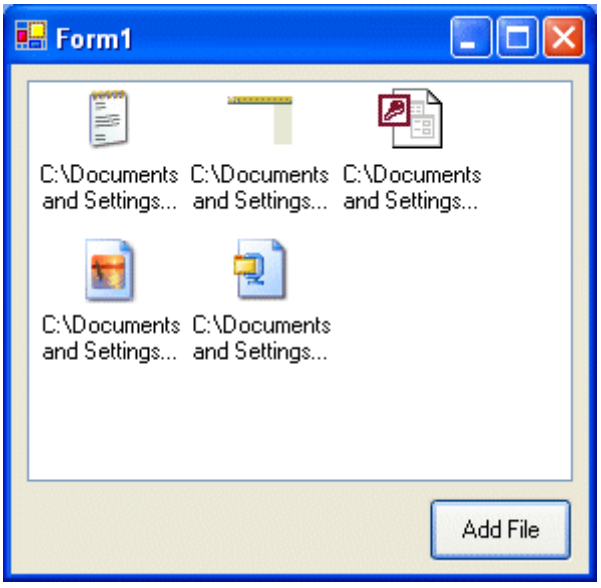
Obtaining (and managing) file and folder icons using SHGetFileInfo in C#

Paul Ingles, 3 Jul 2002

Article showing how to read file and folder icons from C#, and then building a management class to maintain file icons in up to two ImageList objects.

Download source files - 5 Kb

Download demo project - 18 Kb



Introduction

This article is based upon code from the MSDN Cold Rooster Consulting case study. Included in part of the CRC Rich Client is support for file icons, something I wanted to do myself. This article and classes are the result of my attempts to use the MSDN code

in my own application.

The MSDN article explains how functions from Shell32 and User32 were wrapped in detail, but here's a short clip from the article:

> "Interoperability with interfaces exposed by COM objects and the .NET Framework is handled via a proxy called the Runtime Callable Wrapper (RCW). The majority of the marshalling work is handled automatically by the .NET Framework.
>
> C-style functions exported from an unmanaged library are a different matter. They are not wrapped automatically because information about the parameters required by the function is not as rich as the information provided by a COM type library. To call C-style functions exported from an unmanaged library, such as the Microsoft Windows® Shell32 API, you use the Platform Invocation Services (PInvoke)..."

The code was left largely unchanged from the original article, although only `SHGetFileInfo` and `DestroyIcon` were retained.

I personally found it quite hard to incorporate the MSDN code in my own application and after a few hours of wrestling with the masses of code and still getting errors when trying to build my own project I decided I would try and build up some classes around the Shell32 and User32 wrapped functions that I could use myself.

After looking back at the MSDN article the architecture of my solution and theirs is pretty similar, however I found it easier to develop my own classes and incorporate them into my own project.

This article explains how I modified the MSDN article's code so that it can be used to retrieve icons as a stand-alone class in the form of the `IconReader` type, and then the `IconListManager` type which can be used to maintain ImageLists of file icons. It shields you from having to call the `IconReader` type's members directly, instead adding file icons to specified image lists. To prevent icons for the same file type being added more than once, a `HashTable` is used to store the file's extension at the time of adding the icon to the `ImageList`.

Top Level View

The end result is two classes which make use of .NET's Interoperability to call the Win32 API to obtain icons for specified files and or folders. The `IconReader` class enables the caller to obtain icons directly (which may be all you need). However, an `IconListManager` class is then created which maintains icons within two `ImageList` types and shields you from retrieving icons directly.

A couple of additional enumerations were also included to make the library a little more .NET-esque.

IconReader - GetFileIcon Explanation

GetFileIcon is used to obtain icons for files, and uses three parameters:

- `name` - Complete file and path names to read.
- `size` - Whether to obtain 16x16 or 32x32 pixels, uses the IconSize enumeration.
- `linkOverlay` - Specify whether the returned icon should include the small link overlay.

