

# Type Spoofin' For Dummies

## or How To Get VEE To Pass Nearly Anything Over The CFI

### What Is Type Spoofin'?

Well, we saw in [Windows Programming 101](#) how Windows does nearly everything it does (with a window anyway) by passing messages to window objects. Then we saw in [C Programming 101](#) how the all important consideration for the CFI is data size – not necessarily data type. We also saw how Windows defines a whole slew of goofy types that are actually just aliases for far more familiar C types. Which, in turn, are just different names for very familiar VEE types. Then we saw how there are "extended" types, or Record types. That which C calls a structure. These types are also called *compound* types because they are compounded of collections of familiar or *intrinsic* types.

Type Spoofing is passing one type over the CFI when Windows is expecting something totally different. As long as the *size* of the data is the same, VEE is plenty happy calling it one type and Windows is equally happy calling it some other type. We've already seen this in action. I've been calling a pointer an int. This is a simple type spoof. There are of course others. On a 32-bit platform, an int can also be a long. I mentioned how this can also break down and turn into a gotcha. When the bit width of the platform changes, so does the size of two types: int and pointer.

These simple spoofs are handy for passing pointers to objects over the CFI, but using a pointer in VEE involves other operations that we're not quite ready for yet, so lets concentrate on a far more interesting and useful spoof, the compound type spoof.

### A Closer Look At Compound Types.

Consider a simple VEE record. Let's say it contains an Int32 that will be a record number (IRecNo), a Real64 that is a time stamp (tTime), and an array of Real32 values that are each some measurement or other (pfData). This can be a useful structure for a data acquisition application. We can collect a number of these records into an array and permanently store them in a VEE Data Set — though to satisfy VEE's requirements on arrays of records the data arrays must all be the same size. There can be an exact duplicate of this record defined in C, and it's declaration is this:

```
struct _DATARECORD {  
    long    lRecNo;  
    double  tTime;
```

```
float    *pfData;
};
```

The name of this structure is `_DATARECORD`. It's usually useful to define a new alias for this type when it's declared, and in that case you'll see something like this:

```
typedef struct _DATARECORD {
    long    lRecNo;
    double  tTime;
    float    *pfData;
} DATARECORD, *PDATARECORD;
```

It looks nearly the same only that now there are two new types available to the compiler. They're called `DATARECORD` and `PDATARECORD`, and when the C pre-processor replaces those strings before the file is compiled they are replaced with `"struct _DATARECORD"` (meaning "a variable is of type `struct _DATARECORD`") and `"struct *_DATARECORD"` (meaning "a variable of type `struct _DATARECORD` pointer") respectively. Now, the interesting thing is that doing this (typedef-ing a compound type along with declaring its composition) is *precisely* the same as declaring a Record Type variable in VEE. These two variables are laid out exactly the same in VEE and in C. That's an important point. Also, it helps you get used to seeing how record vars are declared in C. You'll see a lot of that type of thing in the Windows header files.

The problem with this is that, as I've already mentioned, VEE Record type variables are not allowed to cross the CFI. So this knowledge doesn't seem to be of much importance. The reason it is important is because we now know that both of these record types are exactly the same in VEE and in C. Before we get into passing compound types across the CFI, take a moment to reflect back on the definition of an array: an array is a pointer to a chunk of memory whose contents are successive data entities, all of the same type. Now you can begin to see why this is so important. A single variable of type `DATARECORD` can't be passed across the CFI, *but a pointer to an array of them can*. And the array doesn't have to contain any more than one record either.

Now hold it! HOLD EVERYTHING. Before you go dashing off and create an array of Record in VEE and try to pass it over the CFI you should know that it won't work. VEE will flag the attempt as a type mismatch. Why? Because there is no way to tell VEE that it's an array of `DATARECORD` that you want to pass. You can't call it a `DATARECORD` array, so you call it an int array instead. Well when you try to pass an array of `DATARECORD` VEE knows you're trying to spoof the type and it says no way. We're still not all the way through the complete explanation.

## A Small Excursion Into Reality

Ok, so far all this has been theory. Rather boring theory at that. Why not show a practical example that's simply bursting with type spoofs of all kinds? We are going to write a small VEE program to convert a network computer name into an IP address. And we're

going to do it all in VEE, without writing anything at all in C or any ActiveX control or library. This operation is called a "DNS Lookup". And why not go the other way too? The program will also lookup a network computer name given an IP address.

The first thing we need to know about this is that there is a Windows Sockets function (taken directly from the original BSD Sockets spec) called "gethostbyname" that accomplishes looking up a host address given it's name. When you look up "gethostbyname" in the Windows SDK you find the following declaration:

```
struct hostent FAR * gethostbyname(const char FAR * name);
```

Now, I'm aware that many of you do not have access to a Windows SDK and would like to follow along. The thing is, see, you don't need an SDK because there's one on line and it's called the MSDN Library or Microsoft Developer Network Library. Just click over to the [MSDN Library](#) and use the "search for" box at the upper left of the page to search for gethostbyname. The MSDN Library is a really huge place and you're going to get a lot of results. We're interested in the Winsock call, so scroll down in the results list until you find [Winsock] at the end of the function name. [This is the page you should have found.](#) At any rate, there are a couple of things you need to notice about this. First, the function returns a pointer to a "hostent" structure. Second, under Return Values in the error table, you see that the return value WSANOTINITIALIZED means that you have not called WSAStartup. So make a mental note of that. We must first call WSAStartup. Third, there's the declaration of the "hostent" structure itself. If you click on the "[hostent](#)" link (either here or there), you see the following declaration:

```
typedef struct hostent {
    char FAR      *h_name;
    char FAR FAR  **h_aliases;
    short         h_addrtype;
    short         h_length;
    char FAR FAR  **h_addr_list;
} hostent;
```

I've cleaned this one up a bit so it's easier to read. The first thing you should know about this is that you can forget about "FAR". It *used* to mean "far pointer", which meant "32:32 segment address". But in a 32-bit world there are no 16:32 segments any more (except in a thunk) so "FAR" is meaningless. So much for that. Next you notice "char \*h\_name". If you look back at the [Type Table](#) in [C Programming 101](#), you'll see that type "char \*" basically means "Text" in VEE. Also, if you read below the Type Table you'll see that "char \*\*\*" means "Text array" in VEE. So that's what these types are: a string and string arrays (though this one is not quite a string array, it's convenient to think of it as such. It's actually a pointer to an array of in\_addr structures, but when we get to that I'll explain). We'll see how to deal with them as we build the VEE program that will accomplish this hearty feat. The other types in this structure are "short", which the Type Table tells us means "Int16" in VEE.

So much for the types contained in the "hostent" structure. Now, look up WSASStartup. Here we find this declaration:

```
int WSASStartup(WORD wVersionRequested, LPWSADATA
lpWSADATA);
```

Ok, so the Type Table tells us that a WORD is actually an Int16, but LPWSADATA isn't anywhere. This is a pointer to another structure. If you look up WSADATA, you'll find:

```
typedef struct WSADATA {
    WORD          wVersion;
    WORD          wHighVersion;
    char          szDescription[WSADESCRIPTION_LEN + 1];
    char          szSystemStatus[WSASYS_STATUS_LEN + 1];
    unsigned short iMaxSockets;
    unsigned short iMaxUpdDg;
    char FAR      *lpVendorInfo;
} WSADATA, FAR *LPWSADATA;
```

Here's a slightly different twist on a string. The members "szDescription" and "szSystemStatus" are character arrays that are embedded in the structure. Further down in the declaration we see that "lpVendorInfo" is what we're use to seeing as a string. All three of these members are strings, but there are two different techniques for handling them (as we'll see later). And then there's "unsigned short". Well, VEE doesn't do unsigned so we're just going to ignore that.

Another little detail you might notice is that the documentation for WSASStartup says that a matching call to WSACleanup should be made. Fortunately that is a straightforward call:

```
int WSACleanup(void);
```

We won't need any special tricks to call this function. Astute readers also might have noticed that the WSADATA structure documentation says that much of this information should be ignored anyway if we're using the Windows Sockets version 2 or higher specification. Since we will be doing so, just be aware that the information returned in these members is not even supposed to be real — we'll simply be handling it to demonstrate how to do so.

