



## Extending VEE via custom menus and DLLs

September 22, 1997

This paper was written to instruct you on how to integrate calls to a DLL into HP VEE. It also shows how to modify the VEE menu to allow your users/customers to use the functionality you are delivering. It will cover options from easiest (less work for you, more for your customer) to hardest (more work for you, less for your customer). It uses as examples the implementation from Data Translation and Computer Boards, Inc. (OEMs for HP VEE) to integrate calling DLLs in order to communicate with their PC plug-in boards. This paper shows how to use Microsoft Visual C++ (4.2) development environment to debug the DLL. Other IDE's are not covered.

The options are (in easiest to hardest order):

1. Include in your DLL, which will be callable from VEE, the functions you think are necessary. This may involve writing some wrapper code to call your DLL from VEE, providing an interface file for those functions, and a few examples showing how to call the functions. HP VEE does not support all the data types that are available in C/C++. Single character types, unsigned values and single precision floating point (32 bit) number data types are not supported by VEE. This will be discussed in detail later in this paper.
2. All of 1 above, but creating an add-on menu to VEE and organizing it in common areas of functionality.
3. All of 2 above, but adding help functionality to the objects (User Objects) you provide. This involves writing a help file and putting the necessary code into the VEE objects so that the customer can get help on the function by using the built-in help system with HP-VEE.

This paper will start with a general tutorial on creating compiled functions in HP VEE. It closely follows the VEE documentation (see the Advanced Programming Techniques manual, chapter 4). This is designed to be the starting point for learning how to integrate PC plug-in cards with HP VEE.

### Compiled Functions in HP VEE

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 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 time slice (i.e. they execute until they are done without interruption) they are only useful for specific purposes that are not available in VEE.

Although you can extend the capabilities of your VEE program by using Compiled Functions, it adds complexity to the VEE programming process. The key design goal should be to keep the purpose of the external routine highly focused on a specific task. You currently cannot access any of the VEE internal functions from within the DLL.

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

- VEE does not normally trap errors originating in the external routine. Because your external routine becomes part of the VEE process, any errors in that routine will propagate back to VEE, and a failure in the external routine may cause 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 VEE. There is another white paper on this titled “Catching Exceptions in a DLL. See the support pages in our web page to get a copy of this document.
- 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. You should use the non-multithreaded memory library, which is unfortunately not the default in MS Visual C.
- Your external routine cannot convert data types the way 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 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 VEE to exit or error with a segmentation violation.

## Importing and Calling a Compiled Function

Once you have created a dynamically linked library, you can import the library into your VEE program with the “Import Library” object and then call the Compiled Function with the “Call” object. 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 DLL. The Definition File field specifies the name of the include file which contains the specification for the functions to be called:

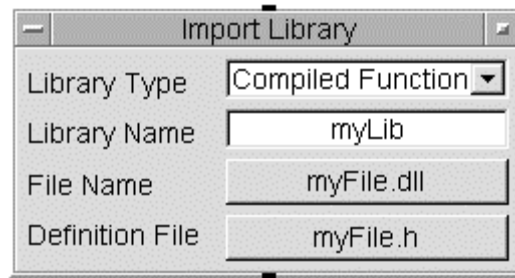


Figure 1: Using Import Library for Compiled Functions

The definition file defines the type of data that is passed between the external routine and 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 “myFunc”.

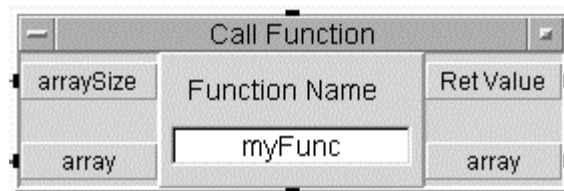


Figure 2: Using Call for Compiled Functions

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 any case, provided VEE recognizes the function, the input and output terminals of the “Call” object are configured automatically for the function. The definition file supplies the necessary information. Or, you can reconfigure the “Call” input and output terminals by selecting “Configure Pinout” in the object menu. Whichever method you use, 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.

The DLL remains bound to the VEE process until VEE terminates, or until the library is expressly deleted.

You delete the DLL from 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 a DLL 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.

## 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 used in the **Import Library** object. The definition file defines the type of data the function returns, the function name, and the arguments the function accepts. When VEE executes an Import Library object, 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”, VEE automatically configures the **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.

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

## Function Definition

Function definitions are of the following general form:

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

Where:

<return type> can be: **int, short, long, double, char\*, or void.**

The optional “[\_hidden]” parameter has the effect of “hiding” the function in VEE although it can be called as with any other function. The difference is that it will not appear in the Explorer view or the Function Selection dialog box. You might want to use this if your DLL / menu picks has entries which are called from encapsulated objects, but you do not want to “clutter” up the Explorer/Function selection dialog box with unnecessary functions.

The **int, short** and **long** types are all passed as a 32-bit value to your function. The **int** and **long** types are 32 bits and the **short** is 16 bits.

<function name> can be a string consisting of an alpha character followed by alphanumeric characters (the ‘\_’ character is also allowed), 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 (the ‘\_’ character is also allowed), up to a total of 512 characters. The parameter names are optional, but it is highly recommended to include them as VEE will just create a name on the **Call** box (a, b, c, etc) if it is missing. If a parameter is to be passed by reference, the parameter name must be preceded by the indirection symbol (\*).

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

If you specify “pass by reference” for a parameter by preceding the parameter name with “\*”, VEE will pass the address of the information to your function. If you specify “pass by value” for a parameter by leaving out the “\*”, VEE will copy the value (rather than the address of the value) to your function. You’ll want to pass the data by reference if your external routine changes that data for propagation back to VEE. Also, all arrays must be passed by reference or VEE will signal an error on the **Call** object.

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 functions return value, plus an output for each input parameter that was passed by reference.

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.

You may include comments in your definition file. 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.

An example definition file might look like this:

```
/*  
    This function performs the snicker doodle transform on  
    the input array.  
*/  
long myFunc(long arraySize, double *array);
```

## Creating a Compiled Function

There are several steps to the process of creating a Compiled Function. First you must write a program in C or C++ and compile it to create a Dynamic Link Library (DLL) containing the Compiled Function, and bind the shared library into the VEE process. We'll look at each step in turn.

## Building a C Function

The following C function accepts a real array and adds 1 to each element in the array. The modified array is returned to VEE on the Array terminal, while the size of the array is returned on the Ret Value terminal. This function, once linked into HP VEE, becomes the Compiled Function called in the VEE program shown in Figure 3 below. All the code shown below comes with VEE and is in the examples/manual directory.

```

/*
   C code from manual49.c file
*/

#include <stdlib.h>

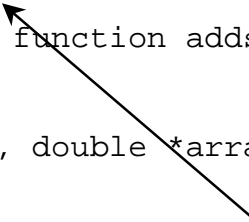
#define DLLEXPORT __declspec(dllexport)

/* 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;
    }
}

```



You can add text to the function selection box by adding this to the DLL. See Figure 4.

You must include any header files on which the routine depends in the source file. The library should link against any other system libraries needed to resolve the system functions it calls.

Notice the `myFunc_desc[ ]` entry above. If you add entries like this for each of your DLL calls, then when the user looks at them with the Function selection box. See figure 4 below. In addition, this string will be displayed in a dialog box if the user selects the "Description" selection with a right-button-down mouse click in the Explorer view on that function. The general form for these strings is:

```
__declspec(dllexport) char <function name>_desc[] = "some
string information"
```

The `<function name>` must be exactly the same name as the function itself with a `"_desc[ ]"` appended to it.

The definition file, `manual49.h`, for the function in `manual49.c` 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.)

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

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

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 VEE. This technique is useful when you need to return an array that has fewer elements than the input array.



