Extending Visual Basic with Microsoft Windows DLLs

Bob Gunderson
Microsoft Developer Network Technology Group

Created: January 18, 1993

Abstract

Microsoft® Visual Basic™ is a very powerful and flexible tool for developing applications for Microsoft Windows™. Despite this tool's capabilities, developers occasionally run into situations where Visual Basic either can't solve the problem directly or the solution in Visual Basic is clumsy and inefficient. In most cases, these problems can be solved fairly easily using a language such as C or C++. Fortunately, Visual Basic allows external functions in dynamic-linked libraries (DLLs) to be called from Visual Basic programs. Writing your own code in C and packaging it in a DLL that can be called by a Visual Basic program is an attractive and efficient way to extend the capabilities of Visual Basic.

Most of the current Visual Basic documentation concerning DLLs discusses the process of calling preexisting DLLs, specifically the native Windows application programming interface (API). Very little documentation exists that describes issues involving the creation of your own DLL that can be called from Visual Basic. This technical article covers the following topics:

- Linking to a DLL function from Visual Basic
- Visual Basic data types as parameters
- How to debug a DLL called from Visual Basic
- VBFILEIO, a sample DLL that gives Visual Basic applications access to C run-time stream input/output (I/O) functions

This technical article is written for developers who create applications for Windows and have used languages such as C in the past. Knowledge of the native Windows API, as well as how DLLs operate and are created, is assumed.

This technical article and the accompanying sample application were created using Microsoft Visual Basic version 2.0 and Microsoft C/C++ version 7.0.

Linking to a DLL Function from Visual Basic

In a C application written for Microsoft® Windows™, linking to a dynamic-link library (DLL) is usually done implicitly using an import library or entries in the application's module definition (DEF) file. There is a way to dynamically load and link to a DLL, but it involves calling Windows application programming interface (API) functions to explicitly load the DLL and locate functions within the DLL.

DLL function linking in Visual Basic is quite different from the way it's done using C. The key to calling a function in a DLL from a Visual Basic application is the **Declare** statement. Like a C function prototype, the Visual Basic **Declare** statement defines the name of the function, its parameters, the data types of the parameters, and values returned by the function. But the **Declare** statement goes one step further and also defines the name of the DLL in which the function resides.

Here are two C function declarations from the VBFILEIO sample and their corresponding **Declare** statements.

First, the C declarations in the DLL:

```
UINT __export CALLBACK CRTfwrite(LPSTR buffer, UINT size,

                                     UINT count, FILE *stream)

void __export CALLBACK CRTrewind(FILE *stream)
```

And the corresponding Visual Basic **Declare** statements:

```
Declare Function CRTfwrite Lib "vbfileio" (ByVal buffer As String,

    ByVal size As Integer, ByVal count As Integer, ByVal stream As
    Long) As Integer

Declare Sub CRTrewind Lib "vbfileio" (ByVal stream As Long)
```

Note Visual Basic does not have the ability to split a statement across multiple source lines. The declaration of the **CRTfwrite** function above was split into multiple lines for readability.

Visual Basic designates functions that return values as **Function** procedures (**Declare Function**) and those that don't as **Sub** procedures (**Declare Sub**). To a C function in a DLL, there is no difference between the two. Nothing can be returned by a Visual Basic subroutine, so any subroutines in a DLL should be declared to return a void.

Where you place the **Declare** statement is important. Placing it in the declarations section of a form restricts the function's use to code within that form. Placing it in a module, on the other hand, allows the function to be used anywhere in the application.

When you create a function in your DLL, watch out for naming restrictions enforced by Visual Basic. All external functions must adhere to Visual Basic's identifier naming conventions (defined in the Microsoft Visual Basic version 2.0 Programming System for Windows *Programmer's Guide*). Watch out for names beginning with the underscore character and names that are the same as a Visual Basic reserved keyword; these can't be used as names for external functions. For preexisting DLLs, you can use the **Alias** option on the **Declare** statement to call external functions with invalid names. You could use the **Alias** option for functions in your own DLL, but it is unnecessary if the names in the DLL conform to Visual Basic's naming conventions.

