# Version Information Manipulator Library User Guide

## *Accessing exported objects*

The DLL exports one function, CreateInstance that takes a CLSID specifying the type of reader / writer object required and passes out a reference to an interface to the required object.

All other interaction with the DLL is made using the provided object reference.

The provided file IntfBinaryVerInfo.pas lists all the required constants, data types, interfaces etc. and provides a function prototype for CreateInstance. The comments in the file explain the use of each method.
There are four distinct objects that can be used for manipulating version information. They are:

| **CLSID** | **Description** |
|---|---|
| CLSID_VerInfoBinaryReaderA | Provides read only access to version information stored in ANSI format. Objects of this type support IVerInfoBinaryReader. |
| CLSID_VerInfoBinaryA | Provides read/write access to version information stored in ANSI format.Objects of this type support IVerInfoBinary. |
| CLSID_VerInfoBinaryReaderW | Provides read only access to version information stored in Unicode format. Objects of this type support IVerInfoBinaryReader. |
| CLSID_VerInfoBinaryW | Provides read/write access to version information stored in Unicode format. Objects of this type support IVerInfoBinary. |

Which object you need depends on what you want to do. If you only want to read version information, use one of the "Reader" CLSIDs, for write access use one of the other CLSIDs. But how do you know whether you need ANSI or Unicode versions? Here are the rules:

- Resources from 16 bit programs are always returned in ANSI format regardless of the operating system. This means we must be able to determine whether the executable being examined is 16 or 32 bit (i.e. NE or PE format)
- Resources from 32 bit programs are returned in ANSI format on Windows 9x operating systems and in Unicode in NT platforms (including Windows 2000 and XP). This means we also have to be able to detect the OS platform.
- 32 bit binary resource files are always in Unicode format.

To detect the OS platform either use the Windows GetVersionEx() API call. Alternatively, if you are using Delphi use the following function:

```
function IsWinNT: Boolean;
begin
```

```
  Result := (SysUtils.Win32Platform =
Windows.VER_PLATFORM_WIN32_NT);
end;
```

To detect the type of an executable file use the following function (which also recognises DOS programs).

```
type
  TExeFileType = (
    etNotExec,  // not an executable file
    etPE,       // PE format file (Windows 32 bit executable)
    etNE,       // NE format file (Windows 16 bit executable)
    etDOS       // DOS format executable
  );

function ExeType(const FileName: string): TExeFileType;
  {Examines given file and returns a code that indicates the type of
  executable file it is (or if it isn't an executable)}
const
  cWinHeaderOffset = $3C; // offset of "pointer" to windows header
in file
  cDOSMagic = $5A4D;      // magic number for DOS executable
  cNEMagic = $454E;       // magic number for NE executable (Win 16)
  cPEMagic = $4550;       // magic number for PE executable (Win 32)
var
  FS: TFileStream;        // stream to executable file
  DOSMagic: Word;         // contains Dos magic number
  Offset: LongInt;        // offset of windows header in exec file
  WinMagic: Word;         // contains PE or NE magic numbers
begin
  // Assume we can't find type of file
  Result := etNotExec;
  // Open file for analysis
  FS := TFileStream.Create(FileName, fmOpenRead + fmShareDenyNone);
  try
    // Try to DOS magic number at start of file
    if FS.Size < SizeOf(Word) then
      Exit;
    FS.ReadBuffer(DOSMagic, SizeOf(Word));
    if DOSMagic <> cDOSMagic then
      Exit;
    // We no we have at least a DOS file
    Result := etDOS;
    // Try to find offset of windows program header
    if FS.Size <= cWinHeaderOffset + SizeOf(LongInt) then
      Exit;
    FS.Position := cWinHeaderOffset;
    FS.ReadBuffer(Offset, SizeOf(LongInt));
    // Now try to read first word of Windows program header
    if FS.Size <= Offset + SizeOf(Word) then
      Exit;
    FS.Position := Offset;
    FS.ReadBuffer(WinMagic, SizeOf(Word));
    // This word should identify either a NE or PE format file
    if WinMagic = cNEMagic then
      Result := etNE
    else if WinMagic = cPEMagic then
      Result := etPE;
  finally
    FS.Free;
```

```
  end;
end;
```

You can test for a 32 bit binary resource file by checking if the first eight bytes in the file are $00, $00, $00, $00, $20, $00, $00, $00.

Once you have decided which CLSID to use, get the required object reference by calling

```
var
  Obj: IVerInfoBinaryReader // or Obj: IVerInfoBinary

if CreateInstance(MyCLSID, Obj) = S_OK then
  {Everything OK}
else
  {Error};
```

Check the HResult from CreateInstance – it will be S_OK if the object was created OK. If the library does not support the CLSID then Obj is set to nil and E_NOTIMPL is returned. If there is an error in creating the object Obj is set to nil and E_FAIL is returned.

## *Interrogating Version Information Data*

Assume you have an object that supports IVerInfoBinaryReader. We will look at how to iterate all the version information in a file. Please see IntfBinaryVerInfo.pas for details of method parameters and a brief description of how to call the methods.

