VBZ

The Electronic Journal on Visual Basic

Copyright 1993 User Friendly, Inc.
Issue 02: March/April 1993

Welcome to the second issue of VBZ, the electronic journal on Visual Basic. We hope that you will find something in this diverse set of articles to interest you. This is the first issue with work from authors other than me (check out the reviews section). Also new to this issue when you hit F1, while in a demo, the appropriate section of the journal, describing the technique, will be activated. Let us know if this helps those of you who like to try before you read. The next issue will have even more, including a new regular column. Please feel free to contact us with your suggestions for ways to improve the journal. Our goal is to make this the premier resource to Visual Basic programmers. Enjoy the issue and keep in touch!

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

Table of Contents

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. An issue of VBZ will come out every two months unless there is great 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
License Information
Using VBZ DLLs
Writing for VBZ

Subscribing to VBZ

If you are not currently a subscriber, you should be. To subscribe to VBZ, send check, money order or purchase order for $69 + $4 s/h for the first subscription, $49 + $4 s/h 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: 76702, 1605

Please specify whether you would like to receive VBZ via email, in which case include your Compuserve account number, or US Post, in which case please include your preference as to media size.

License Information

Disclaimer

Using VBZ DLLs

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 users 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 so that you can simply 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 the 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

Apps that Tile and Cascade

You have probably noticed that when you select either Tile or Cascade from the Task Manager utility, you Visual Basic applications do not comply. For this reason, I have written BEHAVE.DLL to help accomplish this task.

Why Things Don't Work Now
Using BEHAVE.DLL
How It Works

Apps that Tile and Cascade: Why Things Don't Work Now

As you are aware, your VB apps are made up of a series of modeless dialogs, with nothing owning anything else. This is very unusual for a Windows application (and we'll be addressing this in the next article) and results in non-standard behavior. The whole VB application is managed by an "owner" window which is invisible and has no height or width. Consequently, it is *this* window that receives the command to tile and cascade but it ignores it. Task Manager doesn't realize it's being ignored and leaves a hole. You can find the hWnd of this owner window with the GetWindow API function.

```
DefInt A-Z
Declare Function GetWindow Lib "User" (ByVal hwnd, ByVal fuRel)

Const GW_OWNER = 4

Sub Form_Load ()
    hOwner = GetWindow(Me.hWnd, GW_OWNER)
End Sub
```

Once you have this, you can do a number of interesting things. First, if your application is always minimized, you could minimize this window. The purpose of that would be to simply keep it out of the way of the tiling and cascading window. If you don't, this window will still create a hole on the desktop.

Further, if you subclass this window, trapping the WM_WINDOWPOSCHANGING message, you can force another of your forms to emulate the required behavior. If you have a generic subclass control, such as the one in SpyWorks or the one coming up in VBX, you could subclass the control that way. For those of you without such tools, I have written BEHAVE.DLL to do the job.

<u>Apps that Tile and Cascade: Using BEHAVE.DLL</u>

Included with this issue is the BEHAVDEM.MAK sample which demonstrates the simple use of this DLL. There are two functions in the DLL: RegisterMainForm and UnRegisterMainForm. To each you simply pass the hWnd of the form that you want to act as the main form of your application. In most cases, you will call the RegisterMainForm in the Form_Load event handler and call the UnRegisterMainForm in the Form_Unload handler. Be advised that you shouldn't arbitrarily call "End" in VB to close your whole application as this is very unfriendly to DLLs of this type. Better to simply unload all forms to close your application. At the very least, you should unregister the main form before calling the end statement if you must call it.

Apps that Tile and Cascade: How It Works

BEHAVE.DLL simply maintains an array of hWnds and their owners which are found using the technique above. It subclasses the owners so that when one of them receives the WM_WINDOWPOSCHANGING message, it knows to redirect both the coordinates and the sizing information to the "Main form" using the MoveWindow API function. The Pascal source code for the DLL can be found in BEHAVE.PAS.

Creating Modeless Dialogs in VB

It is possible to create a modal dialog in VB, by simply specifying modal when using the .Show method of a form. This is very useful for dialogs that query the user for information and stop the normal processing of an application, to wait for an answer. There is another type of dialog, called a modeless dialog, which remains present while allowing the user access to the underlying application. Examples of modeless dialogs are the floating toolbars that have become so popular and the search/replace dialog, found in WRITE and elsewhere.

Creating Modeless Dialogs: What Are Modeless Dialogs?

What are the characteristics of these dialogs? Well, first of all, they always stay on top of the main form, like a modal dialog but they don't prevent access to the underlying form. At first glance, it might seem that the new .ZOrder method is the way to go, but user actions override these settings. If the user clicks on the bottom form, it will come to the top of the dialog. No good and not standard. Another popular approach is to use the SetWindowPos API function to set the ZOrder to (-1), causing a particular window to *always* be on top. This works fine except when the user tries to activate another application. Our dialog box shows up on top of the *other* application as well! Again, this is a dead give away that this is a non-standard, and unfriendly application. What we need is a form which will stay on top of *our* application but allow itself to be covered up by other applications, should they be come active.

Creating Modeless Dialogs: So What *Can* We Do?!

It turns out that the solution here is quite simple and involves using just one API function, SetWindowWord. Because all VB forms are WS_POPUP-style windows, they know to stay in front of their parent. The only problem is that they don't have a parent. So, if we give a modeless dialog a parent, it will do the rest of the work for us. The MODELESS.MAK sample demonstrates the use of this technique.

In the Form_Load event of the child form, you need to simply use SetWindowWord with the undocumented SWW_HPARENT constant to dynamically change the parent of the form. This is a little unorthodox but the only solution available to us in VB. It is extremely important to undo this change before leaving the application. Therefore, you should unload the dialog form before the parent gets fully unloaded. It's okay to use the unload statement on a form that isn't loaded because no error is generated.

