How to detect the types of executable files (part 2 of 3)

Coding the ExeType function

Developing the function

Now we have an outline design, we can begin creating our function. Recall that it will analyse a given file and return a value to indicate the type of file found. *Listing 2* shows the function prototype:

```
function ExeType(const FileName: string): TExeFileKind;
```

Listing 2

The return value is an enumerated type, which is defined as follows.

```
type
  {
  TExeFileKind:
    The kinds of files recognised.
  }
  TExeFileKind = (
    fkUnknown,  // unknown file kind: not an executable
    fkError,    // error file kind: used for files that don't exist
    fkDOS,      // DOS executable
    fkExe32,    // 32 bit executable
    fkExe16,    // 16 bit executable
    fkDLL32,    // 32 bit DLL
    fkDLL16,    // 16 bit DLL
    fkVXD       // virtual device driver
  );
```

Listing 3

We will now examine the function's implementation. The code follows the logic presented in the first part of the article. First, here's the function prototype again:

```
function ExeType(const FileName: string): TExeFileKind;
  {Examines given file and returns a code that indicates the type of
  executable file it is (or if it isn't an executable)}
```

Listing 4

Immediately following the prototype we declare some constants to hold the fixed offsets, flags and "magic numbers" we will need to use:

```
const
  cDOSRelocOffset = $18;  // offset of "pointer" to DOS relocation table
  cWinHeaderOffset = $3C; // offset of "pointer" to windows header in file
  cNEAppTypeOffset = $0D; // offset in NE windows header of app type field
  cDOSMagic = $5A4D;      // magic number for a DOS executable
  cNEMagic = $454E;       // magic number for a NE executable (Win 16)
  cPEMagic = $4550;       // magic nunber for a PE executable (Win 32)
  cLEMagic = $454C;       // magic number for a Virtual Device Driver
  cNEDLLFlag = $80        // flag in NE app type field indicating a DLL
```

Listing 5

The constants are followed by the the function's local variables. First we declare a variable to reference a file stream object that we will be using to read the file. We then have a variable to store Windows executable "magic numbers", along with another variable to store the offset of the Windows header record. We then declare variables to read information from the various header records - a byte in the case of the NE file format, an *IMAGE_FILE_INFO* structure for PE format files and an *IMAGE_DOS_HEADER* structure for

the DOS file header. Finally we have a variable to store the minimum expected size of the MS-DOS file. *Listing 6* has the the variable declarations:

```
var
  FS: TFileStream;              // stream to executable file
  WinMagic: Word;               // word containing PE or NE magic numbers
  HdrOffset: LongInt;           // offset of windows header in exec file
  ImgHdrPE: IMAGE_FILE_HEADER;  // PE file header record
  DOSHeader: IMAGE_DOS_HEADER;  // DOS header
  AppFlagsNE: Byte;             // byte defining DLLs in NE format
  DOSFileSize: Integer;         // size of DOS file
```

Listing 6

The *IMAGE_DOS_HEADER* structure is not declared in the `Windows` unit, so we must define it ourselves as shown in *Listing 7*.

```
type
  {
  IMAGE_DOS_HEADER:
    DOS .EXE header.
  }
  IMAGE_DOS_HEADER = packed record
    e_magic   : Word;                   // Magic number ("MZ")
    e_cblp    : Word;                   // Bytes on last page of file
    e_cp      : Word;                   // Pages in file
    e_crlc    : Word;                   // Relocations
    e_cparhdr : Word;                   // Size of header in paragraphs
    e_minalloc: Word;                   // Minimum extra paragraphs needed
    e_maxalloc: Word;                   // Maximum extra paragraphs needed
    e_ss      : Word;                   // Initial (relative) SS value
    e_sp      : Word;                   // Initial SP value
    e_csum    : Word;                   // Checksum
    e_ip      : Word;                   // Initial IP value
    e_cs      : Word;                   // Initial (relative) CS value
    e_lfarlc  : Word;                   // Address of relocation table
    e_ovno    : Word;                   // Overlay number
    e_res     : packed array [0..3] of Word;  // Reserved words
    e_oemid   : Word;                   // OEM identifier (for e_oeminfo)
    e_oeminfo : Word;                   // OEM info; e_oemid specific
    e_res2    : packed array [0..9] of Word;  // Reserved words
    e_lfanew  : Longint;                // File address of new exe header
  end;
```

Listing 7

We now start to process the file. Before we can perform any analysis we need to open the file for reading. A read only file stream is used to do this. We also need to handle any exceptions raised when reading the file. A skeletal outline of the body of the function is shown in *Listing 8*. This listing illustrates how we open and close the file stream and handle any exceptions raised.

```
begin
  try
    // Open stream onto file: raises exception if can't be read
    FS := TFileStream.Create(FileName, fmOpenRead + fmShareDenyNone);
    try
      // ... process file here
    finally
      FS.Free;
    end;
  except
    // Exception raised in function => error result
    Result := fkError;
  end;
end;
```

Listing 8

If the file doesn't exist then an exception will be raised by the stream constructor. This, and any other exceptions, are trapped and converted into error results. We use an inner **`try..finally`** block to ensure thefile stream gets closed.