## Taking Stock Of All This Stuff

That's an awful lot to go through, so let's just sit back a minute and think. There's this WSADATA structure that contains embedded strings as well as a pointer to a string. Then there's this "hostent" structure that contains not only a string pointer but two string pointer pointers or string arrays! Combine this with the fact that you can't pass a structure over the CFI and it looks like this experiment is doomed to fail. Apparently there's just no way to handle all this data. The fact is that all of this *can* be done using nothing but VEE

and some clever type spoofing. I chose this example *because* it looks totally impossible. It's possible because as long as we know the members of these structures and how large they are, we don't need to tell VEE anything at all about them! **Remember that: as long as you know what you're doing, VEE doesn't have to.**

## Planning The Example Program

The example will first call WSASStartup requesting Windows Sockets version 2.0 support. We'll have to check the return value from this call to make sure that WS 2.0 is supported on the system. As a side note, I should say that WS 2.0 is not required for this simple example. WS 2.0 is considered the minimum standard supported on all Windows platforms except Windows 95 however, and if you're running Windows 95 I strongly suggest you upgrade to at least Windows 98, if not Windows 2000 or XP. You can add WS 2.0 support to Windows 95 but you're much better off just replacing Windows.

At any rate, provided the call to WSASStartup works the next thing we'll do is decode and display the information returned in the WSADATA structure. This will require some tricky programming! Next, we'll enter an infinite loop that pops up a dialog asking for the name of the computer to look up an IP for, or the IP of the computer to look up a name for. Naturally it's easiest to use the local machine, but that's only a little fun. You can look up the IP or name of any computer on your network, or any network your computer is connected to (i.e., the internet). And privileges don't really come into play here. Looking up an IP or a name happens long, long before any given computer is contacted for any kind of access. Any machine no matter what its privilege level can always look up IPs. This operation is one of the hallmarks of many different types of networks, not just TCP/IP — if it's disallowed then your network becomes non-functional.

But getting back to this dialog box, it will have three buttons: Lookup IP, Lookup Name and Cancel. If the Cancel button is clicked then the loop will exit, cleanup code will be run and the program will exit. If Lookup IP is clicked, we'll call "gethostbyname" with the name entered and decode and display the results of the call located in the returned "hostent" structure. This also is going to take some tricky programming. Trust me: this example has a little bit of everything in it. You won't often find a task that needs this many spoofing techniques. If the Lookup Name button is clicked we'll call "gethostbyaddress" with the IP entered and decode and display the results of the call located in the returned "hostent" structure.

As for why do any of this stuff at all, well that's harder to explain. There are much, much easier ways to look up a computer's IP or name on a network. The trouble is that all of those ways require something else. What we're doing here, and what we're going to be doing throughout this entire series and in most of the examples, is using what's available on *every single Windows computer in the world*. There are plenty of ActiveX controls and libraries that will do all this for you, but unless you have one of those controls or libraries you won't have a clue how to Git 'R Done.

Using any of the various Windows APIs at their basic level always implies a lot of work. But it also implies that the techniques you use are absolutely universal, and that means a

lot when you're trying to sell self-contained software and don't want to give up a portion of your profits to somebody else. Remember that a great many libraries and controls generate royalties for their authors. If you want to use them, you have to pay. And very often you have to pay *per use*. If you sell 10 copies of your program that uses library X, then you pay X's author 10 times.

Actually, I like to look at it the other way around. Or sideways, as it were. Yes, there is a widely available ActiveX library you can use to get the IP of any computer on a network. It isn't a royalty generating library, and you don't have to pay anybody to get it either. But you won't know anything about *how* it does what it does — this is how it does it. And this isn't even the lowest level we can go to. There are lower-level ways to get network names and addresses, but that's for another article.

## The Black Cat's VEE App Framework For Windows

Years ago when [Samantha](#) was learning to work with VEE, she found herself always importing from one Windows dll or another and defining a lot of constants to work with Windows. She had to do spoofed versions of memory and string copy functions and all sorts of things. One day she got totally fed up with duplicating all this work for several VEE applications. She raked the mouse, threw it against the wall and faced me with flaring nostrils and a deep growl in her throat. "Make me a standard library for working with Windows" she said. With that, she jumped down from the computer desk and went out in the back yard. I didn't see her again for a few days, but I *did* find the remains of several species of small furry animals gathered together in a cave down by the creek. The basic point of the story is that if a pissed off Black Cat tells you to do something you really should just shut up and do it. I did, and she never uttered another word about for the rest of her life.

The entire framework has never been publicly published but bits of it have been seen on the VRF at times. For importing libraries from various Windows dlls there's LoadLibs, and it depends on LoadLib. For common type spoofing tasks there's LoadMemLibs. For defining constants there's LoadConstants. In fact, LoadConstants isn't mine. It's adapted from an example provided by Georg Neid of GN Software. I picked this up from the VRF. See, that's one of the things that the VRF is all about: sharing experience. Basically I'm going to skip a lot of explanations I should and concentrate on the spoofing aspects of what's going on with the "hostent" structure. One further point: Whenever you do anything, always use the earliest revision of whatever you're using that's necessary to get the job done. This example needs to use an array of Int16, so we're going to have to use VEE 6. The reason you use the earliest version possible is because you make your software accessible to a wider audience. Sure, it's nice to use the latest and greatest, but if you don't need to it's a bad move. Most of the examples that will eventually be published here are VEE 5 compatible. Most of the Windows functions we'll be using are Win32 functions.

## Calling WSASStartup

The first thing we have to do is initialize the Windows Sockets library by calling WSASStartup. BTW, the WSA stands for "Windows Sockets Architecture". This function takes an Int16 that defines the requested Windows Sockets version and a pointer to a WSADATA structure. The version request is formatted with the high order byte specifying the minor version number and the low order byte specifying the major version number. Since we're asking for Windows Sockets 2.0 functionality, this is fortunately not a big deal. The number "2" works fine. I'm sure some of you who are familiar with byte ordering are scratching your heads now and saying, "but wait! Isn't that stored in memory as 0x0200?" Yes it is. But remember: VEE is storing an Int16 and Windows is reading an unsigned short and besides the fact that one is unsigned and the other isn't, those two types are exactly the same. When Windows gets the passed variable, it's still 0x0002 and everything works out.

This WSADATA structure is another thing entirely. Provided the function works, It is filled out by Windows and returned to the caller to give you information on what's up with sockets for the calling thread. The first two members are WORDs (Int16 values) that specify the version of the Windows Sockets specification that the dll expects the caller to use and the highest version the dll can support, respectively. Next is an embedded zero-terminated character string of length WSADESCRIPTION\_LEN + 1. Well we need to know how large this is. This is where we come to it: you have to have the Windows header files to look this up. As far as I know it's not documented on the MSDN Library web site. You can download the Windows SDK at the [Windows SDK Update Site](#), but it's really huge. Such is the price of cheap computing. You always pay the price somehow.

At any rate, the value we're looking for is 257. And the other string, WSASYS\_STATUS\_LEN + 1, is 129. Then we have two Int16 and a pointer to a string — 32 bits. Now, if we add all of these data sizes up we find that the WSADATA structure is: 2 + 2 + 257 + 129 + 2 + 2 + 4 bytes long, or 398 bytes long. If we divide that by two (because one Int16 is 2 bytes) we find that it's exactly equivalent to 199 Int16s. Well, a pointer to a WSADATA structure is exactly equivalent to an array of 199 Int16s, and it just so happens that's what we're going to use instead of a WSADATA structure.