Accessing the Common Color Dialog

Beginning with Windows 3.1, Microsoft shipped a DLL of common dialogs for applications to use for a greater level of standardization in various components of applications. These dialogs include those for file handling, color selection, font selection and printer selection and setup. Visual Basic Professional includes a fairly easy-to-use custom control with which you can access these dialogs but you may not have the Pro edition of VB (you should, however). Further, there may be times when you don't want the overhead of loading a form in order to gain access to a custom control. For that reason, as it comes up, I will be providing "wrapper" functions for the common dialogs in the form of reusable modules that you can simply add to your projects to use directly. These modules will be cumulative and used in the utilities I provide in order to simplify the distribution process. (Note: If you would like to see more examples using tools available elsewhere, such as in the professional edition, let us know.) This article briefly describes the color selection dialog which will be used in the utility in the next article.

Each of the common dialog functions in COMMDLG.DLL requires a single parameter which is a structure which contains the elements of the call. In the case of the common color dialog, the structure is defined as follows:

```
Type CHOOSECOLORSTRUC
    lStructSize As Long
    hWndOwner As Integer
    hInstance As Integer
    rgbResult As Long
    lpCustColors As Long    'Long pointer to array of LONGs
    Flags As Long
    lCustData As Long
    lpfnHook As Long
    lpTemplateName As Long
End Type
```

So, all we have to do is populate this structure, as we can call the function. The last three elements of the structure are optional and relate to customizing the dialog, which we cannot do without a callback function. The only element that's going to give us trouble is the lpCustColors element which is a pointer to an array of LONGs in which we store our custom colors. In VB we don't have a way to retrieve the address of a variable so we will use the SSegAdd function in QBFUNC.DLL, documented below.

The CLRDLG.BAS module defines the function: GetColors. This function is prototyped as follows:

```
Function GetColors (Clr&, Custom)
```

Where Clr& is the address of a LONG into which to place the selected color. The second parameter, Custom, is a boolean which specifies whether or not to enable custom color selection.

The CLRDLG.MAK sample program demonstrates the use of the common color dialog. In this case, the array of custom colors is maintained by the GetColors function as it is assumed that this array will be constant throughout the application. There may be instances where you want to store this yourself because you want to store the custom colors. In this case it would be a simple matter to add a parameter to the GetColors function which is an arrays of longs.

Pop Up Color Selection

I have two pet peeves when it comes to color selection in Visual Basic. First of all, when selecting a color property from the property bar, I can't select a custom color. What's up with that? Another problem is that when setting colors programmatically, there *is* no color palette to choose from. You are reduced to using constants, the spelling of which you have forgotten, or using QBColor to get back to colors you've already used. Wouldn't it be nice if there was a popup color palette that could be used inside the editor? Of course it would; and POPCLR.EXE is the answer!

Pop Up Color Selection: The POPCLR.EXE Solution

POPCLR.EXE is a pop-up program, much like a TSR under DOS, which will come up with a hotkey. When you select Ctrl-C, the common color dialog comes up, allowing you to select a color (of course!). When you select "OK," hex color value is blasted into the current application using SendKeys. Whatever control has focus will get the color value. So, if you are in the property bar or in the VB code editor you can use this utility. To close the utility, simply use Task Manager to "End Task."

Pop Up Color Selection: How POPCLR.EXE Works

Well, I had a couple of goals when creating this utility. First, I wanted it to be as simple as possible. Loading an invisible form to manage things seemed unnecessarily complex. However, to make use of the HOTKEY.DLL and the COMMDLG.VBX from VB Pro would require this. So, I needed solutions to both the hotkey and color dialog problems that didn't require a VB form. Using our CLRDLG.BAS module we solve the latter problem because we are calling the color dialog routine directly. The second is a little trickier.

<u>**Pop Up Color Selection: Simple Hot-Keys**</u>

Whenever you have a lot of applications, or need to trap a number of hotkeys within the same application, HOTKEY.DLL is certainly the way to go. It is also the solution which causes the least amount of drag on the system. However, in special cases, you might want to trap just one hotkey and you don't want to distribute an additional DLL. How would we do that? Well, it turns out that it isn't too complicated. The GetAsyncKeyState API function will tell us if a key is down. Using this function in a loop, we can tell when a key gets pressed. POPCLR.MAK is the project that contains the code we need to look at. In POPCLR.BAS, we have a Sub Main (you will notice there are no forms in this project) in which we enter an infinite DoEvents loop, constantly checking to see if the "C" key is depressed (and what we can do to cheer it up). As soon as we discover that it is down, we check to see if the Ctrl key is also down. If it is, we know that the user has pressed Ctrl-C and we pop up the common color dialog. If you wanted to be really careful, you might also check to see if any other shift keys were down to make sure that the user wasn't pressing Ctrl-Shift-Alt-C or something like that, but we don't care about that for our purposes cause we are writing a utility for our own use.

Finally, if the user selected "OK" from the color dialog, we put the hex value of the color into the keyboard buffer with SendKeys. What could be simpler. It's a pretty small utility, but demonstrates a lot and I use it all the time now that I have it. I hope you will too.

Dropping Leftover DLLs

If you do any experimenting with DLLs, you have probably faced the problem that when (I mean if) your program crashes, the DLLs are not unloaded from memory. This can eat up memory if it's not your DLL. If it is your DLL, you need to exit Windows in order to make a change to the DLL and try it again. If you don't the old copy simply gets reused, which isn't too useful. It turns out that TOOLHELP.DLL provides the functions you need to determine which DLLs are loaded into memory. Therefore, it's possible to write a little developer's utility which lists them and allows you to unload the ones you want. Hence ZAPLIB.EXE.

