Scratching the surface

With a little know-how, Visual Basic can give the programmer more control over Windows' behind-the-scenes operations. Mike Liardet looks at ways to access DLL files, the backbone of Windows, through a Visual Basic program called VBAPI.

Visual Basic shields the programmer from many of the complexities of Windows programming. This is both a strength and a weakness in the language — a strength because it makes Windows application development comparatively easy, and a weakness because it prevents the programmer from controlling what happens beneath the surface of the programming environment. Or does it?

Anything to Declare?

The VB "Declare" statement is the escape clause that allows a program to get to grips with Windows internals. It provides a great way to extend Visual Basic's functionality. In essence, "Declare" can be used to specify a linkage to the routines in any Dynamic Link Library (DLL) file, so that the DLL routines can then be called from the VB program in the usual way. "So what?" I hear you say. The point is that virtually the whole of Windows is implemented on the back of a handful of DLLs. Knowing how to use them opens doors.

There lies the catch. Ideally, to make full use of the DLLs you should have a good understanding of the way Windows works beneath the surface, and have access to some good supporting documentation for reference and guidance. Visual Basic (Professional)'s API (Application Programming Interface) Help file contains all the common Declares and provides a valuable tool. There is also a useful SDK (Software Development Kit) Help file that gives a description of how the routines can be used from within a C programming environment.

To get a better picture of what to do with these routines, the *Visual Basic Programmer's Guide to Windows API* (Daniel Appleman, Ziff-Davis Press) makes for ideal bedtime reading. The Microsoft Developers Network (MSDN) CD-ROM is another option. It is packed with hints on useful API calls for the Visual Basic programmer, alongside a lot of other information on C, Access, FoxPro and other Microsoft development tools. In an inspired move, Microsoft has extracted the VB-specific material from the MSDN and made it available directly to VB programmers by including the material on the VB 4.0 CD-ROM. Unfortunately it has been excluded from the Standard Edition, but Professional and Enterprise users get it.

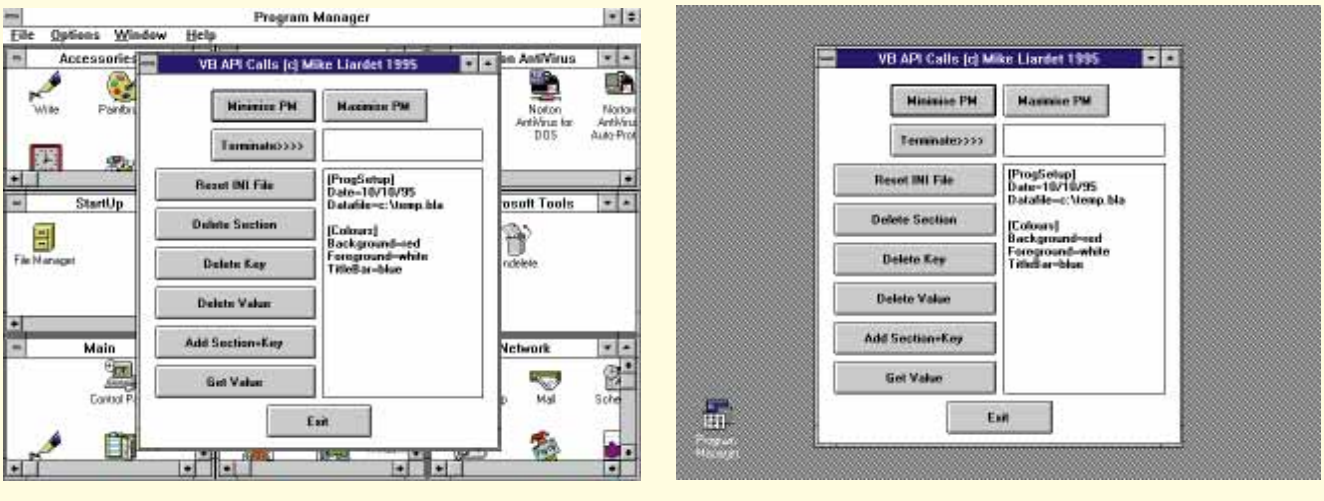
Back to Basics

This month we will pick up on some of the hints given on the MSDN/VB 4.0 and show how they can be applied to a plain and simple Visual Basic program, called VBAPI. It is worth mentioning that there is no need to upgrade to VB 4.0 in order to try out these techniques. They will work quite happily with version 3.0.

Figs 1 to 3 illustrate VBAPI in action. *Fig 1* shows how it can be used to minimise the Program Manager. This is something that might be required just after a program is loaded, as it gets rid of distracting background clutter. In a real application the code to do this would probably be put in the Form_Load event handler.