The DLL (VBFILEIO.DLL in the example code above) must reside in some location where Windows can find it. When Windows looks for a DLL, it first looks in the current working directory, then the Windows directory, followed by the Windows system directory, and finally the directories on the path. Unfortunately, Visual Basic usually runs with the current working directory set to the directory that contains VB.EXE, rather than the directory that contains the .MAK file for your application. Normally, you don't keep your DLL and application in the Visual Basic directory. This means that, unless you place an absolute path on the DLL name in the **Declare** statement, Windows will not be able to locate the DLL. The best solution is to move your DLL to the Windows directory, the Windows system directory, the Visual Basic installation directory, or some directory in the path.

DLL loading and linking under Visual Basic is a bit different than you might be used to. Normally, when a Windows-based application is loaded, any DLLs that the application uses are loaded automatically (assuming the application uses implicit linking). For Visual Basic applications, an external DLL is not loaded until the first call is made to a function in the DLL. Once loaded, the DLL remains loaded until the Visual Basic application is stopped. This is true both for Visual Basic applications run from Visual Basic, as well as Visual Basic applications run as an executable file.

Visual Basic Data Types as Parameters

Visual Basic supports a slightly different set of data types than C does. This, and some differences in parameter passing conventions, has an impact on the way information is passed between a Visual Basic application and an external DLL. The Visual Basic version 2.0 *Programmer's Guide* contains a convenient table that lists common C declarations and their Visual Basic equivalents. On the Developer Network CD, you can find the table in the topic "Converting Common Declarations," the last topic in Chapter 22. (The table is on page 505 of the hard-copy version of Visual Basic version 2.0 *Programmer's Guide*.)

Note Visual Basic always calls external functions using the Pascal calling convention.

Use of ByVal

You will notice that every parameter in the example code above has the **ByVal** keyword preceding the parameter name. By default, Visual Basic passes parameters by reference, that is, it pushes a far pointer to the data onto the stack rather than the actual value. C, on the other hand, passes all but strings and structures by value; it pushes a copy of the actual value onto the stack. You could write your Visual Basic callable DLL functions to accept everything by reference if you wanted to, but it is somewhat inconvenient to have to de-reference the pointer in order to use data. Preceding the parameter name with the **ByVal** keyword in the **Declare** statement forces Visual Basic to pass the parameter by value rather than by reference.

Note All pointers passed from Visual Basic to an external function are 32-bit far pointers.

Problems with String Parameters

Unfortunately for C programmers, Visual Basic does not store strings internally as a standard null-terminated string. Instead, Visual Basic internally manipulates strings using a 4-byte string descriptor. The string descriptor is used to locate the actual storage for the string, as well as its current length. Passing a string variable to an external function by reference actually passes a pointer to the string descriptor. Unless you are using the additional features of the Visual Basic Control Development Kit (discussed below), your application cannot use this string descriptor pointer because the format of the descriptor is not documented.

To create something more palatable to external functions written in languages such as C, Visual Basic will pass a pointer to a null-terminated string instead of a pointer to the string descriptor if the **ByVal** keyword is placed on a string parameter in the **Declare** statement.

Because most external functions assume a specific string buffer size, it is important to understand how Visual Basic stores strings, especially variable length strings. The number of bytes of storage space allocated for variable length strings is never more than the current length of the string. For example, the assignment:

```
mystring = "Hello World!"
```

allocates space for a 12-byte string. Changing the string to:

```
mystring = "Goodbye!"
```

deallocates the space previously allocated and allocates space for an 8-byte string. Exactly how the string space is allocated and managed is irrelevant. The important point is that passing a variable length string to an external function passes a pointer to an area only as long as the current string length. Writing beyond this area will corrupt other areas of data. If the external function needs a certain amount of string space for storing strings, the Visual Basic application must preallocate the string in one of two ways. The Visual Basic application could create a variable length string of the proper length by filling the string with nulls like:

```
tmp = String(200, 0)
```