Using ZAPLIB
How ZAPLIB Works

Dropping Leftover DLLs: Using ZAPLIB

To use ZAPLIB, you simply restore it from its usual state as an icon. At this point, it lists the loaded DLLs in a multi-select list box. Accordingly, you can select as many of these as you like and click "OK." At this point, ZAPLIB will unload these DLLs and re-minimize itself. ZAPLIB filters out the DLLs that it thinks are system DLLs to reduce clutter and temptation. Since you have the source, you can add to this list. There might still be DLLs loaded, the names of which you don't recognize. If in doubt, don't unload. Windows gets very unhappy if you unload a DLL that is in use by another program but you won't hear about it until the program try to call the DLL.

Dropping Leftover DLLs: How ZAPLIB Works

ZAPLIB uses the ModuleFirst and ModuleNext functions, found in TOOLHELP.DLL to enumerate all of the active modules. The declarations for these functions and the MODULEENTRY structure are in the ZAPLIB.BAS module. The key code of ZAPLIB is in the Form_Resize event of the ZAPLIB.FRM source file. This code steps through the module list, getting the names of all the active modules. You will notice that you need to trim the trailing null and blanks from the populated elements of the MODULEENTRY structure.

For each entry, ZAPLIB checks the extension for DLL or VBX. There is a more sophisticated method of determining whether the module is a DLL or EXE but this method will work for our purposes and doesn't require a custom DLL function.   Next ZAPLIB checks to see if the module name falls in a list of "reserved" module names. These are system DLLs that you really shouldn't unload. If you find something on your system that shows up all the time that you will never unload, simply add its module name to the Reserved$ string. The .ItemData property of the list box is used to store the handle of the module if it gets listed. It is this handle which will be used with the FreeLibrary and GetModuleUsage API functions to unload the DLL.

In the Command1_Click event, we simply walk the list box, looking for selected items. For any that are selected, we call FreeLibrary with the hModule until GetModuleUsage returns zero. At this point, the DLL is unloaded. When the end of the list is reached, the application is minimized. That's all there is to it.

The QuickBasic Function Library

There are a number of useful functions in QuickBasic that were left out of VB. For some it seems like pure oversight. For others, it appears as though Microsoft was attempting to protect us from ourselves. Accordingly, it behooves me to say that you use these functions AT YOUR OWN RISK. I'm simply supplying you with the functionality you want but I'm not taking responsibility for your use of these functions. I'll be using these functions on and off in various issues of the journal. This issue simply uses the SSegAdd function which you will find yourself using all the time. What follows is a list of functions in the QBFUNC.DLL and their purpose. QBFUNC.BAS contains the declarations for all of these for use in your own applications. The Pascal source code for the DLL can be found in QBFUNC.PAS. This library isn't complete. If there is something you would like to see in there, let me know. For more details about using these functions, I would strongly recommend Ethan Winer's book on DOS BASIC from Ziff-Davis Press.

BLOAD, BSAVE

These functions are used for two purposes in QB. One is to save screens and the other is to save arrays quickly to disk. The first purpose has no meaning under Windows but the second does and you can use these functions to save your arrays in a compatible format across versions of BASIC. To quickly save an array to disk, simply pass the filename, and the first element of the array (by reference), and the number of bytes to the BSAVE function.   To use the BLOAD function, you need to redimension your array to the correct size by reading storing this information, by file size calculation (using LOF) or by reading the header. See Winer's book for details.

CVD, CVI, CVL

These functions translate the string (or memory) versions of doubles, integers and longs into their equivalent numeric values. You pass the string (retrieve from disk, or created with the MK? functions) to these functions and they return the numeric equivalents.

InterruptX

This is a function which allows you to make interrupt calls from within VB. This is often necessary to call DOS or machine functions that are not part of the API. Network functions are often called in this way as well. We will be using this function in future issues of the journal, but its use is specialized and I would really recommend getting Ethan's book for a good description of its use.

MKD, MKI, MKL

These are the opposite of the CV? functions and return the string equivalents of numeric variables.

PEEK

This function will allow you to determine the value of any byte in memory. You simply pass a long pointer to the memory address and the function returns the integer value of that byte.

POKE

This function will allow you to set a byte of memory (that you own). You pass a long pointer to the memory location and the integer value of the byte to set.

Note: These functions are not very efficient for large chunks of memory. For those, you might want to use the hMemCpy API function.

SSEGADD

This function will return a long pointer to any variable. The function requires you to *pass* a long pointer to a variable. This is done by simply passing the variable by reference in every case except for a string, in which case you need to use the ByVal keyword to call the function.

SWAP, SWAP4, SWAP8

Unfortunately, it isn't possible to provide a single SWAP function that works with all variable types. Instead you use SWAP for variable length strings and integers, the SWAP4 function for LONGs and SINGLEs and finally the SWAP8 function for DOUBLEs and CURRENCYs. The SWAP4 and SWAP8 routines aren't really any faster than those you could create in VB alone and are provided solely for convenience. Using SWAP for variable length strings is *much* faster than the equivalent code could be, however, and should be used whenever possible. ALL parameters should be passed by reference, even strings.

VARPTR, VARSEG

These functions return the offset and the segment accordingly of any VB variable. Since you will most often be using long pointers under Windows, these functions are provided mostly for compatibility. Pass strings "ByVal."

VBINP, VBOUT

These are equivalent to the QB INP and OUT statements and used in the same way to retrieve or set information to a port. These are equivalent to the IN and OUT statements in assembler.

The VBZ Utility Library

Every so often, there are functions in which it becomes 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 people's 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)
```

<u>Usage</u>

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

<u>Comments</u>

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.

<u>GetHInstance</u>

Returns the instance handle of your current application.

<u>Prototype</u>

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

<u>Usage</u>

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

<u>Comments</u>

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

<u>HWnd2Ctl</u>

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

<u>Prototype</u>:

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

<u>Usage</u>:

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

<u>Comments</u>:

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.

<u>HIWORD & LOWORD</u>

Returns the HiWord or LoWord of a LONG integer.

<u>Prototypes</u>

```
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 an 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 VBZ01.

