How to extract version information using the Windows API

Introduction

The Windows API provides a method to extract version information from an executable file. The API is rather archane and some knowledge of how version information is stored in an executable file is helpful.

This article starts with a brief overview of how version information is laid out in a file. It then goes on to develop some code that tests whether version information is present and extracts it if so. We then wrap the code up in a class before finally looking at some of the limitations of the approach we have taken.

Version Information Overview

Version information is stored in an executable file's resources. The binary format of this data is complex and is not described here. Conceptually though, version information contains a record that provides a machine-readable binary description of a file (*fixed file information*) and one or more *string tables* that provide human-readable information. String tables can be localised – so there can be a string table for each supported language or code page. Because of this, version information also contains a table of supported languages / code pages (the *translation table*) that effectively provides an index into the string tables.

Although version information supports multiple languages it is rare to find executables that take advantage of this – version information usually contains only one string table. Unlike most of the code available on the net ours will handle multiple string tables.

Reflecting the organisation discussed above, version information is structured in three main parts:

- **Fixed file information**

  This is a data structure that contains binary information about the executable file. Windows declares this structure as *VS_FIXEDFILEINFO* and the Delphi equivalent is *TVSFixedFileInfo*, defined as follows:

```
type
  tagVS_FIXEDFILEINFO = packed record
    dwSignature: DWORD;         // always $feef04bd
    dwStrucVersion: DWORD;      // version of structure
    dwFileVersionMS: DWORD;     // most significant file version
    dwFileVersionLS: DWORD;     // least significant file version
    dwProductVersionMS: DWORD; // most significant product version
    dwProductVersionLS: DWORD; // last significant product version
    dwFileFlagsMask: DWORD;     // masks valid flags in dwFileFlags
    dwFileFlags: DWORD;         // bit mask of attributes of file
    dwFileOS: DWORD;            // flags describing target OS
    dwFileType: DWORD;          // flag describing type of file
    dwFileSubtype: DWORD;       // sub type for file: keyboard driver
    dwFileDateMS: DWORD;        // most significant part of file date
    dwFileDateLS: DWORD;        // least significant part of file date
  end;
  TVSFixedFileInfo = tagVS_FIXEDFILEINFO;
```

Listing 1

  The fields of this record are described in depth in the Windows help file so we won't discuss further here.

- **Variable file information**

  According to the Windows documentation this section can be user-defined. However, in practise it always contains a table of "translation" information. Each entry in the table is a pair of language and character set identifiers that together provide a key used to specify a string table. We can define an entry in this table with the following record (not defined by Windows):

```
TTransRec = packed record
  Lang,           // language code
```

```
  CharSet: Word;  // character set (code page)
end;
```

Listing 2

- **String file information**

  This section stores a string table for each "translation" listed in the *variable file information* section. The entries in string tables are name / value pairs – i.e. the string values are accessed by specifying a string name. Windows defines several standard names:

  - *Comments*
  - *CompanyName*
  - *FileDescription*
  - *FileVersion*
  - *InternalName*
  - *LegalCopyright*
  - *LegalTrademarks*
  - *OriginalFilename*
  - *PrivateBuild*
  - *ProductName*
  - *ProductVersion*
  - *SpecialBuild*

  The intended purpose and restrictions on use of these names are set out in the Windows API help file under the "VERSIONINFO resource" topic. User defined names are also permitted.

These different sections of the resource are accessed using an addressing system that is similar to a file path. The "paths" are:

- \ – i.e. the "root" path

  Accesses fixed file information.

- `\VarFileInfo\Translation`

  Accesses the "translation" table. The table is a sequence of *TTransRec* records.

- `\StringFileInfo\<translation-code>\<string-name>`

  Accesses a named entry in a given string table. <translation-code> is made up of the hexadecimal representation of the translation's language code and character set code. Unsurprisngly <string-name> is the name of the desired string value. For example to access the "ProductName" string in the string table associated with translation "040904E4" the "path" is `\StringFileInfo\040904E4\ProductName`.