Or, it could define the string as having a fixed length like this:

```
Dim tmp As String * 200
```

Either way, you will get a string that has enough room for 200 bytes. The DLL has no control over the allocation of the string, so there is no way to extend the string when it is passed using **ByVal**. This also means that you can't write an external function that returns a string. An external function can change the contents of a string passed to it, but the function can't return a string—that is, without resorting to the additional features found in the Visual Basic Control Development Kit. These additional features are designed for creating Visual Basic custom controls, but can be used for standard DLLs as well. See the discussion in the section "The Control Development Kit" below.

Note To pass a null string pointer (which is different from a pointer to a null string), pass the value "0&" for the string parameter. This will pass a long value containing all zeros—exactly what you want for a null string.

Passing Structures

Passing structures (called user-defined types in Visual Basic) is the only exception to the "always use the **ByVal** keyword" rule when passing parameters to external functions. Structures are almost always passed by reference, even in C. That is, structures are always referred to by a pointer to the structure. Passing the name of the structure from Visual Basic to an external function will pass a pointer to the structure, just as it does in C. The elements of the user-defined type are packed consecutively in memory, just like C structures when the /Zp compiler option is used.

Strings as part of a structure must be defined as a fixed-length string in Visual Basic. Variable-length strings in a Visual Basic structure are not stored in the structure itself. Instead, a string descriptor (see the previous section on passing strings) is stored in the structure. Again, DLLs that wish to use the capabilities of the Control Development Kit can access this internal information using functions designed for custom controls (see "The Control Development Kit" below).

When coding in C, it is convenient at times to create structures that contain pointers to strings. Many of the native Windows functions accept such structures. Without a little help, it is impossible for Visual Basic to create such a structure because there is no way to obtain the address of a string. However, calling an external function with a string as a parameter marked with the **ByVal** keyword passes just such a pointer. It is fairly easy in C to create a function that does nothing but return this pointer as a **long** data type. In the Visual Basic structure, the string pointer element can be declared as a **Long** data type and the return value from the external function assigned to this element. Here's an example.

First, the Visual Basic structure definition:

```
Type MyStruct

   strptr As Long
End Type
```

Next, the C code in the external DLL:

```
unsigned long __export CALLBACK StrAddr(LPSTR string)

{
   return (unsigned long)string;
}
```

And the Visual Basic **Declare** statement for the function:

```
Declare Function StrAddr Lib "MyDLL" (ByVal strng As String) As Long
```

And finally, the Visual Basic code to use the function:

```
Dim TestStruct As MyStruct

tmpstr = "Hello"
TestStruct.strptr = StrAddr(tmpstr)
```

This code will store a pointer to the string "Hello" in the strptr element of the TestStruct structure. Obviously, you need to ensure that the memory referenced by the pointer stays valid as long as the pointer is in use.

Passing Integer and Long Values

Integer and **Long** values pass easily to external functions. The only restriction is that Visual Basic does not support any sort of unsigned numeric data type. All **Integer** and **Long** data types in Visual Basic are signed values. There is really no easy workaround if your application needs to pass or receive unsigned values to or from a DLL.

Passing Arrays

Arrays of numeric data can be passed by reference by passing the first array element by reference (don't use the **ByVal** option). Arrays of numeric data are packed in Visual Basic just as they are in C. Refer to Chapter 22 (the section on arrays in "Calling DLL Procedures with Specific Data Types") in Visual Basic version 2.0 *Programmer's Guid*e for an example of passing arrays to external DLLs. (See page 499 in the hard copy of the *Programmer's Guide*.)

Passing Currency Values

The Visual Basic **Currency** data type has no corresponding C data type. There is, however, information in the Knowledge Base that describes the format of the **Currency** data type (on the Developer Network CD, search for *Q51414* or the terms *currency AND format.* The currency data type is 8 bytes long. A convenient way to ensure proper stack offsets when passing currency data to an external function is to define the currency parameter as a **double** value in the C function declaration.