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, ByVal 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 and 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?

Letters

Actually, I got about 20 letters hailing the advent of this journal, but I forgot to save them. Oops! Costas Kitos wrote in with a suggestion to pop-up the proper section of the help file from the samples so we have done that where appropriate 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
Articles Broken up into sub-topics (good for browsing, bad for printing)
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

The only fix this issue is in the VBZUTIL.BAS support module. The second parameter of the Str2Ctl function was mis-declared. The CtlName$ parameter needs to be passed by value.

Reviews

Doc-To-Help
3D-Widgets

DOC-To-Help (by David S. Mohler 76450,1642)

WHO:	WHAT: DOC-To-Help
WexTech Systems, Inc.	
60 East 42nd Street, Suite 1733	HOW MUCH: $249.00
New York, NY 10165	
Phone: (212) 949-9595	
Fax: (212) 949-4007	

My reaction to and opinion of Doc-to-Help are very, very positive. The complete cross-referencing process is unparalleled in other packages I have investigated. And, as the name implies, the complete hard copy documentation is generated from the same text as the help file, killing two birds with one stone.

The text generation is done completely in WinWord; Doc-to-Help is built out of WordBasic macros that automate the formatting, bitmaps, cross referencing, indices, contents, glossaries and key-word searches. All text entered appears identical in both the hard copy and on-line docs by default. However, you can mark portions of text that is intended for hard copy only or for help file only.

It is somewhat restrictive in the sense that you cannot use Doc-to-Help to automatically insert jumps to other help files, use bitmaps as jumps, etc., which means you have to do this manually in WinWord. There also needs to be a help-file debugging command that helps to eliminate recursive jumps: that is, jumps which call themselves. This often happens when glossary text contains the word being defined. Doc-to-Help does not smartly eliminate those words in the linking process.

Even manual work in the RTF file is made easier, though, because Doc-to-Help will select 3.0, 3.1 or protect-mode compilers along with allowing you to set graphics compression options to 0, 40% or 50%. It then builds the proper .HPJ file and runs the compiler for you, trapping and generating a listing of any errors.

A minor upgrade to the current version is due out at the end of the month, I believe, which is supposed to include a few enhancements -- hopefully geared more toward the bells and whistles possible with the 3.1 help system.

It is so usable and complete that I would estimate 10-15% of my work now revolves around making help docs for other developers who only occasionally require full blown manuals and on-line help.

3D-Widgets (by Thomas Wagner 73530,1002)

WHO:
Sheridan Software Systems, Inc.
65 Maxess Road
Melville, NY 11747
Phone: (516) 753-0985
Fax: (516) 753-3661

WHAT: 3D-Widgets

HOW MUCH:
Widgets / 1 $49
Widgets / 2 and Widgets / 3 $39 each
Combination of all three $99

There has been much talk in recent times about the changing look of user interfaces, especially in applications that have been created with Visual Basic. The 3-D look - as it is called - has become more and more important, not so much from a program functionality standpoint but from an aesthetic perspective.

To the best of my knowledge a lot of what we see today in the area of 3-D user interface design was first introduced by Steven Jobs in the form of the "burnished gray steel" look of the NEXT computer interface, as well as certain UNIX applications. This grayscale three dimensional look was so slick and different from everything else out at the time, it quickly became a favorite. Interestingly, now that NEXT has ceased operations as a hardware manufacturer, one of its remaining contributions is the 3-D look .

Not surprising, once VB hit the market people began using its inherent graphic abilities to recreate this look. At first by utilizing a work-around or programmatic solutions, and later by utilizing specific add-on libraries. For example the Waite Group's VB "How to " book contains among its many tips a way of recreating the NEXT look on VB forms.

Today, the most comprehensive aid in designing 3-D interfaces is being produced by Sheridan Software. The three component package called "Widget's" lets a programmer create all the raised panels, sunken check boxes and multiple 3-D list boxes that his heart desires. Even three dimensional menus are possible.

Each component is sold separately, thus allowing for a maximum of flexibility when choosing a set of custom controls. Widgets 1 contains the VBX's necessary to create 3-D check boxes, command buttons, Frames, Option Buttons, Panel's and Ribbon Buttons. Of the three sets of Widget's, this is probably the one utilized most often. For example, with the 3-D panel it becomes very easy to implement a status bar in your application. The ribbon buttons allow for fast creation of a toolbar. If you have the professional release of VB you are already familiar with this set, it is included with the program.

Widgets 2 contains 3-D versions of List Boxes, Combo Boxes, as well as File-, Directory-, and Drive List Boxes. And lastly, Widgets 3 contains Enhanced Menus. All of these components give the programmer an unprecedented amount of control over the look of the application's interface.

At this point a question comes to mind: Why buy it? For starters, Microsoft itself will be releasing more and more software incorporating the 3-D look. For example, Excel 4.0 for Windows makes extensive use of it. Another program that comes to mind is MS Mail, which has a toolbar that looks virtually identical to a ribbon bar you can create with "Widgets".

Granted, just because Microsoft wants to change some of the looks of their applications is not much of a   reason for you to spend your hard earned money. However, the same impetus that contributed to the success of 3-D interfaces in the first place may be enough reason for you to look at these custom controls: People like it!

A good number of people think 3-D looks "high tech", "great" and "different". End-users as well as programmers. That goes for me as well. I remember the first time I changed the bevel width of   a panel control and watched the result.....wow. So, it's exciting for the programmer (at least for some), and oftentimes it's exciting for the end user as well. That can translate to higher productivity for you personally - I spent more time working and less dreaming about Tahiti - and a more marketable end product.

You may have heard it said somewhere that a consistent look is very important in designing an interface. That is true, and the fun part in using "Widget's" is that your applications will look consistently different from a lot of programs out there. That in turn generates interest and excitement, and that's what it's all about.

VBZ Table of Contents

VBZ

The Electronic Journal on Visual Basic
Copyright 1993 User Friendly, Inc.
Issue 02: March/April 1993

Welcome to the second issue of VBZ, the electronic journal on Visual Basic. We hope that you will find something in this diverse set of articles to interest you. This is the first issue with work from authors other than me (check out the reviews section). Also new to this issue when you hit F1, while in a demo, the appropriate section of the journal, describing the technique, will be activated. Let us know if this helps those of you who like to try before you read. The next issue will have even more, including a new regular column. Please feel free to contact us with your suggestions for ways to improve the journal. Our goal is to make this the premier resource to Visual Basic programmers. Enjoy the issue and keep in touch!

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

Table of Contents

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. An issue of VBZ will come out every two months unless there is great 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. To subscribe to VBZ, send check, money order or purchase order for $69 + $4 s/h for the first subscription, $49 + $4 s/h for each additional subscription, to:

VBZ Subscription
User Friendly, Inc.
1718 M Street, N.W.
Suite 291
Washington, DC 20036
(202) 387-1949
(202) 785 - 3607 FAX
CIS: 76702, 1605

Please specify whether you would like to receive VBZ via email, in which case include your Compuserve account number, or US Post, in which case please include your preference as to media size.

License Information

Disclaimer

Using VBZ DLLs

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 users 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 so that you can simply 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 the 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

Apps that Tile and Cascade

You have probably noticed that when you select either Tile or Cascade from the Task Manager utility, you Visual Basic applications do not comply. For this reason, I have written BEHAVE.DLL to help accomplish this task.

Why Things Don't Work Now

As you are aware, your VB apps are made up of a series of modeless dialogs, with nothing owning anything else. This is very unusual for a Windows application (and we'll be addressing this in the next article) and results in non-standard behavior. The whole VB application is managed by an "owner" window which is invisible and has no height or width. Consequently, it is *this* window that receives the command to tile and cascade but it ignores it. Task Manager doesn't realize it's being ignored and leaves a hole. You can find the hWnd of this owner window with the GetWindow API function.

```
DefInt A-Z
Declare Function GetWindow Lib "User" (ByVal hwnd, ByVal fuRel)

Const GW_OWNER = 4

Sub Form_Load ()
    hOwner = GetWindow(Me.hWnd, GW_OWNER)
End Sub
```

Once you have this, you can do a number of interesting things. First, if your application is always minimized, you could minimize this window. The purpose of that would be to simply keep it out of the way of the tiling and cascading window. If you don't, this window will still create a hole on the desktop.

Further, if you subclass this window, trapping the WM_WINDOWPOSCHANGING message, you can force another of your forms to emulate the required behavior. If you have a generic subclass control, such as the one in SpyWorks or the one coming up in VBX, you could subclass the control that way. For those of you without such tools, I have written BEHAVE.DLL to do the job.

Using BEHAVE.DLL

Included with this issue is the BEHAVDEM.MAK sample which demonstrates the simple use of this DLL. There are two functions in the DLL: RegisterMainForm and UnRegisterMainForm. To each you simply pass the hWnd of the form that you want to act as the main form of your application. In most cases, you will call the RegisterMainForm in the Form_Load event handler and call the UnRegisterMainForm in the Form_Unload handler. Be advised that you shouldn't arbitrarily call "End" in VB to close your whole application as this is very unfriendly to DLLs of this type. Better to simply unload all forms to close your application. At the very least, you should unregister the main form before calling the end statement if you must call it.

How It Works

BEHAVE.DLL simply maintains an array of hWnds and their owners which are found using the technique above. It subclasses the owners so that when one of them receives the WM_WINDOWPOSCHANGING message, it knows to redirect both the coordinates and the sizing information to the "Main form" using the MoveWindow API function. The Pascal source code for the DLL can be found in BEHAVE.PAS.

Creating Modeless Dialogs in VB

It is possible to create a modal dialog in VB, by simply specifying modal when using the .Show method of a form. This is very useful for dialogs that query the user for information and stop the normal processing of an application, to wait for an answer. There is another type of dialog, called a modeless dialog, which remains present while allowing the user access to the underlying application. Examples of modeless dialogs are the floating toolbars that have become so popular and the search/replace dialog, found in WRITE and elsewhere.

What are Modeless Dialogs?

What are the characteristics of these dialogs? Well, first of all, they always stay on top of the main form, like a modal dialog but they don't prevent access to the underlying form. At first glance, it might seem that the new .ZOrder method is the way to go, but user actions override these settings. If the user clicks on the bottom form, it will come to the top of the dialog. No good and not standard. Another popular approach is to use the SetWindowPos API function to set the ZOrder to (-1), causing a particular window to *always* be on top. This works fine except when the user tries to activate another application. Our dialog box shows up on top of the *other* application as well! Again, this is a dead give away that this is a non-standard, and unfriendly application. What we need is a form which will stay on top of *our* application but allow itself to be covered up by other applications, should they be come active.

So What *Can* We Do?!

It turns out that the solution here is quite simple and involves using just one API function, SetWindowWord. Because all VB forms are WS_POPUP-style windows, they know to stay in front of their parent. The only problem is that they don't have a parent. So, if we give a modeless dialog a parent, it will do the rest of the work for us. The MODELESS.MAK sample demonstrates the use of this technique.

In the Form_Load event of the child form, you need to simply use SetWindowWord with the undocumented SWW_HPARENT constant to dynamically change the parent of the form. This is a little unorthodox but the only solution available to us in VB. It is extremely important to undo this change before leaving the application. Therefore, you should unload the dialog form before the parent gets fully unloaded. It's okay to use the unload statement on a form that isn't loaded because no error is generated.

Accessing the Common Color Dialog

Beginning with Windows 3.1, Microsoft shipped a DLL of common dialogs for applications to use for a greater level of standardization in various components of applications. These dialogs include those for file handling, color selection, font selection and printer selection and setup. Visual Basic Professional includes a fairly easy-to-use custom control with which you can access these dialogs but you may not have the Pro edition of VB (you should, however). Further, there may be times when you don't want the overhead of loading a form in order to gain access to a custom control. For that reason, as it comes up, I will be providing "wrapper" functions for the common dialogs in the form of reusable modules that you can simply add to your projects to use directly. These modules will be cumulative and used in the utilities I provide in order to simplify the distribution process. (Note: If you would like to see more examples using tools available elsewhere, such as in the professional edition, let us know.) This article briefly describes the color selection dialog which will be used in the utility in the next article.

Each of the common dialog functions in COMMDLG.DLL requires a single parameter which is a structure which contains the elements of the call. In the case of the common color dialog, the structure is defined as follows:

```
Type CHOOSECOLORSTRUC
    lStructSize As Long
    hWndOwner As Integer
    hInstance As Integer
    rgbResult As Long
    lpCustColors As Long   'Long pointer to array of LONGs
    Flags As Long
    lCustData As Long
    lpfnHook As Long
    lpTemplateName As Long
End Type
```

So, all we have to do is populate this structure, as we can call the function. The last three elements of the structure are optional and relate to customizing the dialog, which we cannot do without a callback function. The only element that's going to give us trouble is the lpCustColors element which is a pointer to an array of LONGs in which we store our custom colors. In VB we don't have a way to retrieve the address of a variable so we will use the SSegAdd function in QBFUNC.DLL, documented below.

The CLRDLG.BAS module defines the function: GetColors. This function is prototyped as follows:

```
Function GetColors (Clr&, Custom)
```

Where Clr& is the address of a LONG into which to place the selected color. The second parameter, Custom, is a boolean which specifies whether or not to enable custom color selection. The CLRDLG.MAK sample program demonstrates the use of the common color dialog. In this case, the array of custom colors is maintained by the GetColors function as it is assumed that this array will be constant throughout the application. There may be instances where you want to store this yourself because you want to store the custom colors. In this case it would be a simple matter to add a parameter to the GetColors function which is an arrays of longs.

Pop Up Color Selection

The Problem

I have two pet peeves when it comes to color selection in Visual Basic. First of all, when selecting a color property from the property bar, I can't select a custom color. What's up with that? Another problem is that when setting colors programmatically, there *is* no color palette to choose from. You are reduced to using constants, the spelling of which you have forgotten, or using QBColor to get back to colors you've already used. Wouldn't it be nice if there was a popup color palette that could be used inside the editor? Of course it would; and POPCLR.EXE is the answer!

The Solution (or *one* solution)

POPCLR.EXE is a pop-up program, much like a TSR under DOS, which will come up with a hotkey. When you select Ctrl-C, the common color dialog comes up, allowing you to select a color (of course!). When you select "OK," hex color value is blasted into the current application using SendKeys. Whatever control has focus will get the color value. So, if you are in the property bar or in the VB code editor you can use this utility. To close the utility, simply use Task Manager to "End Task."

How it Works

Well, I had a couple of goals when creating this utility. First, I wanted it to be as simple as possible. Loading an invisible form to manage things seemed unnecessarily complex. However, to make use of the HOTKEY.DLL and the COMMDLG.VBX from VB Pro would require this. So, I needed solutions to both the hotkey and color dialog problems that didn't require a VB form. Using our CLRDLG.BAS module we solve the latter problem because we are calling the color dialog routine directly. The second is a little trickier.

Simple Hot-Keys

Whenever you have a lot of applications, or need to trap a number of hotkeys within the same application, HOTKEY.DLL is certainly the way to go. It is also the solution which causes the least amount of drag on the system. However, in special cases, you might want to trap just one hotkey and you don't want to distribute an additional DLL. How would we do that? Well, it turns out that it isn't too complicated. The GetAsyncKeyState API function will tell us if a key is down. Using this function in a loop, we can tell when a key gets pressed. POPCLR.MAK is the project that contains the code we need to look at. In POPCLR.BAS, we have a Sub Main (you will notice there are no forms in this project) in which we enter an infinite DoEvents loop, constantly checking to see if the "C" key is depressed (and what we can do to cheer it up). As soon as we discover that it is down, we check to see if the Ctrl key is also down. If it is, we know that the user has pressed Ctrl-C and we pop up the common color dialog. If you wanted to be really careful, you might also check to see if any other shift keys were down to make sure that the user wasn't pressing Ctrl-Shift-Alt-C or something like that, but we don't care about that for our purposes cause we are writing a utility for our own use.

Finally, if the user selected "OK" from the color dialog, we put the hex value of the color into the keyboard buffer with SendKeys. What could be simpler. It's a pretty small utility, but demonstrates a lot and I use it all the time now that I have it. I hope you will too.

Dropping Leftover DLLs

If you do any experimenting with DLLs, you have probably faced the problem that when (I mean if) your program crashes, the DLLs are not unloaded from memory. This can eat up memory if it's not your DLL. If it is your DLL, you need to exit Windows in order to make a change to the DLL and try it again. If you don't the old copy simply gets reused, which isn't too useful. It turns out that TOOLHELP.DLL provides the functions you need to determine which DLLs are loaded into memory. Therefore, it's possible to write a little developer's utility which lists them and allows you to unload the ones you want. Hence ZAPLIB.EXE.

Using ZAPLIB

To use ZAPLIB, you simply restore it from its usual state as an icon. At this point, it lists the loaded DLLs in a multi-select list box. Accordingly, you can select as many of these as you like and click "OK." At this point, ZAPLIB will unload these DLLs and re-minimize itself. ZAPLIB filters out the DLLs that it thinks are system DLLs to reduce clutter and temptation. Since you have the source, you can add to this list. There might still be DLLs loaded, the names of which you don't recognize. If in doubt, don't unload. Windows gets very unhappy if you unload a DLL that is in use by another program but you won't hear about it until the program try to call the DLL.

How It Works

ZAPLIB uses the ModuleFirst and ModuleNext functions, found in TOOLHELP.DLL to enumerate all of the active modules. The declarations for these functions and the MODULEENTRY structure are in the ZAPLIB.BAS module. The key code of ZAPLIB is in the Form_Resize event of the ZAPLIB.FRM source file. This code steps through the module list, getting the names of all the active modules. You will notice that you need to trim the trailing null and blanks from the populated elements of the MODULEENTRY structure..

For each entry, ZAPLIB checks the extension for DLL or VBX. There is a more sophisticated method of determining whether the module is a DLL or EXE but this method will work for our purposes and doesn't require a custom DLL function.   Next ZAPLIB checks to see if the module name falls in a list of "reserved" module names. These are system DLLs that you really shouldn't unload. If you find something on your system that shows up all the time that you will never unload, simply add its module name to the Reserved$ string. The .ItemData property of the list box is used to store the handle of the module if it gets listed. It is this handle which will be used with the FreeLibrary and GetModuleUsage API functions to unload the DLL.

In the Command1_Click event, we simply walk the list box, looking for selected items. For any that are selected, we call FreeLibrary with the hModule until GetModuleUsage returns zero. At this point, the DLL is unloaded. When the end of the list is reached, the application is minimized. That's all there is to it.

The QuickBasic Function Library

There are a number of useful functions in QuickBasic that were left out of VB. For some it seems like pure oversight. For others, it appears as though Microsoft was attempting to protect us from ourselves. Accordingly, it behooves me to say that you use these functions AT YOUR OWN RISK. I'm simply supplying you with the functionality you want but I'm not taking responsibility for your use of these functions. I'll be using these functions on and off in various issues of the journal. This issue simply uses the SSegAdd function which you will find yourself using all the time. What follows is a list of functions in the QBFUNC.DLL and their purpose. QBFUNC.BAS contains the declarations for all of these for use in your own applications. The Pascal source code for the DLL can be found in QBFUNC.PAS. This library isn't complete. If there is something you would like to see in there, let me know. For more details about using these functions, I would strongly recommend Ethan Winer's book on DOS BASIC from Ziff-Davis Press.

BLOAD, BSAVE

These functions are used for two purposes in QB. One is to save screens and the other is to save arrays quickly to disk. The first purpose has no meaning under Windows but the second does and you can use these functions to save your arrays in a compatible format across versions of BASIC. To quickly save an array to disk, simply pass the filename, and the first element of the array (by reference), and the number of bytes to the BSAVE function.   To use the BLOAD function, you need to redimension your array to the correct size by reading storing this information, by file size calculation (using LOF) or by reading the header. See Winer's book for details.

CVD, CVI, CVL

These functions translate the string (or memory) versions of doubles, integers and longs into their equivalent numeric values. You pass the string (retrieve from disk, or created with the MK? functions) to these functions and they return the numeric equivalents.

InterruptX

This is a function which allows you to make interrupt calls from within VB. This is often necessary to call DOS or machine functions that are not part of the API. Network functions are often called in this way as well. We will be using this function in future issues of the journal, but its use is specialized and I would really recommend getting Ethan's book for a good description of its use.

MKD, MKI, MKL

These are the opposite of the CV? functions and return the string equivalents of numeric variables.

PEEK

This function will allow you to determine the value of any byte in memory. You simply pass a long pointer to the memory address and the function returns the integer value of that byte.

POKE

This function will allow you to set a byte of memory (that you own). You pass a long pointer to the memory location and the integer value of the byte to set.

Note: These functions are not very efficient for large chunks of memory. For those, you might want to use the hMemCpy API function.

SSEGADD

This function will return a long pointer to any variable. The function requires you to *pass* a long pointer to a variable. This is done by simply passing the variable by reference in every case except for a string, in which case you need to use the ByVal keyword to call the function.

SWAP, SWAP4, SWAP8

Unfortunately, it isn't possible to provide a single SWAP function that works with all variable types. Instead you use SWAP for variable length strings and integers, the SWAP4 function for LONGs and SINGLEs and finally the SWAP8 function for DOUBLEs and CURRENCYs. The SWAP4 and SWAP8 routines aren't really any faster than those you could create in VB alone and are provided solely for convenience. Using SWAP for variable length strings is *much* faster than the equivalent code could be, however, and should be used whenever possible. ALL parameters should be passed by reference, even strings.

VARPTR, VARSEG

These functions return the offset and the segment accordingly of any VB variable. Since you will most often be using long pointers under Windows, these functions are provided mostly for compatibility. Pass strings "ByVal."

VBINP, VBOUT

These are equivalent to the QB INP and OUT statements and used in the same way to retrieve or set information to a port. These are equivalent to the IN and OUT statements in assembler.

The VBZ Utility Library

Every so often, there are functions in which it becomes 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 people's 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))

<u>Comments</u>

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.

<u>Prototype</u>

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

<u>Usage</u>

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

<u>Comments</u>

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.

<u>Prototype</u>:

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

<u>Usage</u>:

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

<u>Comments</u>:

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.