This all looks rather horrendous – and if you write code to access the raw version information binary it is. However the Windows API provides three functions that assist in extracting the code. In the next section we'll review the API calls and write some Delphi wrappers to make them easier to use before finally developing our version information class.

Writing the code

Windows API functions

Windows provides the following functions to use to access version information.

- **GetFileVersionInfoSize**

```
function GetFileVersionInfoSize(lptstrFilename: PChar;
  var lpdwHandle: DWORD): DWORD; stdcall;
```

Listing 3

This function returns the size of the given executable file's version information. It returns 0 if there is no version information present. We have to pass a dummy variable as *lpdwHandle* – the function just sets it to 0!

- **GetFileVersionInfo**

```
function GetFileVersionInfo(lptstrFilename: PChar;
  dwHandle, dwLen: DWORD; lpData: Pointer): BOOL; stdcall;
```

Listing 4

Once we know the size of the version information we have to create a buffer the same size and then call this function to copy the version information from the executable file into the buffer. We pass the name of the file, a dummy 0 value to *dwHandle*, followed by the size of the buffer and the buffer pointer itself. The function returns true on success and false on failure.

- **VerQueryValue**

```
function VerQueryValue(pBlock: Pointer; lpSubBlock: PChar;
  var lplpBuffer: Pointer; var puLen: UINT): BOOL; stdcall;
```

Listing 5

This function is used to read data from the version information buffer we filled using *GetFileVersionInfo*. We pass the buffer as the first parameter. The next parameter takes a pointer to the "path" that specifies the information we need (as described above). The function sets the pointer variable passed as *lplpBuffer* to point to the requested data. *puLen* is set to the length of the data. The function returns true on success and false on failure. If the function fails *lplpBuffer* has an indeterminate value and can cause a GPF if read.

A full description of these functions can be read in the Windows API help. As can be seen the calls we need to make are complex and require a lot of pointer manipulation. In the next section we hide some of this complexity in wrapper routines.

Delphi wrapper routines

Let us now define some wrapper functions that (a) hide the API routines and (b) give easy access to the fixed file information, string values and translation table.

We will create five routines:

1. **GetVerInfoSize** – a thin wrapper round the *GetFileVersionInfoSize* API function.
2. **GetVerInfo** – a thin wrapper round the *GetFileVersionInfo* API function.
3. **GetFFI** – gets the fixed file information from the version information buffer using *VerQueryValue*.
4. **GetTransTable** – uses *VerQueryValue* to get and return the translation table as a dynamic array of *TTransRec* records.
5. **GetVerInfoStr** – returns the value of a string table entry using *VerQueryValue*.

GetVerInfoSize

```
function GetVerInfoSize(const FileName: string): Integer;
var
  Dummy: DWORD; // Dummy handle parameter
begin
  Result := GetFileVersionInfoSize(PChar(FileName), Dummy);
end;
```

Listing 6

This function is a thin wrapper round the *GetFileVersionInfoSize* API routine – it simply prevents us from having to provide the dummy parameter that routine requires.

GetVerInfo

```
procedure GetVerInfo(const FileName: string; const Size: Integer;
  const Buffer: Pointer);
begin
  if not GetFileVersionInfo(PChar(FileName), 0, Size, Buffer) then
    raise Exception.Create('Can''t load version information');
end;
```

Listing 7

This is a thin wrapper that calls the *GetFileVersionInfo* API function and raises an exception if the call fails.

GetFFI

```
function GetFFI(const Buffer: Pointer): TVSFixedFileInfo;
var
  Size: DWORD;  // Size of fixed file info read
  Ptr: Pointer; // Pointer to FFI data
begin
  // Read the fixed file info
  if not VerQueryValue(Buffer, '\', Ptr, Size) then
    raise Exception.Create('Can''t read fixed file information');
  // Check that data read is correct size
  if Size <> SizeOf(TVSFixedFileInfo) then
    raise Exception.Create('Fixed file information record wrong size');
  Result := PVSFixedFileInfo(Ptr)^;
end;
```

Listing 8

GetFFI gets fixed file information from the version information data. We call *VerQueryValue* with a "path" of "\" and then check if the call succeeds and that the returned data is the correct size.