It is a static member function since it doesn't need to store any state, and is intended primarily as an added layer of abstraction. If I needed to obtain a file's icon in the future (and not store it in an ImageList etc.) then I could do so using this class. Once I had a type that wrapped up the necessary API functions to obtain file icons I would then build another type to manage large and small ImageLists that would enable me to make a single call to add an icon, and if it was already added, return the index that the icon was in the `ImageList`.

Hide Shrink ▲ Copy Code

```
public static System.Drawing.Icon GetFileIcon(string name, IconSize size,
                                              bool linkOverlay)
{
    Shell32.SHFILEINFO shfi = new Shell32.SHFILEINFO();
    uint flags = Shell32.SHGFI_ICON | Shell32.SHGFI_USEFILEATTRIBUTES;

    if (true == linkOverlay) flags += Shell32.SHGFI_LINKOVERLAY;


    /* Check the size specified for return. */
    if (IconSize.Small == size)
    {
        flags += Shell32.SHGFI_SMALLICON ; // include the small icon flag
```

```
    }
    else
    {
        flags += Shell32.SHGFI_LARGEICON ;  // include the large icon flag
    }

    Shell32.SHGetFileInfo( name,
        Shell32.FILE_ATTRIBUTE_NORMAL,
        ref shfi,
        (uint) System.Runtime.InteropServices.Marshal.SizeOf(shfi),
        flags );


    // Copy (clone) the returned icon to a new object, thus allowing us
    // to call DestroyIcon immediately
    System.Drawing.Icon icon = (System.Drawing.Icon)
                        System.Drawing.Icon.FromHandle(shfi.hIcon).Clone();
    User32.DestroyIcon( shfi.hIcon ); // Cleanup
    return icon;
}
```

Firstly, a SHFILEINFO structure is created from the following definition:

Hide Copy Code

```
[StructLayout(LayoutKind.Sequential)]
public struct SHFILEINFO
{
    public const int NAMESIZE = 80;
    public IntPtr hIcon;
    public int iIcon;
    public uint dwAttributes;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst=MAX_PATH)]
    public string szDisplayName;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst=NAMESIZE)]
    public string szTypeName;
};
```

The SHFILEINFO struct includes an attribute to define a formatted type, which is "a structure or class member annotated with the StructLayoutAttribute to ensure predictable layout information to its members." This ensures that the unmanaged code we call receives the struct as it is intended - i.e. in the order that the members are declared. More details on passing structures are on MSDN

Once the SHFILEINFO struct is created, flags are then set specifying how SHGetFileInfo should behave and what type of icon to retrieve. The code for this part is pretty self explanatory.

Once the various parameters have been finalised, its time to call Shell32.SHGetFileInfo. The code for the Shell32 class was written entirely as part of the MSDN article, and so I cannot take credit for it (and so if you would like more info on how this was done I recommend you take a look at the original CRC article). However as a quick example of how simple it is the unmanaged function is declared as:

Hide Copy Code

```
DWORD_PTR SHGetFileInfo( LPCTSTR pszPath,
   DWORD dwFileAttributes,
   SHFILEINFO* psfi,
   UINT cbFileInfo,
   UINT uFlags
);
```

Which translated to managed code is:

Hide Copy Code

```
[DllImport("Shell32.dll")]
public static extern IntPtr SHGetFileInfo(
    string pszPath,
    uint dwFileAttributes,
    ref SHFILEINFO psfi,
    uint cbFileInfo,
    uint uFlags
);
```

Once the `SHFILEINFO` struct had been populated, its then time to get the `hIcon` that points to the file's icon. This `hIcon` can then be passed as a parameter of `System.Drawing.Icon.FromHandle()` which returns the file's icon. After looking through the original code I noticed that a `DestroyIcon` function was also included, so I looked it up on MSDN and found it was used to (funnily enough) "destroys an icon and frees any memory the icon occupied". I decided it would be a good idea to do this immediately after the icon had been retrieved (since this class was intended to be used in any number of ways). The icon could then be cleaned up as soon as necessary by the GC, or stored in an `ImageList`. If this isn't necessary then please let me know.

