Video Capture in Liberty BASIC

by Callum Lowcay
from Liberty BASIC Newsletter #127- updated 2010
See important correction added in issue #129 at bottom of article.

Table of Contents

Capturing Video

Impossible you say? Not at all, just try the demo code that accompanies this article.

VidCapLB.zip

- Details
- Download
- 3 KB

Video capture in LB is actually quite easy because all the hard work is done by AVICAP32.DLL, which is included with Windows 95 and higher. The possibilities are amazing! You could include a webcam in your application, perhaps even write a video conferencing program. All you have to do is understand the API calls involved, which is why I am writing this article. Please note that this is advanced LB and perhaps best suited to more advanced programmers.

Null Terminated Strings

Before getting into the API calls, a quick note about strings. When Microsoft wrote Windows 95 and Windows NT (98 and ME are based on 95; 2000 and XP are based on NT) the most commonly used programming language for PC was some form of C. C uses null terminated strings, which always end in chr$(0). Thanks to this, Windows API functions and most DLLs expect strings to end in chr$(0). This creates a small problem for LB, because LB strings don't automatically end in chr$(0). It turns out that LB has two solutions. If you pass a string into an API or DLL and it does not end in chr$(0), LB automatically creates a copy of the string and adds a chr$(0). It then passes the copy to the API. The disadvantage of this is that if the API modifies the string, it is modifying a copy rather than the actual string. If you needed to use the modified version of the string you are stuck because you can't get the copy back. The solution to this problem is to manually add a chr$(0) to the end of your string. In this case LB passes the address of the string and the API modifies the actual string rather than the copy. If you need to pass a string to an API and the API will not modify that string, you usually send the string without adding chr$(0) to the end and let LB sort it out for you. It is only if you need to get a string back from an API that you need to manually add a chr$(0). VidCapLB uses both methods for passing strings because some of the strings it passes need to be modified by the API and others don't.

Loading AVICAP32.DLL

The first thing you need to do to include video capture capabilities in your application is open up the correct DLL. As I stated in the previous paragraph, the correct DLL is AVICAP32.DLL which is loaded with the following statement:

```
'I am using the handle #vfw because that's what
'I use in the sample program (VidCapLB)
```

```
'vfw stands for "Video for Windows"
open "AVICAP32.DLL" for DLL as #vfw
```

Creating a Capture Window

Once you have the DLL loaded you can create a capture window. A capture window does all the hard work for you. It interfaces with the video capture driver and displays a preview of the video data. The capture window is a child window which means that you must first create a normal LB window, then create a capture window inside that window. For VidCapLB, I decided to make the capture window 320 pixels by 240 pixels because that is the largest image size my webcam can transmit. Here's the dll call you need:

```
calldll #vfw, "capCreateCaptureWindowA", _
    lpszWindowName$ as ptr, _
    dwStyle as ulong, _
    x as long, _
    y as long, _
    w as long, _
    h as long, _
    hWnd as ulong, _
    id as long, _
    hWndC as ulong
```

lpszWindowName$ is a string that specifies the name for your capture window. Because the API will not modify this string, it is safe to pass it without adding chr$(0) to the end. dwStyle (the window style) should always be _WS_VISIBLE OR _WS_CHILD (i.e. dwStyle = _WS_VISIBLE OR _WS_CHILD). This makes the capture window automatically visible, and a child window. x and y are obviously the x and y coordinates for the capture window, relative to the upper left hand corner of the parent window (the window made by LB). w and h are the width and height of the capture window. As stated above, I set these to 320 and 240. hWnd is the window's handle to the parent window. This is the window in which the capture window will appear. You get this with the HWND(#handle) function, where #handle is the handle to the LB window in which the capture window will appear. id is a unique id number for the capture window. If you have more than one capture window, each should have a unique id number. If you have only one capture window, almost any number will do for an id. hWndC is the windows handle to the capture window you have created. This variable is extrememly important, if you overwrite its value you could crash your program.

