What a drag: Dragging a virtual file (HGLOBAL edition)

devblogs.microsoft.com/oldnewthing/20080318-00

March 18, 2008

Raymond Chen

Now that we've gotten our feet wet with simple data objects, let's do something a smidge more complicated but extremely useful: Dragging a virtual file. There are many ways of doing this, but I'll start with the simplest one, where the virtual file is represented as a block of memory.

Remember, the subtitle of this series is "It's the least you could do." There are a lot of optional things you can (and even should) do, but I'm going to start with the absolute minimum.

Take the drag/drop program we've been working on for a while and make the following changes. First, change the enumeration of data types:

```
enum {
  DATA_FILEGROUPDESCRIPTOR,
  DATA_FILECONTENTS,
  DATA_NUM,
  DATA_INVALID = -1,
};
```

The clipboard format central to dragging a virtual file is the `FILEGROUPDESCRIPTOR`, which describes how many files are being dragged and various information about them. For each file in the file group descriptor, you must provide the associated file contents, represented by the `CFSTR_FILECONTENTS` clipboard format.

```
CTinyDataObject::CTinyDataObject() : m_cRef(1)
{
  SetFORMATETC(&m_rgfe[DATA_FILEGROUPDESCRIPTOR],
               RegisterClipboardFormat(CFSTR_FILEDESCRIPTOR));
  SetFORMATETC(&m_rgfe[DATA_FILECONTENTS],
               RegisterClipboardFormat(CFSTR_FILECONTENTS),
               TYMED_HGLOBAL, /* lindex */ 0);
}
```

Initializing the file group descriptor entry is pretty much what you've seen before. Note that the structure is called `FILEGROUPDESCRIPTOR`, but the clipboard format is `CFSTR_FILEDESCRIPTOR` without the "group". This was probably originally a typographical error, but now we're stuck with it.

The file contents entry has a twist: The `lindex` is zero, not `-1`. The file contents clipboard format uses the `lindex` as a zero-based index which selects which virtual file the caller is talking about. Since we have only one virtual file, its index is zero.

As before, all the real work is in the heart of the data object, the `IDataObject::GetData` method.

```
HRESULT CTinyDataObject::GetData(FORMATETC *pfe, STGMEDIUM *pmed)
{
  ZeroMemory(pmed, sizeof(*pmed));

  switch (GetDataIndex(pfe)) {
  case DATA_FILEGROUPDESCRIPTOR:
  {
    FILEGROUPDESCRIPTOR fgd;
    ZeroMemory(&fgd, sizeof(fgd));
    fgd.cItems = 1;
    StringCchCopy(fgd.fgd[0].cFileName,
                  ARRAYSIZE(fgd.fgd[0].cFileName),
                  TEXT("Dummy"));
    pmed->tymed = TYMED_HGLOBAL;
    return CreateHGlobalFromBlob(&fgd, sizeof(fgd),
                               GMEM_MOVEABLE, &pmed->hGlobal);
  }

  case DATA_FILECONTENTS:
    pmed->tymed = TYMED_HGLOBAL;
    return CreateHGlobalFromBlob("Dummy", 5,
                               GMEM_MOVEABLE, &pmed->hGlobal);
  }

  return DV_E_FORMATETC;
}
```

When the caller asks for the file group descriptor, we fill out a `FILEGROUPDESCRIPTOR` structure, taking care to zero out the bytes we don't care about before filling in the goodies, so as to avoid an information disclosure vulnerability. As I noted, we're starting by doing the absolute minimum necessary, which in the case of virtual file transfer consists merely of specifying how many virtual files there are and their names.

When the caller asks for the contents of file zero (which is the only one we have), we produce a five-byte chunk of memory with the word "Dummy" in it.

Run this program, and drag the invisible object out of the client area and drop it onto the desktop, say. Woo-hoo, your virtual file has been copied to the desktop and has turned into a real file. (You can even drop it onto an Outlook message composition window, and it will appear as an attachment!)

There are still some issues here, but we've at least done the absolute minimum necessary to drag a virtual file represented by a block of memory. Let's look at some of those optional features, some of which turn out to have significant consequences for both you and the end user.

First of all, you may have noticed that the Dummy file that is created might have some garbage bytes at the end. I say "might" because the presence of said garbage bytes depends on how the heap manager feels. If all you provide is an `HGLOBAL`, then the only indication of the size of the memory block is what comes out of the `GlobalSize` function. But the size returned by the `GlobalSize` function need not be equal to the size passed to `GlobalAlloc`; the only guarantee is that it is at least as big as the size requested. It might be bigger as the result of internal heap bookkeeping such as rounding all allocations up to the nearest multiple of 16 bytes. If such rounding has occurred, then the Dummy file that gets created will contain those extra garbage bytes.

To avoid this problem, set the `FD_FILESIZE` flag and specify the exact file size in the `nFileSizeLow` and `nFileSizeHigh` members:

Specifying the file size in the `FILEGROUPDESCRIPTOR` also benefits the end user, because it gives information to the file copy progress bar as to how many bytes it should expect to receive. Without it, the progress bar doesn't know how many bytes are in that virtual file. It eventually finds out when it requests the file contents, but it learns that from each file as it is copied. The progress dialog doesn't have the opportunity to collect this information up front in order to provide meaningful progress feedback.

Another optional detail that you may wish to take advantage of is specifying file attributes and modification times in the `FILEGROUPDESCRIPTOR`. For example, you might want to make the file hidden when it copied, or you might want to customize the last-modified time.

Let's do a few of these things. We'll specify the file size in the file group descriptor to avoid the garbage and improve the progress feedback, and we'll set the last-modified time to a specific date.

```
case DATA_FILEGROUPDESCRIPTOR:
{
  FILEGROUPDESCRIPTOR fgd;
  ZeroMemory(&fgd, sizeof(fgd));
  fgd.cItems = 1;
  fgd.fgd[0].dwFlags = FD_FILESIZE | FD_WRITESTIME;
  fgd.fgd[0].nFileSizeLow = 5;
  fgd.fgd[0].ftLastWriteTime.dwLowDateTime = 0x256d4000;
  fgd.fgd[0].ftLastWriteTime.dwHighDateTime = 0x01bf53eb;
  StringCchCopy(fgd.fgd[0].cFileName,
                ARRAYSIZE(fgd.fgd[0].cFileName),
                TEXT("Dummy"));
  pmed->tymed = TYMED_HGLOBAL;
  return CreateHGlobalFromBlob(&fgd, sizeof(fgd),
                               GMEM_MOVEABLE, &pmed->hGlobal);
}
```

Now, when you drop the file, it will not have any garbage bytes at the end, and the timestamp will be midnight January 1, 2000 UTC. (You won't notice any improvement in the progress bar since the file is so small.)

Even though we haven't done very much, it may already be enough for many people who simply want to allow users to drag an object out of their program and drop it into an Explorer window to create a corresponding file, provided that an `HGLOBAL` is a convenient format for the file contents. This is suitable for small files, but as files get bigger, the fact that you have to generate the entire file at once can become expensive. We'll look at one alternative next time.

Raymond Chen

Follow