The file analysis code fits inside the inner **try..finally** block in the above code fragment. The remainder of this section is devoted to a discussion of how we perform the analysis. We begin by attempting to read the DOS header record. We then use the information in the header to perform various checks. The code is shown in *Listing 9*.

```
// Assume unkown file
Result := fkUnknown;
// Any exec file is at least size of DOS header long
if FS.Size < SizeOf(DOSHeader) then
  Exit;
FS.ReadBuffer(DOSHeader, SizeOf(DOSHeader));
// DOS files begin with "MZ"
if DOSHeader.e_magic <> cDOSMagic then
  Exit;
// DOS files have length >= size indicated at offset $02 and $04
// (offset $02 indicates length of file mod 512 and offset $04
// indicates no. of 512 pages in file)
if (DOSHeader.e_cblp = 0) then
  DOSFileSize := DOSHeader.e_cp * 512
else
  DOSFileSize := (DOSHeader.e_cp - 1) * 512 + DOSHeader.e_cblp;
if FS.Size < DOSFileSize then
  Exit;
// DOS file relocation offset must be within DOS file size.
if DOSHeader.e_lfarlc > DOSFileSize then
  Exit;
// We assume we have an executable file: assume its a DOS program
Result := fkDOS;
```

Listing 9

We first assume the file format is unknown. Then we check that the file is large enough to contain the DOS header and exit if this is not the case. (Remember that the **finally** block is always executed following an *Exit*, so our file stream will be freed). If the file is large enough we read the DOS header before preforming these three checks on it:

1. That the *e_magic* field stores the required magic number.
2. That the file has the required length. The *e_cp* field stores the number of 512 byte pages in the file. The *e_cblp* field stores the number of bytes in the file modulus 512. The expected file length is calculated from these two values and this is checked against the actual size of the file. (Note that the file size can by greater than the expected size since it is possible to append data to an executable file – see *article #7* for further information).
3. That the offset of the DOS relocation table per the *e_lfarlc* field falls within the file.

If all these tests are passed then it is safe to assume we have at least a DOS executable, so we set the function result accordingly.

The next thing to do is to try to find a Windows file header and read the Windows executable magic number at the start of it. *Listing 10* has the code to do this:

```
// Try to find offset of Windows program header
if FS.Size <= cWinHeaderOffset + SizeOf(LongInt) then
  // file too small for windows header "pointer": it's a DOS file
  Exit;
// read the offset
FS.Position := cWinHeaderOffset;
FS.ReadBuffer(HdrOffset, SizeOf(LongInt));
// Now try to read first word of Windows program header
if FS.Size <= HdrOffset + SizeOf(Word) then
  // file too small to contain header: it's a DOS file
  Exit;
FS.Position := HdrOffset;
// This word should be NE, PE or LE per file type: check which
FS.ReadBuffer(WinMagic, SizeOf(Word));
```

Listing 10

We first check that the file is large enough to store the Windows header offset, and bail out if not. If we bail out we assume the file is a DOS executable (the return value was set earlier). We then read in the offset and

once again check that the file is large enough to hold a magic number located at the file position given by the offset. If so we move the file pointer to the start of the header and read the magic number into the *WinMagic* variable.

We now use the magic number to determine what kind of file we have. PE and NE format files are then analysed separately to check whether the fill is a DLL or application. A case statement is used to do the checking. Its outline is as follows:

```
case WinMagic of
  cPEMagic:
    // ... PE format - check whether DLL or application
  cNEMagic:
    // ... NE format - check whether DLL or application
  cLEMagic:
    // ... LE format - return VXD type
  else
    // ... DOS file executable
end;
```

Listing 11

We'll discuss each of the cases separately.

PE Format

For PE format files we attempt to read the Windows header record (of type *IMAGE_FILE_HEADER*) from the file. If the *Characteristics* field of the structure (a bit mask) contains the *IMAGE_FILE_DLL* flag then we have a DLL, otherwise we have an application.

