About Compiled Functions

A Compiled Function is created by dynamically linking a library written in C, C++, FORTRAN, or Pascal, to the VEE process. A library of compiled functions is called a shared library in UNIX and a dynamic link library (DLL) in Microsoft Windows.

Creating a Compiled Function is considerably more difficult than creating a UserFunction. Once you have written a library of functions in C or another language, you will need to compile the functions into a DLL or shared library. You will also have to create a definition file that will provide VEE with information it needs to call your function.

Using a Compiled Function

To use a Compiled Function, you:

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

Note

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

The methods for importing a Compiled Function library and for calling the function are very similar to those for UserFunction libraries. The `Import Library` object attaches the DLL to the VEE process and parses the definition file declarations.

The definition file defines the type of data passed between the external library and VEE. (This file is discussed later in this section.) The Compiled Function can then be called with the `Call` object or from such objects as `Formula` and `If/Then/Else`.

Design Considerations for Compiled Functions

Using Compiled Functions, you can develop time-sensitive routines in another language and integrate them directly into your VEE program. You can also use Compiled Functions to keep proprietary routines secure.

Because Compiled Functions do not timeslice (i.e., they execute until they are done without interruption) they are only useful for specific purposes that are not otherwise available in VEE.

You can extend the capabilities of your VEE program by using Compiled Functions, but it adds complexity to the VEE process. The key design goals should be:

- Keep the purpose of the external routine highly focused on a specific task
- Use Compiled Functions only when the capability or performance you need is not available using a VEE UserFunction or an `Execute Program` escape to the operating system.

You can use any operating system facilities available in the program to be linked, including math routines, instrument I/O, etc. However, you cannot access any VEE internal functions from within the external program to be linked.

Although the use of Compiled Functions provides enhanced VEE capabilities, some problems can occur. A few key ones are:

- VEE cannot trap errors originating in the external routine. Because your external routine becomes part of the VEE process, any errors in that routine propagate back to VEE. A failure in the external routine may cause VEE to "hang" or otherwise fail. You need to be sure of what you want the external routine to do and provide for error checking in the routine. If your external routine exits so will VEE.
- Your routine must manage all memory that it needs. Be sure to deallocate any memory that you may have allocated when the routine was running.

- Your external routine cannot convert data types the way VEE does. You should configure the data input terminals of the `Call` object to accept *only* the type and shape of data that is compatible with the external routine.

- If your external routine accepts arrays, it must have a valid pointer for the type of data it will examine. The routine also must check the size of the array on which it is working. The best way to do this is to pass the size of the array from VEE as an input to the routine, separate from the array itself. If your routine overwrites values of an array passed to it, use the return value of the function to indicate how many of the array elements are valid.

- System I/O resources may become locked. Your external routine is responsible for timeout provisions, etc.

- If your external routine performs an invalid operation, such as overwriting memory beyond the end of an array or dereferencing a null or bad pointer, this can cause VEE to exit or error with a General Protection Fault (MS Windows) or a Segmentation Violation (UNIX).

- If your external routine has arrays or char* parameters, the memory passed to these routines must be allocated in VEE. You should allocate this memory by doing the following:

  - For an array input, use an `Alloc Array` object of the appropriate type, and set the size appropriately.

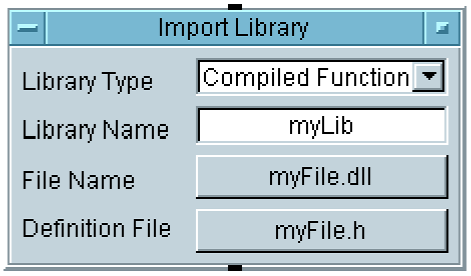
  - For a string input, use a `Formula` object. Delete the data input terminal from the `Formula` object and enter an expression like `256*"a"`. This creates a string that is 256 characters long (plus a null byte) filled with `a`'s. Most VXIplug&play functions will not write more than 256 characters into a `Text` parameter. However, it is best to check the Help on each function panel that requires a `Text` input to be sure.

Importing and Calling a Compiled Function

You can import a DLL into your VEE program with the `Import Library` object, then call the Compiled Function with the `Call` object. The process is very much like importing a library of UserFunctions and calling the functions, as described at the beginning of this chapter.

To import a Compiled Function library, select `Compiled Function` in the `Library Type` field.

Just as for a UserFunction, the `Library Name` field attaches a name to identify the library within the program, and the `File Name` field specifies the file from which to import the library. For a Compiled Function, there is a fourth field, which specifies the name of the `Definition File`, shown in Figure 12-4.


Import Library

Library Type: Compiled Function
Library Name: myLib
File Name: myFile.dll
Definition File: myFile.h

Figure 12-4. Using Import Library for Compiled Functions

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

After importing the library with `Import Library`, you can call the Compiled Function by specifying the function name in the `Call` object. For example, the `Call` object in Figure 12-5 calls the Compiled Function named `myFunction`.

