Using ZLib in Visual Basic

Written by Mike D Sutton Of EDais

Http://EDais.mvps.org/

EDais@mvps.org

- 21.11.2004 -

What is ZLib?

ZLib is a fast and open source compression library which is legally unencumbered by software patents unlike other compression solutions such as Zip and LZW. The ZLib library and source code is available on its site here: http://www.gzip.org/zlib/
Because of its ease of use and portability it has become widely accepted by the internet community and has even become core to a number of file formats such as SWF and PNG, so before you can properly look at either format you first need ZLib which is the focus of this article.

Introduction and compression

At the time of writing the latest version on ZLib is 1.2.2 which is the version I'll be supporting in this article however before we even start we hit our first hurdle which is located in the libraries FAQ located here: http://www.gzip.org/zlib/DLL_FAQ.txt
The problem here is item 6 which reads: "I see that the ZLIB1.DLL functions use the "C" (CDECL) calling convention. Why not use the STDCALL convention?"
The calling convention specifies various things such as how arguments are passed to the DLL and how cleanup is performed, in VB we can only call stdcall DLL's however all is not lost (otherwise this would be a very short and somewhat pointless article!)
Reading on we see the comment: "The ZLIB_WINAPI macro will switch on the WINAPI (STDCALL) convention." which is exactly what we want, however rather than download the source and fiddle about with it ourselves we see that someone has already gone before us on this path (Gilles Vollant, author the WinImage) and has made the result available on his site here: http://www.winimage.com/zLibDll/
You can either download the source and compile it yourself or simply grab the pre-built DLL from the site which is compiled for us using the stdcall calling convention, simply drop it in your system32 directory (or in your application directory) and it will be available for use in VB.

This article assumes you've had experience using the API before and are reasonably familiar with how to port c header files to VB but if not then not to worry.
First off we need to work out what's exposed to us by the DLL and how to call this from VB, heading over to the ZLib programmer's manual page will get us started there: http://www.gzip.org/zlib/manual.html

The first function we'll cover happens to be the first one listed on the page which is the method to compress a data buffer, compress(). Its function header is defined as follows:

```
int compress (Bytef *dest, uLongf *destLen, const Bytef *source, uLong sourceLen);
```

The return value of int means its returning us a Long. The dest and source parameters are both pointers to the destination and source buffers respectively which could be declared using a specific data type, but to keep the code generic I usually declare these as "Any" to avoid typecasting problems in VB. destLen and sourceLen are both Longs, however the destination length is declared as a pointer so it must be declared by reference so the library can both read and write to it. We can now complete the declaration:

```
Private Declare Function compress Lib "ZLibWAPI.dll" ( _
   ByRef dest As Any, ByRef destLen As Long, _
   ByRef source As Any, ByVal sourceLen As Long) As Long
```

Before we can use the function though we need another which isn't actually mentioned on the page, however a quick look through the ZLib main header file will reveal it. In most cases you will have no idea how well the library will compress your data and since you're responsible for allocating the data for the destination buffer how large do you make it? The answer lies in this second function which simply returns the maximum possible size of the destination buffer for the given amount of data:

```
uLong compressBound(uLong sourceLen);
```

This one is nice and easy to port to VB so I'll just provide the declare here:

```
Private Declare Function compressBound Lib "ZLibWAPI.dll" ( _
   ByVal sourceLen As Long) As Long
```

Ok, now we're good to go, time to take her out for a little test drive.
First us we need something to compress, the obvious choice here is just the contents of a file however you can compress anything you wish. If you do choose a file pick something big enough that you give the library something to work with, but not too large that you're sitting around for ages waiting for it to finish every time you test it; around the 5 mb mark should be good.
Here's a quick function for returning the contents of a file as a byte array rather than having to write your own:

```
Private Function FileToBuf(ByRef inFile As String, ByRef outBuf() As Byte) As Long
   Dim FNum As Integer
   Dim RetBuf() As Byte

   ' Make sure file exists
   If (Not FileExist(inFile)) Then Exit Function

   FNum = FreeFile() ' Get a free file handle and open file
   Open inFile For Binary Access Read Lock Write As #FNum
      ReDim RetBuf(0 To (LOF(FNum) - 1)) As Byte ' Allocate buffer
      Get #FNum, , RetBuf() ' Read file data into buffer
   Close #FNum

   ' Return array
   outBuf = RetBuf
   FileToBuf = UBound(RetBuf) + 1
End Function

Private Function FileExist(ByRef inFile As String) As Boolean
   On Error Resume Next
   FileExist = CBool(FileLen(inFile) + 1)
End Function
```