GetTransTable

```
type
  TTransRec = packed record
    Lang,                             // language code
    CharSet: Word;                    // character set (code page)
  end;
  PTransRec = ^TTransRec;             // pointer to TTransRec

  TTransRecArray = array of TTransRec; // translation table

function GetTransTable(const Buffer: Pointer): TTransRecArray;
var
  TransRec: PTransRec;  // pointer to a translation record
  Size: DWORD;          // size of data read
  RecCount: Integer;    // number of translation records
  Idx: Integer;         // loops thru translation records
begin
  // Read translation data
  VerQueryValue(
    Buffer, '\VarFileInfo\Translation', Pointer(TransRec), Size
  );
  // Get record count and set length of array
  RecCount := Size div SizeOf(TTransRec);
  SetLength(Result, RecCount);
  // Loop thru table storing records in array
  for Idx := 0 to Pred(RecCount) do
  begin
    Result[Idx] := TransRec^;
    Inc(TransRec);
  end;
end;
```

Listing 9

This routine fetches the translation table from the version information. It gets the raw data using *VarFileInfo* then calculates the number of translation records by dividing the size of the raw data by the size of a translation record. We then size the dynamic array appropriately and copy each record into the array.

We also need to define the *TTransRec*, *PTransRec* and *TTransRecArray* types required by the routine.

GetVerInfoStr

```
function GetVerInfoStr(const Buffer: Pointer;
  const Trans, StrName: string): string;
```

```
var
  Value: PChar;   // the string value data
  Dummy: DWORD;   // size of value data (unused)
  Path: string;   // "path" to string value
begin
  // Build "path"  from translation and string name
  Path := '\StringFileInfo\' + Trans + '\' + StrName;
  // Read the string: return '' if string doesn't exist
  if VerQueryValue(Buffer, PChar(Path), Pointer(Data), Dummy) then
    Result := Data
  else
    Result := '';
end;
```

Listing 10

This function returns the value of given string from the string table associated with a given translation. We build the "path" needed to access the string value and use *VerQueryValue* to get the string value. If the string exists we return the data, cast to a string. Otherwise we return the empty string.

The *demo program* accompanying this article shows how to use the routines.

The next step is to encapsulate the whole process in a class.

Object oriented solution

The class we will develop is rather simple but has all the needed functionality. It can detect whether a file contains version information and can extract fixed file information, the translation table and strings from one or more string table. Here's the class declaration:

```
type
  TVerInfo = class(TObject)
  private
    fFixedFileInfo: TVSFixedFileInfo; // fixed file info record
    fTransTable: TTransRecArray;      // translation table
    fHasVerInfo: Boolean;             // whether file contains ver info
    fVerInfo: Pointer;                // buffer storing ver info
    function GetString(const Trans, Name: string): string;
    function GetTranslation(Idx: Integer): string;
    function GetTranslationCount: Integer;
  public
    constructor Create(const FileName: string);
    destructor Destroy; override;
    property HasVerInfo: Boolean read fHasVerInfo;
    property FixedFileInfo: TVSFixedFileInfo read fFixedFileInfo;
    property Translations[Idx: Integer]: string read GetTranslation;
    property TranslationCount: Integer read GetTranslationCount;
    property Strings[const Trans, Name: string]: string read GetString;
  end;
```

Listing 11

The constructor receives the name of the file to be examined. The *HasVerInfo* property is true if the file contains version information. The *FixedFileInfo* property contains a copy of the fixed file info record. *TranslationCount* records the number of translations in the file while *Translations* is a zero based array of string values corresponding to the translations. These values are used to specify which string table the *Strings* property uses to get a specified string value.

Internally we store the translation table in a dynamic array of *TTransRec* records. Both the translation and the fixed file information are recorded when the class is created. String file information is read from version information as values are requested from the *Strings* property.