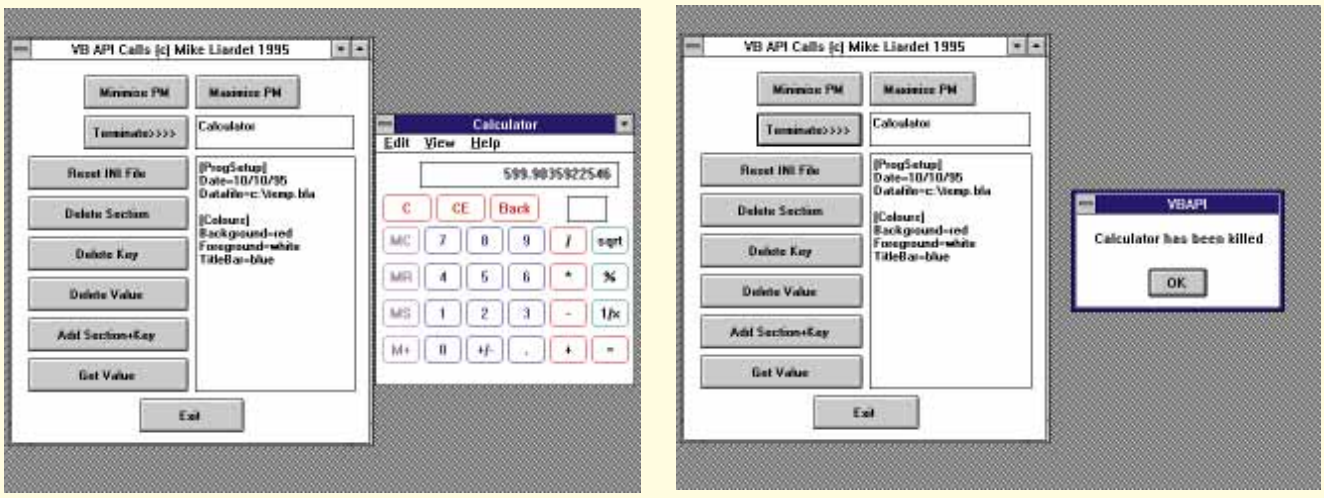
Fig 2 shows how another application can be terminated under VB control. In a real application this could be a useful trick for freeing up memory, or for preventing an application from running alongside some

Fig 1 Minimise/Maximise PM



"Minimise PM" minimises Program Manager to an icon, and "Maximise PM" brings it back to full size. The code behind the "Minimise PM" button would normally be put into an application's Form_Load

Fig 2 The "Terminate" button



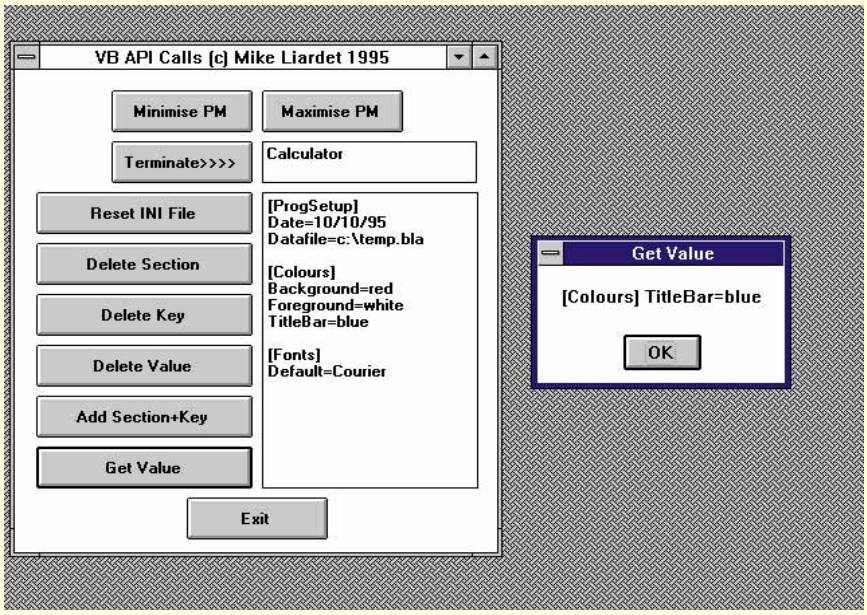
Now you see it, now you don't! The "Terminate" button can terminate any application — just give it the window title

other incompatible task — such as a Backup application. *Fig 3* shows the manipulation of an INI initialisation file. Just about every VB application could make good use of an INI facility. Although it could be implemented by using standard VB file I/O, it is much simpler and more elegant to use the standard Windows APIs instead.