You heard right: **an array can successfully substitute for a pointer to a C compound data structure and that is the secret to passing almost anything at all over the CFI.** *The type of the data entity doesn't matter at all. It's the type of the members that we have to keep in mind.* All we need is a pointer to a chunk of memory that's large enough to contain the target type, and in this case an array of Int16 of size 199 will do it. Magic, huh? Of course accessing the data returned in the WSADATA structure is going to be a hassle but hey! At least it will work. An Int32 array will work as well, but it's not as suitable as an Int16 array. First of all, some of the members themselves are Int16. Second of all, we are closer to being able to divide the embedded string members by 2 rather than 4. And we'll have to deal with that little point too.

## Defining Member Offsets

There is one thing that *must* be done first of all. Before we go calling any of these functions, we first have to define offsets to the members of the structures. These are labels that equal the value of the array index where the members start. You can try to remember what array index these members start at, but if you do try you won't succeed (well, ok. I bet some of you could. Not me though!). You're far better off *never* using magic numbers. Where ever a number must appear, define a constant and name the constant something meaningful. This isn't just a good idea, it's the way almost all successful commercial programming is done. The number 10 doesn't mean much, but the label LF means Line Feed. See?

So lets take these structures in the order that they are going to be used. First is WSADATA:

| Member         | Type      | Label               | Byte Offset | Array Index |
|----------------|-----------|---------------------|-------------|-------------|
| wVersion       | WORD      | m_wd_wVersion       | 0           | 0           |
| wHighVersion   | WORD      | m_wd_wHighVersion   | 2           | 1           |
| szDescription  | char[257] | m_wd_szDescription  | 4           | 2           |
| szSystemStatus | char[129] | m_wd_szSystemStatus | 261         | 130.5       |
| iMaxSockets    | short     | m_wd_iMaxSockets    | 390         | 195         |
| iMaxUdpDg      | short     | m_wd_iMaxUdpDg      | 392         | 196         |
| lpVendorInfo   | char*     | m_wd_lpVendorInfo   | 394         | 197         |

Notice we have a problem with szSystemStatus. The offset of the beginning of the member isn't evenly divisible by two so one of the Int16 array values is split! Half of it belongs to one member and the other half belongs to another member. There's a simple way around this problem that we'll see when we get to it. In the program, we'll simply let m\_wd\_szSystemStatus equal 130. Oh! What's with the m\_wd\_? Well, these are members so that's where the "m\_" comes from, and they're members of the WsaData structure, so that's where the "wd\_" comes from. For such a small program it really doesn't matter, but when you get a whole lot of structures with member offsets that have to be defined, it helps to give them some kind of "pseudo name-space" name. Now lets do the hostent structure:

| Member     | Type   | Label           | Byte Offset | Array Index |
|------------|--------|-----------------|-------------|-------------|
| h_name     | char*  | m_he_h_name     | 0           | 0           |
| h_aliases  | char** | m_he_h_aliases  | 4           | 2           |
| h_addrtype | short  | m_he_h_addrtype | 8           | 4           |



|             |        |                  |    |   |
|-------------|--------|------------------|----|---|
| h_length    | short  | m_he_h_length    | 10 | 5 |
| h_addr_list | char** | m_he_h_addr_list | 12 | 6 |

If you add these up, you'll see that this structure is  $4 + 4 + 2 + 2 + 4 = 16$  bytes long. Remember, a pointer is always four bytes. We'll use an Int16 array of size 8 to handle this structure. But wait! Isn't that wrong? I mean, there are *pointers*! And we'll only be getting 16 bits per array index. Yes, that's right. Here's something else we have to be aware of and provide for. And in this particular case we'll see how the fact that VEE doesn't do unsigned types is going to be a real pain. See? I told you this example had a little of everything!

## Dealing With String Pointers

Ok. We're finally getting some stuff done. Before we start writing the program though, we have to think about how to handle a few of these things. First of all there's char\* (text), then there's char\*\* (text array), and finally there's char[] (embedded string). How are we going to deal with these different string declarations? Let's take char\* first because that's the easiest. The value that we're going to pick out of the hostent array is going to be a pointer to a string. We want to copy it to a VEE Text variable. It just so happens that there's a function that's specifically made for doing this, and it's called "lstrcpy". It's declaration looks like this:

```
LPTSTR lstrcpy(LPTSTR lpString1, LPCTSTR lpString2);
```

Substituting familiar types for Windows self-documenting types, and renaming things to make it a little clearer, we get:

```
char* lstrcpy(char *pDst, char *pSrc);
```

Here, I have differentiated the source and destination strings. Well, for one I don't like the fact that this function returns a string pointer so we're going to take care of that when we write the import definition for VEE. Also, you may have noticed that from VEE's point of view the second argument, the source string to copy, is a Text var, but we have a number in an array! The fact that *it is* a string pointer is lost on VEE, because we used an Int16 array to get it. We're just going to have to spoof VEE into passing the number. *We already know it's a pointer, and that's why it works. As long as you know what you're doing, VEE doesn't have to.* Think of it as a little white lie. Marketing wants to know when the program will be done. Development says "soon". Well, every programmer in existence knows that "soon" means "sometime between now and when I die." Just like when will the [archive search page](#) be functional or when will this article be finished [hmmm... apparently both have come to pass. *Astounding*]? Well, the answer is soon. That is, some time between now and when I die.

Here we get to see our first spoofed function declaration:

```
int lstrcpy(char *pDst, int pSrc);
```

I know it looks completely insane, but trust me: it works! In fact, this is a common spoof and it's included in Sam's library. Now, on to `char[]`. This one is easy. Something may be itching at the back of your head... No? Back in C Programming 101 I said that an array is a chunk of memory filled with data entities all of the same type. I also said that what VEE passes when it passes an array is a pointer. If you think about it, that means that as far as the CFI is concerned an array is a pointer. I keep saying this, and this is one of the reasons why. As far as C is concerned an array is a pointer also. In fact, if you take C classes at some point you learn about *array / pointer duality*. Arrays are a compiler construct. They are bogus. They are false, tricky. They are implemented as a pointer. The compiler simply arranges it so that the pointer points to a chunk of memory (all together now) filled with data entities all of the same type.

So how does that help with copying a string that's been embedded in our array? Simple. Remember that VEE kicks royal butt when it comes to arrays. We will simply pass a subarray to the `lstrcpy` function and VEE takes care of the rest. I'm not kidding. It's that simple. VEE builds a new array containing just the characters we're after (because it's a `char` array, and that means string) and passes a pointer to that array to the `lstrcpy` function. Unfortunately however, we're going to need a different declaration of `lstrcpy` to do it. This version needs to look like this:

```
int lstrcpy(char *pDst, short *pSrc);
```

Because as far as VEE is concerned the source is an `Int16` array. This is also a pretty common spoof and it's included in Sam's library. Confused yet? Just wait. There's plenty more. There's this `char**` for instance. What do we do about that? Well, what this notation means is "pointer to pointer to string". It *means* string array. The data entities in this array are string pointers. *Those* are the buggers we need to get at so we can call `lstrcpy`. What we're going to do with this one is basically implement the pointer array ourselves and get the individual pointers with a very handy function known as `RtlMoveMemory`. I'm afraid I have some bad news. Those of you who downloaded SDKs so that you could look all this up are not going to find `RtlMoveMemory` in it. Now you need to download the Driver Development Kit, or DDK. This function is in `kernel32.dll` for use by user code, but it's also in `ntdll.dll`. In fact, I wouldn't be surprised if it's all over the place. The "Rtl" indicates that this function is actually implemented in the C Run-Time Library. It's really nothing more than a renamed "movemem". If you have any C compiler documentation, you can look that up and you'll find something very similar to the declaration of `RtlMoveMemory`: You can also look it up in the SDK documentation as `MemMove`. `MemMove` is a *C macro* that resolves to `RtlMoveMemory`.

```
VOID RtlMoveMemory(IN VOID UNALIGNED *Destination, IN CONST  
VOID UNALIGNED *Source, IN SIZE_T Length);
```

Whoa! You really don't need to pay attention to any of those silly words. Driver writers are in the habit of prefixing parameters with the words `IN` and `OUT` to indicate whether

they are input or output parameters. Unaligned simply means that the source and destination blocks of memory do not need to be aligned in any special way, as is sometimes a concern. The CONST is simply there to tell the compiler that the source may not be modified. It helps when you forget that all these move and copy functions take a destination *first*, just as the assembly language mnemonics for the move and load instructions for x86 do. If you get the parameters turned around the C compiler will catch it. We don't have that luxury in VEE so you have to be careful when using move or copy functions. The spoofed declaration for this function is:

```
void RtlMoveMemory(long *pDst, int pSrc, long lLen);
```

This version of the declaration is made specifically for getting a 32-bit value into a VEE Int32 variable and lLen will *always* be 4. This is a very useful function. Member I said something about copy? There is an RtlCopyMemory function too. It actually runs faster so it would seem a better choice for this operation. In fact, it is. The reason I use Move instead of Copy is because Move will handle cases where the destination block of memory overlaps the source block and Copy does not. This doesn't happen often, but every once in a while it will so I always use Move to avoid the problem entirely. In most general purpose programming this won't be the case however, so you can go ahead and use Copy if you want to. There *is* one good reason for doing so: If you're copying a whole lot of memory in loops, then Copy beats Move every time. Just make sure that the destination will not overlap the source!