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

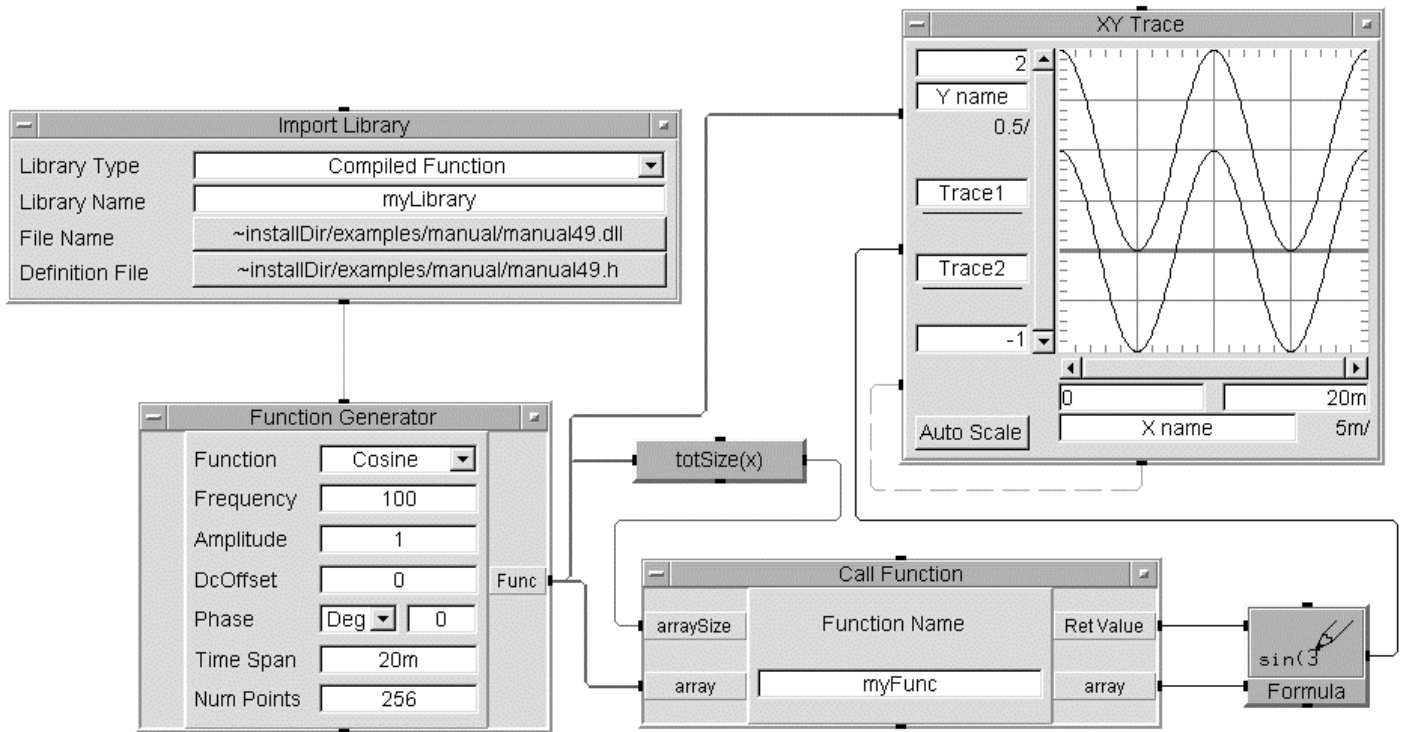


Figure 3: Program Calling a Compiled Function

The example in Figure 3 is located in the file manual49.vee in the examples directory. The C file is in manual49.c, the definition file is in manual49.h, and the shared library is in manual49.dll.

After importing the function (either execute the program or select “Load Lib” on the Import Library object menu) you can see that the function is available on the Function Selection dialog box and the Explorer view in VEE.

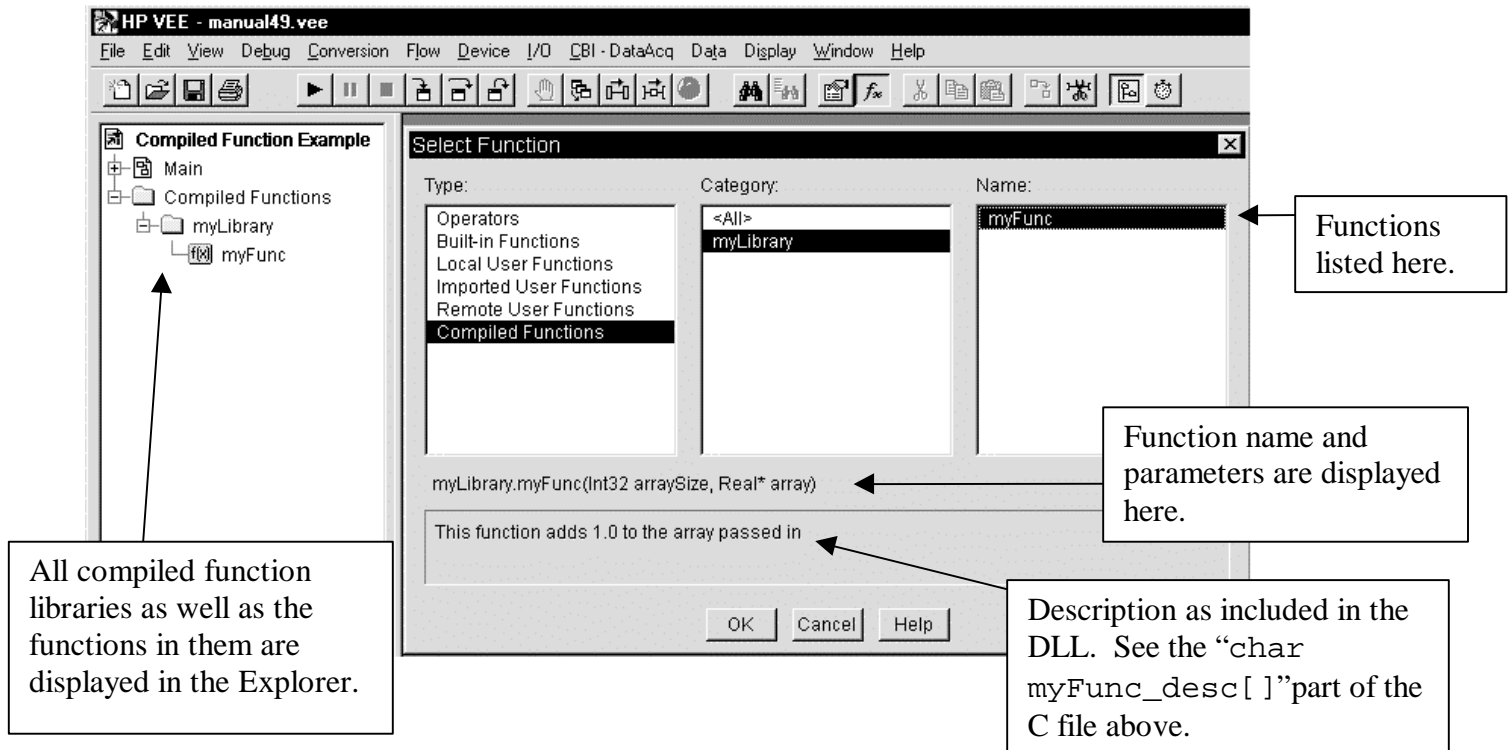


Figure 4. Shows how a compiled function is integrated into VEE.