In pure VB terms there is nothing fancy going on behind the scenes in VBAPI. We just have a number of Declare statements to provide access to the necessary API calls, and each command button is implemented with a few lines of code making the necessary calls. *Figs 4 to 6* give the details of the API calls needed. These show a few of the possibilities, once you get to grips with the Windows API. The major Windows DLLs offer around 1,000 others, then there are other supporting DLLs, DLLs from third-party suppliers, and so on — thousands of calls which are just a declaration away.

Unfortunately, information on DLLs is usually directed at the C programmer, so the VB coder needs to read between the lines when figuring out a DLL's declaration and use. C is richer in types than VB, so a 16-bit quantity in C might be typed as "hWnd", "int", "UINT" or "BOOL", and so on. Visual Basic has to make do with just "integer" for all of these types, but in practice this does not cause major problems. There is a similar variety of C types for "strings" and "longs", but in all cases the API call can still be made to work in VB if plain old "string" or "long" are used in the declaration, as necessary. Notice that all the parameters to API calls must be declared in VB as "ByVal" — this just makes sure that VB delivers the arguments in the right form to the underlying C routine.

Fig 3 Manipulating an INI file



Accessing an INI file. The various command buttons alongside the big text box show how sections, entries and values in the INI file can be added, deleted, changed or accessed

Callbacks

Although most of the DLLs are up for grabs, there are one or two API calls that simply cannot be handled by Visual Basic at all. For example, some Windows functions do "callbacks", which means that one of their parameters takes the address of a function. As Visual Basic functions are compiled differently from C, they cannot be called from it. Thus there is no meaningful address that can be given in this case. This means that API calls with callback cannot be used, or at least not used directly (there is a way round the problem but we won't go into it here).

Also, Visual Basic itself suppresses some of Windows' functionality, and this can render some calls unusable. For example, there are Windows functions to change the shape of the mouse pointer, but the VB environment has its own ideas about mouse display. This means that in general these functions won't work, and you are likely to be stuck with the dozen or so shapes that are predefined in VB itself.

For most common Windows functions the only awkward area is the handling of

Fig 4 Manipulating windows

```
IsWindow

C Declaration
BOOL IsWindow(hWnd)
HWND hWnd; /* handle of window, */

VB Declaration
Declare Function IsWindow Lib "User" (ByVal hWnd As
Integer) As Integer

Parameters
hWnd: identifies a window.
Returns: non-zero if the window handle is valid, other-
wise, returns zero (false).

GetWindow

C Declaration
HWND GetWindow(hWnd, wCmd)
HWND hWnd;/* handle of original window*/
UINT wCmd;/* relationship flag, */

VB Declaration
Declare Function GetWindow Lib "User" (ByVal hWnd As
Integer, ByVal wCmd As Integer) As Integer
'Values for wCmd...
Const GW_CHILD = 5 'Identifies the window's first child
window.

Const GW_HWNDFIRST = 0 'Returns the first sibling win-
dow for a child window; otherwise, it returns the first
top-level window in the list.
Const GW_HWNDLAST = 1 'Returns the last sibling window
for a child window; otherwise, it returns the last top-
level window in the list.
Const GW_HWNDNEXT = 2 'Returns the sibling window that
follows the given window in the list.
Const GW_HWNDPREV = 3 'Returns the previous sibling
window in the list.
Const GW_OWNER =  4 'Identifies the window's owner.

Parameters
hWnd: identifies the original window.
wCmd: specifies the relationship between the original
window and the returned window. It can be set to one of
the constant values given above.
Returns: the handle of the window if the function is
successful, otherwise NULL (integer 0 in VB) - indicat-
ing either the end of the system's list or an invalid
wCmd parameter.
```

An extract from a selection of Windows (with a big "W") functions for manipulating windows (with a small "w"). With the given declarations these can be used from Visual Basic, either to manipulate the VB application's own windows, or the windows belonging to another application

routines that return strings. The problem here is that Visual Basic and C handle strings differently. With C, the memory for the string needs to be allocated before the call, whereas a VB routine can find the space dynamically.

The GetPrivateProfileString routine *(Fig 9)* is a good illustration of this. It returns a string in its fourth argument. To call it from VB, an appropriate-length string must be initialised before the call and passed to the routine in this argument position, with the length given as its next argument. The function itself returns the number of characters it placed in the string, and these characters can easily be extracted using the VB left$() function.