Connecting a Video Device

Right, so you have a capture window. At this stage the capture window appears as a black rectangle in your program. Before it can capture video you need to connect it to a device. Windows can have up to 10 video capture devices, numbered from 0 to 9. Before you can go any further you need to allow the user to select a video capture device. To do this you need to enumerate the devices on the user's system, put them in a

listbox, then when the user selects one calculate the index and connect the device. Enumerating the devices is really easy, you only need one dll call. This commented code demonstrates how to enumerate the devices then put them in an array that can be loaded into a listbox.

```
    dim Devices$(10)  'Because there can be up to 10 devices

    lpszName$ = space$(40)+chr$(0)  'Initialise a string buffer for th
e device names
    'Note that you need to add chr$(0) to the end of this string so th
e API can modify it.

    cbName = len(lpszName$)  'Find the length of that string

    lpszVer$ = space$(40)+chr$(0)  'Initialise a string buffer for the
 device version

    cbVer = len(lpszVer$)  'Find the length of that string

    for wDriverIndex = 0 to 9 'Because Windows numbers the devices fro
m 0 to 9

    'This API call retrives information about the device numbered wDri
verIndex
        calldll #vfw, "capGetDriverDescriptionA",_
            wDriverIndex as word, _
            lpszName$ as ptr, _
            cbName as long, _
            lpszVer$ as ptr, _
            cbVer as long, _
             test as long

    'This puts the name and version strings into the Devices$ array.
    'The strings are offsetted by 1 because LB listboxes start countin
g
    'at 1 rather than 0.
        Devices$(wDriverIndex+1) = lpszName$+" "+lpszVer$

    next
```

VidCapLB creates a listbox from the Devices$ array. When the user double clicks an item in the listbox VidCapLB gets the index of the selection. It then subtracts 1 to get the device index, which is stored in the DevNum variable. It is necassary to subtratct 1 because listboxes start couting at 1 whereas windows starts counting at 0. Once you have the device number from the user, simply use this dll call to connect the device:

```
calldll #user32, "SendMessageA", _
    hWndC as ulong, _
    1034 as ulong, _
    DevNum as long, _
    0 as long, _
    test as long
```

Where hWndC is the variable returned from the capCreateCaptureWindowA call and DevNum is the number of the device the user selected. 1034 is the message number, LB does not include windows constants for AVI capture messages so you either have to define them yourself or just use straight numbers. Using numbers is bad practice, you should use constants to give them descriptive names. I will leave this up to you.

Troubleshooting Hardware

This is all very well, but what if the device the user selected was turned off, not plugged in or malfunctioning? If the connect fails test will equal 0. You should always check to see if the connect failed before continuing otherwise all the video capture routines will simply not work and the user will have no idea why. Here's the code VidCapLB uses:

```
if test = 0 then
    notice "Cannot connect capture device"
    goto [quit.main]
end if
```

By this stage, your capture window will probably be displaying a single still image that it has captured from the video capture device. So how do you make that image move? There are actually two ways, enable preview mode and enable overlay mode. The difference between overlay and preview mode is that preview mode uses a lot more system resources. All devices support preview mode, but only the more expensive ones support overlay mode. To find out if you can use overlay mode requires querying the device to see if it is supported. I'll avoid querying the device for the moment as VidCapLB does not currently support overlay mode anyway. That means we are stuck with preview mode. I have found the following code to enable preview mode on all tested hardware:

```
'This enables preview mode
calldll #user32, "SendMessageA", _
    hWndC as ulong, _
    1074 as ulong, _
    -1 as long, _
    0 as long, _
    test as long
```

```
    'This sets the frame rate to 5.  Most devices can handle a frame r
ate of 5
    calldll #user32, "SendMessageA", _
        hWndC as ulong, _
        1076 as ulong, _
        5 as long, _
        test as long
```

Setting the Frame Rate

Once again, you are sending messages to the capture window. The 1074 message is used to enable and disable preview mode. The -1 in this call enables preview mode, a 0 in its place is used to disable preview mode. The 1076 message sets the frame rate. The example sets the frame rate to five, although with my hardware it makes no difference what I set the frame rate to as the device simply uses it's default frame rate. You need to set the frame rate to something however, or all you'll get in the capture window is that static image captured from the device. As before, you can check the test variables for 0. If either of them are 0, the message failed to do what it was meant to and you should bring up an error dialog to inform the user.

Capturing the Actual Video Data

The title of this article implied that it was about capturing data from the video capture device. All we've done so far is created a capture window, connected a device and enabled preview mode. Now it's time to capture some actual video data. VidCapLB captures two kinds of data, still images and AVI files. Capturing still images is less complex than capturing AVI files, so I'll start with that.

To capture a still image, you need to do two things. Firstly you need to store a frame into the frame buffer of the device, then you need to save it to the hard drive. Windows will automatically save these images as bmps. The two messages to achieve this are:

```
    'This captures a frame in the computers RAM
    calldll #user32, "SendMessageA",_
        hWndC as ulong,_
        1085 as ulong,_
        0 as long,_
        0 as long,_
        test as long

    'This saves that frame to the file the user specified, in BMP form
at
    calldll #user32, "SendMessageA",_
```

```
        hWndC as ulong,_
        1049 as ulong,_
        0 as long,_
        FileName$ as ptr,_
        test as long
```

FileName$ is the name of the file you want to save the frame in. VidCapLB uses the LB filedialog statement to get a filename from the user. As usual, you can check test for 0 in order to trap and report errors.