Passing Date/Time Variant Values

These **Variant** data types obviously have no corresponding C data types. The easiest way to pass date/time data between Visual Basic and external functions is to convert the date/time **Variant** into a **Double** date type and then pass the **Double** to the external function. For example, the following Visual Basic statement will pass the current date and time as a **Double** value to the external function **DTParse**:

```
DTParse (CDbl(Now))
```

The **CDbl** function converts a date/time value to a **Double** value. The integral portion of the **Double** value is defined as the number of days since December 30, 1899, and the fractional portion is the fraction of the day since midnight. Don't worry—there is code in the VBFILEIO sample that takes a date/time **Double** and breaks it into day, month, year, hour, minute, and second values. Look for the function **DateTimeParse** in the file VBFILEIO.C.

The Control Development Kit (CDK)

A Visual Basic custom control, normally referred to as a VBX, is simply a specially built Windows DLL. The main difference between a DLL and a VBX is that the VBX code must follow a set of guidelines described in the Microsoft Visual Basic version 2.0 Professional Edition *Control Development Guide*. The guidelines define the protocol used by Visual Basic to allow the custom control to operate seamlessly with Visual Basic-supplied controls, as well as other custom controls. The *Control Development Guide* and all the supporting include files, link libraries, help files, and samples are part of Microsoft Visual Basic Programming System for Windows, version 2.0, Professional Edition.

The CDK exposes a low-level interface to Visual Basic that allows custom controls to define and manage properties, events, internal data types, and such. Although primarily designed for custom control authors, the functions defined in the CDK can be used by any DLL that wishes this functionality.

Debugging: How to Debug a DLL Called from Visual Basic

Any kind of debugging solution is very dependent on the development environment used to create the DLL. The information below pertains to Microsoft C/C++ version 7.0, but is most likely applicable to other development environments.

Visual Basic has no built-in support for debugging external functions in DLLs. But there is a technique that can be used to debug code in a DLL that is to be called from Visual Basic. You need to overcome two basic problems in order to debug the DLL. The first problem is figuring out how to get a debugger, like CodeView® for Windows (CVW), loaded and breakpoints set in the DLL at the appropriate places. The second problem is getting Visual Basic and the debugger to coexist. To allow the Visual Basic application to be debugged, Visual Basic itself contains a debugger of sorts. Having two debuggers resident in the system at once is fraught with problems and can easily cause system instability.

Normally, to debug DLL code, the application that calls the DLL is loaded into the debugger. This normally forces any DLLs the application calls into memory and makes the DLLs accessible to the debugger. This technique, however, cannot be used to debug DLLs that Visual Basic calls. You can't load VB.EXE into CVW. Trying this will, at first, appear to work. VB.EXE is loaded just fine, and the CVW Run/Load menu command can be used to load the DLL. But Visual Basic will terminate with a GP fault as soon as an attempt is made to load an application. This is most likely due to a conflict between CVW and the debugging portion of Visual Basic.

What is needed is a small application that does nothing but force the DLL to load. This application can then be debugged using CVW. Once appropriate breakpoints are set in the DLL, the Visual Basic application can be started. Because DLL code is shared (that is, all applications using the DLL share the same code segments), the breakpoint will be executed as the Visual Basic application calls the DLL. CVW then traps the breakpoint and allows you to debug the DLL code. This solution works—with one caveat. An .EXE version of the Visual Basic application must be used in order for this scheme to work. Running the Visual Basic application from Visual Basic while CVW is running will usually cause the system to become unstable when the application is terminated. Running the Visual Basic application as an .EXE file works without problems.

The CRTDLL sample accompanying this technical article contains a small application called VBDEBUG. This application does nothing but create a blank window. The module definition file (VBDEBUG.DEF) contains the following lines, which force VBFILEIO.DLL to be loaded:

```
IMPORTS

    vbfileio.CRTfopen
```

The statement "vbfileio.CRTfopen" forces the Windows loader to load CRTDLL.DLL and look for the function **CRTfopen** in that DLL. For your own DLL, just change "vbfileio" to the name of your DLL and "CRTfopen" to the name of an exported function in your DLL.

