefeed characters will not terminate number building early. For example, this transaction:

```
READ TEXT x OCT ARRAY:4
```

interprets each line of data in Figure D-35 as the same set of four octal numbers.

```
0345 067 003<LF>0377<LF>
345 67 3 377<EOF>
345,67,3,377,45,67<EOF>
```

Figure D-35. Octal Data

The symbol `<LF>` represents the single character linefeed (ASCII 10 decimal). The symbol `<EOF>` represents the end-of-file condition.

HEX Format

`READ TEXT HEX` transactions are of this form:

`READ TEXT` *VarList* `HEX ARRAY:`*NumElements*
`-or-`
`READ TEXT` *VarList* `HEX MAXFW:`*NumChars*
`ARRAY:`*NumElements*

VarList is a single Integer variable or a comma-separated list of Integer variables.

NumChars specifies the number of 8-bit characters that can be read in an attempt to build a number.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

`READ TEXT HEX` transactions interpret incoming characters as hexadecimal digits representing 32-bit, two's complement integers. The valid range for these integers is 2 147 483 647 decimal to -2 147 483 648 decimal.

If the transaction specifies a `MAX NUM CHARS` (`MAXFW`), the hexadecimal number read may contain more than 32 bits of data. For example, assume HP VEE reads the data in Figure D-36 using this transaction:

```
READ TEXT x HEX MAXFW:21
```

```
ad2469Ff725BCdef37964
```

Figure D-36. Hexadecimal Data

HP VEE reads all the digits in Figure D-36, but uses only the last 8 digits (`def37964`) to build a number for the value of `x`. This is because each digit corresponds to 4 bits and the hexadecimal number must be stored in an HP VEE Integer, which contains 32 bits. Eight hexadecimal digits yields exactly 32 bits. There is no possibility of overflow.

Assume HP VEE reads the same data in Figure D-36, but with a different `MAX NUM CHARS`, as in this transaction:

```
READ TEXT x HEX MAXFW:3 ARRAY:7
```

In this case, the transaction reads the same data and interprets it as seven Integers, each comprised of three hexadecimal digits.

If the transaction specifies `DEFAULT NUM CHARS`, it will continue to read characters until it builds enough numbers to fill each variable in *VarList*. Each number will read *exactly* 8 hexadecimal digits. Linefeed characters will not terminate number building early.

Assume HP VEE reads the same data in Figure D-36, but with `DEFAULT NUM CHARS`, as in this transaction:

```
READ TEXT x HEX ARRAY:2
```

In this case, the transaction reads the same data and interprets it as two Integers, each comprised of eight hexadecimal digits. The last five digits (37946) are not read.

REAL Format

`READ TEXT REAL` transactions are of this form:

```
READ TEXT VarList REAL ARRAY:NumElements
-or-
READ TEXT VarList REAL MAXFW:NumChars
ARRAY:NumElements
```

VarList is a single Real variable or a comma-separated list of Real variables.

NumChars specifies the maximum number of 8-bit characters that can be read in an attempt to build a number.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

The decimal number read by this transaction is interpreted as an HP VEE Real which is a 64-bit IEEE 754 floating-point number. The range of these numbers is:

```
-1.797 693 134 862 315E+308
-2.225 073 858 507 202E-307
 0
 2.225 073 858 507 202E-307
 1.797 693 134 862 315E+308
```

If the transaction specifies a `MAX NUM CHARS` (`MAXFW`), the Real number read may contain more than 17 digits of data. For example, assume HP VEE reads the data in Figure D-37 using this transaction:

```
READ TEXT x REAL MAXFW:19
```

```
1.234567890123456789
```

Figure D-37. Real Data

HP VEE reads all the digits in Figure D-37, but uses only the 17 most-significant digits of the mantissa to build a number for the value of `x`. This is because each Real contains a 54-bit mantissa, which is equivalent to more than 16 but less than 17 decimal digits. As a result, x has the value 1.2345678901234567. Text to Real conversions are not guaranteed to yield the same value to the least-significant digit. Comparisons of the two least-significant bits is unadvisable.