Capturing an AVI File

Now for the more complex part, capturing an AVI file. In order to capture video data in real time, you must preallocate some hard drive space. If you do not allocate enough space Windows will attempt to allocate more space and capture video at the same time. This results in poor video quality, so you should always allocate enough space. Windows will create a file called CAPTURE.AVI in the allocated space. CAPTURE.AVI is not a standard AVI and cannot be read with any media player, it is only a buffer to store the captured data. Once the capture is complete you use a message to copy the data from CAPTURE.AVI to the AVI file the user specified. After this it is safe to delete the CAPTURE.AVI file. Once you have sent Windows the message to start capturing video it will suspend all other applications (including LB) until the user presses ESC. It is possible to change the key, it is also possible to set up time limits for video capture. VidCapLB does not support these features so I won't discuss them here. Here's some sample code that allocates 50MB for a CAPTURE.AVI file, captures video data, saves it to an AVI file and finaly deletes the CAPTURE.AVI file:

```
    'This allocates 50MB of hard drive space for a capture file
    'Note that you specify the capture size in bytes.  50MB is 5242880
0 bytes
    'This is because 1MB = 1024 x 1024 bytes, therefore 50MB is 50 x 1
024 x 1024 bytes
    calldll #user32, "SendMessageA", _
        hWndC as ulong, _
        1046 as ulong, _
        0 as long, _
        52428800 as long, _
         test as long

    'If the user is out of hard drive space...
    if test = 0 then
        notice "Error"+chr$(13)+"Could not allocate hard drive
'space"+"Make sure you have at least 50MB free space"
        goto [quit.main]
```

```
    end if

    'This starts the capture, it also jams up all other programs in th
e system.
    'Including LB, so we just have to wait now for the user to press E
SC
    calldll #user32, "SendMessageA", _
        hWndC as ulong, _
        1086 as ulong, _
        0 as long, _
        0 as long, _
         test as long

    'This saves the video data in the file the user specified
    calldll #user32, "SendMessageA", _
        hWndC as ulong, _
        1047 as ulong, _
        0 as long, _
        FileName$ as ptr, _
         test as long

    'If the user is out of hard drive space...
    if test = 0 then
        'notice "Error"+chr$(13)+"Could not save file,
'you may not have enough hard drive space"
        goto [quit.main]
    end if

    'And this gets rid of the capture file, which is no longer needed
    kill "\CAPTURE.AVI"
```

Ending the Program

Before I end this article, there is one more vital thing to discuss. You absolutely must execute this code before ending your program:

```
    calldll #user32, "DestroyWindow",_
        hWndC as ulong, _
        ret as long
```

Failure to do so will result in an embarrassing crash when your program is ended. This code safely disconnects the device then destroys the capture window.

Device Configuration

There is one capability of VidCapLB not discussed in this article, device configuration. This involves checking the device capabilities and providing the user with a menu of configuration dialog boxes that are relevant to the connected device. The code in VidCapLB is clear and resonably well documented. I sugest you look at it closely if you want to see how to apply the video capture APIs described in this article. If you need more information there is probably some at the MSDN website. The free API documentation from [Borland] also contains a lot of useful information. It is where I learned to do this video capture. Also useful would be a file called VFW.H. This is a C/C++ header file that contains the values of the messages used in video capture amoungst other things. Being a C/C++ header file you can find it in the include folder of most good win32 C/C++ compilers. There is probably also an SDK somewhere on the Microsoft website that includes this file.

CORRECTION FROM NL129
In issue #129, Callum offers the following correction to the article above:

I recently ordered the Microsoft Platform SDK CD in order to get more up to date API documentation. After reading the documentation regarding the frame rate for preview mode I realised that the article I wrote in the newsletter was misleading on frame rate. It turns out the value referred to as frame rate in that article is actually the interval in milliseconds between frames. In other words, the computer waits the specified number of milliseconds before capturing each frame. By using the term 'frame rate' I feel my article implied that this value was the number of frames per second, which it is not.

In theory if the interval is too low you can waste a lot of system resources, but I think this applies more to computers from the Win 3.1 days when Video for Windows was introduced. If you set the interval lower than your device can go, the device simply defaults to its lowest supported interval. The example in the SDK documentation suggested an interval of 60 milliseconds as opposed to the 5 I used in the article.

I would like to apologise if my article mislead or confused anyone on this issue. Hopefully I will be able to avoid any similar errors in the future now that I have complete and up to date documentation.