Fig 4 gives a selection of the Windows functions which are used in the VBAPI program by both the commands that minimise and maximise Program Manager, and the command that terminates other applications. IsWindow and ShowWindow are straightforward. IsWindow determines whether or not the given window handle is valid, as opposed to being some arbitrary integer value. ShowWindow sets the given window's visibility state (maximised or minimised).

Can you handle it?

Unless the windows functions are being used to manipulate a VB application's

Fig 5 Definitions of Windows profile functions

```
WritePrivateProfileString

C Declaration
BOOL WritePrivateProfileString(lpAppName,lpKeyName, lpString, lpFilename)
LPCSTR lpAppName;/* address of section*/
LPCSTR lpKeyName;/* address of entry*/
LPCSTR lpString;/* address of string to add*/
LPCSTR lpFilename;/* address of initialization filename*/

VB Declaration
Declare Function GetPrivateProfileString% Lib "Kernel" (ByVal lpAppName As
String, ByVal lpKeyName As Any, Byval lpDefault as Any, ByVal
lpReturnBuffer As String, cbReturnBuffer as Integer, ByVal lpFileName As
String)

Parameters
All string parameters point to null terminated strings, which are case
independent, and so can contain any combination of upper and lower case
characters
lpAppName: specifies the section containing the entry.
lpKeyName: specifies the entry whose associated string is to be retrieved.
lpDefault: specifies the default value for the given entry if the entry
cannot be found in the initialization file- must never be NULL.
lpReturnBuffer: receives the character string.
cbReturnBuffer: receives the size, in bytes, of the buffer pointed to by
the lpReturnBuffer parameter.
lpFilename: names the initialization file.
Returns: the number of bytes copied to the specified buffer, not including
the terminating null character.
```

Definitions of two of the Windows "profile" functions that can be used to manipulate INI initialisation files

Fig 6 PostMessage

```
PostMessage

C Declaration
BOOL PostMessage(hWnd, uMsg, wParam, lParam)
HWND hWnd; /* handle of the destination window*/
UINT wMsg; /* message to post, */
WPARAM wParam; /* first message parameter*/
LPARAM lParam; /* second message parameter*/

VB Declaration
Declare Function PostMessage Lib "User" (ByVal hWnd As Integer, ByVal wMsg
As Integer, ByVal wParam As Integer, ByVal lParam As Long) As Integer
'Alterantive value for hWnd
Const HWND_BROADCAST = &HFFFF

Parameters
hWnd: identifies the window to which the message will be posted
wMsg: specifies the message to be posted.
wParam: specifies 16 bits of additional message-dependent information.
lParam: specifies 32 bits of additional message-dependent information.
Returns: nonzero if the function is successful, otherwise, zero (= false).
```

The PostMessage function, used for communicating with other windows or applications

own windows (where the control's hWnd property can return a window handle), it is first necessary to find the window's handle. FindWindow is needed here. It retrieves the handle of a window given either its class name (not so useful from VB) or title. GetWindowLong retrieves extra information on a window, as a long value at the specified offset into the extra window memory. There are many ways of using this, but in VBAPI this function is just used to determine whether or not a window is disabled.

It is often necessary to iterate through all the windows to find one with a particular attribute, or else to process them all in some way. GetWindow can be used for this. It retrieves the handle of a window that has the specified relationship to the given window, searching the system's list of top-level windows, their associated child windows, the child windows of any child windows, or any siblings of the owner of a window.

Windows has several APIs for manipulating INI files, but there are just two main functions that can do almost everything between them — WritePrivateProfileString and GetPrivateProfileString (Fig 5). WritePrivateProfileString copies a character string into the specified entry of a section of the specified initialisation file. If the file does not exist, it is created. If the section does not exist, it is created; and (surprise, surprise) if the entry does not exist in the specified section, then that too is created. It is also possible to delete information from the INI file using this function. If the entry parameter is given as 0, the entire section is deleted. If the string parameter is given as 0, then only the entry specified by the lpKeyName argument is removed.