Most of the work is done in the constructor – and we use the routines developed above to simplify the code. So, let us start by looking at the constructor:

```
constructor TVerInfo.Create(const FileName: string);
var
  BufSize: Integer;     // size of ver info buffer
begin
  inherited Create;
  // Get size of buffer: no ver info if size = 0
```

```
  BufSize := GetVerInfoSize(FileName);
  fHasVerInfo := BufSize > 0;
  if fHasVerInfo then
  begin
    // Read ver info into buffer
    GetMem(fVerInfo, BufSize);
    GetVerInfo(FileName, BufSize, fVerInfo);
    // Read fixed file info and translation table
    fFixedFileInfo := GetFFI(fVerInfo);
    fTransTable := GetTransTable(fVerInfo);
  end;
end;
```

Listing 12

We first use *GetVerInfoSize* to find the size of the version information data. We set the *HasVerInfo* property to record if there is version information (by checking the size of the data), and if there is none, we skip the rest of the code.

If we do have version information we allocate a buffer to hold it then load the version information from the file using the *GetVerInfo* routine. We next use the *GetFFI* and *GetTransTable* routines to store the fixed file information and translation table in fields.

The destructor is very simple: we just free the version info buffer:

```
destructor TVerInfo.Destroy;
begin
  // Free ver info buffer
  FreeMem(fVerInfo);
  inherited;
end;
```

Listing 13

We'll look at the *GetTranslation* method next. This simple method returns a string representation of the translation at a given index in the table. The string representation is simply the concatenation of the translation's language code and character set / code page code in hexadecimal:

```
function TVerInfo.GetTranslation(Idx: Integer): string;
begin
  Assert(fHasVerInfo);
  Assert((Idx >= 0) and (Idx < fTranslationCount));
  // Return string representation of translation at given index
  Result := Format(
    '%4.4x%4.4x', [fTransTable[Idx].Lang, fTransTable[Idx].CharSet]
  );
end;
```

Listing 14

The *GetTranslationCount* method simply returns the size of the dynamic array that stores the translation table:

```
function TVerInfo.GetTranslationCount: Integer;
begin
  Result := Length(fTransTable);
end;
```

Listing 15

This leaves just the *GetString* method. Most of the work is done by the *GetVerInfoStr* routine. We simply check we have version information and then return the result of calling *GetVerInfoStr*.

```
function TVerInfo.GetString(const Trans, Name: string): string;
begin
  Assert(fHasVerInfo);
  Result := GetVerInfoStr(fVerInfo, Trans, Name);
end;
```

Listing 16

That's the class completed. Again this can be tested using the demo code (below).

Demo Code

Please *download the demo code* that accompanies this article. It includes the class, types and routines presented here, along with a test program that exercises them.

The demo program can load version information from one of four supplied executables. One of these is the demo program itself while the others are do-nothing console programs whose sole purpose is to have their version information checked by the demo. The example programs contain various different configurations of version information as follows:

- the demo program has a single translation / string table
- one console program has two translations / string tables
- another has no translations (just fixed file information)
- the last has no version information at all

For a more comprehensive solution to the problem of extracting version information see my *Version Information Component*.

Limitations

While the solution works well in the majority of cases, it is by no means perfect. There are two major shortcomings:

1. While standard and non-standard string information can be accessed, the user must know in advance what non-standard string names are present in the version information. The code cannot enumerate the string names. This can't be done using the official API and much more advanced methods are required. My *Version Information Spy* program can enumerate the strings in a string table. If you need to do this please examine this program's source code.
2. Several programs contain version information that does not follow the Microsoft guidelines – i.e. the structure of the binary version information is non-standard. The API functions can fail on some of these programs. Again the source to Version Information Spy illustrates how to get round this problem by partially ignoring the API and parsing the binary data directly.

Summary

In this article we have reviewed how Windows handles version information and makes it accessible to programmers via the API. We noted that the API is rather cumbersome and so went on to develop wrapper routines to hide some of the complexity. We then wrote a class that encapsulates the version information and provides a simpler interface for the developer. A demo program was provided that implements and tests the code. Finally we reviewed some of the limitations of the code.

I hope the article has been useful. If you have any comments or suggestions please *contact me*.