## Multiple Libraries, Similar Functions

There's just one little detail left to talk about: as you can see, we need to declare different parameters to the same functions — lstrcpy and RtlMoveMemory specifically. Not to mention the fact that this will be done quite a bit. So how's that going to work? Most languages have some concept of *namespace* and VEE is no exception. A namespace is a space wherein names are unique, and separate from other namespaces. So as long as the different declarations of the same function are kept in different namespaces everything is fine. The namespace of a VEE compiled function import is the library name, and that's what we'll use to keep these different declarations separate. I use two main conventions: when copying stuff from VEE to Windows, I use the namespace prefix V2W. When copying stuff from Windows to VEE, I use W2V. Other characters are added to indicate the type of the value being copied and the number of bytes in the type but you get the general idea. All this can be seen when the program is run. Samantha's VEE memory libraries create a whole slew of namespaces with all sorts of declarations for copying various data to and from Windows.

## Are We There Yet?

Now see? There's a reason for all this too. VEE lulls you into a false sense of security. It's *very* easy to throw something together and make it work quickly. Programming in general isn't like that. The more planning and forethought you put into any given project the better it's going to turn out to be. When you start whipping around huge structures and fake them with arrays and start passing pointers to API functions... well, you're not really

programming in VEE any more and you have to be careful. Mistakes will *always* be made. After 30 years of programming in all kinds of languages I can count on one hand the number of times that everything worked the first time I ran something. It's much more rare than once in a Blue Moon. You agro and astro enthusiasts know what that means, but for everybody else it means it almost never happens and when it does you should mark it on the calendar, write about it in your journal and celebrate with a hearty [Guinness Stout](#) (HINT: get the pint bottles. Avoid the stupid spritzer. It does nothing but ruin the beer). Steak & potatoes is optional but always a good choice too.

There is such a thing as planning a project to death and that's bad too. You don't want to worry about every little detail, but you should at least have a very good idea of what data structures you'll be using and how you're going to handle them. You might want to scribble down what BCS libraries will be used to copy structure members to and from Windows when those members can't be directly manipulated by VEE. Parameters going IN will use V2W functions if VEE can't directly write their values into the arrays used to spoof their type. Parameters coming OUT will use W2V. Parameters that are defined as IN / OUT (i.e. they both pass data to the function and receive output from the function) will use V2W before the call and W2V after the call. Gather all your constants together and look up their values. Get them all typed into the BC's Constants text array.

## **Can We *PLEASE* Just Do The Program Now?**

Yes. It is time. Figure 1 shows the constants I'm defining for this example. Oh, the program? You want the program you say? Well, ok. You can download it from [here](#). It's got a VEE 6.2 header but it only needs VEE 6.0 compatibility.

Now if you look up these errors, you'll find that I tinkered with the names. Normally one should never do this, but I don't like it when names are all run together with no separating cues, so I stuck a few underscores in here and there. The comments after the error labels are the text of the error message that applications *should* display when they encounter these errors. They're all easy to get from the system and there's no excuse for not displaying them. Except that one is lazy. We'll be lazy. If an error occurs, the example program will tell you the code but you'll have to return to the LoadConstants function to look it up.

The comments after the array index labels are the data type of the member. I do this because I'm extremely stupid and constantly forget what type I'm trying to deal with. That's part of the reason that self documentation is so important. Oh! The name! I forgot to say anything about the name. The name of this example is "DNSLookup" because the operation of looking up an IP address for a computer name is called a "DNS Lookup" (clever, huh?). That of looking up a computer name given it's

Figure 1: Constants for DNSLookup.

```

// WSAStartup errors.
WSA_SYS_NOT_READY      = 10091 //WSAStartup cannot
WSA_VER_NOT_SUPPORTED  = 10092 //The Windows Socket
WSA_E_IN_PROGRESS      = 10036 //A blocking operati
WSA_E_PROC_LIM         = 10067 //A Windows Sockets
WSA_E_FAULT            = 10014 //The system detected

// gethostbyname errors.
WSA_NOT_INITIALIZED    = 10093 //Either the applica
WSA_HOST_NOT_FOUND     = 11001 //No such host is kn
WSA_TRY_AGAIN          = 11002 //This is usually a
WSA_NO_RECOVERY        = 11003 //A non-recoverable
WSA_NO_DATA            = 11004 //The requested name
WSA_E_NET_DOWN         = 10050 //A socket operatio
WSA_E_INTR             = 10004 //A blocking operati

// Array indexes for WSADATA.
sizeof_WD              = 398 //Size of WSADATA in byt
m_wd_wVersion          = 0 //WORD
m_wd_wHighVersion      = 1 //WORD
m_wd_szDescription     = 2 //char[257]
m_wd_szSystemStatus    = 130 //char[129]
m_wd_iMaxSockets       = 195 //short
m_wd_iMaxUdpDg         = 196 //short
m_wd_lpVendorInfo      = 197 //char*

// Array indexes for hostent.
sizeof_HE              = 16 //Size of hostent in byt
m_he_h_name            = 0 //char*
m_he_h_aliases         = 2 //char**
m_he_h_addrtype        = 4 //short
m_he_h_length          = 5 //short
m_he_h_addr_list       = 6 //char**

// Address type constants.
AF_INET = 2 //Defined in winsock2.h.

```

IP is called a Reverse DNS Lookup. A DNS is a "Domain Name Server". Everything in IP land is based on an IP address... the dotted quartet of numbers between 0 and 255. There is no such thing as a computer called "oswegosw.com". It's just that the world's DNSs have been told that "oswegosw.com" means IP address <whatever it happens to be now>. It's not always that simple. Sometimes a site will have a web server on one computer, an ftp server on another computer and so on. In this case, the different prefixes (like www or ftp for instance) will lead to different IPs and so different computers. Sometimes the routing is done internally by protocol request.. There are a lot of different ways to do it. But here at Black Cat Software, all servers run on one computer and so anything at all that you type in your browser's address bar with "oswegosw.com" in it comes to this IP.

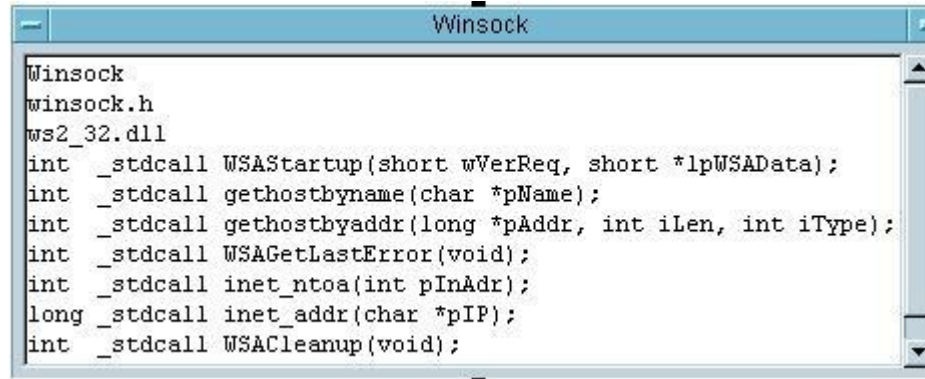
## **The Spoofed Functions**

Figure 2 shows how the main functions I'm using in this example have been spoofed.