Assume HP VEE reads the same data in Figure D-37, but with a different MAX NUM CHARS, as in this transaction:

```
READ TEXT x REAL MAXFW:6 ARRAY:3
```

In this case, the transaction reads the same data and interprets it as 3 Real numbers, each comprised of six decimal characters. The last two characters are not read.

If the transaction specifies DEFAULT NUM CHARS, it will continue to read characters until it builds enough numbers to fill each variable in *VarList*. Each number will read at most 17 decimal digits. Linefeed characters, white space and other non-numeric characters will terminate number building before 17 digits have been read.

READ TEXT REAL transactions recognize most commonly used decimal notations for Real numbers including leading signs, decimal points, and signed exponents. The characters +-.0123456789Ee are recognized as valid parts of a Real number by *all* READ TEXT REAL transactions. If the transaction specifies DEFAULT NUM CHARS, the suffix characters shown in Table D-13 are also recognized. The suffix character must immediately follow the last digit of the number with no intervening white space.

Table D-13. Suffixes for `REAL` Numbers

Suffix	Multiplier
`P`	10^{15}
`T`	10^{12}
`G`	10^{9}
`M`	10^{6}
`k` or `K`	10^{3}
`m`	10^{-3}
`u`	10^{-6}
`n`	10^{-9}
`p`	10^{-12}
`f`	10^{-15}

The Text data in Figure D-38 represents six real numbers.

```
1001
+1001.
1001.0
1.001E3
+1.001E+03
1.001K
```

Figure D-38. Example of Real Notations

If HP VEE reads the data in Figure D-38 with this transaction:

```
READ TEXT x REAL ARRAY:6
```

then each element of the Real variable `x` contains the value `1001`.

If HP VEE reads the data in Figure D-38 with this transaction:

```
READ TEXT x REAL MAXFW:20 ARRAY:6
```

then the first five elements of the Real variable `x` contain the value `1001` and the sixth element contains the value `1.001`.

COMPLEX, PCOMPLEX, and COORD Formats

`COMPLEX`, `PCOMPLEX`, and `COORD` correspond to the HP VEE multi-field data types with the same names. The behavior of all three `READ` formats is very similar. The behaviors described in this section apply to all three formats except as noted.

Just as the HP VEE data types Complex, PComplex, and Coord are composed of multiple Real numbers, the `COMPLEX`, `PCOMPLEX`, and `COORD` formats are compound forms of the `REAL` format. Each constituent Real value of the mult-field data types is read using the same rules that apply to an individual `REAL` formatted value.

COMPLEX Format. `READ TEXT COMPLEX` transactions are of this form:

`READ TEXT` *VarList* `CPX ARRAY:`*NumElements*

Each `READ TEXT COMPLEX` transaction reads the equivalent of two `REAL` formatted numbers. The first number read is interpreted as the real part and the second number read is interpreted as the imaginary part.

PCOMPLEX Format. `READ TEXT PCOMPLEX` transactions are of this form:

`READ TEXT` *VarList* `PCX:`*PUnit* `ARRAY:`*NumElements*

PUnit specifies the units of angular measure in which the phase of the PComplex is measured.

Each `READ TEXT PCOMPLEX` transaction reads the equivalent of two `REAL` formatted numbers. The first number read is interpreted as the magnitude and the second number read is interpreted as the phase.

If any transaction reading COMPLEX, PCOMPLEX, or COORD formats encounters an opening parenthesis, it expects to find a closing parenthesis.

Assume you wish to read a file containing the data shown in Figure D-39.

```
   (1.23 , 3.45  (6.78 , 9.01) (1.23 , 4.56)
```

Figure D-39. Data Containing Parentheses

If HP VEE reads the data in Figure D-39 with this transaction:

```
READ TEXT x,y CPX
```

then the variables x and y contain these Complex values:

```
x = (1.23 , 3.45)
y = (1.23 , 4.56)
```

Note that the transaction read past 6.78 and 9.01 to find the closing parenthesis. If parentheses had been omitted from the data entirely, y would have the value (6.78 , 9.01).

COORD Format. READ TEXT COORD transactions are of this form:

READ TEXT *VarList* COORD:*NumFields* ARRAY:*NumElements*

VarList is a single Coord variable or a comma-separated list of Coord variables.