VBFILEIO Sample: C Run-time Stream I/O Functions

Coming from a C programming background, I was somewhat disappointed at how different the Visual Basic I/O functions were from the standard C run-time I/O functions. I took this as an opportunity to demonstrate how to create a Visual Basic-callable DLL. The result is a DLL that allows Visual Basic applications to use the standard C run-time stream I/O functions. The DLL does nothing more than create an external callable interface to the run-time functions. For example, here is the code in the DLL for the **fopen** function:

```
unsigned long __export CALLBACK CRTfopen(LPSTR filename, LPSTR mode)

{
    return (unsigned long)fopen(filename, mode);
}
```

The corresponding Visual Basic **Declare** statement for **fopen** looks like this:

```
Declare Function CRTfopen Lib "crtdll.dll" (ByVal filename As

   String, ByVal mode As String) As Long
```

The (FILE *) data type used by stream I/O run-time functions is represented in Visual Basic by a **Long** data type. Since the Visual Basic application only uses this value to pass to other DLL functions, encapsulating the (FILE *) value in a Visual Basic **Long** variable works just fine.

A few of the stream I/O functions were not included in the DLL, most notably **printf** and all its variations. The **printf** functions accept variable numbers of parameters. It is not possible to pass a variable number of parameters using the PASCAL-style calling convention, which is what Visual Basic uses to call external functions.

VBFILEIO.DLL is built as a large-model DLL. Because all pointer values passed to the DLL from Visual Basic are __far pointers, the large-model run-time library was used. (It is the only one that accepts far pointers for pointer parameters.) It would have been possible to use small- or medium-memory models and copy data into near buffers.

A small Visual Basic sample application (CRTTEST) that demonstrates calling each of the functions is some of the sample code accompanying this article. The file GLOBAL.BAS included with the sample contains **Declare** statements for all functions in the DLL along with a few useful constants. The following table is a quick reference for each of the functions implemented in the DLL. Full descriptions of the C run-time function and its parameters can be found in the Microsoft C/C++ versions 7.0 *Run-Time Library Reference* that comes with the C/C++ product. This manual is also on the Developer Network CD (Product Documentation, C/C++ 7.0).

Table 1. C/C++ Functions Supported in CRTDLL.DLL

Function	**Description**
CRT_fcloseall	Calls **_fcloseall**. Closes all files opened with **fopen**.
CRT_flushall	Calls **_flushall**. Flushes buffered I/O for all open streams to disk.
CRT_rmtmp	Calls **_rmtmp**. Deletes all temporary files created by **tmpfile** in the current directory.
CRTclearerr	Calls **clearerr**. Resets the error indicator for a stream.
CRTfclose	Calls **fclose**. Closes an open stream.
CRTfeof	Calls **feof**. Tests for end-of-file on a stream.
CRTferror	Calls **ferror**. Tests for an error on a stream.
CRTfflush	Calls **fflush**. Flushes buffered I/O to a particular stream to disk.
CRT	Calls **fgetc**. Reads a

fgetc	single character from a stream.
CRTfgets	Calls **fgets**. Reads a line of characters from a stream.
CRTfopen	Calls **fopen**. Opens a file for I/O.
CRTfputc	Calls **fputc**. Writes a single character to a stream.
CRTfputs	Calls **fputs**. Writes a line of characters to a stream.
CRTfread	Calls **fread**. Reads an arbitrary number of characters from a stream.
CRTfseek	Calls **fseek**. Moves the file pointer to a specified location.
CRTftell	Calls **ftell**. Gets the current position of a file pointer.
CRTfwrite	Calls **fwrite**. Writes an arbitrary number of characters to a stream.
CRTrewind	Calls **rewind**. Moves the file pointer to the beginning of a file.
CRTtmpfile	Calls **tmpfile.** Creates a temporary file in the current directory and opens it.
CRTtmpnam	Calls **tmpnam**. Creates a temporary filename.