Since we're  
using Int16  
arrays to  
substitute for the

*Figure 2: Spoofed Function Definitions.*

structures, anywhere a structure is mentioned in the prototypes they're replaced with short pointers. Note that the `gethostbyname` call itself is interesting in that it returns a pointer to a `hostent` structure. Well, VEE won't handle that so what we're going to do is copy the `hostent` structure to an `Int16` array when it returns. We'll get to just how to do that shortly. For the CFI definition, we're going to tell VEE this function returns an "int". And in order to copy this structure to an array, we need `RtlMoveMemory` in it's `W2VI2` form. This form is specifically used to copy individual `Int16` values and `Int16` arrays to VEE from Windows. Note also that

A screenshot of a Notepad window titled "Winsock". The window contains the following text:

```
Winsock
winsock.h
ws2_32.dll
int _stdcall WSAStartup(short wVerReq, short *lpWSAData);
int _stdcall gethostbyname(char *pName);
int _stdcall gethostbyaddr(long *pAddr, int iLen, int iType);
int _stdcall WSAGetLastError(void);
int _stdcall inet_ntoa(int pInAddr);
long _stdcall inet_addr(char *pIP);
int _stdcall WSACleanup(void);
```

gethostbyname  
is sort of special  
in that it doesn't  
have "A" and  
"W" versions.  
This is an  
ancient leftover  
from long, long  
ago that doesn't  
take wide  
characters into  
account.

## Wrapper Functions

The very next thing I always do is stick a formula in main that calls LoadConstants(True) and LoadLibs(LL\_LOAD) and run it. This will go through all the code and imports and make sure there are no mistakes. It also imports the libraries so that VEE knows about the functions I'm calling and I can create call objects from them. This is important because the next thing I'm going to do is build wrapper functions that handle the calls to the dlls.

Calling a dll is usually something of a big deal. There are sometimes return buffers to set up, input parameters to prep and all kinds of stuff. These details are subordinate to the main point of simply calling the function and getting some kind of useful return value, so what I usually do is put all the prep work along with the actual function call in a *wrapper function*. A wrapper function is a function that you call that hides all the tedious details of the prep work. It does the prep, calls the actual target function, then cleans up after the call (if necessary) and returns some useful information. Notice that in the case of the WSASStartup call, we're interested not only in the return value of the call, but what Windows filled out the WSADATA structure with. So I want the function to return not only the return value, but the return array as well.

Many API calls are like this. What I do is consider the two different pieces of information the function returns separately: there's a success / failure indicator and a piece of data. I'm actually more interested in the data, so I make that the primary return from the wrapper function. The success / failure indicator I send off to a global variable that I can test in any context. In general global variables are a bad idea, but this is a legitimate use so I'm not worried about it. And while I'm at it I should say something about GetLastError too.

## GetLastError?

Yeah. See, one of the more primitive error reporting facilities is SetLastError. Many (very many, like most) API functions will call SetLastError when they complete. If there was an error they set the last error to whatever the error was. This is great when you're executing compiled code because you get a general purpose success / failure value from the function, but it's usually a boolean value indicating whether or not the function



succeeded. You call GetLastError to find out what the error actually was. Now, when you call dll functions from VEE, there is a whole lot of stuff that goes on behind the scenes that you don't see. And when a dll function returns to VEE there are again a whole lot of things that go on that you don't see. So, by the time VEE is ready to execute the very next object or statement in a Formula there are probably hundreds (if not thousands) of things that have gone on in the mean time. Unfortunately this means that by the time you realize there has been an error the specific error information has probably been overwritten with other error information. So calling GetLastError usually doesn't get you the information you want. You want to know why your call failed, and GetLastError is reporting some other error that the VEE runtime has run into in the mean time.

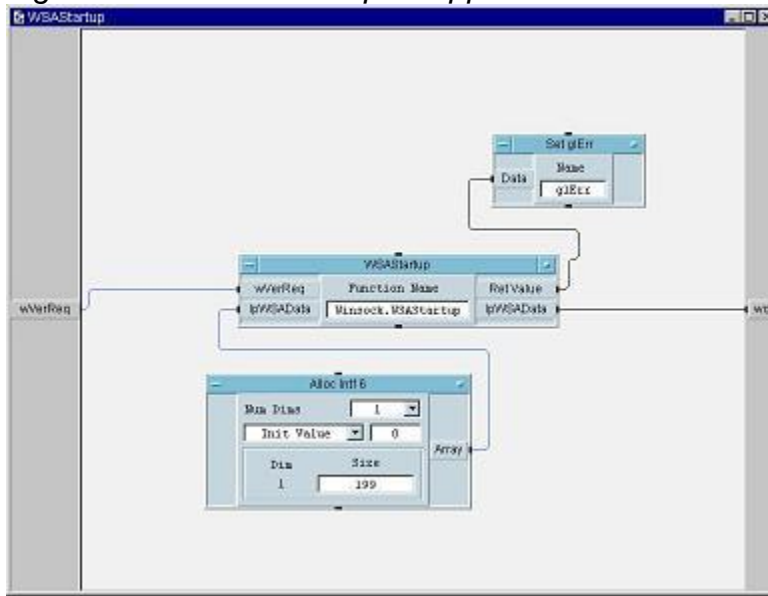
In order for GetLastError to work, VEE would have to call GetLastError *immediately* after it executes the call requested from your program and make that information available to your program somehow. This it does not do (at least it doesn't make that information available. I think. I don't know if it calls GetLastError immediately after executing a call from a user program). Sometimes GetLastError *will* work, but most of the time it won't. Whenever I call GetLastError I get information that a function was not found. That doesn't have anything to do with the error I committed. It has to do with the VEE runtime calling GetProcAddress for a function that wasn't found in the particular dll it was trying to find that function in. Realize that this is a normal and acceptable programming technique: retry on error. Sometimes it's the best way to get something done. In these cases, the error is expected behavior and is used to sequence a loop or some other control structure.

The best example I can think of at the moment is the implementation of what we think of as Virtual Memory. Here I'm talking about VM in the paging context. When memory is allocated for a process, it's allocated in pages. On Intel hardware Windows NT allocates memory in 8K pages. In order to save time when setting up the process, the memory requirement is simply noted - memory from the pool is not actually mapped at any specific address. The first time a process tries to physically touch that memory a page fault is generated. The *Virtual Memory Manager* (VMM) handles the exception and requests the MMU of the CPU to map memory at that address. This way, processes don't actually consume memory unless they need it. It's a much more efficient way of doling out memory.

## **Back To The Wrapper Functions**

Getting back to the wrapper functions, we need at least three of them. One for WSASStartup, one for gethostbyname and one for gethostbyaddress. I just usually name the wrapper function the same as the original call. The original calls are in a library (VEE namespace "Winsock" in this example) so there's no name conflict. I define the inputs and outputs that are convenient for VEE. Then I build the wrapper function around a VEE call object to the actual function in the dll. WSASStartup is shown in Figure 3, gethostbyname is shown in Figure 4 and gethostbyaddress is shown in Figure 6. Figure 5 is the wrapper function used to get an Int16 array. WSACleanup doesn't need a wrapper because it doesn't have any parameters and it's very simple to call in-line.

Figure 3: The WSAShutdown Wrapper Function



You know, take a look at Figure 3 (click the pic to see the full size function). It's very simple. All we do is allocate an Int16 array of size 199 and pass it to WSAShutdown. The version requested is a parameter to the wrapper function but it could have easily been supplied inside the wrapper. The input to the function by the way is not only named properly, but it's also constrained to take only an Int16 scalar value. When doing Windows Programming in VEE, it's important to constrain inputs to dll functions like this. This will save you endless hours of troubleshooting only to find that you've mistakenly passed an array to a function that was expecting a scalar. VEE itself will catch the error when you try to execute the call object, but remember that not all of your inputs will always be going directly to the call object.

