 ***VBZ*** **Table of Contents (Unregistered Copy)**

VBZ

The Electronic Journal for Visual Basic Programmers

Issue 01: January/February 1993

Unregistered Copy

Welcome to the first issue of *VBZ*, the electronic journal on Visual Basic. Bear in mind that this issue is bigger than others will be because it represents a compilation of most of our submissions to Compuserve and elsewhere over the past year. Please feel free to contact us with your suggestions for ways to improve the journal. Our goal is to make this the premier resource for Visual Basic programmers. Enjoy the first issue and keep in touch!

Jonathan Zuck, President
User Friendly, Inc.
CIS: 76702,1605

Table of Contents

Fixes and Updates

About *VBZ*

VBZ is a completely electronic journal for Visual Basic programmers. Each issue is in the form of a help file, such as the one you are reading, with lots of techniques, sample code, DLLs, custom controls and reviews. A new issue of *VBZ* will come out every two months unless there is strong demand for greater frequency at the expense of content in each issue. We think it's better to wait and get some substantial stuff into the journal rather than just try to get it out every month as some others do.

Subscribing to *VBZ*

If you are not currently a subscriber, you should be. We accept MasterCard or Visa, or send check, money order or purchase order (payable to User Friendly, Inc.) for $69 + $4 shipping and handling for the first subscription, $49 + $4 shipping and handling for each additional subscription, to:

User Friendly, Inc.
1718 M Street, N.W.
Suite 291
Washington, DC 20036
(202) 387-1949
(202) 785 - 3607 FAX
CIS: 71652,2657

Please specify whether you would like to receive *VBZ* via email (in which case please include your Compuserve account number), or US Mail (in which case please include your preferred media size).

VBZ may also be registered on CompuServe's Shareware Registration Forum (GO SWREG). *VBZ's* Registration ID Number is 1005. $74.00 will be billed to your CompuServe account, and we will be notified to email you your subscription.

License Information (Unregistered Copy)

Just like a paper journal, *VBZ* should not be in two places at once. Therefore, its license information is quite simple. If you pass the journal around, you must do it in its entirety, erasing it from your own hard disk. In this way, many can read a single issue of *VBZ*, but only one at a time.

As a subscriber you have full rights to distribute the DLLs in their binary form (no source code) with your programs. You may not document their internal use, however. If more than one developer will be actively using the DLLs, each developer must subscribe to *VBZ*.

PLEASE NOTE! These rights only apply to registered subscribers of *VBZ*. This is an unregistered copy of *VBZ*. You have the right to read the journal and experiment with the software but you must subscribe in order to distribute the DLLs. The good news is that this will only be the first of many issues of *VBZ*, all of which will be useful.

Disclaimer

The techniques and software presented in this journal are offered "as is" with no warranty expressed or implied as to their suitability to the task to which they are applied or reliability under diverse conditions. The primary purpose of this journal is education. You use the software and techniques in this journal at **your own risk.**

Using *VBZ* Dynamic Link Libraries

Each DLL that comes with an issue of *VBZ* contains a version resource so that you can use the Windows version verification procedures when installing one of our DLLs over an existing one as part of a setup procedure. Please make every effort not to overwrite a newer copy with an older one that another developer might have installed on the user's system.

Whenever a DLL is fixed, only the minor version of the DLL will go up. Only when substantial functionality has been added to the library will the major version increase.

Each DLL also has a module (.BAS) of the same file name with the necessary declarations and constants to allow you simply to add this file to your project and start programming. Let us know if there is anything else we can do to make the use of our DLLs any easier.

Writing for *VBZ*

If you are interested in getting your name in print or simply looking for a way to share your findings, we are interested in hearing from you. Because of *VBZ's* format, there is no real limit as to the number or size of submissions. Even if it's just a little technique you have discovered, I'm sure readers would be interested in hearing about it.

In addition to the notoriety, one benefit of using *VBZ* to get your idea out is that you have an organized way to fix or improve the idea later on. We will do our best to work with you to get your idea working in the first place, but we will also make sure that corrections go out to all our subscribers. This kind of dynamic approach is one of the unique benefits of electronic publishing.

If you wish to make a submission of some sort, please try to make it problem-solution oriented versus "general discussion." We will be happy to look at anything but the journal is targeted at those with specific problems and needs.

Please send your ideas, either by mail or email. We look forward to getting your name in print!

VBZ Submission
User Friendly, Inc.
1718 M Street, N.W.
Washington, DC. 20036
(202) 387-1949
(202) 785-3607 FAX
CIS: 76702,1605

System Level Hotkeys

HOTKEY.DLL is designed to provide hotkey support to Visual Basic (VB) as this is not directly supported in VB. You can define multiple hot keys in your application and the DLL can be used by multiple applications simultaneously.

The Point:

The point is that you might want to create a TSR-type utility in VB that "pops up" or performs some background task given a certain "hot key." For example, let's say that you want to be able to insert the current date at the cursor location of whatever text editor you are using. This is normally impossible to accomplish in VB. Not anymore! Now you can create a utility that has a hot key of Ctrl-D that, while as an icon, inserts today's date in whatever application you are using. Another example might be a popup calendar application. Normally, the user would have to double-click on your icon, but now you can define a key that brings your application to the fore front for immediate use. I am certain that when you think about it, you will be able to come up with much more interesting ideas, but those are just a couple.

How Do You Do It?

Well, HOTKEY.DLL has two functions: CreateHK and KillHK. The former establishes a hotkey and the latter gets rid of them. The syntax for them is as follows:

hHotKey = CreateHK (VKCode, Shift, Wnd, UserVK)

where VKCode is an valid virtual key code. These are the same codes used in the KeyDown and KeyUp event handlers. The values for the different keys can be found in CONSTANT.TXT with the KEY_ prefix.

Shift is the mask for the shift keys you want included in your hot key. These are the same as those used in the KeyDown and KeyUp events in the "Shift" parameter. These values can be added together for multiple shift keys. This concept is explained on page 277 of the Programmer's Guide.

Wnd is the hWnd property of the Form that you want to receive "hot key events." Therefore, you would normally simply pass hWnd as this parameter.