Originally, I didn't use the `Clone` member function to obtain a copy of the icon, and just left it at `FromHandle`. However, the call to `DestroyIcon` immediately after then meant that the returned `Icon` was now useless and generated an exception. Since I thought this class could be used in any number of ways, I decided to stick with a static call which would obtain a copy of the icon, and then call `DestroyIcon` immediately after. It suited what I needed to do, and this was something different to the original MSDN code.

The function then returns with the specified icon.

IconReader - GetFolderIcon

The code for `GetFolderIcon` is very similar to `GetFileIcon`, except that the `dwFileAttributes` parameter is passed `Shell32.FILE_ATTRIBUTE_DIRECTORY` as opposed to `Shell32.FILE_ATTRIBUTE_NORMAL` for files.

It also requires fewer parameters, specifying whether a large or small icon is desired, and whether to retrieve the open or closed version.

IconListManager - Overview

`IconListManager` was created after I had produced `IconReader`, and was designed to manage up to two `ImageList` types with file icons. The type requires itself to be instantiated, and can be passed up to two parameters when constructed - specifying `ImageList` objects.

Firstly, there are some member fields which are declared as:

Hide Copy Code

```
private Hashtable _extensionList = new Hashtable();

//will hold ImageList objects
private System.Collections.ArrayList _imageLists = new ArrayList();
private IconHelper.IconReader.IconSize _iconSize;

//flag, used to determine whether to create two ImageLists.
bool ManageBothSizes = false;
```

The `HashTable` is used to contain a list of extensions that have been added to the `ImageList`. We only need to store each icon once, so a `HashTable` can be used to look up whether an extension exists, if so, whereabouts the icon is in the `ImageList`.

The `ArrayList` is used to contain references to `ImageList` objects, this is so that two constructors can be provided. The first allows the caller to manage a single `ImageList` with a specified size. The second constructor uses two `ImageList` parameters, allowing the type to manage both large and small icons.

The first constructor looks like:

Hide Copy Code

```
public IconListManager(System.Windows.Forms.ImageList imageList,
                       IconReader.IconSize iconSize )
{
    // Initialise the members of the class that will hold the image list we're
    // targeting, as well as the icon size (32 or 16)
    _imageLists.Add( imageList );   // add ImageList reference to the array list
    _iconSize = iconSize;           // specify what size to retrieve
}
```

This stores icons only for a single size in a single `ImageList`.

The second constructor (which fill allow the type to be used for both large and small icons) looks like:

Hide Copy Code

```
public IconListManager(System.Windows.Forms.ImageList smallImageList,
                       System.Windows.Forms.ImageList largeImageList )
{
    //add both our image lists
    _imageLists.Add( smallImageList );
    _imageLists.Add( largeImageList );

    //set flag
    ManageBothSizes = true;
}
```

This adds both `ImageList` types to the `ArrayList`, and then sets a flag specifying that calls to `IconReader` class's member functions should retrieve both sizes. Its not the neatest way to do it, but it worked, and if I have enough time I'll go through and tidy a few things up.

The class has a few internal functions which are used to make the code a little cleaner, the first of which is `AddExtension`. This adds a file extension to the `HashTable`, along with a number which is used to hold the icon's position in the `ImageList`.

`AddFileIcon` adds a file's icon to the `ImageList`, and forms the majority of the code for `IconListManager`:

Hide Shrink Copy Code

```
public int AddFileIcon( string filePath )
{
    // Check if the file exists, otherwise, throw exception.
    if (!System.IO.File.Exists( filePath ))
         throw new System.IO.FileNotFoundException("File    does not exist");

    // Split it down so we can get the extension
    string[] splitPath = filePath.Split(new Char[] {'.'});
    string extension = (string)splitPath.GetValue( splitPath.GetUpperBound(0) );

    //Check that we haven't already got the extension, if we have, then
    //return back its index
    if (_extensionList.ContainsKey( extension.ToUpper() ))
    {
        // it already exists
        return (int)_extensionList[extension.ToUpper()]; //return existing index
    }
    else
    {
        // It's not already been added, so add it and record its position.
        //store current count -- new item's index
        int pos = ((ImageList)_imageLists[0]).Images.Count;

        if (ManageBothSizes == true)
        {
            //managing two lists, so add it to small first, then large
            ((ImageList)_imageLists[0]).Images.Add(
                        IconReader.GetFileIcon( filePath,
                                                IconReader.IconSize.Small,
                                                false ) );
            ((ImageList)_imageLists[1]).Images.Add(
                        IconReader.GetFileIcon( filePath,
                                                IconReader.IconSize.Large,
                                                false ) );
        }
        else
        {
            //only doing one size, so use IconSize as specified in _iconSize.
            //add to image list
            ((ImageList)_imageLists[0]).Images.Add(
                        IconReader.GetFileIcon( filePath,
                                                _iconSize, false ) );
        }

        AddExtension( extension.ToUpper(), pos ); // add to hash table
        return pos;                               // return its position
    }
}
```