There's a subtle problem here that could turn into something ugly at some point in time. Remember how I said I hated magic numbers? Well, there's a magic number, right there in the Dim Size 1 of the Allocate Array object. This *should* be taken from a label. See, if the size of the WSADATA structure ever changes then this won't work anymore. There's one specifically set up for it, and it's called `SIZEOF_WD`. Of course `SIZEOF_WD` is in bytes, and we want to calculate the size of the array in elements, so it's `SIZEOF_WD / sizeof(short)`, or `sizeof(Int16)`, which happens to be 2 bytes. The trouble is you can't put a formula in the Dim Size. You have to add a control input. But then that requires another object to calculate the size, and then you have to constrain the sequence input of the Allocate Array object to wait until after the formula executes... it gets very messy. Fortunately there's another way to do it. You can use a formula to calculate the size of the array and allocate the array all at once. It's `"asInt16(ramp(SIZEOF_WD / 2, 0, 0));"`. But

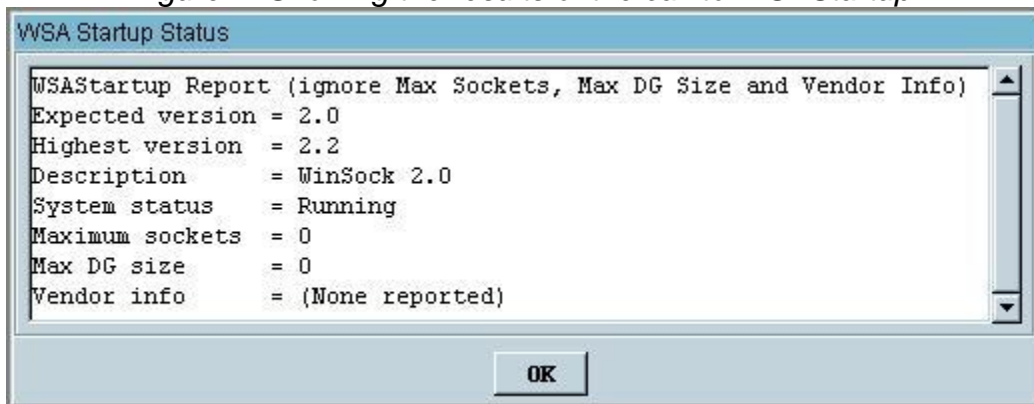
isn't 2 a magic number also? No. The size of an Int16 will never ever change. The size of the WSADATA structure will probably never change either, but never count on that. Several structures that are sometimes (improperly) considered immutable have indeed changed. BITMAP is one of them. It has changed several times.

But anyway what I wanted to say was look at the size of it! Yes, these three objects can be closed and they take up much less room, but it's still got wires running around and three interconnected objects. This is another purpose of Wrapper Functions. They not only hide unnecessary details that get in the way of your main thought process, but they compact all that stuff into one neat little package. There are no interconnected objects. Just one, nice neat line of code that executes all of this stuff. One of the goals of any kind of programming is clarity of purpose. When you look at any one function, it's nature and purpose should be immediately obvious. Or at least after only a few moments study. If it's not obvious then it's not done well. VEE excels at self-documentation — to a point. Once you get a ton of objects on the screen and wires running all over the place, it totally breaks down into a raw mess. That is to be avoided at all costs. I have a lot more to say about Wrapper Functions but that's another article.

## Showing WSASStartup Results

This is where things start to get interesting. The task now is to show the results of the WSASStartup call, provided it worked. The results of the call are shown in Figure 4. Probably most of you will get exactly the same result. The first line of the report states that we should ignore Max Sockets, Max DG Size and Vendor Info. This is because we are using the version of Windows Sockets that is capable of multi-vendor service providers ( $\geq 2.0$ ). The Expected version is 2.0, meaning that any calls made to any ws2\_32.dll function from this thread must conform to Windows Sockets 2.0 specifications. Next, just as a point of information, it says that the highest version of the Windows Sockets specification that this version of ws2\_32.dll can support is 2.2. This may well vary on y'all's systems because I ran this on a Windows 2000 system and it's getting old and tired now. The description string shows up as WinSock 2.0. The status string shows up as Running. The interesting thing is how those strings got there.

*Figure 4: Showing the results of the call to WSASStartup.*



## REWRITE FROM HERE

The code is in one of my famous gigantic MoFOs in the user function ReportStartup. You can almost see all the code in the screen shot (Figure 5), but we'll take it a section at a time so you can see what's going on. First, we get the version numbers returned from the call and pass them to a function called MkVerStr. This function is shown in Figure 6. It breaks the passed WORD into hi and lo bytes, then assembles the numeric representation of the value of the bytes with a period between them. The function itself is quite bizarre actually. Theoretically it shouldn't be this big a deal except that the value 0xff00 (the hi-byte mask) "cannot be converted to Int16" so the original WORD has to be converted to Int32 so that the bitAnd function will operate on Int32 values instead of Int16. Believe it or not this is not all that uncommon. No matter what the language, sometimes the *operand promotion rules* do not allow you to do what seems obvious, so you wind up explicitly converting types back and forth.

Now, in case you're wondering why this seems to be an "illegal value", consider that the constant 0xff00, when typed into a formula, is automatically promoted to Int32. So it's actually 0x0000ff00. This is a positive number. OTOH, 0xff00 as Int16 is negative. VEE realizes it cannot represent 0x0000ff00 as Int16 and says so. Outside the wrapper functions, the code is very neat and clean and so obvious that you can't help but understand exactly what it does immediately:

```
1111 1111 0000 0000
```

```
sExpVer = MkVerStr(wd[m_wd_wVersion]);  
sMaxVer = MkVerStr(wd[m_wd_wHighVersion]);
```

Next we have to get the description. What we have here is a sequence of 257 characters beginning at wd[m\_wd\_szDescription] and ending in the low order byte of wd[m\_wd\_szSystemStatus]. Since these members names begin with sz, we are guaranteed that they will be zero terminated so we can safely use the C Run Time string functions to manipulate them. The Black Cat's standard VEE library has several functions explicitly for dealing with strings stored in numerical arrays. The one we need is pi22Str. This reads as "convert pointer to two byte int to string". The "pointer" part means that the input is an array. The "two byte int" part means that the input array is an Int16 array, and the output will be a VEE Text string. A wrapper function called GetAryI2Str contains all the allocation prep work, and it's shown in Figure 6. First, a buffer is allocated with "sDst = "\*" \* pi22Str.lstrlenA(pi2)". This will make a string of asterisks the same length as the string stored in the array. Then all we have to do is call lstrcpyA with the source as the array and the destination as the buffer and it's done.

```
sDesc = GetAryI2Str(wd[m_wd_szDescription :  
m_wd_szSystemStatus]);
```

Now the fun part. Remember how the last byte of the description is actually stored at the lsb of the system status? Well, that last byte is going to be a zero character and if we try

to pass the status string directly to GetAryI2Str it will return a zero-length string. So, first we have to copy the array members that contain the status and then set the lsb to something other than 0. I chose the space character, because then I can run the output of GetAryI2Str directly through strTrim and the space will go away. Oh, why is it the first character and not the second? Well, these are Intel machines you see. They store the lsb of a multibyte value first, and the msb second. We're dealing with two different types here, one with byte ordering and one without (string). So when VEE operates on the string it's operating on it with every pair of characters reversed. Confusing, huh?

```
pi2SysStat      = wd[m_wd_szSystemStatus : m_wd_iMaxSockets -  
1];  
pi2sysStat[0] = bitOr(pi2SysStat[0], 0x0020);  
sSysStat       = strTrim(GetAryI2Str(pi2SysStat));
```

This sequence isn't so obvious. And actually I broke one of my own rules about not using magic numbers. 0x0020 should be called ASCII\_SPACE or something. I didn't notice it until just now and I'm not going to go back and fix it.