UserVK is the value you want sent to your window when your hot key comes up. This can be any value you want. It is best to start with values over 144 as this is where the normal Key_Codes end. This can be *any* valid integer.

hHotKey is returned by the function as the "handle" to the hotkey. This is used only to "kill" the hotkey later on. This will be set to (-1) if the hot key you selected is already in use. It will return a (-2) if

there are no more hot keys. You are limited to 1024 simultaneous hot keys on your desktop. Note: this does not include any hot keys that might be defined in the Windows Macro Recorder. Those are independent of HOTKEY.DLL. Also, HOTKEY.DLL *cannot* detect if a hot key is already taken by the Windows Macro Recorder. Be careful of overlap!

Once you have created as many hot keys as you want, you need to create a KeyDown event handler for that form. This event is rarely used because normally it requires that the form have focus. This can only happen when there are no controls on the form or they are all disabled. Never fear, however: HOTKEY.DLL can easily coexist with normal KeyDown processing as long as you define UserVK codes that do not represent valid virtual key codes. Anyway, in the KeyDown handler, you will see two parameters: KeyCode and Shift. The only one of interest to us is KeyCode because the event will only be triggered if all the proper shift keys are pressed! What the KeyCode parameter will contain is the UserVK code that you specified when creating your hot key. You can then use SELECT CASE, for example, to process the different hot keys that you have created. All this can transpire with your app as an icon and without your app ever receiving focus!

When Unloading Your Form:

When you are unloading your form, please call KillHK and pass the "handle" returned to you by CreatHK. Not only will this allow other applications to use those hot keys but HOTKEY.DLL does a lot of cleanup when a hot key is "killed" so that key processing is as fast as possible. When you consider that the DLL has to filter *every* keystroke, it is imperative to process keys as fast as possible. I think it's pretty quick but I would like to hear comments from you on any performance hits you feel you are taking. I don't think the human eye will be able to notice.

The Example:

HOTKEY demo (HOTDEMO.* source files) is a *very* simplistic demo of some of the features of HOTKEY.DLL. Essentially, it loads itself and looks for two hot keys (Ctrl-Alt-F1 and Shift-F12) and displays a message box when those keys are detected. This takes place whether the demo application is the active one or not.

How did I do it? (For those who care...)

HOTKEY was written in Turbo Pascal for Windows. I like Pascal better than C and, as a matter of principle, I would choose to use the tool that cost me $199 and included a compiler, windows based debugger and the Whitewater Resource Toolkit all in a Windows based development environment (hey, I thought it was MS who was hot on Windows!) that allows me to create *one* source file instead of four (with all sorts of crazy link options and import libraries!). The C/SDK combo cost me close to $1,000 and it is a PITA to use.

HOTKEY was not implemented as a custom control because: 1) it would have been more trouble than it was worth; 2) it probably would have required multiple keyboard hooks instead of one, which would degrade performance; 3)it would only work with VB and we use

lots of other similar tools; and finally 4) it would have to have been done in C! Argh! Besides, Crescent Software is creating a low-level keyboard handler control that I will simply buy.

Essentially, HOTKEY has four routines. There is an initialization routine that dims an array of record variables (you know TYPE..END TYPE) to 1024 elements. It then sets some global variables. Finally, it makes a call out to SetWindowsHook, telling it to trap all keys and to filter them through my keyboard handler: KBProc

CreateHK searches through the array to see if the hot key is already taken. If not, it inserts the passed parameters into the next possible element of the record variable array and increments the count of hotkeys in the system. The long integer returned by CreateHK is actually a combination of your hWnd and your UserVK. This way, the hotkey is uniquely identified.

KillHK moves everything, below the hot key being removed, up one element in the record variable array and decrements the total. I chose to take the extra time to do this here rather than make the search longer for every keystroke.

KBProc constructs a mask using GetKeyState for the SHIFT and Ctrl keys and using lParam for the ALT key. It then scans through the record variable array looking for the same keycode and mask. If KBProc finds it, it passes the WM_KEYDOWN message to the corresponding hWhd, passing the UserVK that you provided as the parameter. If the key is not found, it is passed on to the next keyboard handler, which is usually the application you are working in.

The exit procedure unhooks KBProc from the keyboard chain and frees up the memory taken by the record variable array. That's about it. I hope you are able to find this useful in your applications and I welcome any suggestions or comments.

Aldus Format Metafiles

A number of advanced users have been creating metafile pictures on the fly in Visual Basic. There are a number of advantages to metafiles, but VB only has the ability to retrieve them from disk or the clipboard. There is no way to save them if you create or modify them. In order to modify a metafile, you would need to be able to load them via code rather than into a picture box. To save them you need to be able to save them from code, rather than from the contents of a picture box. Therefore, I have provided AMETA.DLL, which will take a metafile handle and save it to disk or retrieve a metafile into a memory. For more information on creating and modifying metafiles, I recommend getting a copy of *Visual Basic Programmer's Guide to the Windows API*, by Dan Appleman, from Ziff-Davis Press.

Using the DLL

Using the functions in AMETA.DLL is simplicity itself. If you know how to manipulate metafiles (which is explained in Appleman's book and not here!), a single function call is all that is required to save your metafile in the placeable format or to load metafiles stored in this format.

Function Summary:

Meta2Aldus	Creates a Disk Metafile from a Metafile Handle
LoadWMF	Loads a Disk Metafile into a Metafile Handle

The functions are declared in VB in the following fashion:

```
Declare Sub Meta2Aldus Lib "ameta.dll" (ByVal hMeta, Place As Rect, ByVal nUnit, ByVal FName$)

Declare Function LoadWMF Lib "ameta.dll" (ByVal FName$,  Place As Rect, nUnitInteger%)
```

The Place parameter in each case is the bounding rectangle for the metafile so that the relative size and aspect ratio of the metafile is preserved. nUnit specifies the units that are to be used.

Sample Files

META.MAK is the sample that demonstrates how to create a metafile and save it to disk.

A Meta-WHAT?

Perhaps it would be helpful to begin with a brief discussion of metafiles and their function (how they exist in memory). Metafiles, or Windows vector images, are a special picture format that have a number of advantages over bitmaps. The two primary benefits of metafiles are that they are a fixed size in memory and their scale mode is set dynamically at playback time, allowing for run-time flexibility, fixed resolution and device independence. Bitmaps, on the other hand, can attain higher resolutions, but can grow to be very large and the flexibility with which they can be displayed is severely limited.

When you make a call to CreateMetafile, an hDC is returned to you. This is not really a true hDC but rather an alias to which you send graphics commands to be "compiled" into p-code. Therefore, when you issue a command like LineTo or Circle, rather than executing the command,

Windows compiles and stores the command in the metafile. It is only with a call later to PlayMetafile that these commands are actually interpreted and executed. In essence, you are creating a kind of script that will be used later to actually draw an image.