Listing 12 has the code for the PE part of the above case statement:

```
// 32 bit Windows application: now check whether app or DLL
if FS.Size < HdrOffset + SizeOf(LongWord) + SizeOf(ImgHdrPE) then
  // file not large enough for image header: assume DOS
  Exit;
// read Windows image header
FS.Position := HdrOffset + SizeOf(LongWord);
FS.ReadBuffer(ImgHdrPE, SizeOf(ImgHdrPE));
if (ImgHdrPE.Characteristics and IMAGE_FILE_DLL)
  = IMAGE_FILE_DLL then
  // characteristics indicate a 32 bit DLL
  Result := fkDLL32
else
  // characteristics indicate a 32 bit application
  Result := fkExe32;
```

Listing 12

Note that, once again, before attempting to read the header we check the file is large enough to contain it and bail out if not, assuming a DOS executable. If we successfully read the header we check the *Characteristics* field for the required flag.

NE Format

For NE format files we read the byte at offset $0D from the start of the header and check to see if it contains a bit flag $80. If so, we have a DLL and if not we have an application.

The code for the NE part of the above case statement, which follows similar logic to that for the PE header, is:

```
// We have 16 bit Windows executable: check whether app or DLL
if FS.Size <= HdrOffset + cNEAppTypeOffset
  + SizeOf(AppFlagsNE) then
  // app flags field would be beyond EOF: assume DOS
  Exit;
// read app flags byte
FS.Position := HdrOffset + cNEAppTypeOffset;
FS.ReadBuffer(AppFlagsNE, SizeOf(AppFlagsNE));
if (AppFlagsNE and cNEDLLFlag) = cNEDLLFlag then
```

```
            // app flags indicate DLL
            Result := fkDLL16
          else
            // app flags indicate program
            Result := fkExe16;
```

Listing 13

LE Format

For LE Format files there is no further checking to be done. We simply return that we have found a virtual device driver:

```
          // We have a Virtual Device Driver
          Result := fkVXD;
```

Listing 14

DOS Format

This just leaves the trivial case of when none of the magic numbers are present. In this case we assume that the file is a DOS application. Since we have already set the function result to the DOS file type there is nothing to do. We simply place a comment to document this fact:

```
          // DOS application
          {Do nothing - DOS result already set};
```

Listing 15

Putting it all together

Our *ExeType* function is now complete. *Listing 16* shows the complete function, along with the required type definitions.

```
type

  {
  IMAGE_DOS_HEADER:
    DOS .EXE header.
  }
  IMAGE_DOS_HEADER = packed record
    e_magic    : Word;               // Magic number ("MZ")
    e_cblp     : Word;               // Bytes on last page of file
    e_cp       : Word;               // Pages in file
    e_crlc     : Word;               // Relocations
    e_cparhdr  : Word;               // Size of header in paragraphs
    e_minalloc: Word;                // Minimum extra paragraphs needed
    e_maxalloc: Word;                // Maximum extra paragraphs needed
    e_ss       : Word;               // Initial (relative) SS value
    e_sp       : Word;               // Initial SP value
    e_csum     : Word;               // Checksum
    e_ip       : Word;               // Initial IP value
    e_cs       : Word;               // Initial (relative) CS value
    e_lfarlc   : Word;               // Address of relocation table
    e_ovno     : Word;               // Overlay number
    e_res      : packed array [0..3] of Word;  // Reserved words
    e_oemid    : Word;               // OEM identifier (for e_oeminfo)
    e_oeminfo : Word;                // OEM info; e_oemid specific
    e_res2     : packed array [0..9] of Word;  // Reserved words
    e_lfanew   : Longint;            // File address of new exe header
  end;

  {
  TExeFileKind:
    The kinds of files recognised.
  }
  TExeFileKind = (
    fkUnknown,  // unknown file kind: not an executable
    fkError,    // error file kind: used for files that don't exist
```

```
    fkDOS,      // DOS executable
    fkExe32,    // 32 bit executable
    fkExe16,    // 16 bit executable
    fkDLL32,    // 32 bit DLL
    fkDLL16,    // 16 bit DLL
    fkVXD       // virtual device driver
  );

function ExeType(const FileName: string): TExeFileKind;
  {Examines given file and returns a code that indicates the type of
  executable file it is (or if it isn't an executable)}
const
  cDOSRelocOffset = $18;  // offset of "pointer" to DOS relocation table
  cWinHeaderOffset = $3C; // offset of "pointer" to windows header in file
  cNEAppTypeOffset = $0D; // offset in NE windows header of app type field
  cDOSMagic = $5A4D;      // magic number for a DOS executable
  cNEMagic = $454E;       // magic number for a NE executable (Win 16)
  cPEMagic = $4550;       // magic nunber for a PE executable (Win 32)
  cLEMagic = $454C;       // magic number for a Virtual Device Driver
  cNEDLLFlag = $80        // flag in NE app type field indicating a DLL
var
  FS: TFileStream;              // stream to executable file
  WinMagic: Word;               // word containing PE or NE magic numbers
  HdrOffset: LongInt;           // offset of windows header in exec file
  ImgHdrPE: IMAGE_FILE_HEADER;  // PE file header record
  DOSHeader: IMAGE_DOS_HEADER;  // DOS header
  AppFlagsNE: Byte;             // byte defining DLLs in NE format
  DOSFileSize: Integer;         // size of DOS file
begin
  try
    // Open stream onto file: raises exception if can't be read
    FS := TFileStream.Create(FileName, fmOpenRead + fmShareDenyNone);
    try
      // Assume unkown file
      Result := fkUnknown;
      // Any exec file is at least size of DOS header long
      if FS.Size < SizeOf(DOSHeader) then
        Exit;
      FS.ReadBuffer(DOSHeader, SizeOf(DOSHeader));
      // DOS files begin with "MZ"
      if DOSHeader.e_magic <> cDOSMagic then
        Exit;
      // DOS files have length >= size indicated at offset $02 and $04
      // (offset $02 indicates length of file mod 512 and offset $04
      // indicates no. of 512 pages in file)
      if (DOSHeader.e_cblp = 0) then
        DOSFileSize := DOSHeader.e_cp * 512
      else
        DOSFileSize := (DOSHeader.e_cp - 1) * 512 + DOSHeader.e_cblp;
      if FS.Size <  DOSFileSize then
        Exit;
      // DOS file relocation offset must be within DOS file size.
      if DOSHeader.e_lfarlc > DOSFileSize then
        Exit;
      // We assume we have an executable file: assume its a DOS program
      Result := fkDOS;
      // Try to find offset of Windows program header
      if FS.Size <= cWinHeaderOffset + SizeOf(LongInt) then
        // file too small for windows header "pointer": it's a DOS file
        Exit;
      // read it
      FS.Position := cWinHeaderOffset;
      FS.ReadBuffer(HdrOffset, SizeOf(LongInt));
      // Now try to read first word of Windows program header
      if FS.Size <= HdrOffset + SizeOf(Word) then
        // file too small to contain header: it's a DOS file
        Exit;
      FS.Position := HdrOffset;
      // This word should be NE, PE or LE per file type: check which
      FS.ReadBuffer(WinMagic, SizeOf(Word));
      case WinMagic of
        cPEMagic:
        begin
          // 32 bit Windows application: now check whether app or DLL
```

```
          if FS.Size < HdrOffset + SizeOf(LongWord) + SizeOf(ImgHdrPE) then
            // file not large enough for image header: assume DOS
            Exit;
          // read Windows image header
          FS.Position := HdrOffset + SizeOf(LongWord);
          FS.ReadBuffer(ImgHdrPE, SizeOf(ImgHdrPE));
          if (ImgHdrPE.Characteristics and IMAGE_FILE_DLL)
            = IMAGE_FILE_DLL then
            // characteristics indicate a 32 bit DLL
            Result := fkDLL32
          else
            // characteristics indicate a 32 bit application
            Result := fkExe32;
        end;
        cNEMagic:
        begin
          // We have 16 bit Windows executable: check whether app or DLL
          if FS.Size <= HdrOffset + cNEAppTypeOffset
            + SizeOf(AppFlagsNE) then
            // app flags field would be beyond EOF: assume DOS
            Exit;
          // read app flags byte
          FS.Position := HdrOffset + cNEAppTypeOffset;
          FS.ReadBuffer(AppFlagsNE, SizeOf(AppFlagsNE));
          if (AppFlagsNE and cNEDLLFlag) = cNEDLLFlag then
            // app flags indicate DLL
            Result := fkDLL16
          else
            // app flags indicate program
            Result := fkExe16;
        end;
        cLEMagic:
          // We have a Virtual Device Driver
          Result := fkVXD;
        else
          // DOS application
          {Do nothing - DOS result already set};
      end;
    finally
      FS.Free;
    end;
  except
    // Exception raised in function => error result
    Result := fkError;
  end;
end;
```

Listing 16

In the *final part* we present a demo program that exercises the *ExeType* function before concluding the article.