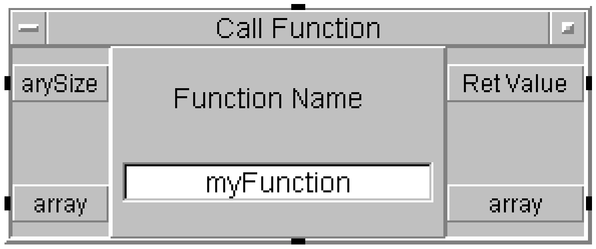


Figure 12-5. Using Call for Compiled Functions

Select the desired function using `Select Function` from the `Call` object menu or from the `Function & Object Browser` (under `Device` ⇒ `Function & Object Browser`), or type the name in the `Call` object.

If VEE recognizes the function, the input and output terminals of the `Call` object are configured automatically for the function. (The necessary information is supplied by the definition file.) You can reconfigure the `Call` input and output terminals by selecting `Configure Pinout` in the object menu.

VEE configures the `Call` object with the input terminals required by the function and with a `Ret Value` output terminal for the return value of the function. There also will be an output terminal corresponding to each input that is passed by reference.

You can also call the Compiled Function by name from an expression in a `Formula` object or from other expressions evaluated at run time. For example, you could call a Compiled Function by including its name in an expression in a `Sequencer` or `ToFile` transaction.

However, only the Compiled Function's return value (`Ret Value` in the `Call` object) can be obtained from within an expression. If you want to obtain other parameters from the function, you have to use the `Call` object.

The Definition File

The `Call` object or Formula expression determines the type of data it should pass to the function based on the contents of the definition file. The definition file defines the type of data the function returns, the function name, and the arguments the function accepts. The data has the following form:

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

Where:

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

The valid return types are:

- character strings (`char*`, corresponding to the VEE Text data type)
- integers (`short`, `int`, `long`, corresponding to the VEE `Int16` and `Int32` data types)
- single and double precision floating point real numbers (`float` and `double` corresponding to the VEE `Real32` and `Real64` data types).

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

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

VEE pushes 144 bytes on the stack. This allows up to 36 parameters to be passed by reference to a Compiled Function. Up to 36 long integer parameters or 18 double-precision floating-point parameters may be passed by value.

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.

VEE allows both "enclosed" comments and "to-end-of-line" comments in the definition file.

"Enclosed" comments use the delimiter sequence `/*comments*/`, where `/*` and `*/` mark the beginning and end of the comment, respectively.

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

Building a C Function

The following C function accepts a real array and adds `1` to each element in the array. The modified array is returned to VEE on the `Array` terminal, while the size of the array is returned on the `Ret Value` terminal. This function, once linked into VEE, becomes the Compiled Function called in the VEE program shown in Figure 12-6.

```
/*
  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 the same as the ANSI C prototype definition in the C file.)

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

The example program uses the ANSI C function prototype. The function prototype declares the data types that VEE should pass into the function.

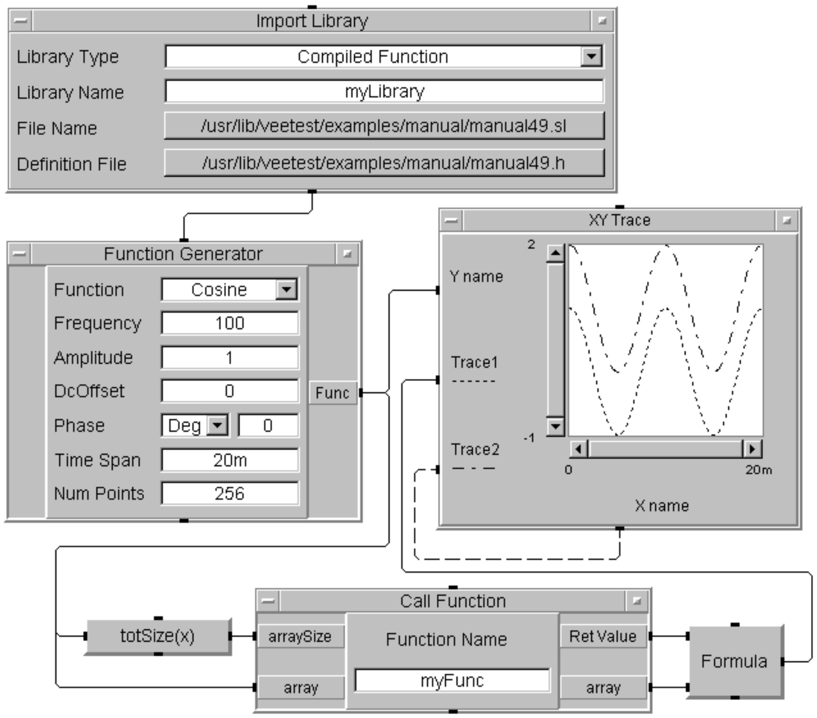
The array has been declared as a pointer variable. VEE will put the addresses of the information appearing on the `Call` data in terminals into this variable. The array size has been declared as a long integer. VEE will put the value (not the address) of the size of the array into this variable. The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

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

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

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

The program in Figure 12-6 calls the Compiled Function created from the 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

Call Function
Function Name: myFunc

Figure 12-6. Program Calling a Compiled Function

The example in Figure 12-6 is saved in the file `manual49.vee` in the `examples` directory. The C file is saved as `manual49.c`, the definition file as `manual49.h` and the shared library as `manual49.sl`.

Creating a Compiled Function (UNIX)

To create a Compiled Function 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 VEE process.

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 VEE environment. It must be linked with a special option to create a shared library.

Suppose the example C routine is in the file named `dLink.c`. To compile the file to be position independent, use the `+z` compiler option. You also need to prevent the compiler from performing the link phase by using the `-c` option. 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 (using the `-o` option) called `dLink.sl`.

Binding the Shared Library

VEE binds the shared library into the VEE process. All you need to do is include an `Import Library` object in your program, specifying the library to import, then call the function by name (i.e., with a `Call` object). When `Import Library` executes, VEE binds the shared library and makes the appropriate input and output terminals available to the `Call` object.

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 VEE process until VEE terminates or until the library is expressly deleted.

Delete the shared library from VEE either by selecting `Delete Lib` from the `Import Library` object menu, or by including the `Delete Library` object in your program. You may have more than one library name pointing to the same shared library file. If so, use the `Delete Library` object to delete each library. The shared library remains bound until the last library pointing to it is deleted.

The `Delete Lib` selection in the `Import Library` object menu unbinds the shared library without regard to other `Import Library` objects.

When 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`, 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. 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, etc. This is normally not a problem unless you add terminals after the automatic pin configuration.

Creating a Dynamic Link Library (MS Windows)

VEE for Windows provides access to DLLs through the `Call` object and through formula objects.

Note This section describes how to call a DLL, not how to write a DLL. VEE Version 3.2 and greater only calls 32-bit DLLs, not 16-bit DLLs.

Creating the DLL

Create the DLL before writing the VEE program. Create the DLL as you would any other DLL, except that only a subset of C types are allowed. (See “Creating the Definition File” on page 390.)

Declaring DLL Functions. If you are using Microsoft Visual C++ Version 2.0 or greater, the function definition should be:

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

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

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

`/LD` creates a DLL. Use `/Zi` to generate debug information.

The MS linker links to the C multi-threaded Runtime Library by default. If you use functions like `GetComputerName()`, you need to link against `Kernel32.lib`. The compile/link line would look like:

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

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

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

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

```
EXPORTS genericFunc
```

And the function definition looks like:

```
long genericFunc(long a);
```

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

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

where:

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

For example:

Pass in four parameters, return a long:

```
long aFunc(double *,long param2,long *param3, char *);
```

No input parameters, return a double:

```
double aFunc();
```

Pass in a string, return a long:

```
long aFunc(char *aString);
```

Pass in an array of strings, return a long:

```
long aFunc(char **aString);
```

Parameter Limitations

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

The Import Library Object

Before you can use a Call object or Formula box to execute a DLL function you must import the function into the VEE environment via the Import Library object. On the Import Library object, select Compiled Function under Library Type. Enter the correct definition file name using the Definition File button. Finally, select the correct file using the File Name button. The Library Name button assigns a logical name to a set of functions and does not need to be changed.

The Call Object

Before using a DLL function with the Call object you must configure the Call object. The easiest way to do this is to select Load Lib on the Import Library object menu to load the DLL file into the VEE environment. Then, select Select Function on the Call object menu.

VEE will bring up a dialog box with a list of all the functions listed in the definitions file. When you select a function, VEE automatically configures the Call object with the correct input and output terminals and function name.

You can also configure the Call object manually by modifying the function name and adding the appropriate input and output terminals:

1. Configure the same number of input terminals as there are parameters passed to the function. The top input terminal is the first parameter passed to the function. The next terminal down from the top is the second parameter, etc.

2. Configure the output terminals so the parameters passed by reference appear as output terminals on the Call object. Parameters passed by value cannot be assigned as output terminals. The top output terminal is the value returned by the function. The next terminal down is the first parameter passed by reference, etc.

3. Enter the correct DLL function name in the Function Name field.

For example, for a DLL function defined as

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

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

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

The Delete Library Object

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

Using DLL Functions in Formula Objects

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

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

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