Since we don't really need a UI for this test I'm going to be developing this using a single module with a Sub Main() declared however you could use the Form_Load handler of a form if you wish.
Grab the file data and size first:

```
Dim FileData() As Byte, FileSize As Long

...

Const FileName As String = "X:\Path\File.xyz"

FileSize = FileToBuf(FileName, FileData())
If (FileSize > 0) Then
```

```
    ...
End If
```

Now we need to know how large to make the buffer for ZLib so call compressBound() and allocate the buffer:

```
Dim CompressBuf() As Byte, CompressLen As Long

...

CompressLen = compressBound(FileSize)
ReDim CompressBuf(0 To (CompressLen - 1)) As Byte
```

Now all that’s left to do it to call the compression function itself and see what it makes of our data:

```
Dim RetVal As Long

...

RetVal = compress(CompressBuf(0), CompressLen, FileData(0), FileSize)
If (RetVal = Z_OK) Then
    Debug.Print "Compression succeeded, went from " & _
        FileSize & " to " & CompressLen & " bytes (" & _
        Format$(CompressLen / FileSize, "0.0%") & " original size)"
Else
    Debug.Print "Compression failed.."
End If
```

You’ll see that I’m testing the return value for “Z_OK” here which is a return code passed back from the library to indicate that all went well, the manual provides a handy hyperlink to the declaration which is simply 0 so that maps straight to a VB constant:

```
Private Const Z_OK As Long = &H0
```

Give it a whirl, and as long as everything goes smoothly after a couple of seconds (depending on the speed of your machine and the size of the input buffer) you should see the results of the compression printed to the debug window (Ctrl+G)
If you get “Bad DLL calling convention” then check that you’ve declared your functions the same way as described above, if so then you may have a version of the DLL compiled with cdecl calling convention (although it would be very strange if it had the “wapi” suffix..) so make sure you grab one compiled with the stdcall calling convention.

Decompression and validation

Ok, we've got the data compressed but it's not really of much use to us unless we can decompress it again so lets move on to dealing with decompression.
A quick look through the manual again shows us that the decompression method is called uncompress and has the following function signature:

```
int uncompress (Bytef *dest, uLongf *destLen, const Bytef *source, uLong sourceLen);
```

This matches the compress function signature and as such makes porting the call to VB simple:

```
Private Declare Function uncompress Lib "ZLibWAPI.dll" ( _
   ByRef dest As Any, ByRef destLen As Long, _
   ByRef source As Any, ByVal sourceLen As Long) As Long
```

We'll go on from the previous chapter's code, and simply decompress the compressed data buffer (assuming the compression operation succeeded.)
ZLib requires the output buffer to be large enough to store all the decompressed data, so whenever you compress data with the library you must also store the size of the original buffer.
Note*; the function returns Z_BUF_ERROR if the supplied buffer was too small so it is possible to decompress a ZLib compressed buffer without knowing its uncompressed size, by sending it various buffer sizes until it no longer returns Z_BUF_ERROR.*
This approach is very inefficient since it means the data has to be decompressed each time so should only be used as a last resort, in most cases you should already know the length of the decompressed buffer.

In this case we know the size of the decompressed data (the file data) so we can simply allocate the buffer:

```
Dim DecompressBuf() As Byte, DecompressLen As Long

...

DecompressLen = FileSize
ReDim DecompressBuf(0 To (DecompressLen - 1)) As Byte
```

Now simply call the decompression method:

```
RetVal = uncompress(DecompressBuf(0), DecompressLen, CompressBuf(0), CompressLen)
If (RetVal = Z_OK) Then
   Debug.Print "Decompression succeeded, result size: " & DecompressLen & " bytes"
Else
   Debug.Print "Decompress failed.."
End If
```

Assuming all went well the size of the decompressed buffer should be the same as the original fie size.
To further verify that the data is indeed correct we can use what's known as a cyclical redundancy check or CRC which takes a buffer and performs some mathematics on

each byte to get a final result. If even one byte of the two buffers differs then the CRC checks will be different which allows us to detect the validity of the data.
The ZLib library exposes two CRC methods, the first performs a full CRC check on the data, where as the second performs a much quicker (but less accurate) check. If you want to find out more about how these two methods work then the full source code is available, you'll find the full CRC method implemented in crc32.c and it's corresponding header file, and the Adler CRC method in adler.c
The two function signatures can be found near the bottom of the programmer's manual:

```
uLong adler32 (uLong adler, const Bytef *buf, uInt len);
uLong crc32 (uLong crc, const Bytef *buf, uInt len);
```