The code is pretty well covered through comments but works as follows. Firstly, it splits the `filePath` so that the extension can be obtained (string after the final period - ".", i.e. the string at the highest position in the array). Once this has been done, a check is done on the `HashTable` to determine whether that extension has already been added. If it has, then return the contents of the `HashTable` for the given key (the file extension). So, if "TXT" exists, the "TXT" key is looked up and the contents returned, which is the position of the icon in the `ImageList`.

If it doesn't exist in the `HashTable` it hasn't been added, so obtain the current count of items (and thus determine the index the new icon will be inserted at). Then, if it's managing both large and small `ImageList` objects, then call `GetFileIcon` twice. If it isn't for both sizes, then just retrieve the specified size icon.

Once this has been done, the extension can then be added to the `ImageList` with its position, and the position then returned to the caller. This position can then be used when adding icons to `ListView` or `TreeView` types when specifying the icon index.

`ClearList` is included in case its necessary to start over,

Hide Copy Code

```
public void ClearLists()
{
    foreach( ImageList imageList in _imageLists )
    {
        imageList.Images.Clear(); //clear current imagelist.
    }

    _extensionList.Clear(); //empty hashtable of entries too.
}
```

Firstly it iterates through the `ArrayList` and clears the respective `ImageList`, and then clears the `HashTable` that contained the file extensions.

That covers the classes. I had originally wanted to produce a FileIconImageList control that derived from `ImageList`. This would have incorporated the functionality that `IconListManager` did, but would have been a slightly neater way of doing it (i.e. instantiate an ImageList, and then call AddFileIcon to add the icon like with `IconListManager`). However, when I tried this I found I couldn't derive from `ImageList` and so this wasn't possible. Producing `IconListManager` was the next best thing I could do.

In the end, a calling application only needs to create an object of type `IconListManager`, pass it the `ImageList` references you are using, and then use the `AddFileIcon` method. I haven't yet added an `AddFolderIcon` member, since there are only a couple of folder icons (and they would probably go in a separate ImageList to file icons) the calls to obtain them could be made directly from `IconReader`. However, if this is something people would like added its very easy to do.

The demo application shows how to use the classes, and includes a `ListView` and `Button`. When you click the `Button` an`OpenFileDialog` is displayed. The filename is then retrieved, and the icon added to the `ListView`. The snippet below gives you the basic code. Note that I set color depth to 32-bit to ensure support for alpha channel smoothing.

Hide Shrink ▲ Copy Code

```
public class Form1 : System.Windows.Forms.Form
{
    private ImageList _smallImageList = new ImageList();
    private ImageList _largeImageList = new ImageList();
    private IconListManager _iconListManager;

    .
    .
    .

    public Form1()
    {
        //
        // Required for Windows Form Designer support
        //
        InitializeComponent();

        _smallImageList.ColorDepth = ColorDepth.Depth32Bit;
        _largeImageList.ColorDepth = ColorDepth.Depth32Bit;

        _smallImageList.ImageSize = new System.Drawing.Size( 16, 16 );
        _largeImageList.ImageSize = new System.Drawing.Size( 32, 32 );
        _iconListManager = new IconListManager( _smallImageList, _largeImageList );
```

```
        listView1.SmallImageList = _smallImageList;
        listView1.LargeImageList = _largeImageList;
    }
    .
    .
    .
    private void addButton_Click(object sender, System.EventArgs e)
    {
        OpenFileDialog dlgOpenFile = new OpenFileDialog();
        if(dlgOpenFile.ShowDialog() == DialogResult.OK)
        {
            listView1.Items.Add( dlgOpenFile.FileName,
                                 _iconListManager.AddFileIcon( dlgOpenFile.FileName ) );
        }
    }
```