If Memory Serves . . .

Metafiles are actually stored in a kind of p-code, much like the code generated by a product like VB or MSC7. In memory, or on disk, metafiles begin with a header of type METAHEADER which is prototyped like this, C:

```
typedef struct tagMETAHEADER {
    WORD   mtType;
    WORD   mtHeaderSize;
    WORD   mtVersion;
    DWORD mtSize;
    WORD   mtNoObjects;
    DWORD mtMaxRecord;
    WORD   mtNoParameters;
} METAHEADER;
```

The most important field for us here is the mtSize parameter which will tell us how many bytes to read from a file. More on that later. After this header there are a series of variable length records that contain the p-code implementation of the metafile. The typical record looks like this:

```
struct {
    DWORD rdSize;
    WORD   rdFunction;
    WORD   rdParm[];
}
```

where rdSize is the size of the record, rdFunction is the id of the function (ex. 0x0817 is the ARC function) and rdParm is an array of integer containing the parameters to the function, in reverse order.

A metafile, therefore, is an array of these structures that are interpreted and played back, using the PlayMetafile function. The hDC returned by CreateMetafile is actually a handle to a blob of memory that contains this format. The Aldus Placeable Metafile format, as we will see, simply adds a header to the top of this format.

The Aldus Advantage

Missing from the METAHEADER structure is any sort of information about the original size or aspect ratio of the metafile. Thus, the Placeable Metafile format was born. This format adds the following 22-byte header to the top of the METAHEADER:

```
typedef struct {
    DWORD    key;
    HANDLE   hmf;
```

```
    RECT      bbox;
    WORD      inch;
    DWORD     reserved;
    WORD      checksum;
} ALDUSHEADER;
```

where key is a reserved value that is intended to identify the Aldus format: 0x9AC6CDD7L, hMF is unused and should be set to zero, bbox is the bounding rectangle of the metafile used for reproducing the aspect ratio and position of the metafile, inch refers to the number of metafile units (remember that they are all relative) to the inch, reserved should be zero and checksum is the checksum of the first 10 words of the header. Once we know this much, we're half way there so let's get started with the DLL: AMETA.DLL.

Loading Metafiles

Because I can't be sure that the WMF extension is necessarily an Aldus Placeable Metafile, I need to determine which metafile format a file contains, or give it my best shot, anyway. The first step is to try and read the first 22 bytes into an AldusHeader. The Key field is then checked for the special constant: $9AC6CDD7. If I find the key, I copy the rectangle in BBox to the rectangle passed to the routine. Next the value of the Inch field is copied into the nUnit parameter. This way, the caller can make use of the aspect ratio and size information. Next, the METAHEADER is read into the MHeader structure and the cursor is repositioned at the top of the METAHEADER in the file.

If the Aldus key is not found, the cursor is repositioned to the start of the file, the METAHEADER is read and the cursor is again moved to the beginning of the file. There is no good way to identify a non-Aldus metafile so you will need to hope that the metafile is stored in one of these two formats. In either case, the net result is that the MHeader record variable has been populated and the cursor moved to the beginning of this header in the file.

At this point, the process of creating either type of metafile is identical so I use the mtSize field to determine the amount of memory that I need to allocate and load those many bytes into the memory area. I have used the new Windows 3.1 _hRead function so that I can accommodate metafiles greater than 64k such as Corel might produce. If you plan to run on Windows 3.0, you will have to do a little more checking of the size and use the lRead function, simply preventing users from loading large metafiles. Finally, I use SetMetaFileBits to create the actual metafile that is returned by the function. The memory is not then freed because is used to contain the contents of the metafile, the handle of which is returned as the function result.

Saving the Metafile

The process of saving the metafile is more direct because I know what format I want. It is merely the process of populating the Aldus header with the correct information, most of which is passed by the calling program, and calculating the checksum value by Xor-ing everything. Next I write out the Aldus header, followed by the data returned by the GetMetafileBits function, again using the new Win 3.1 huge memory routine: _hWrite.

Creating a Drag 'n Drop Server Application

While the process of being a Drag 'n Drop client application is clearly documented for C programmers, the process of becoming a server application isn't documented at all, even for those using the SDK. Therefore, I have written a DLL which provides the necessary functionality to Visual Basic. D&DSERVE.DLL is designed to take all of the heartache out of being a D&D server. Once you get a _MouseDown event, the DLL takes over and does the rest, including the changing the cursor and eventually dropping the files.

Function Summaries:
DropSelItems Drops the selected files in a multi-select List Box
DropAllItems Drops all the files in a list box
DropBuff Drops files contained in a passed buffer

Function Declarations:

```
Declare Function DropSelItems Lib "D&DSERVE.DLL" (ByVal hList, ByVal Path$, ByVal
    nButton)
Declare Function DropAllItems Lib "D&DSERVE.DLL" (ByVal hList, ByVal nButton)
Declare Function DropBuff Lib "D&DSERVE.DLL" (ByVal Buff$, ByVal nButton)
```

Function Details:

DropSelItems: (D&DTEST1.MAK)

You must use a multi-select list box to use this function. The demo code for this function looks something like this:

```
Sub MyFileList_MouseDown (Button As Integer, X As Single, Y as Single)
    Path$ = Dir1.Path + "\"
    hTarget = DropSelItems (MyFileList.hWnd, Path$, Button)
End Sub
```

where hTarget will be the hWnd of the Window on which the selected files were dropped or Zero if the mouse was not released over a valid window. One caveat is that the left mouse button will generally remove the selections from an extended selection list box (the File Manager does a ton of special processing). Therefore, you might want to specify for the user that the Right mouse button be used to drag from the list box.

DropAllItems: (D&DTEST2.MAK)

This function has two advantages. First, it does not require a multi-select list box so those of you still using VB1.0 and without add-ons will be able to use it. Second, it is capable of dealing with files from multiple directories which DropSelItems (or even FileManager) is not capable of. Keep in mind that since you do not pass the path to this function, each file name in the list must be fully qualified. When used, this function will drop all of the items in the list on the target window.

DropBuff: (D&DTEST3.MAK)

This function is provided primarily for those programming systems that are not able to use a standard list box or have special needs. In this case, you are essentially constructing the buffer that will be sent to the target window. The format of this buffer is a fully qualified filename, followed by a null, followed by another filename+NULL, etc., and finally terminated by another null.

Visual Basic will add the terminating null so you don't need to worry about the second one. You just need to construct a string like so:

```
N$ = Chr$(0)
Buff$ = "C:\AUTOEXEC.BAT" + N$  + "C:\DOS\README.TXT" + N$
```

and then pass it ByVal to the function. I don't generally recommend this function as it is the least intuitive to the user but it might be helpful under special circumstances.

Testing your Code:

I have found WRITE to be the best place to test these functions as it is one of the only WINAPPs that supports the dropping of multiple files (into the client area, *not* on the icon!).

Source Code:

The source code for D&DSERVE.DLL was written in Turbo Pascal for Windows.

Creating a Drag 'n Drop Client Application

One of the most exciting new features of Windows 3.1 is support for an inter-application Drag 'n Drop protocol. When working in the File Manager utility that ships with Windows 3.1, you can select one or more files and, using the mouse, "drag" those files and drop them on another application. The program from which the files are dragged is called the D&D Server and the program which receives the files is referred to as the D&D Client. The examples of clients that ship with Windows 3.1 include Notepad, PaintBrush, Write, Program Manager and Print Manager. In the case of Notepad and PaintBrush, the dropped file is simply loaded. In Write, if a file is dropped in the client area (the editing area) then the file is added as a packaged OLE object in the open document. Only if the file is dropped in the non-client area (the title bar or perhaps the icon of a minimized instance of Write) is it actually loaded as the active document. Dragging a file to the Program Manager adds it to a group and dropping a file on the Print Manager queues that file for printing.

While these applications only scratch the surface of the potential of Drag and Drop, they do fall neatly into three general categories of D&D functionality: document loading, document embedding and document management. The purpose of this article is to reveal what is involved in exploiting this technology from Visual Basic as well as some of the considerations that need to be addressed when providing functionality in one of these three categories.

Ease vs. Flexibility

We are often confronted with a trade-off between ease of use and flexibility. Such a conflict becomes obvious to Visual Basic programmers with the introduction of version 3.1 of Windows. Because VB is designed to hide much of the intricacy of the operating system, it is consequently resistant to improvements or additions to that system. Drag and Drop is an example of added functionality that is easy for a typical C programmer to implement but difficult for a VB programmer because Windows is not designed around the VB programming paradigm.

Communication with applications under Windows is accomplished via series of messages that are sent either by other applications or by the operating system itself. A typical C program is built around a single routine. This function has the equivalent of an enormous SELECT CASE statement that takes action based on received messages. Because these messages are sent with a standard set of parameters, it is often the job of the C programmer to manipulate these parameters depending on the message. For example, the last parameter is always a long integer but sometimes it need to be treated as a pointer to an array or perhaps as two short integers.

In VB these messages have all been built into the development environment as standard events with meaningful names and parameters. For example, when a C programmer deals with the WM_MOUSEDOWN message, he must remember that the wParam specifies which mouse button and the lParam contains the X and Y coordinates of the mouse. The VB programmer simply gets the _MouseDown (Button, X, Y) event. This makes the processing of events a much more straightforward exercise. The problem arises, however, when a particular message isn't supported by VB such as the new WM_DROPFILES in Windows 3.1. While a C programmer would simply add another test to the big switch statement, there is no way to add events to a form in Visual Basic . . . or is there?

Trapping Messages Directly

There are two ways that a message gets to a Windows application: it is either "sent" or

"posted." High priority messages are sent by the operating system simply by calling the main window procedure (the one with the big Select Case statement). Lower priority messages are posted to the application and sit in a queue of up to eight messages, and are retrieved when the application has a chance to get to them. Ultimately, the application retrieves the messages from the queue and sends them to itself but that doesn't really concern us here. What does concern us is that we can't touch messages that are sent to a VB application (unless we use a custom control like CSFORM in Crescent Software's QuickPak or the subclass control in Desaware's Spyworks look for a generic subclass control in a future issue of *VBZ*) but we can take a peek at the queue for messages that have been posted to us. Luckily, the WM_DROPFILES message gets posted so we can intercept it.

The API function we use to peek in on the queue of waiting messages is appropriately named PeekMessage. This function is declared as follows:

```
Declare Function PeekMessage Lib "User" (lpMsg As Msg, ByVal hwnd%, ByVal
wMsgFilterMin%, ByVal wMsgFilterMax%, ByVal wRemoveMsg%)
```

where Msg is a record variable which is defined in the following manner:

```
Type Msg
    hWnd As Integer
    MsgNum As Integer
    wParam As Integer
    lParam As Long
    Time As Long
    Pt As POINTSTRUC
End Type
```

and POINTSTRUC is another record variable which is simply two integer values used to specify the X and Y coordinates of the message:

```
Type POINTSRUC
    X As Integer
    Y As Integer
End Type
```

PeekMessage is a boolean function that returns a 1 if the first message in the queue for the window specified with hWnd% is within the range we set with wMsgFilterMin% and wMsgFilterMax%. The Msg structure is then populated with the information about the message and its parameters. The problem arises, however, when we need to look at the next message in the queue, since PeekMessage only looks at the first message. The only way to look at the next message is to allow the application to process the first message. However, your application will not pull messages from the queue (using GetMessage coincidentally) until your routine is finished processing. The way to make your routine take a gasp is with a call to DoEvents. You would therefore construct a loop to look for the WM_DROPFILES message (563) like this:

```
While DoEvents ()
```

```
    Res = PeekMessage(NewMsg, hWnd, 563, 563, 3)
    if Res <> 0 then
        'process message here . . .
    end if
Wend
```

The best place to put this sort of loop is Sub Main in a VB program. This allows us to minimize the code in the form and keep it separate. A number of messages are posted to applications and are therefore accessible from a PeekMessage loop. For example, it is possible to centralize keyboard handling in a VB program using this same technique.

Handling the WM_DROPFILES Message

Now that you have looked at the micro level and determined how to handle message processing in VB, the next step is to determine what's involved in processing files dropped on your application by the File Manager. There are four important functions in the new SHELL.DLL for use when providing D&D functionality. These functions are listed in Table 1.

The first step is to register your window as a recipient of files. You don't want the user dropping files on just any old application and the way to alert the system to your willingness to accept files is to call the DragAcceptFiles function which is declared the following way:

```
Declare Sub DragAcceptFiles Lib "Shell" (ByVal Wnd%, ByVal Accept%)
```

where Wnd is the hWnd of the form that is to receive files and Accept is a boolean value which specifies whether or not to accept files. Once you have made a call to this function, File Manager knows that it can drop files on your form or icon and will send you the WM_DROPFILES message to notify you that it has done so. Using the PeekMessage loop outlined above, you will be able to determine when this happens.

Now that you have the message, you need to process it. That's actually the easy part! Once you know that you have the message you need, you use the various fields in the MSG structure to determine which files were dropped on your window. Actually, the only field of importance here is the wParam which contains a handle to an internal data structure containing information about the files that have been dropped. This is actually just a handle to global memory that we could walk directly but since SHELL.DLL provides an API for using this handle, who am I to do something else?

The first function of interest is the DragQueryPoint function which is declared like so:

```
Declare Function DragQueryPoint Lib "Shell" (ByVal hDrop%, lpPNT as POINTSRUC)
```

where hDrop% is the handle that was sent as the wParam of the WM_DROPFILES message and lpPNT is a long pointer to a POINT structure (shown above). The lpPNT will be populated with the X and Y coordinates of the dropped files relative to the client area of the window. The DragQueryPoint function is a boolean function that specifies whether the files were dropped in the client area of the window or not. It is with this information that WRITE decides whether to load the file, if it's in the non-client area, or to simply add the file as an OLE object at the appropriate location in the document.

Now that you know where the files were dropped, you need to find out which files were

dropped and that is the purpose of the DragQueryFile function which is declared as below:

```
Declare Function DragQueryFile Lib "Shell" (ByVal hDrop%, ByVal iFile%, ByVal lpBuff$,
ByVal BuffSize%)
```

where iFile% is the file number (zero based), lpBuff$ is a pointer to a buffer in which to place the file name and BuffSize% is the size of the buffer. You first call the function with a -1 as the file number and it will return the total number of files. If iFile% contains a valid number then DragQueryFile returns the number of bytes that were copied into the buffer.

Finally, you will want to free the dropfile handle and this is accomplished with a simple call to the DragFinish function in SHELL.DLL.

A Generalized Module

Whenever I implement new technology in VB, I try to encapsulate it entirely into a module that can be reused. Sometimes this is more difficult than others. This module, ~D&DMAIN.BAS (the tilde forces the module to the bottom of the project module list), can be added to any project to add client Drag and Drop capability. This module contains a fairly generic Sub Main that can be added to your project. Of course to make it work, you will have to specify that Sub Main should be the "Startup Form." To a make a re-usable module I had to use a couple of tricks (see the Sidebar entitled Stack Substitution) and make a couple of assumptions.

The first assumption is that the FormName property of the main form is set to "ClientForm" rather than its default "Form1." There is no clean way to determine the FormName dynamically and this method allows you to be certain about which form is being used for Drag and Drop purposes.

The second assumption is that you have a subroutine in another module called Form_FileDrop which is prototyped as follows:

```
Sub Form_FileDrop (FileArr$(), nFiles%, X%, Y%, Client%)
```

where FileArr$() is a string array that will contain the names of the files that were dropped, nFiles% contains the number of files (I suppose you could use UBOUND but this seemed cleaner when I wrote it), X% and Y% contain the point at which the files were dropped and Client% is a boolean variable which specifies whether or not the files were dropped into the client area of the target window. I chose to name the routine this way in hopeful anticipation of this becoming a standard event in the next version of VB. You need a separate module because, while you can create a routine of this name as part of your form code, it cannot be called from an outside module. Bummer.

At first glance this seems like extra work because you need to have two different module in your project; and, in fact, it is more work initially. However, after you have built a module of this sort once, the process of building a D&D client application becomes trivial as you will see in the two examples below.

A Punk Rock Trash Can (an esoteric New Wave reference?)

As I said earlier, one of the type of applications that take advantage of D&D is File Management. By this I mean that the utility exists simply as an icon at the bottom of the screen and when one or more files is dropped on the icon, something gets done with them. A classic

example would be a trash can which deleted the files that were dropped on it. Other examples include loading multiple instances of the application that uses that file (because you can only drag one file to Notepad and PaintBrush, for example), zipping the files, appending the files and many more. For my example, I chose to write a trash can utility because the amount of processing code is minimal.

My first goal was to create a program that existed only as a utility that couldn't be restored to a normal window by double-clicking. Once again, this is a task which is easily accomplished in C. You just return a zero when you get the WM_QUERYOPEN message. At first, you might think that you could use the PeekMessage method to deal with this message but there are two problems. First, this message is sent so you can't see it and second, there is no good way for you to respond to a message. Therefore I needed an alternative. Through experimentation I found that as I removed elements from the window such as .MinBox, corresponding options disappeared from the system menu. Accordingly, I took this to an extreme and turned off everything and wouldn't you know the restore option disappeared as well and double-clicking to restore went away in the bargain. Table 1 contains the definition of VBTRASH.FRM and if you set these properties the only options in the system menu are Move, Close and Switch To. . . . Cool!

Table 1. VBTRASH.FRM Definition

Property	Setting
BorderStyle	1 - Fixed Single
Caption	TRASH
FormName	ClientForm
Icon	Icon {icons\computer\trash01.ico}
MaxButton	False
MinButton	False
WindowState	1 - Minimized

One thing which requires care is the WindowState property. It seems to change back to normal when you least expect it. Therefore, make sure that you set it to "1-Minimized" just before you build your EXE. Don't forget that the Sub Main depends on the fact that FormName is set to "ClientForm."

The final set is to add a new module to the project. I called mine VBTRASH.BAS. As you can see, the process of deleting all of the files is simply a matter of a stepping through the FileArr string array. Creating new utilities is simply a matter of changing the contents of this module and the icon property of the form. It took me an hour to write the first utility and fifteen minutes to write three more!

Status Bar Help on Controls and Cursors

The Problem:

A number of people have asked for a way to provide a status bar for the user, depending on the position of the mouse. There are two aspects to this. First, it is necessary to know when a menu item has been highlighted by the user (although not yet selected). Both MS EXCEL and MS WORD for WINDOWS provide a short help prompt depending on which menu item is highlighted.

The second part of this task is to provide some help prompt when the mouse is hovering over a particular control. While the Mouse_Enter event is available under HyperCard and Toolbook, it is missing from VB (and from Windows, in general).

The Solution:

As you might guess, MWATCH.DLL is the solution! <g> or at least a partial one. Once you register your Form with MWATCH, specifying the control (or form to be notified) a _KeyDown event will be triggered every time the cursor is someplace different, either over a new control or in a menu. If the cursor has been moved over a control, the hWnd of that control is passed to the _KeyDown event in the KeyCode parameter. If a menu item has been selected than the Menu ID of that menu item is passed as a negative number in the KeyCode parameter.

The Cursor Enters a Control:

Once the cursor moves over a control, the _KeyDown event is triggered with the hWnd of the control in the KeyCode parameter, as noted above. You can determine the hWnd of various controls at start-up and then use a Select Case loop to determine the help text to display.

Alternatively, MWATCH also exports a couple of useful utility functions to assist you: HWndCtlName and HWndTag. HWndCtlName$ will return the name of a control, given its hWnd and HWndTag will return the text in the .Tag property of the passed hWnd. Using this technique, you can either use a Select Case loop through the CtlName's to determine the help text or, if you don't need the .Tag property for something else, the easiest thing is to put your help text in the .Tag property and simply use HWndTag to retrieve this text at run-time. Please see the sample app MENTER.MAK for demonstrates of these functions in context.

The Cursor Enters a Menu:

Once the cursor highlights a menu item (or it is triggered with the keyboard), the Form_KeyDown event is triggered with the MenuId in the KeyCode parameter. Once again, you can check these once when you are finished designing your menu because they will be the same (except for the top-level menu heading) every time the app is run. You could then set up a Select Case statement to check KeyCode against various MenuIds.

Alternatively, MWATCH exports a function called: MenuCaption. The MenuCaption$ function takes a MenuId as a parameter and returns the caption of the menu item. See MENTER.MAK for an example of this function in use. PLEASE NOTE: the MenuCaption function was written specifically for this DLL to save you calling a bunch of Windows API functions and it WILL NOT work outside of your Form_KeyDown handler! The HWndCtlName and HWndTag functions can be used in other contexts but this is NOT THE CASE with the MenuCaption function.

Also, at this point, MWATCH doesn't generate an event when the user is in the system (or

control) menu. I was just lazy. If a lot of people request it, I can add it.

A PLAY Command for Visual Basic

Those of you familiar with the PLAY statement already know how to use it. However, it is missing from VB. WINPLAY.DLL provides this function in a DLL that can be called easily from VB.

Function Declaration:

Declare Sub Play Lib "WINPLAY.DLL" (ByVal Song$)

For those of you who are not familiar with the syntax of the PLAY statement, here is a table of the things you can use when constructing this string:

Table 2 PLAY Commands

Command	Action
A-Gnn	Play that note for that length.
#,+	Sharp the immediately previous note.
-	Flat the immediately previous note.
.	Dot the immediately previous note (3/2 length).
>	Up one octave.
<	Down one octave.
Lnn	Set default length for notes that follow.Default is L4.
MB	Music in background.
MF	Music in foreground (default).
ML	Music Legato (full length).
MN	Music Normal (7/8 length).
MS	Music Staccato (3/4 length).
Nnn	Play this numeric note (1-84).
On	Set Octave. (0 - 6).Default is O4, where C is an octave above middle C. Pitches in an octave begin at C and work upwards to B.
Pnn	Pause for that duration.
Tnnn	Set tempo (32-255). The number of quarter notes in one minute. Default is T120.

Details:

This function is not entirely self-contained as it is in QuickPak and VBTOOLS. The reason was to provide complete flexibility, in terms of the buffer used by Windows to hold the notes. When playing notes in the background, a bigger buffer allows more notes to be stored and the function returns to the calling program *much* faster.

Therefore, you need to call a couple of Windows API functions to actually make this DLL work. Of course, you could make your own PLAY function in VB which encapsulates all of this.

Here is a summary of the Windows API functions that you will need to use:

Table 1. Sound Functions in the Windows API

OpenSound	Opens the sound driver for exclusive use.
CloseSound	Closes the sound driver, making it available.
SetVoiceQueue	Sets the size of the voice queue.

Hopefully the PLAYDEMO.MAK sample will be sufficient to explain the use of each of these functions. One caveat that you should keep in mind, however, is that once you have opened the sound queue with OpenSound it is not available to other programs until you issue a closesound. For this reason, it is a good idea to call CloseSound as soon as possible after you are done making a particular sound.

Source Code:

The source code for WINPLAY.DLL is in Turbo Pascal for Windows.

Musical MsgBox and Beep Commands

Under Windows 3.1 we have seen that the wType parameter of the MessageBeep statement is to play a particular .WAV file and that these same constants correspond to the ones used to put icons in message boxes. For those without Windows 3.1, or without Sound cards, I have written these functions to provide similar functionality.

Function Summaries:

MusicBeep Same as MessageBeep but plays a sound.
MusicBox Same as MessageBox but plays a sound.

Function Declarations:

```
Declare Sub MusicBeep Lib "msgmusic.dll" (ByVal wType)
Declare Function MusicBox Lib "msgmusic.dll" (ByVal hWnd, ByVal Text$, ByVal Caption$,
ByVal wType)
```

The wType parameter in both cases corresponds to the values used with a MessageBox (MsgBox in VB). .They are the following:

```
Const MB_ICONSTOP = 16              ' Critical message
Const MB_ICONQUESTION = 32          ' Warning query
Const MB_ICONEXCLAMATION = 48       ' Warning message
Const MB_ICONINFORMATION = 64       ' Information message
```

which you "Or" with other constants to specify which buttons should be displayed in the message box and which button should be the default.

The DLL functions work in an identical fashion except that they produce a sound which roughly corresponds to the icon that appears in the message box. These are the constants and the sounds that they produce with these functions:

MB_NOICON (&H0)	Doorbell
MB_ICONSTOP (&H10)	British 2-tone siren
MB_ICONQUESTION (&H20)	"Huh?!"
MB_ICONEXCLAMATION (&H30)	"Oh, oh!"
MB_ICONINFORMATION (&H40)	Phone

Differences with VB:

The BEEP statement in VB actually calls the MessageBeep function in the Windows API and passes a 0 as the wType. You will need to use the declaration as above when calling MusicBeep instead. This can be useful if you want to associate these sounds with something other than a message box, perhaps your own VB form. I also use them a lot in debugging code. Please note that none of these constants actually produces the boring beep. You will have to call out to BEEP to accomplish that.

The MessageBox statement in the Windows API has an additional parameter at the

beginning which is the hWnd of the parent window that is to own the MessageBox. This can be set to the hWnd of the current form or even to zero. It is in this function to maintain compatibility with the MessageBox statement in the API. You could construct a function in VB which is similar to the MsgBox function in the following way:

```
Function MusicMsgBox (Msg$, Type%, Title$)
        MusicMsgBox = MusicBox (Screen.ActiveForm.hWnd, Msg$, Title$, Type%)
End Function
```

Please note that none of these parameters are optional as they are in VB.

As you can see, it's possible to get some pretty nice sound effects out of Windows without using any multimedia functions.

Source Code:
The course code for MSGMUSIC was written in QuickC for Windows

Creating Windows 3.1 Screen Savers

A great many people have asked about how to create a Windows 3.1 screen saver in VB. It's actually not so tough. There are only a few things you need to know.

Command Line Parameters

First, screen savers are really just EXEs that have been renamed. Second, when these EXEs are run, they are called with command line arguments based on what operation is needed. If the screen is to be saved, the "-s" parameter is passed on the command line (retrievable via COMMAND$). If the screen saver is to be configured, a "-c" is passed on the command line. Therefore, your "Startup Form" should be Sub Main in a module rather than a form. From within this routine, you check Command$ and .Show either the save form or the config form.

The Configuration Form

The config form should have some way of saving configuration info like a private INI file (I don't believe in adding things to the WIN.INI that don't represent system information but that's just me).

The Screen Save Form

The "save" form should be created without the title bar, the control box, without a border and without the Min/Max buttons. When launched, it should maximize to take up the whole screen. You can then draw all over it in any way you wish. The _KeyDown, _MouseMove and _MouseDown events should all have the END statement in them.

Showing up in Control Panel

The final tidbit is that you need to specially name the EXE in the "Make EXE" dialog. The name must be in the form

SCRNSAVE SaverName

where SaverName is the name you want to have appear in the control panel. After you create the EXE, you rename it to .SCR and copy it to the Windows directory and you're done.

Using the Windows Screen Saver Password

You will notice that the screen savers that come with Windows give you the option of using a password for re-entry into the desktop after the screen save is finished. This process is provided to C developers via a static link library SCRNSAVE.LIB. Accordingly, since this library is useless to VB programmers, I have written SSAVE.DLL to provide the password dialogs and functionality. There are two functions in this DLL. One is GetSSPassword which you call when you are about to end the screen saving process. This will put up a dialog box, prompting the user for a password. The function will return True if the correct password was entered, otherwise it will return False. The second function is SetSSPassword which is a subroutine which will put up a dialog box for the user to change the screen save password. There is no return from this as this activity is independent of your application.

Building a Screen Saver

I have tried to provide a generalized module for creating screen savers called SCRNSAVE.BAS. You need to add this to your project and take the following steps:

1. Create a config dialog and call it ConfigForm. Note that this is the "name" of the form, not the file name.

2. Create a saver form (no caption, control box, min/max buttons and maximized) and call it SaveForm.

3. In the _KeyDown, _MouseDown and _MouseMove event handles of SaveForm, you need to make a call to **TryToGo**. This will pop up the dialog to retrieve the password, if there is one and end the program if there isn't.

4. Set the startup form of your application to Sub Main.

SCRNSAVE.BAS will handle the command line parameters and launch the appropriate form of the screen saver. This module also disables screen saving when you are running so that multiple instances will not be called and restores the screen saver functionality when you call TryToGo.

The *VBZ* Utility Library

Every so often, there are functions that become necessary to throw in a DLL. Many of you have requested that we stop creating one-function DLLs and this makes sense. The result of this request is VBZUTIL.DLL which is designed to contain all of these little utility functions that don't quite belong anywhere else. Don't mistake these for trivial functions because if they were, they wouldn't have to be in a DLL. In fact, it is in this library that some of the most important and creative functions will be found.

What goes into this DLL is guided both by our discoveries and needs and requests from you. All that we ask is that you don't request standard functions that can be found in any commercial add-on package. In other words, don't look for a routine to sort integer arrays any time soon. Other than that, the sky's the limit. You never know what's possible unless you ask and we can never anticipate all of your needs. The functions in this library will often be used in context in the samples programs that come with *VBZ* but there will be no samples devoted to these functions. If you would like this to be otherwise, let us know.

The function reference for VBZUTIL.DLL will always be updated instead of being fragmented over every issue. This is the one example where you can get the latest issue and know that you are not missing anything which is not the case for the DLLs and samples in the features.

VBZUTIL Function Reference

The declarations can be found in VBZUTIL.BAS.

CtlName

Returns the name of a control as a string.

Prototype

```
Declare Function CtlName$ Lib "VBZUTIL.DLL" (C As Control)
```

Usage

```
tbName$ = CtlName$ (Text1)
```

Comments

This function exists because of the unavailability of the .Name property of a control at runtime. You use it in much the same way you would a control, however.

CtlUBound & CtlLBound

Returns the upper and lower bounds of a control array.

Prototypes

```
Declare Function CtlUBound Lib "VBZUTIL.DLL" (C As Control)
Declare Function CtlLBound Lib "VBZUTIL.DLL" (C As Control)
```

Usage

```
UpperBound% = CtlUBound (MyControl (0))
LowerBound% = CtlLBound (MyControl (7))
```

Comments

Don't pass a control to either of these functions unless you know it to be a member of a control array. *Don't* pass the control array with an open paren. You must pass a valid control to the function but it can be any element in the array.

GetHInstance

Returns the instance handle of your current application.

Prototype

```
Declare Function GetHInstance Lib "VBZUTIL.DLL" ()
```

Usage

```
myhInstance% = GetHInstance()
```

Comments

This value is necessary for a number of Windows API functions so it is provided here.

HWnd2Ctl

Returns the element into the VB2 .Controls() collection of a form, given the control's hWnd.

Prototype:

```
Declare Function HWnd2Ctl% Lib "VBZUTIL.DLL" (ByVal hWnd%)
```

Usage:

```
Dim C As Control
'you got the Wnd% from someplace, like MWATCH
El = HWnd2Ctl% (Wnd%)
If El > -1 then Set C = Me.Controls(El)
```

Comments:

At this point, you may use C as if it were a normal control and access all of its properties directly. This function is very useful in applications when you have an hWnd but you need access to the underlying properties of the control.

HIWORD & LOWORD

Returns the HiWord or LoWord of a LONG integer.

Prototypes

```
Declare Function HIWORD Lib "VBZUTIL.DLL" (ByVal LongVal&)
Declare Function LOWORD Lib "VBZUTIL.DLL" (ByVal LongVal&)
```

Usage

```
HW% = HIWORD (MyLong&)
LW% = LOWORD (MyLong&)
```

Comments

Often a DLL function will return a long integer which contains two short integers. These functions will help you to parse out those values.

LPSTR2Str

Creates a Visual Basic string from a C null terminated string (LPSTR)

Prototype

```
Declare Function LPSTR2Str$ Lib "VBZUTIL.DLL" (ByVal LPSTR&)
```

Usage

```
MyString$ = LPSTR2Str$ (lpstr&)
```

Comments

Sometimes a DLL function will return a C language string and you will want to get this into a VB string.

LP2Str

Creates a VB language string from any block of memory.

Prototype

```
Declare Function LP2Str$ Lib "VBZUTIL.DLL" (lp As Any, ByVal nBytes)
```

Usage

```
MyString$ = LP2Str$ (ByVal lp, 283)
MKI$ = LP2Str$ (MyInt, 2)
```

MAKELONG

Create a LONG integer from two short integers.

Prototype

```
Declare Function MAKELONG Lib "VBZUTIL.DLL" (ByVal loword%, ByVal hiword%) As Long
```

Usage

```
MyLong& = MAKELONG (Var1%, Var2%)
```

Comments

Often a DLL function requires a LONG integer parameter which is really two short integers. This function will allow you to call those functions. It is identical to the MAKELONG macro which is listed in the SDK documentation but that is unavailable in a Windows DLL.

ReCreateControlhWnd

Recreates the hWnd associated with a particular control.

Prototype
Declare Sub ReCreateControlhWnd Lib "VBZUTIL.DLL" (C As Control)

Usage
This sample changes a single-column, single-select list box to multi-column, multi-select. While this is no longer necessary with a standard list in VB2 (though File Lists still can't be multi-column for some reason), it demonstrates the use of the DLL.

```
OldLong& = GetWindowLong&(File1.hWnd, GWL_STYLE)
OldLong& = OldLong& Or LBS_EXTENDEDSEL Or LBS_MULTICOLUMN
SetWindowLong File1.hWnd, GWL_STYLE, OldLong&
RecreateControlHwnd File1
File1.Refresh
```

You can also look at the D&DTEST1.MAK sample that came with *VBZ*01.

Comments
This function is useful when you need to change a pre-hwnd style of a control dynamically or when you need to gain access to a style that has not been exposed by VB, such as justification in a text box. This is an advanced function. Use it with care!

Str2Ctl
Returns the element into the VB2 .Controls() collection of a form, given the control name as a string.

Prototype:
Declare Function Str2Ctl% Lib "VBZUTIL.DLL" (Frm As Form, CtlName$)

Usage:

```
Dim C As Control
CtlName$ = "Text1"
El = Str2Ctl% (Me, CtlName$)
If El > -1 then Set C = Me.Controls(El)
```

Comments:
At this point, you may use C as if it were a normal control and access all of its properties directly. This function is very useful in database applications when you want to be able to relate control names to corresponding column names in a database table.

USHORT
Returns the unsigned version of an integer into a long integer.

Prototype:
Declare Function USHORT& Lib "VBZUTIL.DLL" (ByVal Word)

Usage:
Unsigned& = USHORT& (MyWord%)

Comments:

VB lacks the unsigned integer data type (often called a WORD). This function will allow you to calculate the actual value from an integer, even one which has moved into negative numbers.

Recent Press Releases

This space will contain press releases for recent VB-related products. If you have a press release that you would like included in this section, please send it via email to CIS:76702.1605 or on disk to:

User Friendly, Inc.
1718 M Street, N.W.
Suite 291
Washington, DC 20036

We will not transcribe press releases from hard copy, so you must get it to us in binary form. Thanks a lot!

What's Coming Up?

While we have a lot already in the works, the content of *VBZ* is largely up to you, the reader, to dictate. If you are having a problem, send it in. If you want more of a particular type of article beginner, advanced, more DLLs, more VB code let us know and we'll do our best to accommodate your needs and wants. Here's our list thus far:

VB Developers Utilities
Button Bitmap Builder
Palette Builder (for PicClip among other things)
Stub Replacement
Object Manager
Code Generators
. . . and much more!

Visual Basic Techniques
Making VB Applications Behave
Calling DLL functions
Advanced Printing
Metafile Creation
Simplified Hotkeys
An Improved FindWindow Function
Waiting for other Apps to Execute
. . . and much more!

Dynamic Link Libraries
Custom Cursors
Tile/Cascade
SendKeys function for DOS Applications
Replacement for missing QB Functions (QBFUNC.DLL)
Additional VBZUTIL functions
. . . and much more!

Custom Controls
Generic Subclassing Control
Clipboard Viewer Control
Huge Scrollbars
. . . and many other specialized controls!

Improved Help Files
Binary Sample Extraction
More links

The rest is up to you! Be sure to let us know which of these things are of greatest interest to you so that we can bump them to the top of the list to complete.

What's Gone Before?

This section will be devoted to a summary of the features in previous issues. As there are no previous issues at this time, there's nothing here but this boring explanation. Clicking on topics listed in this space will actually launch the appropriate issue of *VBZ* and jump right to the article of interest.

Letters

No letters in this issue! Write soon and write often. We want to hear your suggestions on how to improve the journal, topics you want covered, solutions you have come up with, mistakes we have made or anything else you can come up with. We look forward to hearing from you and making *VBZ* the best it can be for you! Some possible suggestions might be:

More standardized look and feel for the journal
Shorter/longer articles
Better explanation of the workings of the DLLs/controls
More/fewer add-on reviews
Inclusion of the sample code in the VBZ??.HLP file

Address letters to:

VBZ Response
User Friendly, Inc.
1718 M Street, N.W.
Suite 291
Washington, DC 20036

or fax them to:

(202) 785-3607
Attn: *VBZ* Response

or email them to Jonathan Zuck on Compuserve:
76702,1605

Fixes and Updates

This section is reserved for the inevitable fixes and upgrades that will be necessary to the samples, custom controls and DLLs that will be posted. Included will be a new version number (as is in the version resource), updated files, and a list of things that have changed. This will not be complete documentation for the files so this section will be primarily for those of you with the original issue.