Again these are pretty simple to port to VB, each taking an initial value then a pointer to a data buffer and its length:

```
Private Declare Function adler32 Lib "ZLibWAPI.dll" ( _
   ByVal adler As Long, ByRef buf As Any, ByVal length As Long) As Long
Private Declare Function crc32 Lib "ZLibWAPI.dll" ( _
   ByVal crc As Long, ByRef buf As Any, ByVal length As Long) As Long
```

The way these methods work is to take an initial value and use that as a base to calculate the rest of the CRC from the given buffer. The reason for this is it allows CRC calculation of multi-part buffers rather than having to send the entire thing in one go. The only problem here is that what initial value do we start the CRC buffer on for the first piece of data we sent to it, does it even matter? The answer depends on what you're using the check for, if you simply want to check inside your own application then as long as you specify the same initial value for both the source CRC check and the destination CRC check then it really doesn't matter which initial value you specify. If however you're receiving the CRC as calculated by another application (common in things such as network transfer where data is prone to 'go missing' or get corrupted) then you must be sure to specify the same initial value as the other application. While you could get the application to send its initial CRC value to you, there is no reason that that data wouldn't get corrupted but luckily there is a better way.
By calling the functions and specifying a null pointer to the buffer, it will simply return its preferred initial value so as long as the other application is using this too then you know you're starting from the correct value.
For this test we'll use the full CRC method, however the Adler method works in exactly the same way, so go ahead and find the initial value:

```
Dim FileCRC As Long

...

' Get initial value
FileCRC = crc32(0, ByVal 0&, 0)
```

With the version of the library I'm using, the initial value of the CRC is zero, however it's always best to get the library to tell you rather than hard coding it since this could (but shouldn't) change in future versions.

Once you have the initial value you can send it the rest of the file buffer to compute its CRC:

```
FileCRC = crc32(FileCRC, FileData(0), FileSize)
```

Now calculate the CRC of the decompressed buffer (I'll be using a slightly more condensed version by getting the initial value inline) and compare them:

```
Dim DecompressCRC As Long

...

DecompressCRC = crc32(crc32(0, ByVal 0&, 0), DecompressBuf(0), DecompressLen)

Debug.Print "File CRC: 0x" & Hex(FileCRC) & ", Decompressed CRC: 0x" & _
   Hex(DecompressCRC) & " (" & IIf(FileCRC = DecompressCRC, "Match", "Diferent") & ")"
```

As long as all went well the CRC's should match.

Additional functionality

We've accomplished our goal, being able to read and write ZLib compressed buffers however there is more functionality the library exposes that can come in handy.

Taking a peek through the main ZLib header file shows us another interesting little function which returns us the parameters that were used to compile the library with, this can be very handy to make sure we can actually call the library (since VB can still call cdecl functions which take no parameters.)
The function header is as follows:

```
uLong zlibCompileFlags(void);
```

This simply translates to VB as:

```
Private Declare Function zlibCompileFlags Lib "ZLibWAPI.dll" () As Long
```

The function returns us a 32-bit value which contains the information about how the library was compiled packed into various bits; the return value is defined as follows:

Bit	Meaning
0, 1	Size of uInt
2, 3	Size of uLong
4, 5	Size of voidpf (pointer)
6, 7	Size of z_off_t
8	Defines whether the library was compiled in debug mode
9	Use ASM code
10	Exported functions use the WINAPI calling convention
11	[Reserved]
12	Build static block decoding tables when needed
13	Build CRC calculation tables when needed
14, 15	[Reserved]
16	gz* functions cannot compress
17	Deflate can't write gzip streams, inflate can't detect and decode gzip streams
18, 19	[Reserved]
20	Slightly more permissive inflate
21	Deflate algorithm with only one, lowest compression level
22, 23	[Reserved]
24	1 means limited to 20 arguments after the format
25	1 means gzprintf() not secure!
26	1 means inferred string length returned
27-31	[Reserved]

Note*; I'm not going to be dealing with the GZip file functions here however based on the introduction laid by this article you're welcome to have a hack at them yourself. If you do decide to have a look at them as a result of this article then I'd love to hear how you get on with it and may present a second article later on using them.*

The one we're really interested in here is bit 10 which specifies whether the library is compiled using a calling convention compatible with VB, however some of the other details are also worth checking.
Since we're working at the bit-level now it makes sense to have a look at the value in binary, so here's a quick method which converts a DWord to binary (sign-bit safe):

```
Private Function DWordToBinary(ByVal inDWord As Long) As String
   Dim LoopBits As Long, ThisBit As Long

   Const HighBit As Long = &H80000000

   DWordToBinary = CStr(((inDWord And HighBit) = HighBit) And &H1)
   For LoopBits = 30 To 0 Step -1
      ThisBit = 2 ^ LoopBits
      DWordToBinary = DWordToBinary & _
         CStr(((inDWord And ThisBit) = ThisBit) And &H1)
   Next LoopBits
End Function
```