To get the fixed file information use one of the following methods:

- GetFixedFileInfo
- GetFixedFileInfoArray
- GetFixedFileInfoItem

To iterate the string file information we first need to find the number of "translations" in the version information – there should be a string table for each translation (there is usually just one). We find the number of translations by calling:

- GetTranslationCount

We can find details of each translation using these methods:

- GetTranslation
- GetTranslationAsString

We can also check if a specified translation exists by calling:

- IndexOfTranslation

There should be the same number of string tables as there are translations (but in malformed version information this may not be true). We can check this is the case using:

- GetStringTableCount

We can find out which translation a string table belongs to by using one of:

- GetStringTableTransString

- GetStringTableTransCode

We can also find if a string table exists for a given translation by using:

- IndexOfStringTable
- IndexOfStringTableByCode

At last we can start to find the entries in a string table. The number of entries in a specified string table by using:

- GetStringCount

The next three methods are used to get the name and value of a specified string item within a string table:

- GetStringName
- GetStringValue
- GetStringValueByName

Finally, there are some miscellaneous methods. We can clear version information by using the Clear method, copy version information from another IVerInfoBinaryReader using the Assign method.

Use the ReadFromStream method to read binary version information from a IStream.

The last error message can be found using LastErrorMessage. The message is set whenever a method returns an error HResult. The message is cleared when a method returns successfully, so LastErrorMessage must be called before calling another method.

## *Opening a IStream onto version information*

From the above, it is obvious that to read version information into a IVerInfoBinaryReader or IVerInfoBinary object we must be able to open a stream onto the version information data, and to provide an IStream interface to the stream. Here's how to open a stream onto version information stored in an executable file.

The version information API returns a buffer containing the whole of the version information data. Once we have a buffer we can access this easily from Delphi using a memory stream. Alternatively we can define a new TCustomMemoryStream that can read version information data. This is the approach taken here:

```
TVerInfoFileStream = class(TCustomMemoryStream)
  private
    fInfoBuffer: Pointer;
    fInfoBufferSize: Integer;
    procedure AllocBuffer(const Size: Integer);
    procedure FreeBuffer;
  public
    constructor Create(const FileName: string);
    destructor Destroy; override;
    function Write(const Buffer; Count: LongInt): LongInt; override;
  end;

{ TVerInfoFileStream }
```

```
resourcestring
  // Error messages
  sNoFile = 'File "%s" does not exist';
  sCantWrite =
    'Can''t write version information into an executable file';
  sNoVerInfo = 'No version information present in file "%s"';

procedure TVerInfoFileStream.AllocBuffer(const Size: Integer);
  {Allocates buffer of given size to store version information data.
  Any pre-existing buffer is first freed}
begin
  if (fInfoBufferSize <> Size) or (fInfoBuffer = nil) then
  begin
    // We need to allocate buffer wrong size
    // first free any old buffer
    FreeBuffer;
    if Size > 0 then
      // non-zero size specified: allocate the buffer
      GetMem(fInfoBuffer, Size);
    // record new buffer size
    fInfoBufferSize := Size;
  end;
end;

constructor TVerInfoFileStream.Create(const FileName: string);
  {Class constructor: accesses version information in given file and
  stores this in internal buffer that stream reads. If version info
  can't be accessed exceptions are raised}
var
  Dummy: DWORD;           // used in call to GetFileVersionInfoSize
  VerInfoSize: Integer;   // size of version information data
begin
  inherited Create;
  // Check file exists
  if not FileExists(FileName) then
    raise EStreamError.CreateFmt(sNoFile, [FileName]);
  // Get size of version information data in file
  VerInfoSize := GetFileVersionInfoSize(PChar(FileName), Dummy);
  if VerInfoSize > 0 then
  begin
    // Allocate buffer of required size to hold ver info
    AllocBuffer(VerInfoSize);
    // Read version info into memory stream
    if not GetFileVersionInfo(
      PChar(FileName), Dummy, fInfoBufferSize, fInfoBuffer
    ) then
      // read failed: free the allocated buffer
      FreeBuffer;
  end;
  // If we didn't get version info we raise exception
  if fInfoBufferSize = 0 then
    raise EStreamError.CreateFmt(sNoVerInfo, [FileName]);
  // Set the stream's memory pointer to buffer where ver info is
  SetPointer(fInfoBuffer, fInfoBufferSize);
end;

destructor TVerInfoFileStream.Destroy;
  {Class destructor: frees version info data buffer}
begin
  FreeBuffer;
  inherited;
```

```
end;

procedure TVerInfoFileStream.FreeBuffer;
  {Frees the version information data buffer, if allocated}
begin
  if fInfoBufferSize > 0 then
  begin
    // Buffer size > 0 => we must have buffer, so free it
    Assert(Assigned(fInfoBuffer));
    FreeMem(fInfoBuffer, fInfoBufferSize);
    // Reset buffer and size to indicate buffer not assigned
    fInfoBuffer := nil;
    fInfoBufferSize := 0;
  end;
end;

function TVerInfoFileStream.Write(const Buffer;
  Count: LongInt): LongInt;
  {Override of TStream's abstract Write method: raise exception
since
  this stream is read only}
begin
  raise EStreamError.Create(sCantWrite);
end;
```

That's all very well, but we need to support the IStream interface. We do this easily by utilising IStream wrapper classes from the DelphiDabbler code library (see http://www.delphidabbler.com/software?id=streams).

## *Reading / writing binary resource file data.*

If you need to know how to read binary data from and write binary data to a .res file then please see the Resource File Unit, also from the DelphiDabbler code library (http://www.delphidabbler.com/software?id=resfile).