### Creating the DLL

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 manual49.c /LD /Zi /link Kernel32.lib
```

“/LDd” creates the debug version of the DLL. Use “/Zi” to generate debug information. This DLL can easily be created in one of the commercial development environments (instead of using the command line interface) if you so choose.

## More on the Call object

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 on the “Call” object so that the parameters passed by reference appear as output terminals. 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 glarch(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 “glarch”. If the number of input and output terminals do not exactly match the number of parameters in the function VEE generates an error. If the DLL library has already been loaded and you enter the function name in the “Function Name” field you can also use the “Configure Pinout” selection on the “Call” object menu to configure the terminals.

## The Delete Library Object

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

## Using DLL Functions in Formula Objects

You can also use DLL functions in formula objects (Formula box, If/Then box, I/O objects, etc). With formula objects, only the return value is used in the formula; the parameters passed by reference cannot be accessed. For example, using the DLL function defined above in a formula:

$$4.5 + \text{glarch}(a, b, c) * 10$$

where “a” is the top input terminal on the formula object, “b” is next and “c” is last. The call to “glarch” must have the correct number of parameters or VEE generates an error. If the function “glarch” returns the value 0, the calculation would then proceed computing 4.5 as the answer.

## DLL specifics

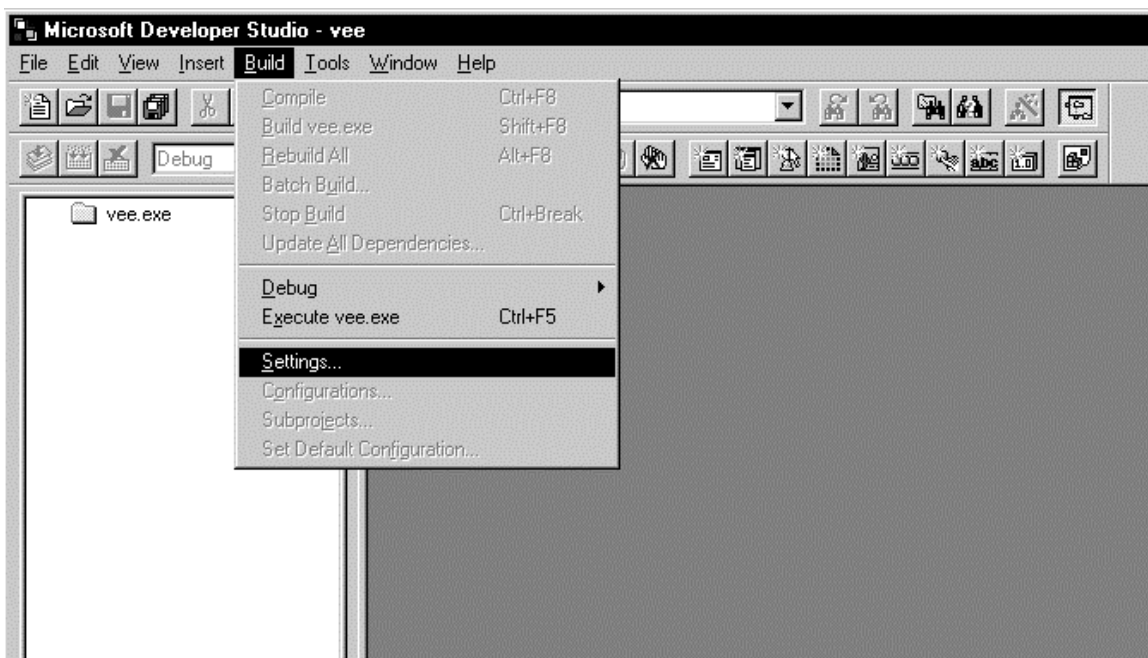
### Debugging your DLL

This section will show you how to debug your DLL running inside of HP VEE. I used the Microsoft Visual C (version 4.2) environment. I leave it as an exercise of the reader to do the same thing with other packages.

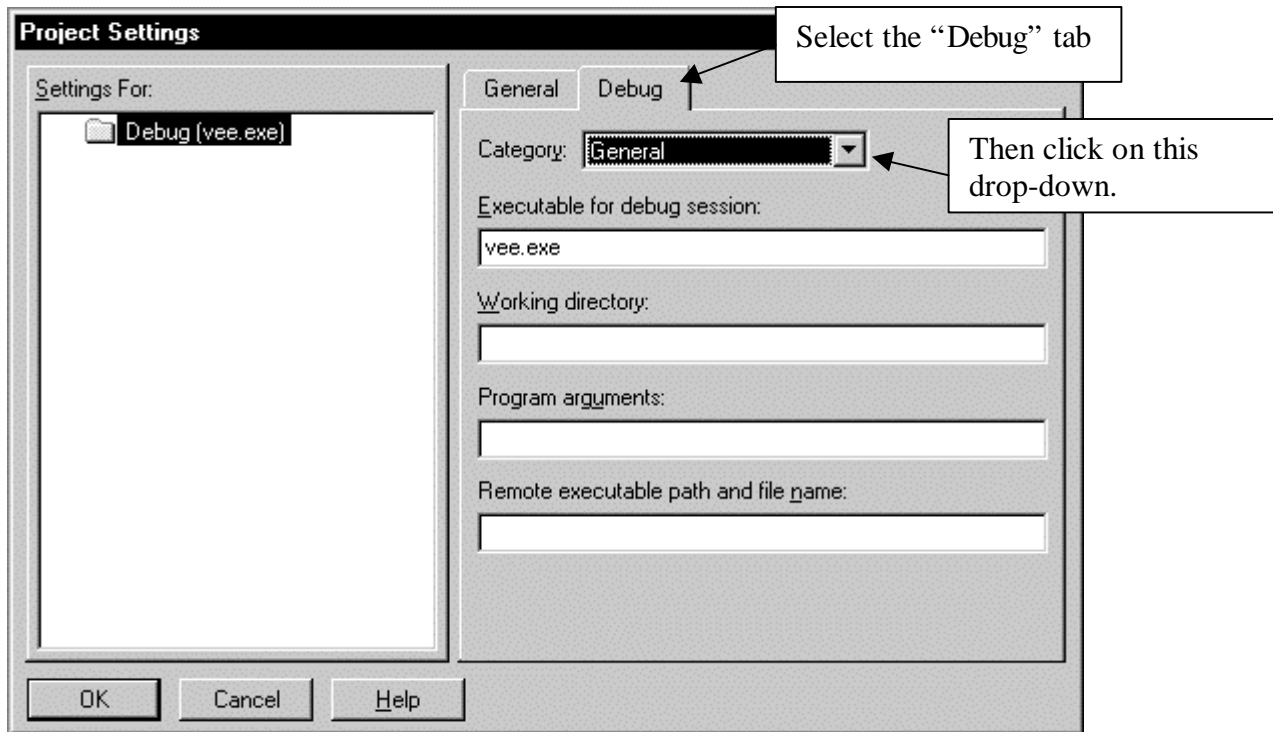
1. From the command line I issued the command:

```
Msdev "d:\Program Files\Hewlett Packard\VEE 4.0\vee.exe"
```

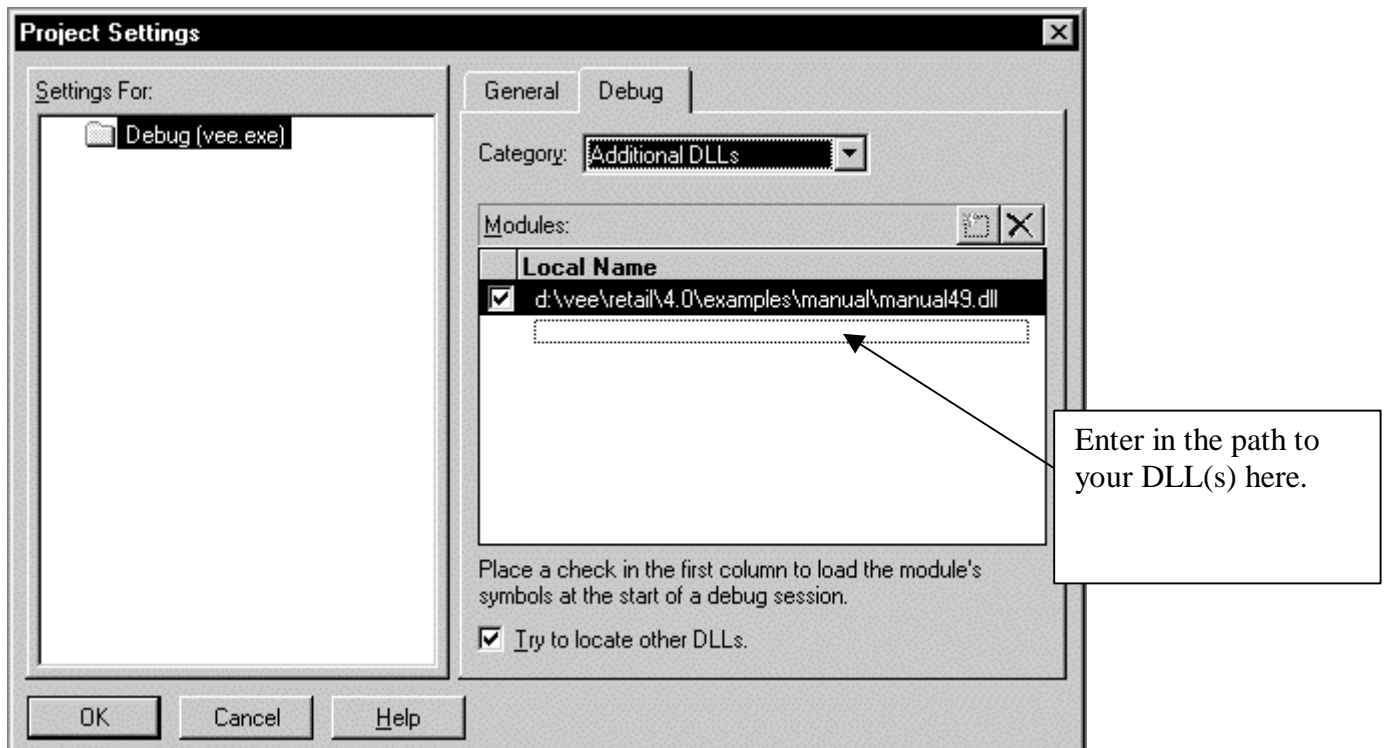
2. This brought up the environment that looks like this, where I selected the Build → Settings menu pick:



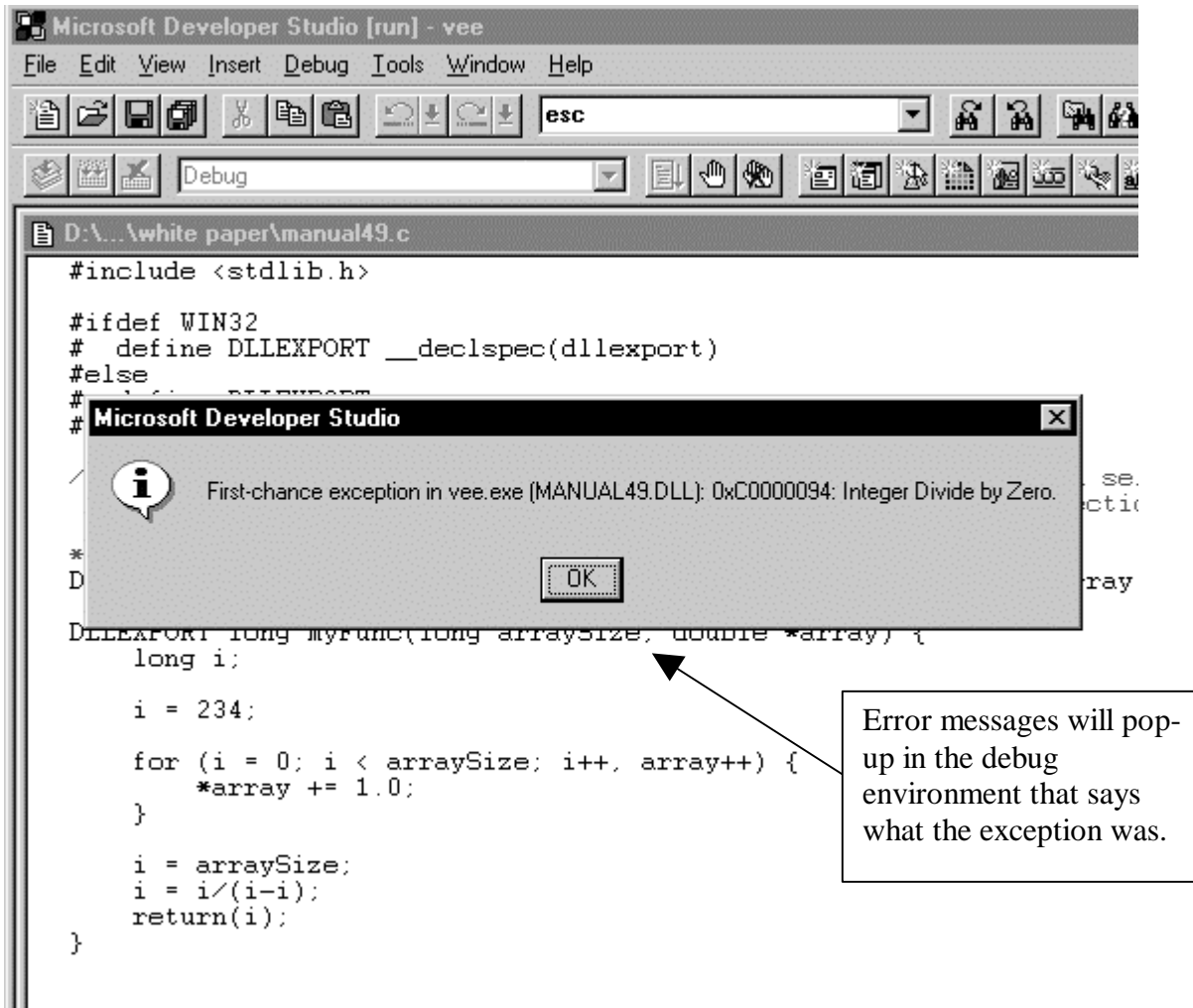
3. Which brings up a dialog box like this:



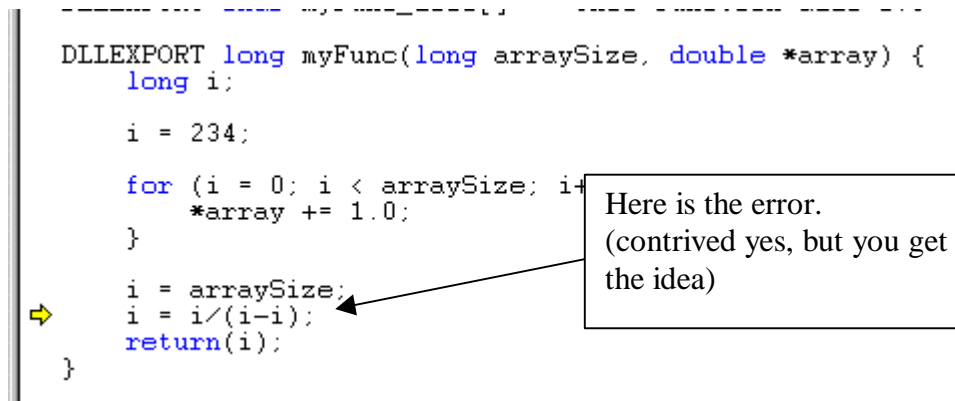
Select the "Debug" tab and click on the "General" drop-down selection and pick "Additional DLLs" and the dialog box will look like this:



4. Enter in the DLL (or DLLs) that will be used when your VEE program runs. You might want to review the exceptions that your debug environment will catch (Debug → Exceptions... for MSVC).
5. Press F5 to run VEE.
6. Load your VEE program as normal and run it.
7. Let's say that the DLL has an error in it. You will get some kind of message box from the debug environment that will look something like this:



8. After clicking on “OK”, the offending source code line will be displayed. You may have to set some options saying where the source code is for the DLL.



In this fashion you can debug your DLL using the shipping version (versus getting a special “debug” version of the bits) of VEE. In addition to this you might want to consider getting a copy of the Pure-Atria’s Purify software. It excels at finding all kinds of problems with memory overwrites, stomps, leaks, API problems. I have used it with great success on VEE and highly recommend it. A word of caution though – you will need a fairly decent machine to run it on. Get lots of memory, too.

### DLL initialization and cleanup

You might want to do some initialization work when your DLL is first loaded and cleanup work right before the DLL is unloaded. HP VEE will automatically look for a function called “libEntry( )” in your DLL and call it when VEE loads (via the LoadLibrary call) the DLL. This is useful for doing any initialization when the DLL is first loaded. It will only get called once. Conversely VEE will look for and call a function called “libExit( )” when the DLL is unloaded. The prototypes for these two functions is: “void libEntry(void)” and “void libExit(void)”. That is, no parameters and no return values. Make sure you export these functions correctly.

### Neat Debugger trick

If your DLL has some kind of error condition in it you can have your program start the debugger directly by calling the DebugBreak() Win32 function at the point in your code that you want debugger to start. Calling this function causes the program to display a dialog box as if it had crashed. Click Cancel to start the debugger and continue on in debug mode. Just insert the following line of code in your DLL where you want the debugger to start.

```
VOID DebugBreak(VOID)
```

## Alternate debug process

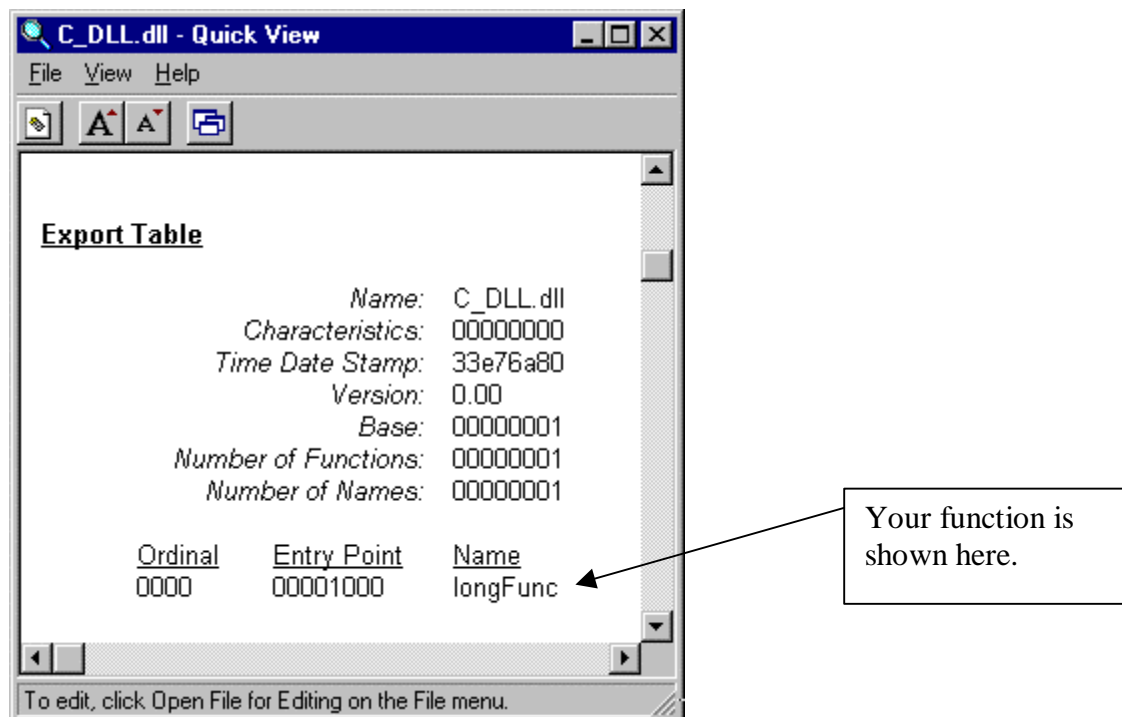
There is an alternate way to debug your freshly built DLL. Assuming that the DLL is built in the visual environment (MS Visual C++, etc.) then you can simply run enter the path to VEE to execute in the “Executable for debug session” part of the Project Settings dialog box. Run the executable (F5) and make sure the Import box points to the Debug version of the DLL.

## C or C++ file?

The file containing the code for your DLL can be in either a C (.c extension) or a C++ (.cpp extension) file. Which one you choose does have an effect on your DLL. Suppose you have the code as follows:

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

in a file called C\_DLL.c. When you compile the DLL the function “longFunc” will be exported and available in VEE. Using the Explorer in Windows, find the DLL and select “Quick View” via the right mouse button. You will see something like this:



If you put the same exact function in a file with the extension .cpp (C++) file and compile it without declaring it as an extern C function, C++ will “mangle” the function name to something that is not useful in VEE. Suppose the code looks like this:

```
/* This function name will be mangled by C++ */  
__declspec( dllexport ) long longFunc(long a, long b) {  
    long c;
```



```

        c = a + b;
        return c;
    }

    /* This function is properly exported as a C function. */
    extern "C" {
    __declspec( dllexport ) long longFunc2(long a, long b) {
        long c;
        c = a + b;
        return c;
    }
    }
}

```

Using “Quick View” again to examine the DLL shows the following two entry points. Notice that the function “longFunc” name is completely “mangled” into something that only this version of C++ understands. This is a normal (and one of the problems of) thing for the C++ compiler to do. Each compiler does the mangling differently. There can even be variations from revision to revision of the same vendor’s compiler!

**Export Table**

Name: CPP\_DLL.dll  
 Characteristics: 00000000  
 Time Date Stamp: 33e76b06  
 Version: 0.00  
 Base: 00000001  
 Number of Functions: 00000002  
 Number of Names: 00000002

Ordinal	Entry Point	Name
0000	00001005	?longFunc@@YAJJJ@Z
0001	00001000	longFunc2

To edit, click Open File for Editing on the File menu.

Annotations:

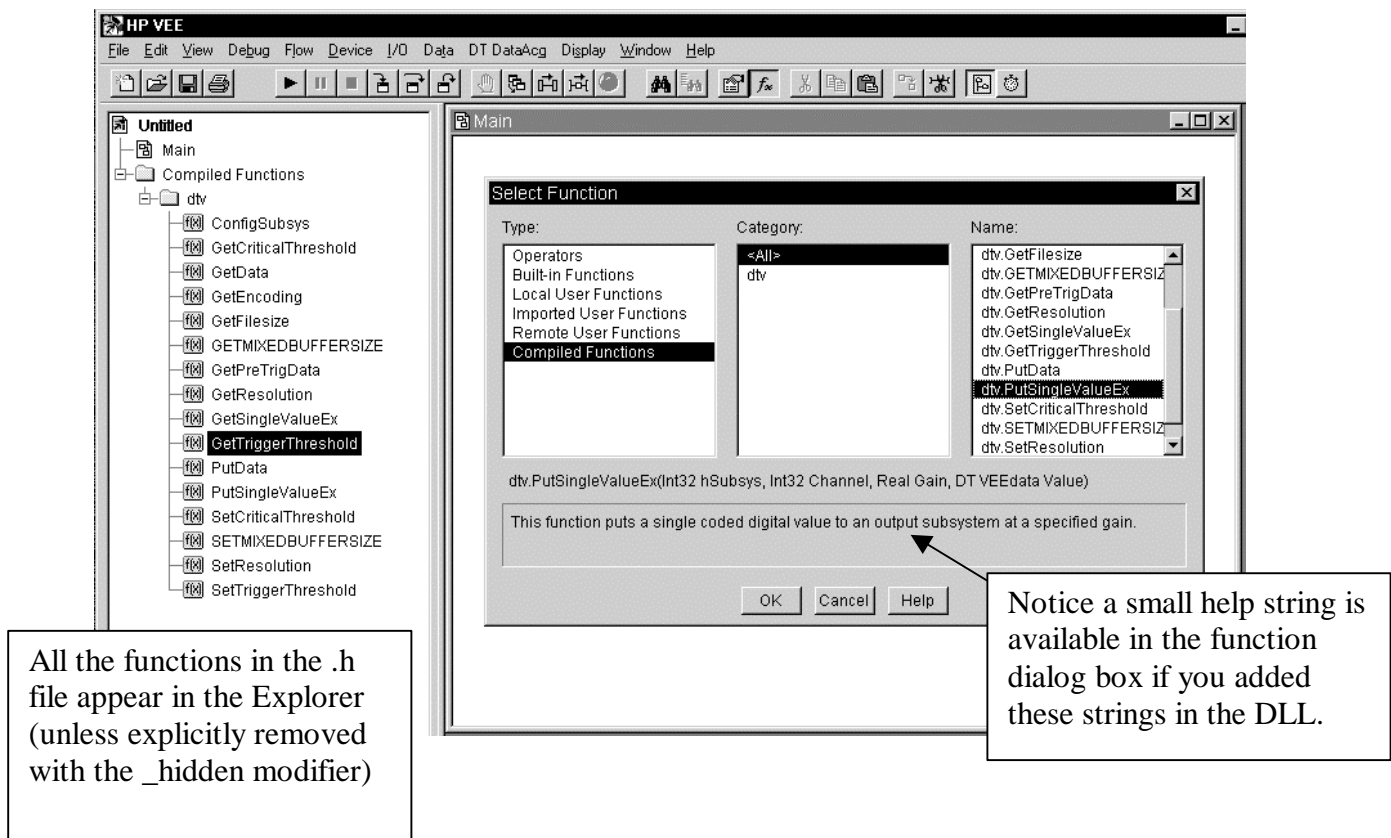
- longFunc name was mangled by C++ (points to ?longFunc@@YAJJJ@Z)
- The “longFunc2” function was not. It is declared as “extern C” in the code. (points to longFunc2)

Now let’s look at the different options of integrating the DLL into VEE.

## Option 1. Integrating the DLL into VEE

This section will show how to integrate your DLL into VEE doing the bare minimum work. Provide your customers the DLL, a header file and perhaps a few example programs on how to talk to the plug-in card with the DLL.

When you import the DLL (using one of the examples) you will see new entries on the VEE 4.0 Explorer and also one the Function Selection dialog box. It might look like the following:



So where here do I put the DLL and examples?

If you have some kind of installation program (I highly recommend InstallShield) you will have to decide where to put the DLL(s), VxD/Driver, VEE examples. The recommended place to put the VEE examples would be relative to the VEE installation directory in **your own** subdirectory, say for example, in the directory:

```
"C:\ProgramFiles\Hewlett Packard\VEE 4.01\cbi".
```

Here the examples would go into the "cbi" subdirectory and your example programs could be constructed to find other items needed in that subdirectory. Say for instance one of the VEE examples needs to load another DLL, or execute a program or load a library of UserFunctions. You should use ~installDir\cbi\<whatever\_filename> directive in your VEE program to get to the files instead of relying on the installation forcing the user to put the files in a certain spot.

If you writing an install program, you can find out where VEE is installed by looking at the registry key: HKEY\_LOCAL\_MACHINE\SOFTWARE\Hewlett-Packard\VEE. There may be one or more versions of VEE installed there which means you will have to enumerate all the version of VEE available and either let the user choose one of entries or determine which is newest (highest numeric value).

Or you can query the key:

HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\App Paths\vee.exe

To find out the path of the most recently installed version of VEE. Each time you install VEE this value is overwritten.

That's it! The user will use your DLL and the examples to construct the appropriate VEE program to talk to your card.

## Option 2. Integration using the VEE menu

The next step would be to modify the VEE menu to include blocks of functionality. The menu objects can be entire example programs, single **Call** objects, or small User Objects configured to call directly to your DLL. This section will show you how to create a file that will modify the VEE menu. There is a file that ships with VEE in the lib subdirectory named customMenu.mnu, which contains a working example of code that will modify your VEE menu. If you copy this file into the install directory, then when you start VEE the menu will have additional items in it. This section of the paper will use these items from customMenu.mnu as a working example.

First some general rules about the menu files:

- Files with the extension ".mnu" that exist in the VEE installation directory will be loaded as VEE menu modifications. The files are loaded in alphabetical order.
- Lines starting with '#' are comments.
- There **MUST** be a line of this form for each menu modification entry:

**"MENU->xxxxx" .**

There must be double quotes around the string and it must start with the characters **MENU->**. The **xxxxx** characters describe where in the menu the item will appear. VEE will parse the string and look for the menu or menu item between each set of '->' characters and put the new menu item BEFORE the one specified in the string. For instance, an entry of the form **"MENU->Data->Constant->Text"** will insert the item into the Data menu, the Constant submenu BEFORE the Text entry. To put a menu item at the end of a menu, use END keyword. See example below.

- There may be an optional visibility selector of the form:

```
[visibleWhen notRunning]
```

This entry will “gray-out” the menu picks when VEE is running. Generally most (but not all) VEE menu items have this. It is not required.

- An optional description that shows up on the status bar when the cursor is over that menu item. It is not required but highly recommended.

```
[desc "The rain in Spain stays mainly on the plain"]
```

- The item itself. You can either specify it via a line like:

```
getDevicesFrom: "~installDir\dataacq\adconfig.vlo"
```

This is the easiest way to build a menu because you can maintain all the menu items in separate files. I used this method here to minimize clutter.

Note the "~installDir" modifier. This tells this version of VEE to look in the directory that the user installed VEE into to find the file "dataacq\adconfig.vlo" and load the object there.

**- or -**

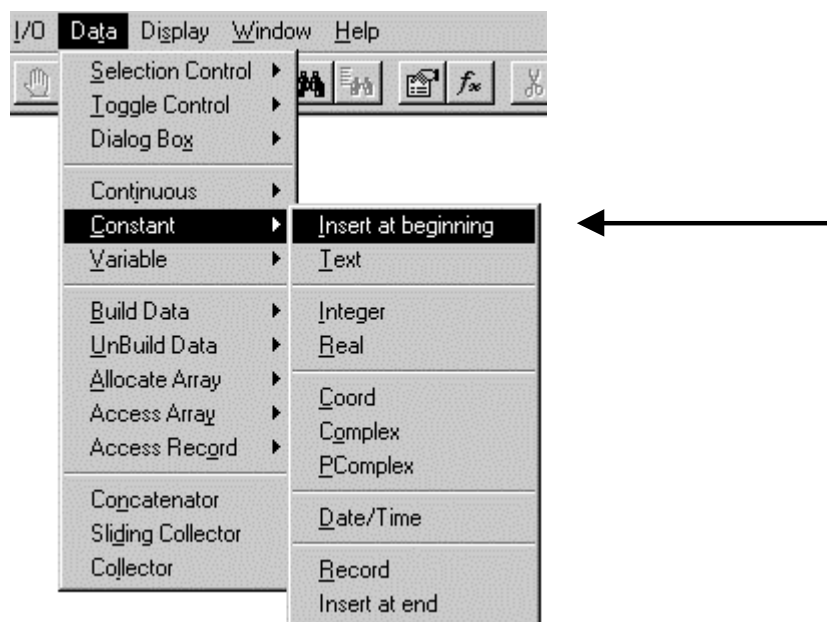
You can do it the hard way ☺ and embed the object directly in the menu as shown in an example below.

It is generally better to put your files to be loaded in a subdirectory relative to the VEE install directory instead of using absolute file names. It makes coding your examples easier (they don't have to depend on making sure that the hardcoded paths are correct) and your installation process easier. The installation process can look in the LOCAL\_MACHINE registry entries for the Software\Hewlett-Packard\VEE registry entries, enumerate them and decide what the most recent version is. Based on that there are subkeys that specify what the installation directory for VEE is. You can install your software in a sub-directory relative to that.

Here are several examples of VEE menu file modifications and how they look in VEE.

Install of single item in cascaded sub-menu structure.

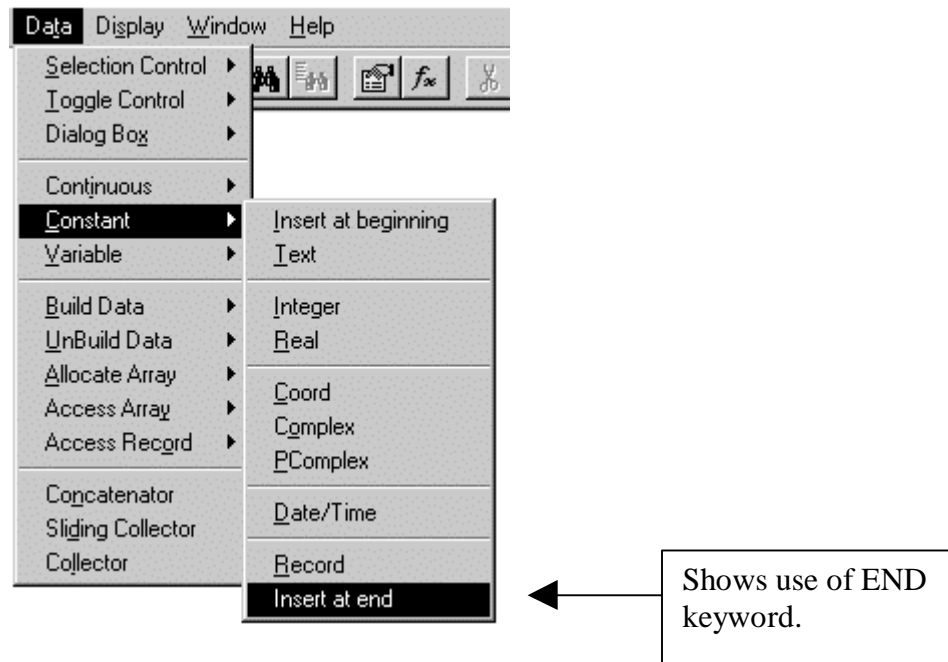
```
"MENU->Data->Constant->Text "  
( "&Insert at beginning"  
  [visibleWhen notRunning]  
  [desc "Test install of single item in menu structure"]  
  getDevicesFrom: "~installDir/lib/convert/Rad2Grad.vee"  
)
```



Install of single item at the end of a cascaded sub-menu.

```
"MENU->Data->Constant->END "  
( "Insert at end"  
  [visibleWhen notRunning]  
  [desc "Install a single item at end of Constant menu"]  
  getDevicesFrom: "~installDir/lib/convert/Grad2Rad.vee"
```

)



Install of entire main level cascaded menu.

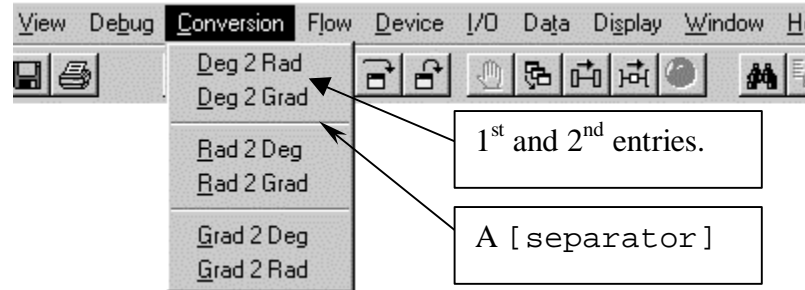
This shows how to insert an entire “top-level” menu pick on the VEE menu bar.

```
"MENU->Flow"
("&Conversion"
 ("&Deg 2 Rad"
    [visibleWhen notRunning]
    getDevicesFrom: "~installDir/lib/convert/Deg2Rad.vee"
    [desc "converts number from degrees to radians"]
  )
 ("&Deg 2 Grad"
    [visibleWhen notRunning]
    getDevicesFrom: "~installDir/lib/convert/Deg2Grad.vee"
    [desc "converts number from degrees to gradians"]
  )
  [separator]
 ("&Rad 2 Deg"
    [visibleWhen notRunning]
    getDevicesFrom: "~installDir/lib/convert/Rad2Deg.vee"
    [desc "converts number from radians to degrees"]
  )
 ("&Rad 2 Grad"
    getDevicesFrom: "~installDir/lib/convert/Rad2Grad.vee"
    [visibleWhen notRunning]
  )
)
```

```

    [desc "converts number from radians to gradians"]
)
... Rest was deleted ...
)

```



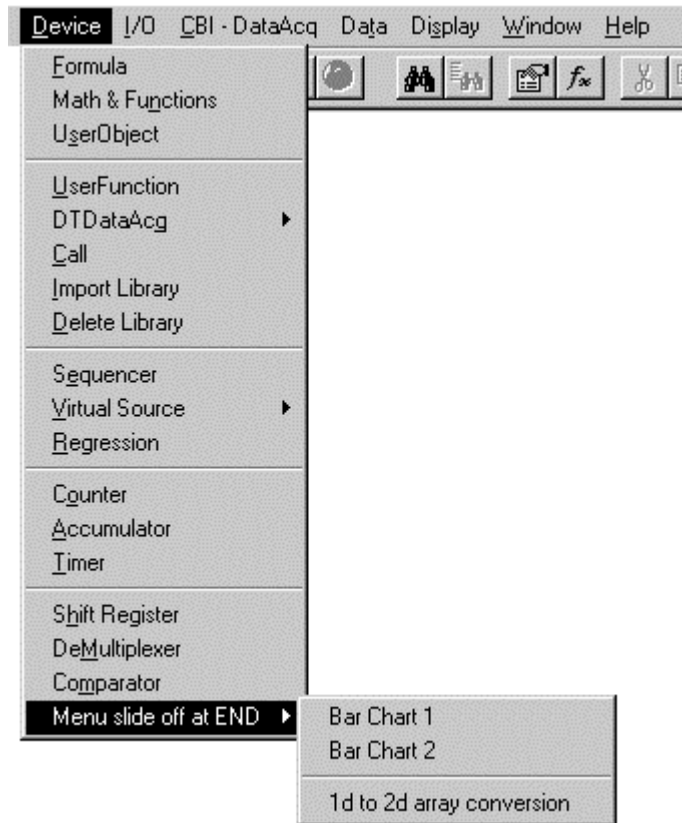
Install of cascaded sub-menu at END (of Device menu).

This shows an example of a cascaded sub-menu that is installed at the END of a VEE main menu selection.

```

"MENU->Device->END"
("Menu slide off at END"
  ("Bar Chart 1"
    [visibleWhen notRunning]
    getDevicesFrom: "~installDir/examples/lib/barcht1.vee"
    [desc "bar chart 1"]
  )
  ("Bar Chart 2"
    [visibleWhen notRunning]
    getDevicesFrom: "~installDir/examples/lib/barcht2.vee"
    [desc "bar chart 2"]
  )
  ("1d to 2d array conversion"
    [visibleWhen notRunning]
    getDevicesFrom: "~installDir/examples/lib/1Dto2D.vee"
    [desc "1D to 2D array conversion"]
  )
))

```



Insert all information directly in menu file.

You may also install an item in the menu by encoding the VEE save information directly in the menu file. This is the "hard" way - i.e. having to put the item here directly. You must save the object(s) first and then cut the information from the save file and paste it correctly into the menu file. This example is here mainly to show how you can link in your help file to a specific VEE User Object or Call object. I don't recommend it, but I mention it, as this is the way the VEE menu is built and shipped (internally) in the product. The "helpLink" entry is how you link a help file to a VEE object that you create. This topic is covered in the next section.

```
"MENU->Device->Call"
( "DTDataAc&q"
  ("Get &Single Value"
    [visibleWhen notRunning]
    [desc "Test install of cascaded menu on main-submenu."]
    createObject: `(component 0 "CALL"
      (helpLink "Get-Single-Val@~installDir/DTVPI.HLP")
      (name "Get Single Value")
      (interface
        (sequence in)
        (sequence out)
        (input 1
          (type data)
          (name "hSubsys")
          (requires (datatype Int32)))
```



```
(lock name constraints))  
... Rest was deleted ...
```

Install of single item at end of Main menu item (Display).

```
"MENU->Display->END"  
( "&Pie chart"  
  [visibleWhen notRunning]  
  [desc "An encapsulated UserObject that displays a Pie Chart"]  
  getDevicesFrom: "~installDir\lib\PieChart.vee")
```



Menu entry

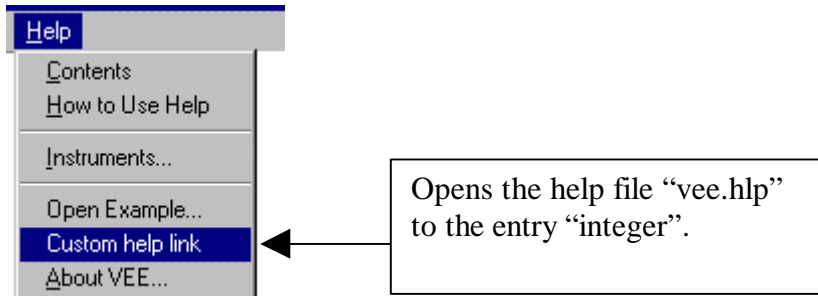
An encapsulated UserObject that displays a Pie Chart

This text appears in the status bar at the bottom of the VEE window. Notice it is the same text entered in the "desc" part of the menu entry.

Install a custom help menu link.

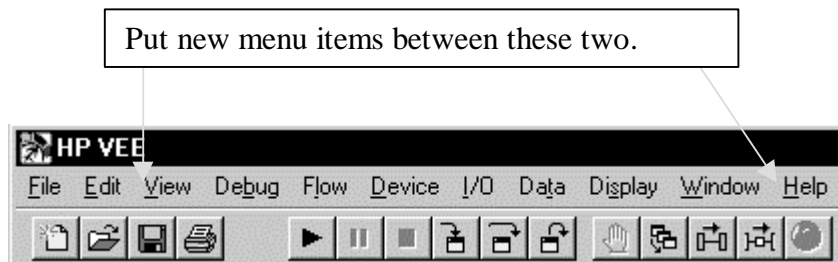
There is a way to provide custom help links in VEE. This example shows how to add an entry to the "Help" menu itself. For demonstration purposes, I have used the "vee.hlp" file and the link in that file called "integer" to show how to do it. You can substitute any valid help file for "vee.hlp" and any valid help link for "integer".

```
"MENU->Help->About VEE..."
("Custom help link"
 [desc "Creates a link to a custom help file"]
 "help:" "integer@~installDir/vee.hlp"
)
```



### ***Menu bar protocol***

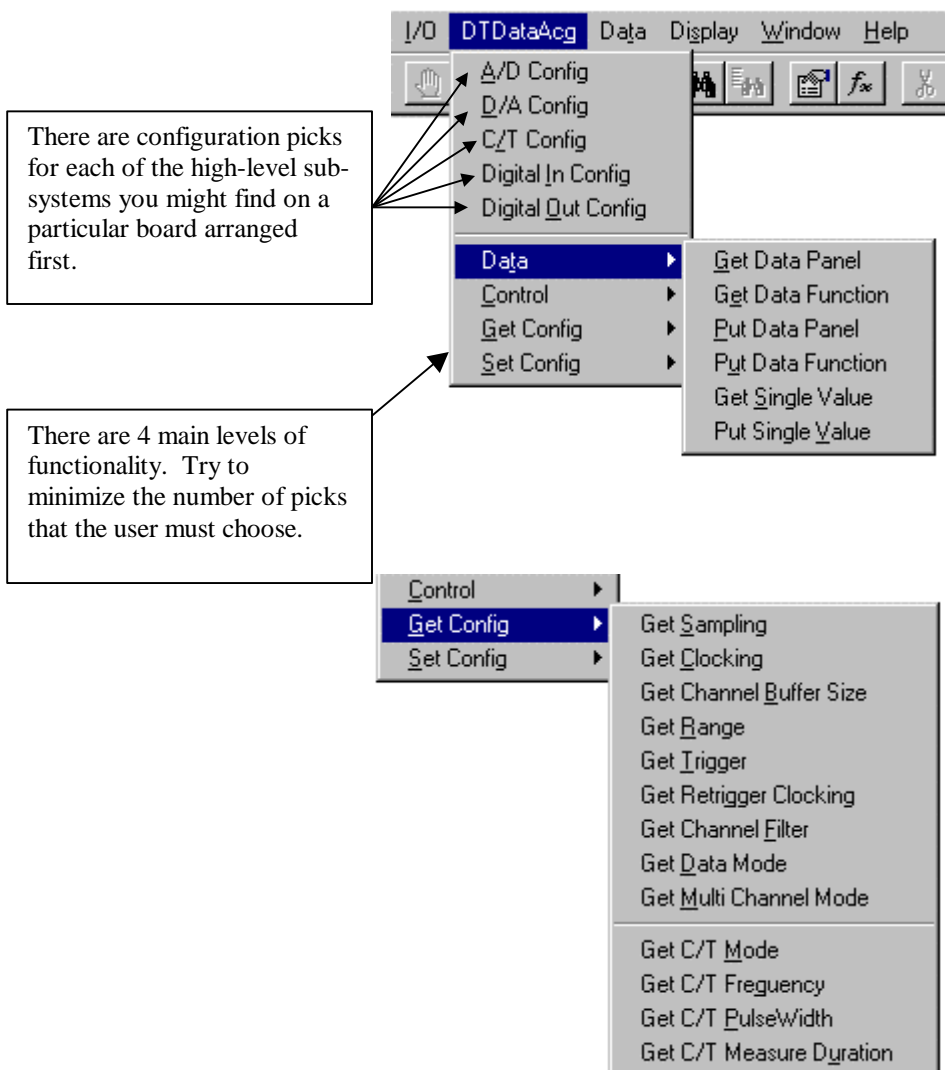
It is considered "against the standard" to insert items before the "Edit" menu bar pick or after the "Help" menu bar pick. Try not to do this.



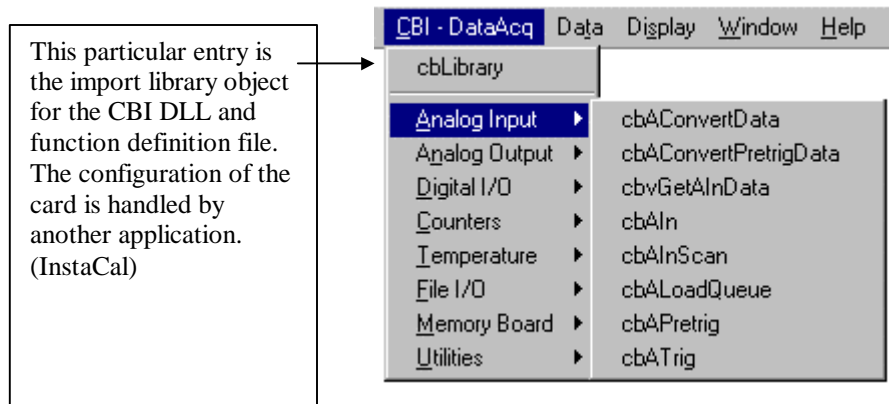
## Vendor examples

OK. Let's get specific now. Here are a couple of examples show how Data Translation and Computer Boards have integrated their menus into VEE.

The first is from Data Translation. They install a top-level menu that is divided into two sections. The first is for configuration of card functionality. Individual cards have different areas of functionality so you might have to use several of these Config objects to get your card set up. The second section includes all the calls you need to make to set or get data from the card as well as doing some other low-level chores.



This example is how Computer Boards (CBI) integrated their cards into VEE:



The other menus/sub-menus are calls to the DLL Library, which is the method they use to talk with their VxD and hardware.

### Option 3. Integration incorporating VEE help

This is an easy addition to the VEE menus, but requires more work on your part so I broke it out as a separate item. Notice the item from the prior section that has an entry similar to this:

```
(helpLink "Get-Single-Val@~installDir/DTVPI.HLP")
```

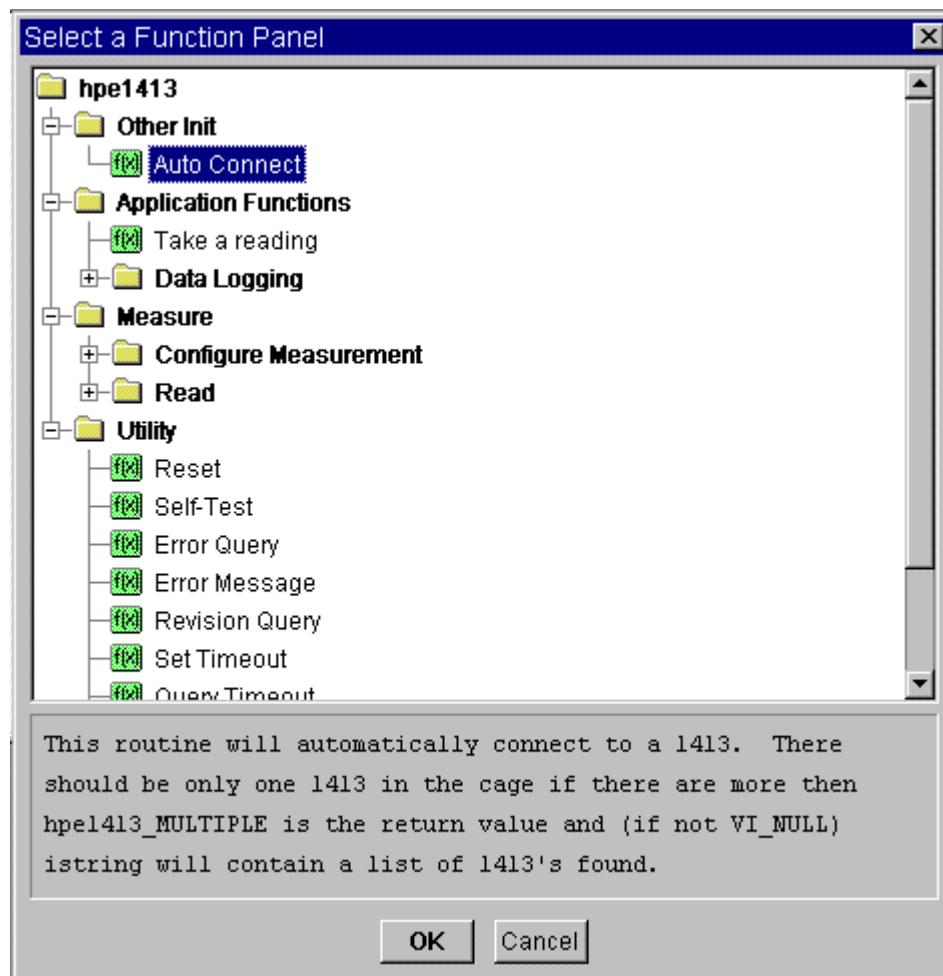
This is the way you can get VEE to open YOUR custom help file (dtvpi.hlp in this case) and go to the specified help link (Get-Single-Val). The down side to this is that you currently have to manually edit the program file and insert this entry into the object that you want to put help on. If you are putting secured User Objects on the menu, put the entries in the User Object and it will be preserved across program save/open cycles. The up side is that when the user selects "Help" on that object, VEE will open the correct help file. In this case the help file must be in the VEE install directory and contain a valid link called "Get-Single-Val". It is very easy to build help files but is beyond the scope of this paper to tell you how to do this. There are many commercial products available to help build help files such as ForeHelp, RoboHelp and others.

Note that you must provide a valid path to the help file. Use of ~installDir is the preferred way to do this or else you must force the help file to the correct place when you install your custom object and or menus.

You can also embed a call to the function "help( )" inside of VEE to do the same thing using the proper OK button or other mechanism to call it. Select the help function on the Function selection dialog box and hit the Help button for more information on this function.

## Option 4. Writing a VXI plug&play *style* driver

There is another option if you have a PC plug-in card. You could write a VXI plug&play *style* (not to be confused with Microsoft Plug and Play hardware/software specification) driver. These drivers provide all of 1-3 outlined on the first page but in a different format. Think of it as a “Visual Function Call”. It provides a nice tree view of the functions in your DLL (really an instrument driver). The big advantage of this kind of driver is that it can be used with HP VEE, Microsoft C, Microsoft Visual Basic, National Instruments LabView and National Instruments LabWindows CVI. It is much more work, but the benefits for a single driver for multiple software packages are obvious. HP will be writing instrument drivers of this type for all its new hardware – including traditional HPIB interface boxes. They look something like this in HP VEE:



But, this is such an enormous topic that I am not going to cover it in this paper.

<b>EXTENDING VEE VIA CUSTOM MENUS AND DLLS</b>	<b>1</b>
Importing and Calling a Compiled Function	2
Function Definition	4
Creating a Compiled Function	6
Building a C Function	6
Creating the DLL	10
More on the Call object	11
The Delete Library Object	11
Using DLL Functions in Formula Objects	11
 <b>DLL SPECIFICS</b>	 <b>12</b>
Debugging your DLL	12
DLL initialization and cleanup	15
Neat Debugger trick	15
Alternate debug process	16
C or C++ file?	16
 <b>OPTION 1. INTEGRATING THE DLL INTO VEE</b>	 <b>18</b>
So where here do I put the DLL and examples?	18
 <b>OPTION 2. INTEGRATION USING THE VEE MENU</b>	 <b>19</b>
Install of single item in cascaded sub-menu structure.	21
Install of single item at the end of a cascaded sub-menu.	21
Install of entire main level cascaded menu.	22
Install of cascaded sub-menu at END (of Device menu).	23
Insert all information directly in menu file.	24
Install of single item at end of Main menu item (Display).	25
Install a custom help menu link.	25
 <b>Menu bar protocol</b>	 <b>26</b>
 <b>VENDOR EXAMPLES</b>	 <b>27</b>
 <b>OPTION 3. INTEGRATION INCORPORATING VEE HELP</b>	 <b>29</b>
 <b>OPTION 4. WRITING A VXI PLUG&amp;PLAY <i>STYLE</i> DRIVER</b>	 <b>30</b>