Then there's more of really easy stuff:

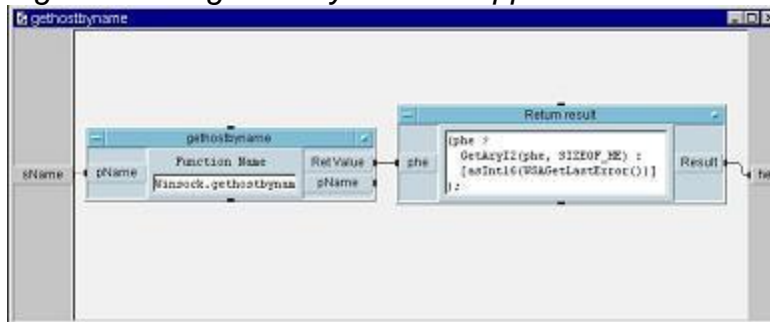
```
iMaxSockets = wd[m_wd_iMaxSockets];  
iMaxUdpDg   = wd[m_wd_iMaxUdpDg];
```

And finally some not so easy stuff. For vendor info, we have to deal with a string pointer. As you might expect, the Black Cat has a library function for that called GetAryI2Ptr. It's job is to get a 32-bit pointer from two successive Int16 array elements. This wouldn't be such a big deal except that this time we can not ignore the fact that VEE does not do unsigned values. If the most significant bit of the least significant word of the pointer is 1 (and it very well may be), then it becomes impossible to convert it to a UInt16 and combine with the msw to form the complete 32-bit pointer. The library has a function called WORD that takes care of converting signed 16 bit numbers to their unsigned 32-bit equivalents. The WORD function itself is very simple. If the input is negative it adds 32768 to the value to make it positive, then explicitly converts the result to a long (Int32). The GetAryI2Ptr function uses WORD to get the least significant word and most significant word of the pointer, then multiplies the msw by 65536 and adds it to the lsw. And that's that. The result is the original pointer.

That's not all though. If the pointer is zero, as it will be for sockets 2.0 & higher specification, we do not dare try to dereference that pointer. Doing so will cause an access violation. A simple test for the value 0 selects the action to take. Either dereference the pointer with GetStr or return a harmless string such as (None reported):

```
pVendInfo = GetAryI2Ptr(wd, m_wd_lpVendorInfo);  
sVendInfo = (pVendInfo ? GetStr(pVendInfo) : "(None  
reported)");
```

Figure 4: The gethostbyname Wrapper Function

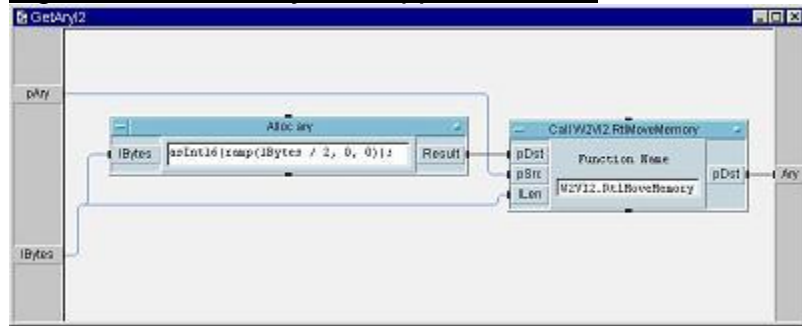


Then there's Figure 4. The gethostbyname wrapper. Calling the winsock function is only the first thing to do here. Provided the call worked, we have to copy the result to an Int16 array. If the call didn't work, we call WSAGetLastError and return the error as element 0 of the array. If you take a look at all the errors that can occur, you see that none are over 12,000. That number will fit nicely in an Int16 (which ranges from -32,768 to 32,767) so we're ok with doing this. Wait? Didn't I just say that GetLastError was useless? Yes. But this isn't GetLastError, it's WSAGetLastError. They're two different functions and the VEE runtime won't be overwriting this error value, so it makes sense to go ahead and have the wrapper function take care of getting it for us.

The return value from this call, which hopefully is a pointer to a hostent struct, is fed to the input of a formula with one purpose: if the call worked, get the resulting array. If it didn't work, get the error and return it as a single element array. We could use an If/Then/Else object to test the value and then wire over to separate objects that do the post processing, but that involves more objects and that means less speed. We're not concerned about speed here, but why waste time if you don't have to? This single formula tests the value of phe. If it's anything but 0, then we call another wrapper function, GetAryI2, with the address of the returned hostent struct and the number of bytes of data to copy. Naturally, since we want to do this so that nothing will ever get out of whack, we use the sizeof\_HE label. So if the size of the hostent struct ever changes all we have to do is change it in LoadConstants and everything still works.

If the function did fail, it returns 0 and in that case, we call WSAGetLastError, convert it's return to an Int16, and return it in a single element array. The GetAryI2 function (Figure 5) allocates an Int16 array large enough to contain the number of bytes being copied (using asInt16 with ramp) and calls RtlMoveMemory to copy the "array" to the VEE variable that we just generated.. Remember, the "phe" input to the function will actually be a pointer to a hostent struct. The RtlMoveMemory spoof takes an Int16 pointer (array) and an int that points to the source (which the gethostbyname function got for us). So far we've been spoofing the 'nads off VEE and it hasn't complained one bit. And everything has actually worked too. Really! We'll see it after some more very impressive spoofing.

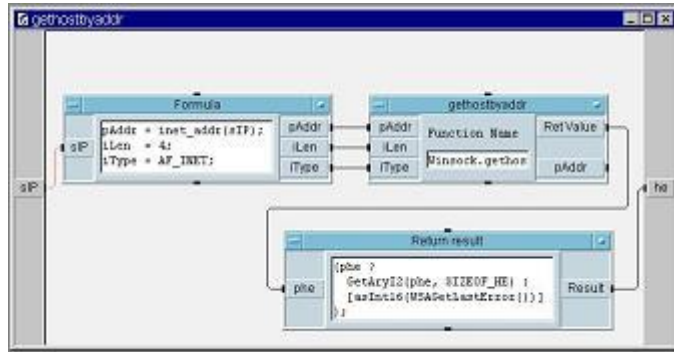
*Figure 5: The GetAryI2 Wrapper Function*



The gethostbyaddr user function (Figure 6) is quite different looking. There's some prep work to do, in addition to all the same post-processing as gethostbyname. The main thing is that the string

*Figure 6: The gethostbyaddr Wrapper Function.*

containing the dotted internet address has to be converted to an `in_addr` struct. The `inet_addr` function accomplishes this, and returns a pointer to the converted structure. Then we tell `gethostbyaddr` that the address is 4 bytes long and that it's an `AF_INET` type address and make the call. The post processing is the same as for `gethostbyname`.

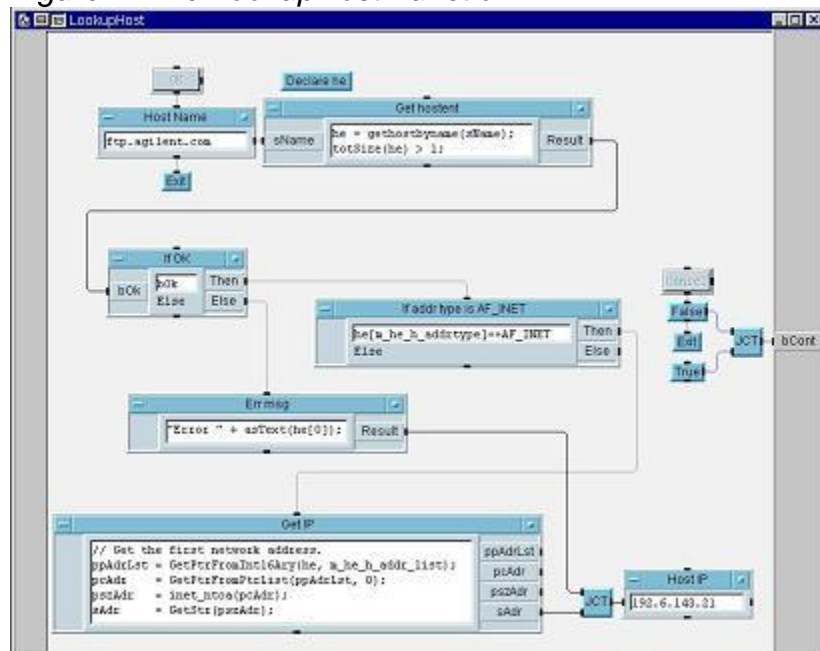


## Interpreting The Returned hostent Structure.

Ok, here it is. This, finally, is the main point of this example.