NumFields is a single variable or expression that specifies the number of rectangular dimensions in each Coord value. This value must be 2 or more for the READ to execute without error.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the ARRAY keyword does not appear in the transaction. Note that ARRAY:1 is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

BINARY Encoding

`READ BINARY` transactions are of this form:

`READ BINARY` *VarList* *DataType* `ARRAY:`*NumElements*

VarList is a single variable or a comma-separated list of variables.

DataType is one of the following pre-defined formats corresponding to the HP VEE data type with the same name:

- `BYTE` - 8-bit byte
- `INT16` - 16-bit two's complement integer
- `INT32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating-point number
- `REAL64` - 64-bit IEEE 754 floating-point number
- `STRING` - null terminated string
- `COMPLEX` - equivalent to two `REAL`s
- `PCOMPLEX` -equivalent to two `REAL`s
- `COORD` - equivalent to two or more `REAL`s

Note

HP VEE stores and manipulates all integer values as the `INT32` data type, and all real numbers as the `Real` data type, also known as `REAL64`. Thus, the `INT16` and `REAL32` data types are provided for I/O only. HP VEE performs the following data-type conversions for instrument I/O:

- On an input transaction `INT16` values from an instrument are *individually* converted to `INT32` values by HP VEE. This conversion assumes that the `INT16` data was *signed* data. If you need the resulting `INT32` data in *unsigned* form, simply pass the data through a formula object with the formula

```
BITAND(a, 0xFFFF)
```

- On an input transaction `REAL32` values from an instrument are *individually* converted to `REAL64` values by HP VEE.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the first expression is an asterisk (*), the transaction will read data until an EOF is encountered. Read to end is supported only for `From File`, `From String`, `From StdIn`, `Execute Program`, `To/From Named Pipe`, and `To/From HP BASIC/UX` transactions.

Only the first dimension can have an asterisk rather than a number. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

For example, the following transaction, reading from a file:

```
READ BINARY a REAL64 ARRAY:*,10
```

will read until EOF is encountered, resulting in a two dimensional array with 10 columns. The number of rows is dependent on the amount of data in the file. The total number of data elements read must be evenly divisible by the product of the known dimension sizes, in this example: 10.

`READ BINARY` transactions expect that incoming data is in *exactly* the same format that would be produced by an equivalent `WRITE BINARY` transaction.

`BINARY` encoded data has the advantage of being very compact, but it is not easily shared with non-HP VEE applications.

BINBLOCK Encoding

`READ BINBLOCK` transactions are of this form:

`READ BINBLOCK` *VarList DataType* `ARRAY:`*NumElements*

VarList is a single variable or a comma-separated list of variables.

DataType is one of these pre-defined HP VEE data types:

- `BYTE` - 8-bit byte
- `INT16` - 16-bit two's complement integer
- `INT32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating-point number
- `REAL64` - 64-bit IEEE 754 floating-point number
- `COMPLEX` - equivalent to two `REAL`s
- `PCOMPLEX` -equivalent to two `REAL`s
- `COORD` - equivalent to two or more `REAL`s

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. The number of columns is equal to the number of channels contained by the binblock. The number of rows is equal to the number of readings per channel. Only the first dimension can have an asterisk rather than a number.

If the first expression is an asterisk (*), the transaction will read data until an EOF is encountered. Read to end is supported only for `From File`, `From String`, `From StdIn`, `Execute Program`, `To/From Named Pipe`, `To/From Socket`, and `To/From HP BASIC/UX` transactions.

If the transaction is configured to read a one-dimension array, for a single channel, the single dimension represents rows and can have an asterisk.

For example, the following transaction, reading from a file:

```
READ BINBLOCK a REAL64 ARRAY:*,10
```

will read until EOF is encountered, resulting in a two dimensional array with 10 columns. Each column represents an instrument channel. The number of rows is dependent on the amount of data in each channel. The total number of data elements contained by the binblock must be evenly divisible by the number of columns, in this example: 10.

You do not need to specify any additional information about the format of incoming data; the block header contains sufficient information. `READ BINBLOCK` can read any of the block formats described previously with `WRITE BINBLOCK` transactions.

The following transaction reads two traces from an oscilloscope that formats its traces as IEEE 488.2 Definite Length Arbitrary Block Response Data:

```
READ BINBLOCK a,b REAL
```

CONTAINER Encoding

`READ CONTAINER` transactions are of the form:

`READ CONTAINER` *VarList*

VarList is a single variable or a comma-separated list of variables.

`READ CONTAINER` transactions reads data stored in the special special text representation written by `WRITE CONTAINER` transactions. No additional specifications, such as format, need to be specified with `READ CONTAINER` since that information is part of the container.

REGISTER Encoding

`READ REGISTER` is used to read values from a VXI device's A16 memory.

`READ REGISTER` transactions are of this form:

> `READ REG:` *SymbolicName ExpressionList* `INCR`
> `ARRAY:`*NumElements*
> `-or-`
> `READ REG:` *SymbolicName ExpressionList*
> `ARRAY:`*NumElements*

where:

SymbolicName is a name defined during configuration of a VXI device. The name refers to a specific address within a device's register space. Specific data types for `READ REGISTER` transactions are:

- `BYTE` - 8 bit byte
- `WORD16` - 16-bit two's complement integer
- `WORD32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating point number

These data types are also specified during configuration of a VXI device and do not appear in the transaction.

ExpressionList is a single expression or a comma-separated list of expressions.

`INCR` specifies that array data is to be read from the register incrementally starting at the address specified by *SymbolicName*. The first element of the array is read from the starting address, the second from that address plus an offset equal to the length in bytes of the data type, and so forth until all array elements have been read. If `INCR` is not specified in the transaction, the entire array is read from the single location specified by *SymbolicName*.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

MEMORY Encoding

`READ MEMORY` is used to read values from a VXI device's A24 or A32 memory.

`READ MEMORY` transactions are of this form:

> `READ MEM:` *SymbolicName ExpressionList* `INCR`
> `ARRAY:`*NumElements*
> `-or-`
> `READ MEM:` *SymbolicName ExpressionList*
> `ARRAY:`*NumElements*

where:

SymbolicName is a name defined during configuration of a VXI device. The name refers to a specific address within a device's extended memory. Specific data types for `READ MEMORY` transactions are:

- `BYTE` - 8 bit byte
- `WORD16` - 16-bit two's complement integer
- `WORD32` - 32-bit two's complement integer
- `REAL32` - 32-bit IEEE 754 floating point number

These data types are also specified during configuration of a VXI device and do not appear in the transaction.

ExpressionList is a single expression or a comma-separated list of expressions.

`INCR` specifies that array data is to be read from the memory location incrementally starting at the location specified by *SymbolicName*. The first element of the array is read from the starting location, the second from that location plus an offset equal to the length in bytes of the data type, and so forth until all array elements have been read. If `INCR` is not specified in the transaction, the entire array is read from the single memory location specified by *SymbolicName*.

NumElements is a single expression or a comma-separated list of expressions that specifies the dimensions of each variable in *VarList*. If the transaction is configured to read a scalar, the `ARRAY` keyword does not appear in the transaction. Note that `ARRAY:1` is a one-dimensional array with one element. HP VEE makes a distinction between scalars and one-dimensional arrays containing only one element.

IOSTATUS Encoding

`READ IOSTATUS` transactions are of this form:

```
READ IOSTATUS STS Bits VarList
-or-
READ IOSTATUS DATA READY VarList
```

VarList is a single Integer variable or a comma-separated list of Integer variables.

`READ IOSTATUS` transactions are used by `Direct I/O` for GPIO interfaces, `From StdIn`, `To/From Named Pipe`, `To/From Socket`, and `To/From HP BASIC/UX`.

`READ IOSTATUS` transactions for GPIO reads the peripheral status bits available on the interface. The number of bits read is dependent on the model number of the interface. A single integer value is returned that is the weighted sum of all the status bits.

For example, the HP 98622A GPIO interface supports two peripheral status lines, STI0 and STI1. Table D-14 illustrates how to interpret the value of `x` in this transaction:

```
READ IOSTATUS STS Bits a
```

Table D-14. `IOSTATUS` **Values**

Value Read	STI1	STI0
0	0	0
1	0	1
2	1	0
3	1	1

`READ IOSTATUS` transactions read the instantaneous values of the status lines; the status line are not latched or buffered in any way.

`READ IOSTATUS` transactions for `To/From Named Pipe`, `To/From Socket`, `To/From HP BASIC/UX` and `From StdIn` returns a Boolean YES (1) if there is data ready to read. If no data is present, a Boolean NO (0) is returned. The `READ IOSTATUS` transaction can be used to avoid a READ that will block program execution until data is available.

EXECUTE Transactions

EXECUTE transactions send low-level commands to control the file, instrument, or interface associated with a particular object. EXECUTE is used to adjust file pointers, clear buffers, and provide low-level control of hardware interfaces. The various EXECUTE commands available are summarized in Table D-15.

Table D-15. Summary of EXECUTE Commands

Commands	Description
To File, From File	
REWIND	Sets the read pointer (From File) or write pointer (To File) to the beginning of the file without changing the data in the file.
CLEAR	(To File only). Erases existing data in the file and sets the write pointer to the beginning of the file.
CLOSE	Explicitly closes the file. Useful when multiple processes are reading and writing the same file.
DELETE	Explicitly deletes the file. Useful for deleting temporary files.

Table D-15. Summary of EXECUTE Commands (continued)

Commands	Description
Interface Operations	
CLEAR	For HP-IB clears all devices by sending DCL (Device Clear). For VXI, resets the interface and runs the resource manager
TRIGGER	For HP-IB triggers all devices addressed to listen by sending GET (Group Execute Trigger). For VXI triggers specified backplane trigger lines or external triggers on an embedded controller.
LOCAL	For HP-IB releases the REN (Remote Enable) line, and puts instrument into local mode.
REMOTE	For HP-IB asserts the REN (Remote Enable) line.
LOCAL LOCKOUT	For HP-IB sends the LLO (Local Lockout) message. Any device in remote at the time LLO is sent will lock out front panel operation.
ABORT	Clears the HP-IB interface by asserting the IFC (Interface Clear) line.
LOCK INTERFACE	In a multiprocess system with shared resources, lets one process lock the resources for its own use during a critical section to prevent another process from trying to use them.
UNLOCK INTERFACE	In a multiprocess system where a process has locked shared resources for its own use, unlocks the resources to allow other processes access to them.
Direct I/O to HP-IB	
CLEAR	Clears device at the address of a Direct I/O object by sending the SDC (Selected Device Clear).
TRIGGER	Triggers the device at the address of a Direct I/O object by addressing it to listen and sending GET (Group Execute Trigger).
LOCAL	Places the device at the address of the Direct I/O object in the local state.
REMOTE	Places the device at the address of the Direct I/O object in the remote state.

Table D-15. Summary of EXECUTE Commands (continued)

Commands	Description
Direct I/O to GPIO	
RESET	Resets the GPIO interface associated with the Direct I/O object by pulsing the PRESET line (Peripheral Reset).
Direct I/O to message-based VXI	
CLEAR	Clears the VXI device associated with the Direct I/O object by sending the word-serial command Clear (0xffff).
TRIGGER	Triggers the VXI device associated with the Direct I/O object by sending the word-serial command Trigger (0xedff).
LOCAL	Places the VXI device associated with the Direct I/O object into local state by sending the word-serial command Clear Lock (0xefff).
REMOTE	Places the VXI device associated with the Direct I/O object into local state by sending the word-serial command Set Lock (0xeeff). in the remote state.
Direct I/O to Serial Interfices	
RESET	Resets the serial interface associated with the Direct I/O object.
BREAK	Transmits a signal on the Data Out line of the serial interface associated with the Direct I/O object as follows: 1. A logical High for 400 milliseconds 2. A logical Low for 60 milliseconds

Table D-15. Summary of `EXECUTE` Commands (continued)

Commands	Description
Execute Program, To/From Named Pipe, To/From HP BASIC/UX	
`CLOSE READ PIPE`	Closes the read named pipe associated with the (To/From) object or the stdin pipe associated with the (Execute Program).
`CLOSE WRITE PIPE`	Closes the write named pipe associated with the (To/From) object or the stdout pipe associated with the (Execute Program).
To/From Socket	
`CLOSE`	Closes the connection between client and server sockets. To re-establish the connection, the client and server must repeat the bind-accept and connect-to protocols.
Direct I/O, MultiDevice Direct I/O, Interface Operations to HP-IB, GPIB, VXI, Serial, GPIO	
`LOCK`	In a multiprocess system with shared resources, lets one process lock the resources for its own use during a critical section to prevent another process from trying to use them.
`UNLOCK`	In a multiprocess system where a process has locked shared resources for its own use, unlocks the resources to allow other processes access to them.

Details About HP-IB

The `EXECUTE` commands used by `Direct I/O` to HP-IB devices and `Interface Operations` are similar but different.

- `Direct I/O` `EXECUTE` commands address an instrument to receive the command.
- `Interface Operations` `EXECUTE` commands may affect multiple instruments already addressed to listen.

The following series of tables indicate the exact bus actions conducted by `Direct I/O` and `Interface Operations` `EXECUTE` transactions.

Table D-16. `EXECUTE ABORT` **HP-IB Actions**

`Direct I/O`	`Interface Operations`
Not applicable.	IFC ($\geq$ 100 μsec) REN $\overline{\text{ATN}}$

Table D-17. `EXECUTE CLEAR` **HP-IB Actions**

`Direct I/O`	`Interface Operations`
ATN	ATN
MTA	DCL
UNL	
LAG	
SDC	

Table D-18. `EXECUTE TRIGGER` **HP-IB Actions**

`Direct I/O`	`Interface Operations`
ATN	ATN
MTA	GET
UNL	
LAG	
GET	

Table D-19. `EXECUTE LOCAL` **HP-IB Actions**

`Direct I/O`	`Interface Operations`
ATN	$\overline{\text{REN}}$
MTA	$\overline{\text{ATN}}$
UNL	
LAG	
GTL	

Table D-20. `EXECUTE REMOTE` **HP-IB Actions**

`Direct I/O`	`Interface Operations`
REN	REN
ATN	$\overline{\text{ATN}}$
MTA	
UNL	
LAG	

Table D-21. `EXECUTE LOCAL LOCKOUT` **HP-IB Actions**

`Direct I/O`	`Interface Operations`
Not applicable.	ATN
	LLO

Details About VXI

The EXECUTE commands used by Direct I/O to VXI devices and Interface Operations are similar, but different. References to message-based VXI devices apply to register-based devices that are supported by I-SCPI.

- Direct I/O EXECUTE commands address a message based VXI device to receive a word-serial command.
- Interface Operations EXECUTE commands affect the VXI interface directly and may affect VXI devices within the interfaces servant area.

EXECUTE TRIGGER transactions for the Interface Operations object are of the form:

```
EXECUTE TRIGGER TriggerType Expression TriggerMode
```

TriggerType specifies which trigger group will be used by the

EXECUTE TRIGGER transaction. The groups are:

- TTL - Specifies the eight TTL trigger lines on the VXI backplane.
- ECL - Specifies the four ECL trigger lines on the VXI backplane.
- EXT - Specifies the external triggers on a embedded VXI controller.

Expression evaluates to a single Integer variable that represents a bit pattern indicating which trigger lines for a particular *TriggerType* are to be triggered. A value of 5, represented in binary as 101, indicates that TTL lines 0 and 2 are to be triggered. A value of 255 triggers all eight TTL lines.

TriggerMode indicates the way the trigger lines are to be asserted:

- PULSE - Lines are to be pulsed for a discreet time limit (*TriggerType* dependent).
- ON - Asserts the trigger lines and leaves them asserted.
- OFF - Removes the assertion from trigger lines that were asserted by a previous ON transaction.

The following series of tables indicate the exact bus actions conducted by `Direct I/O` and `Interface Operations` `EXECUTE` transactions.

Table D-22. `EXECUTE CLEAR` **VXI Actions**

`Direct I/O`	`Interface Operations`
Word-serial command Clear(0xffff)	Pulse SYSRESET line, rerun Resource Manager

Table D-23. `EXECUTE TRIGGER` **VXI Actions**

`Direct I/O`	`Interface Operations`
Word-serial command Trigger(0xedff)	Triggers either the TTL or ECL trigger lines in the backplane, or the external trigger(s) on the embedded VXI controller. You can specify which lines are to be triggered for each trigger type.

Table D-24. `EXECUTE LOCAL` **VXI Actions**

`Direct I/O`	`Interface Operations`
Word-serial command Set Lock(0xeeff)	Not applicable.

Table D-25. `EXECUTE REMOTE` **VXI Actions**

`Direct I/O`	`Interface Operations`
Word-serial command Clear Lock(0xefff)	Not applicable.

WAIT Transactions

There are four types of `WAIT` transactions:

- `WAIT INTERVAL`
- `WAIT SPOLL` (`Direct I/O` to HP-IB and message based VXI devices only)
- `WAIT REGISTER` (`Direct I/O` to VXI devices only)
- `WAIT MEMORY` (`Direct I/O` to VXI devices only)

`WAIT INTERVAL` transactions simply wait for the specified number of seconds before executing the next transaction listed in the open view of the object. For example, this transaction waits for 10 seconds:

```
WAIT INTERVAL:10
```

`WAIT SPOLL` transactions are of the form:

`WAIT SPOLL` *Expression Sense*

Expression is an expression that evaluates to an integer. The integer will be used as as a bit mask.

Sense is a field with two possible values.

- `ANY SET`
- `ALL CLEAR`

`WAIT SPOLL` transactions wait until the serial poll response byte of the associated instrument meets a specific condition. The serial poll response is tested by bitwise ANDing it with the specified mask and ORing the resulting bits into a single test bit. The transaction following `WAIT SPOLL` executes when one of the following conditions is met:

- The transaction specifies `ANY` (`ANY SET`) and the test bit is true (1).
- The transaction specifies `CLEAR` (`ALL CLEAR`) and the test bit is false (0).

The following transactions show how to use `WAIT SPOLL`:

`WAIT SPOLL:256 ANY`	*Wait until any bit is set.*
`WAIT SPOLL:256 CLEAR`	*Wait until all are clear.*
`WAIT SPOLL:0x40 ANY`	*Wait until bit 6 is set.*
`WAIT SPOLL:0x40 CLEAR`	*Wait until bit 6 is clear.*

`WAIT REGISTER` and `WAIT MEMORY` transactions are of the form:

WAIT REG:*SymbolicName* MASK:*Expression* *Sense*
[*Expression*]
-or-
WAIT MEM:*SymbolicName* MASK:*Expression* *Sense*
[*Expression*]

where:

SymbolicName is a name defined during configuration of a VXI device. The name refers to a specific address within a device's A16 or extended memory.

`MASK:`*Expression* is an expression that evaluates to an integer. The integer will be used as as a bit mask. The size in bytes of this mask value depends on the data type for which *SymbolicName* has been configured.

Sense is a field with three possible values.

- `ANY SET`
- `ALL CLEAR`
- `EQUAL`

[*Expression*] is an optional compare value that evaluates to an integer. The integer is used only when *Sense* is `EQUAL`.

`WAIT REGISTER` or `MEMORY` transactions wait until the value read from the register or memory location specified by *SymbolicNames* in the associated VXI device meets a certain condition. The value read is logically ANDed with the bit mask specified in `MASK:`*Expression*, resulting in a test value. The size of the test value is dependent on the data type configured for the specified register or memory location. The transaction following `WAIT SPOLL` executes when one of the following conditions is met:

- The transaction specifies `ANY` (`ANY SET`) and the test value has at least one bit true (1).
- The transaction specifies `CLEAR` (`ALL CLEAR`) and the test value has all bits false (0).
- The transaction specifies `EQUAL` and the test value is equal bit-for-bit with the compare value specified in [*Expression*].

SEND Transactions

`SEND` transactions are of this form:

`SEND` *BusCmd*

BusCmd is one of the bus commands listed in Table D-26.