Important Notes

It's taken me a long time to figure this out, but gave me real grief at one point. Windows XP introduced Visual Styles, enabling you to use icons with alpha channel blending to produce nice smooth icons. However, to include support for this you must include a manifest file. Without one, you get a really ugly black border. For more information on including visual styles support, you ought to read the MSDN Article "Using Windows XP Visual Styles With Controls on Windows Forms". As I said, I forgot to include a manifest and it drove me crazy for weeks.

Thanks

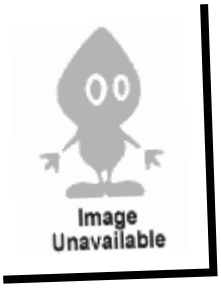
Well this is my first article to CodeProject (finally), although I've not been a registered member here long I've been a quiet lurking one, and even used CodeGuru in the good old days for my MFC learning. I'm not a massively accomplished programmer, but I hope this has been of help to you. Reading file icons is something I've noticed being mentioned a few times on the MS Newsgroups, and so the included classes should help you on your way.

If you have any questions about this article (particularly if I've done something in a bad way), please feel free to email me.

License

Share

About the Author



Paul Ingles
Web Developer
United Kingdom

I graduated with a first class BSc honours degree in a Computer Science/Management Science hybrid from Loughborough University. I live in London and currently work as a .NET Developer in central London.

I'm also currently blogging my progress at further developing my articles into commercial components. Read my blog here.

I've also recently started dabbling at digital photography, and digital re-touching, and developing small simple multiplayer games in Flash.

You may also be interested in...

Public, Private, and Hybrid Cloud: What's the difference?

A Solution Blueprint for DevOps

Iconizer

SAPrefs - Netscape-like Preferences Dialog

An ShGetFileInfo Wrapper Class

Generate and add keyword variations using AdWords API

Comments and Discussions

Add a Comment or Question

Search Comments

First Prev Next

There's black border in the icon
Member 11834968 14-Jul-15 14:34

Nice work
James IV 11-May-14 12:30

Get icon with file extension only
KianK7Gmail 27-Jan-13 17:14

Re: Get icon with file extension only
Meganubis 1-Nov-13 9:19

64-bit support
mike800 15-Jan-13 21:27

My vote of 5
Marc-Anthony Taylor 21-Sep-12 21:33

My vote of 5
Hex01 15-Aug-12 22:51

Good demo
George Hendrickson 17-Mar-12 2:36

How to get special directories icons .. ??
bboy_markor 29-Jan-12 18:20

Re: How to get special directories icons .. ??
James IV 11-May-14 12:29

How to work with window 7 with this code
vijaykumar107 10-Jul-11 20:50

Re: How to work with window 7 with this code [modified]
Danny DB 22-Jul-11 20:45

Re: How to work with window 7 with this code
vijaykumar107 22-Jul-11 23:06

Files With No Extensions SHow Folder Icon
scptech 28-Dec-09 2:25

Alpha Channel
Joel Hess 8-Dec-09 21:40

Re: Alpha Channel
DavidCinadr 10-Jun-10 3:49

Thank you :-)
Aybe 21-Sep-09 22:30

Unicode support
Member 4314808 26-May-09 7:22

Re: Unicode support
Member 4314808 26-May-09 22:46

hidden icons [modified]
hugopq 20-May-09 23:45

Does this code work with X64?
Bishman 11-Oct-08 1:45

License
Tim Dyck 19-Sep-08 3:16

Good article - with DestroyIcon mention!
T-C 31-Aug-08 20:29

MAX_PATH should be 260;
WuJunyin 14-May-08 23:39

Most of this is unnecessary
Dominick O'Dierno 29-Sep-07 0:36

Refresh 1 2 3 Next »

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.