The LookupHost User Function (Figure 7) waits for input in the "Host Name" text box. When OK is clicked, the name goes to the "Get hostent" Formula. The first thing we do is attempt to get the hostent structure corresponding to the name. If that operation works, then the returned array will have more than one value in it so we check to make sure this is so. Meanwhile, a function local variable, `he`, has been set to the returned value (which we hope is a pointer to a hostent structure). The test is done with `totSize` testing the size of the returned array, and the return value of the formula is True if we

Figure 7: The LookupHost Function.





got a valid hostent struct. If not, the return value of the formula is False.

If the returned value is valid, next we check is to make sure that the address type (in member `h_addrtype`) returned is `AF_INET`. The value of this constant is 2. We're only interested in getting IP addresses. If we get an identifier for a different network then we don't really want to know about it. And you know what? This version of the function doesn't say anything about that! That's not real nice, so why don't you go ahead and modify it so that if we get a different *Address Family* it says so. One thing about the structure of the If/Then/Else objects that are used to check these conditions is that you might be tempted to combine both checks in one If/Then/Else, but you can't do that. First, we have to know that the `h_addrtype` member exists before we try to access it. If the returned hostent is *not* valid, then we report the error number in the "Host IP" display. I used a Text Constant here because it's entirely possible that you may want to paste the returned value into your browser just to confirm that all this works. Using an Alphanumeric Display precludes copying anything out of it, so I opted for a Text Constant with a default value control input instead.

Provided we *do* have a valid hostent structure, next we want to get the IP address of the host. This is the first entry in the `h_addr_list` member. We really should check all addresses returned, but we're not going to. You can figure out how to do so if you want to. It's a worthwhile exercise. There is often major confusion about the `h_addr_list` member though and we have to talk about that.

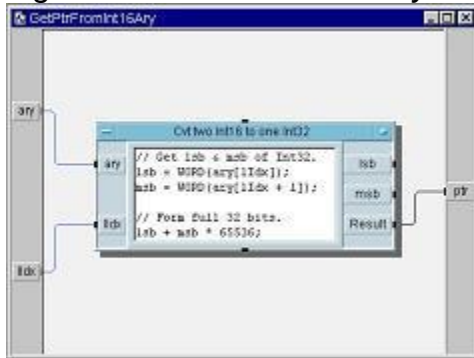
The type of `h_addr_list` is `char**`. That usually means string array but not *quite* this time. What we have here is a pointer to a list of pointers that point to a block of bytes, and each block is `h_length` bytes long. We are going to assume that this length will always be 4 bytes, and for a while yet it will be. This is called being IPv4 compatible. The new IP address specification, IPv6, is different. We're using library routines that can only deal with IPv4, so that's what we're going to use. Each of the four bytes pointed to is one of the numbers of the IP address. What we have to do is get each one of those bytes, convert it to a decimal string and stick periods between them. Fortunately there's a function that does this and it's called `inet_ntoa`. This function takes a pointer to an "in\_addr" structure and returns a pointer to the converted string IP address. Well, I didn't say anything about `in_addr` until now because an `in_addr` structure is simply four consecutive bytes, each one denoting one quarter of an IPv4 IP address. And gee, that pointer is in the list pointed to by `h_addr_list`. Slick, huh?

So all we have to do is get the first pointer pointed to by `h_addr_list`, pass it to `inet_ntoa` and then get the string that the returned pointer points to and we're all set! As you might imagine, there are intermediate steps so it's not really that straightforward.

## Dealing With A Pointer To A Pointer List

The big challenge here is dereferencing all these pointers correctly. The member, `h_addr_list`, is a pointer to a list of pointers, each 4 bytes long. Our first problem is to get the pointer (a 4 byte quantity) from the `Int16` array that is substituting for the hostent structure.

Figure 7: *GetPtrFromInt16Ary*



There's a User Function called `GetPtrFromInt16Ary` (Figure 7) that does this. As parameters, it takes the array and the index of the member that you want to get. Since this all runs on Intel machines, the pointer will be stored in LSB MSB order, so basically we get the number at the index of the pointer member and add it to the number at the index of the pointer member + 1 multiplied by 65536 (or  $2^{16}$ ). The result is the address of the first of the pointers to IP addresses (which are each a block of 4 bytes).

Figure 8: *WORD*.

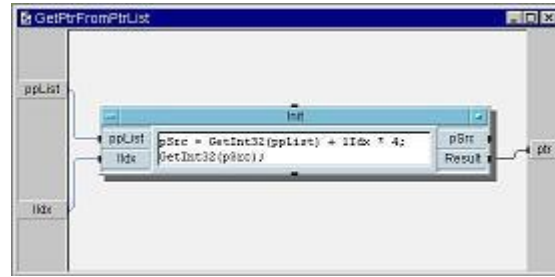


Wait though. This is one of those situations where you can't ignore the fact that VEE doesn't do unsigned arithmetic. It's entirely possible the the most significant bit of the least significant word of any of these addresses is set, so we *must* convert at least this number to an unsigned `Int32` before we add it to anything. This is handled by the User Function `WORD` (Figure 8). It's name is somewhat deceptive. If the input number is negative, it adds 65536 to make it positive. If the input is not negative, no adjustment is necessary. The last thing it does is explicitly convert the 16-bit value to a 32-bit value. This is solely to provide for consistent output. If the number is negative when it comes in, then it will be an `Int32` when it leaves. If it's not negative however, the result would be an `Int16` unless explicitly converted to `Int32`.

So we've called `GetPtrFromInt16Ary` and we now have a pointer that points to a list of pointers. It seems logical that since this pointer we have points to a list of pointers,

Figure 9: *GetPtrFromPtrList*.

we should construct a User Function that is capable of returning any one of the pointers in the list. This is the job of the User Function `GetPtrFromPtrList` (Figure 9). As input parameters, it takes the pointer to the pointer list and the number of the pointer that you want to get. 0 is the first, 1 is the second and so on. This is a hard ride, so hang on. First we get the address of the list by calling `GetInt32` with the pointer to the pointer list. The return from `GetInt32` is a pointer to the first pointer in the pointer list. Next, we add to that to four multiplied by the desired pointer offset (because each pointer is 4 bytes) to get the address of that pointer (a.k.a. the pointer to the pointer we ultimately want). Now all we have to do is call `GetInt32` again with that address and the result is the address of the four bytes that describe the network IPv4 IP address. Wow. Read that as many times as necessary. Double dereferencing is hard to explain but once you see how it works in memory it comes naturally and you almost never think about it.



At the end of all these machinations we have the desired address of the four bytes we're looking for. The reason this all looks so goobered up is because all this came from a very long time ago when Stupid C Tricks were highly valued. Well, I guess I really shouldn't say that. It's all very difficult to explain, but this actually is an excellent way to do this. If this were being written now-a-days, `h_addr_list` would be called `ppcAddrList` and it would be typed `BYTE**`. Checking with the Type Table, we see that `BYTE` is just a synonym for "unsigned char" so it actually works out to almost the same thing, but it looks confusing here. If you want to see some real Stupid C Tricks in action, here is the function that I use to convert plus signs to spaces in the search page CGI client. I absolutely refuse to write anything more cryptic than this:

```
VOID DecodeSpaces (PTSTR p)
{
    TCHAR c;
    if(p) {
        while(c = *p) {
            if(c == '+') *p = ' ';
            p++;
        }
    }
}
```

```
}  
}
```

Shawn Fessenden

Copyright © 2005 [Black Cat Software]. All rights reserved.

Revised: