Beating The System: Delphi Hails A CAB

by Dave Jewell

As you'll have gathered from the title, this month's column is all about accessing CAB files from Delphi. Back when the world was young, the only archive file format of any consequence was .ZIP, and possibly .ARC files if you're as long in the tooth as me! Then Microsoft decided they wanted an archive file format of their own and the CAB file was born. As you'll appreciate, CAB files are used extensively by Microsoft setup programs, particularly for new operating system installations.

Maybe you want to write a utility which scans a set of cabinet files for a particular item peeking inside each file as it goes? Or maybe you want to write a replacement for the excellent WinZip shareware program, and you wish to add CAB capabilities to your code? Either way, being able to access CAB files from a Delphi application is a useful capability.

A Cabinet Of Curiosities

Fortunately, manipulating CAB files isn't massively complicated. Microsoft document the internal file format of CAB files, and they even go into a lengthy explanation of the data compression algorithm that's used. Even so, it's easier to make use of the 32-bit CABINET.DLL file which contains all the necessary routines for CAB file manipulation. You can download Microsoft's CAB SDK from

```
http://msdn.microsoft.com/
  workshop/management/cab/
  cabdl.asp
```

By Microsoft standards, it's relatively small at around 500Kb.

Once you've installed the SDK, you'll see that the available API is neatly split into two groups of related functions. On the one hand, there are the FCI (File Compression Interface) routines, and on other hand there are the FDI (File Decompression Interface) calls. In this article, we'll be concentrating on the latter. The official set of FDI routines are: `FDICreate`, `FDIDestroy`, `FDIIsCabinet` and `FDICopy`.

The DLL also contains another routine, `FDITruncateCabinet`, which is undocumented by Microsoft and I haven't figured out what this routine does yet.

Contrary to what you might expect, `FDICreate` doesn't actually create a new CAB file. Instead, it creates what Microsoft refer to as an FDI context. All this really means is that, behind the scenes, the code inside CABINET.DLL allocates a chunk of memory used to store per-instance data. Here's the function prototype for `FDICreate`:

```
function FDICreate (pAlloc,
  pFree, pOpen, pRead, pWrite,
  pClose, pSeek: Pointer;
  cpuType: Integer; var erf:
  TERF): THandle; cdecl;
```

As you can see, `FDICreate` takes rather a lot of parameters! The design of the CABINET.DLL API is heavily based around application callback routines, and this is the purpose of the first seven parameters to `FDICreate`. Briefly, these routines must provide memory allocation and de-allocation capabilities, and the ability to open, read, write and close files, together with the ability to seek to a specific file position. The format of these routines will be discussed a little later. The next parameter to `FDICreate`, `cpuType`, is a hangover from an earlier 16-bit version of the CAB libraries: it's ignored by the 32-bit code. Finally, the `erf` parameter points to a data structure of type `TERF` which provides error code information back to the caller.

In a similar vein, the `FDIDestroy` routine is used to destroy an existing FDI context. It takes a single parameter, the 'handle' (it's actually just a pointer) to the context which was created by `FDICreate`. It's responsible for freeing any internal data structures and closing any internal files associated with the FDI context:

```
function FDIDestroy (h:
  THandle): Bool; cdecl;
```

The next routine, `FDIIsCabinet`, is used to determine whether a specific file is a valid CAB file. If so, it returns information relating to the archive in a data structure of type `TFDICabinetInfo`. The function prototype for the routine is:

```
function FDIIsCabinet(
  h: THandle; fd: Integer;
  var info: TFDICabinetInfo):
  Bool; cdecl;
```

The first parameter, `h`, is an FDI context handle obtained from `FDICreate`. The second parameter, `fd`, is a file handle which is used to identify the file we're interested in. Finally, `info` is used to return the aforementioned file information. One might reasonably expect that this routine would take a filename and internally open and close the file, but this isn't the case. The caller has to take responsibility for opening and closing the file. The format of the `TFDICabinetInfo` data structure is shown in Listing 1.

Briefly, the `cbCabinet` member corresponds to the size of the enclosing CAB file. The `cFiles` field indicates how many files are present and `setID` is an application-defined number associated with the CAB file. You might expect that `cFolders` tells us how many pathname entries are stored in the archive, but this isn't the case at all. In CAB-speak, a 'folder' is a decompression unit, ie a chunk of bytes inside the CAB file which contains one or more files, or parts of a file. To quote directly from the Microsoft documentation:

'A cabinet file contains one or more folders. A folder contains one or more (pieces of) files. A

folder is by definition a decompression unit, ie, to extract a file from a folder, all of the data from the start of the folder up through and including the desired file must be read and decompressed.'

Clear? Well, not terribly, no. The only point I'm making here is that a CAB file folder has nothing to do with a folder in the normal sense of the word. A better word might have been cluster or bucket.

Like ZIP and RAR archives, CAB files have been designed so that a single logical archive can be spread across multiple physical files. This 'disk-spanning' feature means you can have a single 10Mb CAB file spread over a number of 1.44Mb floppies. To support this, the `iCabinet` field specifies the number of this CAB file in a logical set. In the same way, the `hasPrev` and `hasNext` fields (which are Boolean quantities despite being stored as 32-bit integers) indicate whether a file contained in this CAB is partially stored in the previous and next CAB files in the set.

The `fReserve` field requires a little more explanation. As with `hasPrev` and `hasNext`, it's simply a Boolean field indicating whether the archive contains a reserved area. CAB files can include reserved areas within the file for special purposes such as code-signing (as suggested by the Microsoft documentation) and so forth. Reserved areas can be stored in the CAB file on a per-cabinet basis (ie there's only one of them!), on a per-folder basis (but remember the foregoing remarks: we're talking decompression units, not pathnames), or even on a per-datablock basis. Typically, only the per-cabinet option is likely to be encountered.

The last FDI call, `FDICopy`, is the real meat of the story. This is the routine that actually extracts files from the CAB file, and more besides. The prototype for this function is given below:

```
function FDICopy(
  h: THandle; CabName, CabPath:
  PChar; Flags: Integer;
  pNotify, pEncrypt, pUser:
  Pointer): Bool; cdecl;
```

```
TFDICabinetInfo = record
  cbCabinet: Integer;  // size of the archive
  cFolders: Word;      // number of folders
  cFiles: Word;        // number of files
  setID: Word;         // application-defined magic #
  iCabinet: Word;      // number of cabinet in set
  fReserve: Integer;   // has reserved area?
  hasprev: Integer;    // chained to previous?
  hasnext: Integer;    // chained to next?
end;
```

➤ *Listing 1*

As ever, the `h` parameter specifies an FDI context. The next two parameters, `CabName` and `CabPath`, are used to specify the location of the CAB file that we're working with. `CabName` specifies the actual file while `CabPath` is the pathname of the file. Now why didn't Microsoft do what everyone else does and just put the pathname and filename into a single string?

The `Flags` parameter is essentially unused and can be set to zero. The `pNotify` parameter is used to point to an application-supplied notification routine which is called during the copy operation. Finally, the `pEncrypt` parameter is reserved for encryption enhancements and should be set to nil. The `pUser` field takes any arbitrary value which is passed on to the notification routine.

The FDI Notification Routine

The really important thing here is the notification routine. Rather counter-intuitively, the notification routine gets called for each file that's about to be extracted, giving the application an opportunity to perform the extraction or reject it. Thus, the `FDICopy` call is essentially the only way of enumerating the contents of a cabinet, because the notification routine is called for every file inside the archive.

The function prototype for the notification routine looks like this:

```
function FDINotify(
  NotifyType: Integer;
  var Info: TFDINotification):
  Integer; cdecl;
```

This might be a good time to point out the importance of those `cdecl` attributes on all the callback routines used by the CAB API. As I've already observed, the original design of this API goes back to 16-bit days and the `cdecl` calling convention is pretty antediluvian.

In the Windows API, almost all callback routines should be defined as `stdcall`. These routines are a rare exception. I should also confess that, for readability and clarity, I've been changing the names of Microsoft's identifiers as I've gone along: if you look in the FDI.H file (in the CAB SDK) you'll see they define the notification routine as taking two parameters called `fdint` and `pfdin`. Enough said.

The `NotifyType` parameter tells us what type of notification is being received. This can be one of those shown in Listing 2.

Briefly, the `ncCABInfo` code provides general information about the CAB archive. When this notification is received, the various fields in the `TFDINotification` data structure (discussed below) will contain information about the path/name of the next cabinet file, the number of the current cabinet in the set, and so on. The `ncPartialFile` notification is triggered if the first file in a cabinet is a continuation from a previous cabinet. Similarly, the `ncNextCabinet` notification is triggered whenever the decompression process moves to the next CAB. The `ncCopyFile` notification occurs when the decompressor is asking the application whether a certain file should be decompressed, and `ncCloseFileInfo` occurs when the file has been decompressed and needs to be closed; at this point, the onus is on the application to close the file and update the file's modification date/time using the values provided in the `TFDINotification` data structure.

The `ncEnumerate` notify code has been listed for completeness, but the CAB SDK has no information on how to use it, or what it does.

For each of the notification codes in Listing 2, the contents of

the Info data structure are filled out with information whose meaning depends on the exact notify code that we're dealing with.

At this point, things are starting to sound messy, aren't they? Fortunately, we can substantially simplify things if we make the design decision that we're not going to handle disk-spanned CAB files, not unreasonable given the size of today's storage devices.

The fact is, only two of the aforementioned notification codes are especially useful: ncCopyFile and ncFileClose. With these, we can enumerate the contents of a CAB and perform decompression. Another benefit of simplifying things in this way is that most of the fields in the TFDINotification record have the same meaning for both notification codes. The (simplified) record definition for TFDINotification is in Listing 3.

The first field, FileSize, gives us the uncompressed size of the current file. There is not, as far as I can tell, any way of determining the compressed size of a file using the existing API. You should note carefully that FileSize is only valid if the notification code is ncCopyFile. For ncFileClose, this is used as a flag which indicates whether or not the application should attempt to execute the freshly decompressed file. This is presumably for use by installer applications; we effectively ignore this 'auto-run' flag.

The next field, FileName, obviously gives us the name of the file. The next two fields, psz2 and psz3, are unused by both of the notification codes we're interested in. Next comes AppValue, which corresponds to the pUser argument to FDICopy. We can use this value to recover the instance value of a Delphi component from within our notification handler. The fd field is only relevant for ncFileClose: it's the handle to the freshly decompressed file which should be closed by the application.

Date, Time and Attribs all correspond to the usual MSDOS 16-bit file system properties, while the next three fields relate to which cabinet/folder we're currently decompressing. Finally, FDIError returns an error code, if any.

➤ *Listing 3*

```
TFDINotification = record
  FileSize: Integer;  // uncomp size of the file (ncCopyFile only)
  FileName: PChar;    // name of a file in the CAB
  psz2: PChar;
  psz3: PChar;
  AppValue: Pointer;  // application supplied value
  fd: Integer;        // file handle
  Date: Word;         // file's 16-bit FAT date
  Time: Word;         // file's 16-bit FAT time
  Attribs: Word;      // file's 16-bit FAT attributes
  setID: Word;        // application-defined magic #
  iCabinet: Word;     // number of this CAB
  iFolder: Word;      // number of current 'folder'
  FDIError: Integer;  // error code, if any
end;
```

The CABAPI Unit

Let's start to bring this together. Rather than writing a component that directly talks to the CAB API (as exported by CABINET.DLL), I decided to introduce an intermediate unit called CABAPI.PAS, the interface for which is shown in Listing 4 (the full source is on the disk: it's just too long to print).

The CABAPI unit wraps the somewhat idiosyncratic FDI routines with a number of easy-to-call routines which are far more convenient to use than the Microsoft equivalents. I did things this way because, after all, Microsoft do provide the byte format of CAB files, and I reasoned that some enterprising individual might want to eliminate the need for CABINET.DLL altogether, simply using Delphi stream I/O to access the CAB file directly and re-implementing the compression/decompression algorithms in Delphi. If you do alter CABAPI.PAS in this way, presenting the same interface to the outside world, then

```
ncCABInfo  = 0;          // General information about cabinet
ncPartialFile  = 1;      // First file in cabinet is continuation
ncCopyFile  = 2;         // File to be copied
ncCloseFileInfo  = 3;    // close the file, set relevant info
ncNextCabinet  = 4;      // File continued to next cabinet
ncEnumerate  = 5;        // Enumeration status
```

➤ *Listing 2*

any higher level code will be unaffected.

As you can see from Listing 4, CABAPI exports just a few simple routines, all of which are prefixed by the letters CAB. The first routine, CABIsFile, is used to determine whether or not a specific file is a valid CAB archive, returning True or False as appropriate. Similarly, CABIsMultiPart can be used to determine if a specific CAB archive is a multi-part (disk spanning, if you prefer) file. As I said, I don't support disk spanning archives in this code, but it's useful to be able to recognise them.

CABGetFileCount does exactly what it says on the tin, returning a count of the number of files contained within a specified CAB file while the CABGetFileList routine can be used to populate a TStringList object with a list of the files in the CAB. I could have used additional interface routines to determine the size of each file and its corresponding modification date/time, but I used one of my favourite techniques (it's easy and efficient), encoding all the pertinent information into a single string. So, a typical entry in the returned string list might look like this:

```
1394bus.sys|45568|688592763
```

This tells us that the archive contains a file, 1394bus.sys, which has an uncompressed size of 45568 bytes and a DOS timestamp of 688592763. This timestamp can be converted into the infinitely more useful TFileTime format by using the FileDateToDateTime routine.

Finally, the CABExtractFile routine is used to extract a single designated file from a CAB archive, whereas the CABExtractMultipleFiles extracts all the files which are named in the string list passed to the routine. You will notice that none of the interface routines 'ex-

```
unit CABAPI;
interface
uses
  Classes;
function  CABIsFile(const CABFileName: String): Boolean;
function  CABIsMultiPart(const CABFileName: String): Boolean;
function  CABGetFileCount(const CABFileName: String): Integer;
procedure CABGetFileList(const CABFileName: String; List: TStringList);
procedure CABExtractFile(const CABFileName, DestPath, FileName: String);
procedure CABExtractMultipleFiles(const CABFileName, DestPath: String;
   List: TStringList);
implementation
{... see disk for full source code...}
```

➤ *Listing 4*

ported' from CABAPI contain any hint of implementation details specific to the FDI API. That's just as it should be.

The implementation part of the unit (check the file on the disk) begins with assorted declarations, including the aforementioned TFDICabinetInfo and TFDINotification data structures which we've already discussed. This is followed by a number of private variables, including function pointers for each of the FDIxxxx routines defined in the CABINET.DLL library. These function pointers are automatically initialised in the initialization part of the unit by using GetProcAddress on the cabinet library. If you're feeling especially paranoid, you might like to check if any of these function pointers are Nil before using them, but I haven't bothered; one gets the impression that Microsoft doesn't update this particular DLL very often...

As I mentioned earlier, the FDI routines work by using numerous application callbacks, some of which are responsible for implementing memory management and file I/O. In their documentation, Microsoft boast that this means CABINET.DLL doesn't make use of any C runtime calls at all, implying that it's a deeply cool feature. *Au contraire*, it strikes me as a pain in the neck, since it increases the amount of support code needed in the application. Thus, Listing 4 continues with MyAlloc, MyFree, MyOpen, MyClose, MyRead, MyWrite and MySeek! As noted previously, these *must* be cdecl routines. If not, it'll be tears before bedtime.

Next comes NewFDIContext, a little helper routine whose job is to ensure that we only ever have to make the ghastly FDICreate call once! The fun stuff starts with CABIsFile which demonstrates how to create an FDI context using NewFDIContext. The code first creates the context, then uses MyOpen (and, thus, the ancient _lopen routine!) to create a read-only file handle to the archive, and then calls FDIIsCabinet to do the actual work of recognising the file type. The handle is then closed, the FDI context destroyed, and the result is returned to the caller.

As an aside, you'll notice that CABIsFile is extensively used in the other CABAPI files, and you might suspect that this makes things slow and inefficient. In fact, that's not the case. Internally, FDIIsCabinet and FDICreate do very little work and calling them once or twice more than strictly necessary is neither here nor there. In my tests, I've been using a 51Mb CAB file: retrieving a list of the files contained inside this archive is almost instantaneous.

Moving on, the CABIsMultiPart routine relies on the fact that CABIsFile initialises the contents of the Info record, checking to see if the internal cabinet number is greater than zero, or if the file has a forward or backward link to another cabinet. Yes, I know Info looks suspiciously like a global variable, but give me a break! ☺ It's only visible within the implementation part of the unit and without it the code would be somewhat more convoluted.

Again, CABGetFileCount makes use of the Info data structure, retrieving the file count from the cFiles field. As I mentioned earlier, the cFolders count has got absolutely nothing to do with folders or sub-directories in the normal sense, and therefore I have not bothered to implement a CABGetFolderCount routine, but if you wanted to do this, it'd be dead easy.

Things get a bit more complex when it comes to determining the contents of the CAB file. The CABGetFileList routine gets passed a TStringList object, which is first cleared of any existing contents. Next, CABIsFile is called to validate the archive and a new FDI context is created. Finally, the FDICopy routine is called, using the ExtractFileName and ExtractFilePath routines to carve up the pathname into its directory and filename components. Notice that the GetFileListCallback routine is used as the notification procedure, and we pass the TStringList object as the final parameter to FDICopy.

Inside the notification routine, this means that we can retrieve the string list object by referencing the AppValue field of the TFDINotification data structure. If the notification type is ncCopyFile (the DLL is requesting whether the current file should be decompressed) then we retrieve the name of the file and concatenate the uncompressed file size and DOS date/time as two ASCII strings using '|' as a separator character. The resulting string is then added to the string list object. Notice that in all cases we return zero as the function result. This tells CABINET.DLL that we don't actually want to decompress this file.

The structure of the CABExtractFile routine is very similar. As before, the archive is validated and an FDI context created. We save the destination path string into a variable where it can be accessed from the notification routine which, this time round, is called FileExtractSingleCallback. You'll also notice that we pass the name of the file to be extracted as the last parameter to FDICopy. Inside the notification routine, we retrieve this from Info.AppValue.

If we've got a notification type of ncCopyFile, then the code checks to see if the current filename matches the target filename. If so, then the routine builds a full

pathname for the file to be decompressed, and this is passed to the `_lcreat` routine. This file handle is returned as the function result. In a nutshell, CABINET.DLL looks at the notification procedure function result and (as we've seen) if zero is returned, then it's interpreted as 'skip file'. If it's less than zero it's interpreted as an error. Anything else is taken to be a file handle to use for decompressing the file.

However, if we get a `ncCloseFileInfo` notification the code is requesting the application close the file and set its timestamp. Here we're only extracting one file so only one `ncCloseFileInfo` notification should ever be received. So, it shouldn't be necessary to check for a filename match, but I've included one for good measure. The `DosDateTimeToFileTime` routine is used to convert the date/time to a format compatible with `SetFileTime`, and the file time is updated. Finally, the file is closed. Notice that, regardless of whether or not we got a file match, we always return a value of one in response to an `ncCloseFileInfo` code. This is interpreted as 'success' by the library code. Anything else aborts the entire `FDICopy` operation.

Lastly, there's the `CABExtractMultipleFiles` routine and its sidekick, the `FileExtractMultipleCallback` notification routine. This time, we pass a string list instance to the notification routine and, for each received notification, we use the `IndexOf` method to verify that the current file exists in the list of files to be extracted. If it doesn't, the file is skipped. Here again, the filename check in the `ncCloseFileInfo` case is probably redundant, as we'll probably only get this notification for files that we've decompressed, but the odd sanity check here and there does no harm at all.

Conclusions

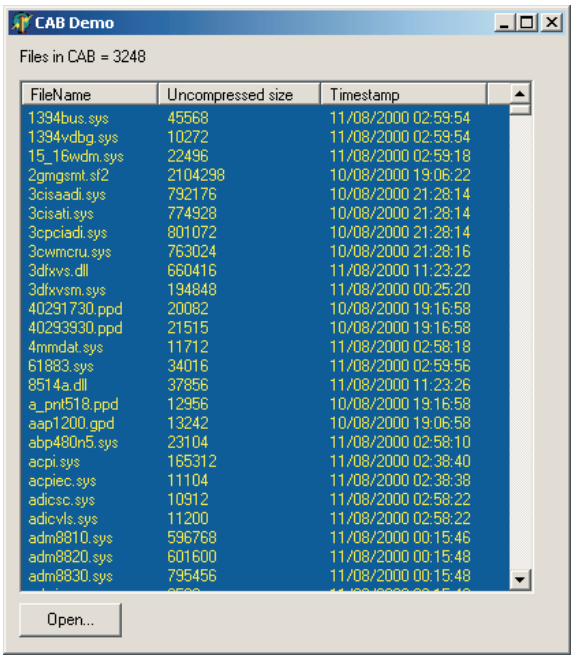
As ever, the proof of the pudding is in the eating. Included on the disk (in the file CabFile.pas) is a minimalist Delphi class, `TCabinet`, that can be used to browse the contents of a CAB file. I haven't included data decompression capabilities since you may well have your own idea about how you want this to work. Using the `CABExtractFile` and `CABExtractMultipleFiles` methods of the `TCabinet` class, you can easily implement method calls to extract a single file, a specific set of files, or everything in the archive. Listing 5 is a snippet from my CAB demo (also on the disk) showing how easy it is to fill a `TListView` control with the contents of a CAB archive. This is shown working in Figure 1.

One thing you should be aware of is the fact that stored filenames (within a CAB file) can contain pathnames ahead of the filename itself. This is why the decompression code in CABAPI is written the way it is. You might pass a destination pathname of c:\wombat, but if the filename entry in the CAB file is com\ms\util\Atom.class, this pathname will obviously be appended to the c:\wombat directory. If you want all-singin', all-dancin' functionality, you could always pass an additional `boolean` variable to `CABExtractFile` and `CABExtractMultipleFiles` which would tell them whether or not to honour embedded pathname information within the archive.

I haven't included any code for creating CAB files (the FCI routines) because I suspect that you will be much more interested in programmatically extracting information from CAB files than in creating them yourself. However, if you have a requirement to do this, all the information is in the Microsoft CAB SDK (see the earlier download URL) and it's no more difficult to implement than decompression.

Have fun!

➤ *Figure 1: My CAB browser.*




```
procedure TForm1.Button1Click(
  Sender: TObject);
var
  Idx: Integer;
  Item: TListItem;
begin
  if OpenDialog.Execute then begin
    FileList.Items.Clear;
    cab.CABFileName :=
      OpenDialog.FileName;
    FileCount.Caption :=
      'Files in CAB = ' +
      IntToStr(cab.FileCount);
    FileList.Items.BeginUpdate;
    try
      for Idx := 0 to
        cab.FileCount-1 do begin
        Item := FileList.Items.Add;
        Item.Caption := cab[Idx];
        Item.SubItems.Add(
          cab.FileSize[Idx]);
        Item.SubItems.Add(
          cab.FileDate[Idx]);
      end;
    finally
      FileList.Items.EndUpdate;
    end;
  end;
end;
```

➤ *Listing 5*

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level work. He is Technical Editor of *Developers Review* which is also published by iTec. Email Dave at TechEditor@itecuk.com