`SEND` transactions are used within `Interface Operations` objects to transmit low-level bus messages via an HP-IB interface. These messages are defined in detail in IEEE 488.1.

Table D-26. `SEND` Bus Commands

Command	Description
`COMMAND`	Sets ATN true and transmits the specified data bytes. ATN true indicates that the data represents a bus command.
`DATA`	Sets ATN false and transmits the specified data bytes. ATN false indicates that the data represents device-dependent information.
`TALK`	Addresses a device at the specified primary bus address (0-31) to talk.
`LISTEN`	Addresses a device at the specified primary bus address (0-31) to listen.
`SECONDARY`	Specifies a secondary bus address following a TALK or LISTEN command. Secondary addresses are typically used by cardcage instruments where the cardcage is at a primary address and each plug-in module is at a secondary address.
`UNLISTEN`	Forces all devices to stop listening; sends UNL.

Table D-26. `SEND` **Bus Commands (continued)**

Command	Description
`UNTALK`	Forces all devices to stop talking; sends UNT.
`MY LISTEN ADDR`	Addresses the computer running HP VEE to listen; sends MLA.
`MY TALK ADDR`	Addresses the computer running HP VEE to talk; sends MTA.
`MESSAGE`	Sends a multi-line bus message. Consult IEEE 488.1 for details. The multi-line messages are: DCL Device Clear SDC Selected Device Clear GET Group Execute Trigger GTL Go To Local LLO Local Lockout SPE Serial Poll Enable SPD Serial Poll Disable TCT Take Control

WRITE(POKE) Transactions

The `WRITE(POKE)` transaction is very similar to the `WRITE` transaction, except that it applies only to the `To/From DDE` object. The main difference of `WRITE(POKE)` is that you must specify an item name. For example:

```
   WRITE ITEM:"r2c3" TEXT a EOL
```

`WRITE(POKE)` transactions are supported by HP VEE for Windows only.

The following encodings are allowed:

- `TEXT`
- `BYTE`
- `CASE`
- `CONTAINER`

For more specific information about these formats see the `WRITE` transaction.

READ(REQUEST) Transactions

The `READ(REQUEST)` transaction is very similar to the `READ` transaction, except that it applies only to the `To/From DDE` object. The main difference of `READ(REQUEST)` is that you must specify an item name. For example:

```
READ ITEM:"r2c3" TEXT a EOL
```

`READ(REQUEST)` transactions are supported by HP VEE for Windows only.

The following encodings are allowed:

- `TEXT`
- `CONTAINER`

For more specific information about these formats see the `READ` transaction.

E

HP VEE for UNIX and HP VEE for Windows Differences

In general, programs written in HP VEE on one platform will work on any other supported platform. The only difficulties that arise are when you use programs that access features specific to the underlying platform, such as DLL's on PC's or named pipes on UNIX. This appendix contains information on the differences between HP VEE on UNIX and PC platforms.

Execute Program

There is an `Execute Program` object for both the UNIX and PC platforms. Note that you can determine which platform you are executing on by using the `whichPlatform()` or `whichOS() Formula` objects. You can then programmatically determine which `Execute Program` object to use.

DLL versus Shared Library

There are several differences that must be noted when creating DLL's and Shared Libraries for Compiled Functions.

I/O	From a Shared Library you do I/O through SICL, DIL or `TERMIO`. For DLL's use SICL. To avoid systemic resource conflicts, be sure your source code uses library commands that support the platform and interface system the compiled function will run on.
Graphics	Shared Libraries use X11 graphics while DLL's use Microsoft Windows GDI calls. Link Shared Libraries against the X Windows Release 5 of the library. While a compiled function runs in an X Window, HP VEE cannot service its human interface.

Data Files

No binary files will work across platforms since byte ordering is reversed between UNIX and PC platforms. However, ASCII data files written using `To File` objects are readable by `From File` objects on other platforms. Also, HP VEE program files are compatible since they are stored in ASCII. Note that when moving ASCII data files from one platform to another, UNIX files use the linefeed character to terminate lines while MS Windows uses the carriage return/linefeed sequence to terminate lines.