INI file manipulation made easy

WritePrivateProfileString can also help track down the INI file. If the lpFilename argument does not contain a fully qualified path and filename for the file, it searches the Windows directory for the file. If the file does not exist, it creates

Fig 7 ShowWindow

```
Sub cmdMinPM_Click ()
    Dim hWnd As Integer, I As Integer
    hWnd = FindWindow(0&, "Program Manager")
    If hWnd <> 0 Then
        I = ShowWindow(hWnd, SW_SHOWMINNOACTIVE)
    End If
End Sub
```

ShowWindow can be used to minimise any application. Here is how it minimises the Program Manager

Fig 8 The PostMessage API

```
Sub cmdTerminate_Click ()
    txtTerminee = Trim$(txtTerminee)
    Select Case TaskKill(txtTerminee)
    Case TASK_NOT_FOUND
        MsgBox txtTerminee & " is not running"
    Case TASK_KILL_FAILED
        MsgBox txtTerminee & " is not answering"
    Case TASK_WAS_DISABLED
        MsgBox txtTerminee & " is disabled"
    Case TASK_WAS_ME
        MsgBox txtTerminee & " is myself"
    Case TASK_KILLED_OK
        MsgBox txtTerminee & " has been killed"
    End Select
End Sub

Function TaskKill (ByVal vsTaskTitleOrClass As String)
As Integer

    (Our cover disk has full details of the remainder of this code)

End Function
```

Using the PostMessage API call to terminate another application

the file in the Windows directory. If lpFilename contains a fully qualified path and filename and the file does not exist, it creates the file in that directory as long as the directory exists. Note that it is considered bad practice to create INI files in the Windows directory, as it makes it difficult to fully de-install an application.

GetPrivateProfileString retrieves a character string for the specified key from the specified section in the specified initialisation file. It searches the file for an entry that matches the name specified by the lpKeyName parameter under the section heading specified by the lpAppName parameter. If the entry is found, its corresponding string is copied to the buffer. If the entry does not exist, the default character string specified by the lpDefault parameter is copied. If the key is given as 0, all entries in the section specified by the lpAppName parameter are copied to the buffer specified by the lpReturnBuffer parameter. If the filename parameter does not contain a full path, it searches for the file in the Windows directory.

Much of the low level communication between applications, and even within an application, is handled by "messages". Windows message handling is a complex topic, but you don't need to know the ins and outs in detail to use messages. PostMessage *(Fig 6)* is the key routine. It places a message in a window's message queue and returns without waiting for it to process the message. Messages in a message queue are retrieved by calls to GetMessage or PeekMessage (neither needed by VBAPI). If the hWnd parameter is set to HWND_BROADCAST, the message is posted to all top-level windows, including disabled or invisible unowned windows. The routine should not be used to post a message to a control.

Figs 7 to 9 show how these functions can be put together to implement the various commands in VBAPI. Only the key code is shown, but the full program is available on the cover disk. *Fig 7* shows how to minimise Program Manager. First, find the handle for the window whose title is "Program Manager", then use ShowWindow with the command SW_SHOWMINNOACTIVE to minimise it. This code won't work if the title of the Program Manager window is changed, as it is for example when running under Windows NT, but there are ways around this.

Fig 8 shows how to kill a task. The main code to do this is in the routineTaskKill, which kills the window with the given task or class name. Again, FindWindow is used to find the window. There is some extra complexity in the routine to handle the case when the task being killed is the application itself or if the window is disabled, but for most cases it posts two messages, WM_CANCELMODE and WM_CLOSE, to the window, in order to kill it. Note the use of DoEvents: this makes sure that the recipient window gets a chance to clear its message queue and process the two new messages, when VBAPI relinquishes the processor.

One example of the INI file manipulation code is given in *Fig 9*. The other INI commands are very similar to this. Notice the RefreshINI command at the end of the routine. This is placed at the end of all the INI commands. It reads the INI file, using standard VB file I/O, into the VBAPI form's text box alongside the command buttons, providing instant verification that the command is working correctly.

Fig 9 Accessing an INI file value

```
Sub cmdGetValue_Click ()
Dim FileName As String
Dim lpAppName As String
Dim lpKeyName As String
Dim lpDefault As String
Dim lpString As String
Dim nBytes As Integer
Dim x As Integer
    FileName = app.Path & "\demo.ini"
    lpAppName = "Colours"
    lpKeyName = "TitleBar"
    lpDefault = "Not Found!!!"
    lpString = Space$(100)
    nBytes = 100
    x = GetPrivateProfileString(lpAppName, lpKeyName, lpDefault, lpString,
nBytes, FileName)
    MsgBox "[Colours] TitleBar=" & Left$(lpString, x), 0, "Get Value"
    RefreshINI
End Sub
```

GetPrivateProfileString can access any value in an INI file, but it's a little tricky to use from within VB

PCW Cover Disk

The full code for this month's Low Level is on the cover disk given with this issue of *Personal Computer World*.

PCW Contacts

Mike Liardet is a freelance programmer and writer. He can be contacted via the *PCW* Editorial office or on email as **mliardet@cix.compulink.co.uk**