The first four values in the compile flags define the size of the various types used by the library so lets extract those and find out what they are. To extract the bit values from the DWord we first mask the desired bits then shift them to the right until there are no trailing zeros. Since VB has no shift operator we must instead use an integer divide using powers of two to shift the value the desired number of bits right.
The first value is already at the far right so only requires a mask:

```
Dim CompileFlags As Long
Dim IntSize As Long

CompileFlags = zlibCompileFlags()
IntSize = CompileFlags And &H3
```

However this isn't the size of the value, but the index in a choice of values:

Index	Value
0	16-Bit
1	32-Bit
2	64-Bit
3	Other

Since the given sizes are all adjacent powers of 2 (with the exception of the last index which effectively has no value) we can get the value by using the power operator:

```
IntSize = 2 ^ (4 + IntSize)
```

Of course there's still that last value which will annoyingly come back as 128 using the above technique, however a simple modulus divide will fix that:

```
IntSize  = (2 ^ (4 + IntSize)) Mod &H80
```

The next size requires a mask and shift, the binary value of the mask is 1100 which translates to 0xC in hex (if you need to look these up then the Windows calculator in scientific mode will do that for you.) Since this is sitting 2 places from the right we must then shift the value two places right using an integer divide of 2^2 or 4 (If you need to look these up then VB's own debug window will tell you the answer, simply type in "?hex(2 ^ 2)" and press enter):

```
Dim LongSize As Long

...

LongSize = (CompileFlags And &HC) \ &H4
```

The next two are calculated in much the same way:

```
Dim VoidPfSize As Long
Dim ZOffTSize As Long

...

VoidPfSize = (CompileFlags And &H30) \ &H10
ZOffTSize = (CompileFlags And &HC0) \ &H40
```

After that we're onto the single bit flags which are easier to work with since they don't need to be shifted, again Windows calculator can be used to get the bit-masks

```
Dim UseWINAPI As Boolean

...

UseWINAPI = CBool(CompileFlags And &H400)
```

There are many more flags other than the one declared above which can be accessed in the same way however for the time being we're only interested in this one.
***Note**; The accompanying code extracts all the flags*

From this point we can tell whether the library is compatible with VB by comparing the sizes of the various types and testing for the presence of the WINAPI flag:

```
Dim VBCompatible As Boolean

...

VBCompatible = _
   (IntSize = 32) And _
   (LongSize = 32) And _
   (VoidPfSize = 32) And _
   UseWINAPI
```

One other function catches our eye after looking through the documentation once more and that's the compress2() method, what does it do and how does it differ from the standard compress() method?
The answer lies in its final parameter, level, which specifies how well the data is compressed. The values defined for compression are located in the header file and listed at the bottom of the programmer's manual:

```
#define Z_NO_COMPRESSION       0
#define Z_BEST_SPEED           1
```

```
#define Z_BEST_COMPRESSION       9
#define Z_DEFAULT_COMPRESSION  (-1)
```

As we can see the compression level goes from 1 to 9 with 0 being a special value to indicate that no compression should be applied and -1 indicating that the library should choose a default compression level (which is what the standard compress() function uses.)
Converting these constants to VB gives us:

```
' Compression levels
Private Const Z_NO_COMPRESSION As Long = 0
Private Const Z_BEST_SPEED As Long = 1
Private Const Z_BEST_COMPRESSION As Long = 9
Private Const Z_DEFAULT_COMPRESSION As Long = (-1)
```

The function header is exactly the same as the standard compress method but with the additional parameter so again porting it is a trivial task:

```
Private Declare Function compress2 Lib "ZLibWAPI.dll" ( _
   ByRef dest As Any, ByRef destLen As Long, _
   ByRef source As Any, ByVal sourceLen As Long, _
   ByVal level As Long) As Long
```

It’s used in exactly the same way as the standard compress() method so I wont present any additional sample code here, however be aware that as the compression level increases the time taken to compress the buffer also increases.

Encapsulating this functionality into a class or even just wrapping the compression/decompression/CRC methods in simple functions to work directly on byte arrays is an obvious next step, however I’ll leave that as an exercise for the reader!
I hope this article has been of some use to you and as always any comments, suggestions or feedback are welcome.

Mike D Sutton (Visual Basic MVP)