<u>Prototypes</u>

```
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 an 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 VBZ01.

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, ByVal 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:

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

<u>Visual Basic Techniques</u>
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!

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

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

<u>Improved Help Files</u>
Binary Sample Extraction
More links

The rest is up to you! Be sure and 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?

System Level Hotkeys
Aldus Format Metafiles
Creating a Drag 'n Drop Server Application
Creating a Drag 'n Drop Client Application
Status Bar Help on Controls and Cursors
A PLAY Command for Visual Basic
Musical MsgBox and Beep Commands
Creating Windows 3.1 Screen Savers

Letters

Actually, I got about 20 letters hailing the advent of this journal, but I forgot to save them. Oops! Costas Kitos wrote in with a suggestion to pop-up the proper section of the help file from the samples so we have done that where appropriate 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
Articles Broken up into sub-topics (good for browsing, bad for printing)
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

The only fix this issue is in the VBZUTIL.BAS support module. The second parameter of the Str2Ctl function was mis-declared. The CtlName$ parameter needs to be passed by value.

Reviews

Doc-To-Help
3D-Widgets

DOC-To-Help (by David S. Mohler 76450,1642)

WHO:	WHAT: DOC-To-Help
WexTech Systems, Inc.	
60 East 42nd Street, Suite 1733	HOW MUCH: $249.00
New York, NY 10165	
Phone: (212) 949-9595	
Fax: (212) 949-4007	

My reaction to and opinion of Doc-to-Help are very, very positive. The complete cross-referencing process is unparalleled in other packages I have investigated. And, as the name implies, the complete hard copy documentation is generated from the same text as the help file, killing two birds with one stone.

The text generation is done completely in WinWord; Doc-to-Help is built out of WordBasic macros that automate the formatting, bitmaps, cross referencing, indices, contents, glossaries and key-word searches. All text entered appears identical in both the hard copy and on-line docs by default. However, you can mark portions of text that is intended for hard copy only or for help file only.

It is somewhat restrictive in the sense that you cannot use Doc-to-Help to automatically insert jumps to other help files, use bitmaps as jumps, etc., which means you have to do this manually in WinWord. There also needs to be a help-file debugging command that helps to eliminate recursive jumps: that is, jumps which call themselves. This often happens when glossary text contains the word being defined. Doc-to-Help does not smartly eliminate those words in the linking process.

Even manual work in the RTF file is made easier, though, because Doc-to-Help will select 3.0, 3.1 or protect-mode compilers along with allowing you to set graphics compression options to 0, 40% or 50%. It then builds the proper .HPJ file and runs the compiler for you, trapping and generating a listing of any errors.

A minor upgrade to the current version is due out at the end of the month, I believe, which is supposed to include a few enhancements -- hopefully geared more toward the bells and whistles possible with the 3.1 help system.

It is so usable and complete that I would estimate 10-15% of my work now revolves around making help docs for other developers who only occasionally require full blown manuals and on-line help.

3D-Widgets (by Thomas Wagner 73530,1002)

WHO:	WHAT: 3D-Widgets
Sheridan Software Systems, Inc.	
65 Maxess Road	HOW MUCH:
Melville, NY 11747	Widgets / 1 $49
Phone: (516) 753-0985	Widgets / 2 and Widgets / 3 $39 each
Fax: (516) 753-3661	Combination of all three $99

There has been much talk in recent times about the changing look of user interfaces, especially in applications that have been created with Visual Basic. The 3-D look - as it is called - has become more and more important, not so much from a program functionality standpoint but from an aesthetic perspective.

To the best of my knowledge a lot of what we see today in the area of 3-D user interface design was first introduced by Steven Jobs in the form of the "burnished gray steel" look of the NEXT computer interface, as well as certain UNIX applications. This grayscale three dimensional look was so slick and different from everything else out at the time, it quickly became a favorite. Interestingly, now that NEXT has ceased operations as a hardware manufacturer, one of its remaining contributions is the 3-D look .

Not surprising, once VB hit the market people began using its inherent graphic abilities to recreate this look. At first by utilizing a work-around or programmatic solutions, and later by utilizing specific add-on libraries. For example the Waite Group's VB "How to " book contains among its many tips a way of recreating the NEXT look on VB forms.

Today, the most comprehensive aid in designing 3-D interfaces is being produced by Sheridan Software. The three component package called "Widget's" lets a programmer create all the raised panels, sunken check boxes and multiple 3-D list boxes that his heart desires. Even three dimensional menus are possible.

Each component is sold separately, thus allowing for a maximum of flexibility when choosing a set of custom controls. Widgets 1 contains the VBX's necessary to create 3-D check boxes, command buttons, Frames, Option Buttons, Panel's and Ribbon Buttons. Of the three sets of Widget's, this is probably the one utilized most often. For example, with the 3-D panel it becomes very easy to implement a status bar in your application. The ribbon buttons allow for fast creation of a toolbar. If you have the professional release of VB you are already familiar with this set, it is included with the program.

Widgets 2 contains 3-D versions of List Boxes, Combo Boxes, as well as File-, Directory-, and Drive List Boxes. And lastly, Widgets 3 contains Enhanced Menus. All of these components give the programmer an unprecedented amount of control over the look of the application's interface.

At this point a question comes to mind: Why buy it? For starters, Microsoft itself will be releasing more and more software incorporating the 3-D look. For example, Excel 4.0 for Windows makes extensive use of it. Another program that comes to mind is MS Mail, which has a toolbar that looks virtually identical to a ribbon bar you can create with "Widgets".

Granted, just because Microsoft wants to change some of the looks of their applications is not much of a   reason for you to spend your hard earned money. However, the same impetus that contributed to the success of 3-D interfaces in the first place may be enough reason for you to look at these custom controls: People like it!

A good number of people think 3-D looks "high tech", "great" and "different". End-users as well as programmers. That goes for me as well. I remember the first time I changed the bevel width of   a panel control and watched the result.....wow. So, it's exciting for the programmer (at least for some), and oftentimes it's exciting for the end user as well. That can translate to higher productivity for you personally - I spent more time working and less dreaming about Tahiti - and a more marketable end product.

You may have heard it said somewhere that a consistent look is very important in designing an interface. That is true, and the fun part in using "Widget's" is that your applications will look consistently different from a lot of programs out there. That in turn generates interest and excitement, and that's what it's all about.