IconLib - Icons Unfolded (MultiIcon and Windows Vista supported)

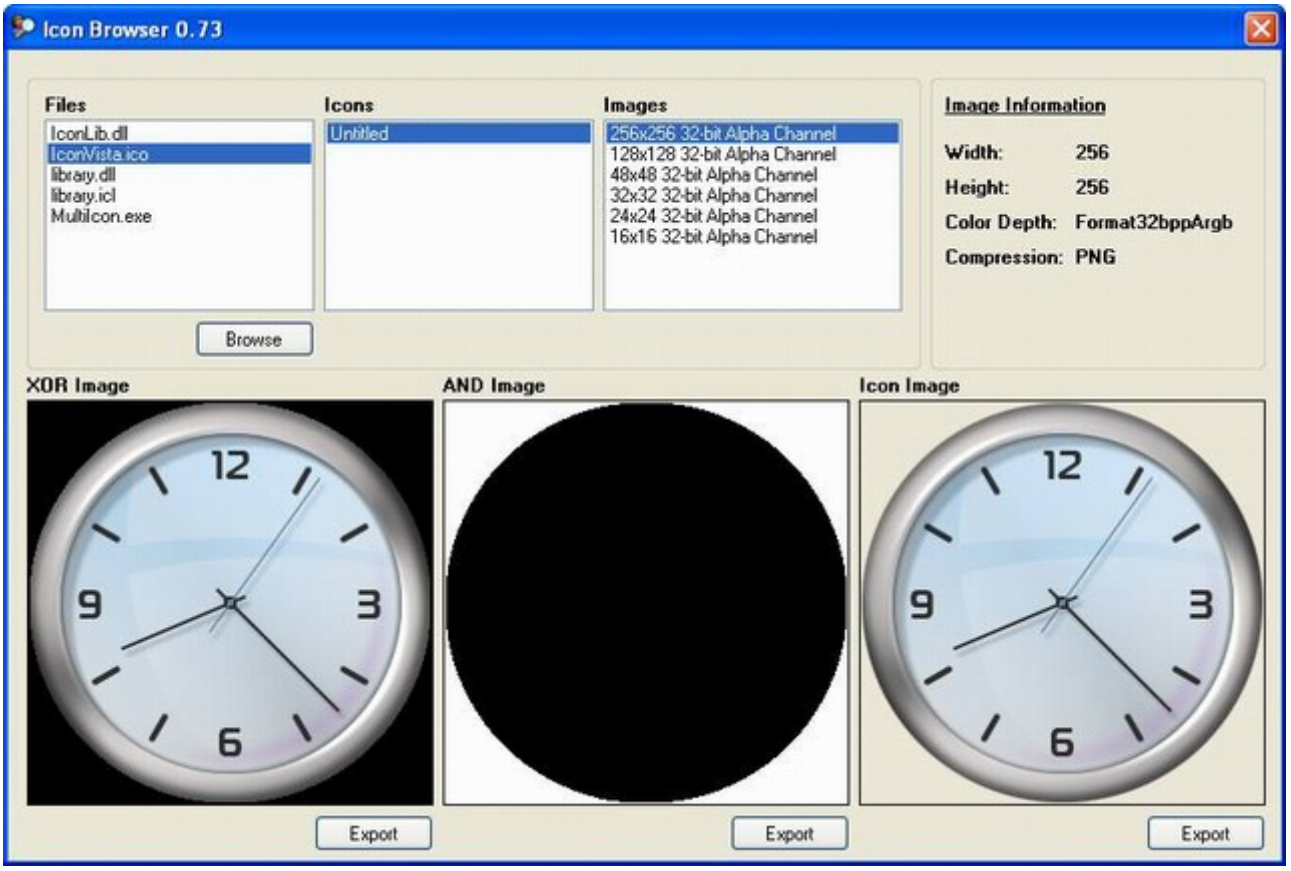
CastorTiu, 15 Feb 2008

4.96 (664 votes) Rate:

Library to manipulate icons and icons libraries with support to create, load, save, import and export icons in ico, icl, dll, exe, cpl and src format. (Windows Vista icons supported).

Download source files - 118.5 KB

Download demo project - 1.52 MB



Introduction

At some point in the last month, I needed to create an .ICO file on the fly with a couple of images inside; preferably I was looking at code in C#. The .NET Framework 2.0 only supports **HICON** that basically is one Icon with just a single image in it. When I was searching out there, to my frustration, I did not find any Icon Editor with the source code. The only thing I found was closed commercial products charging from $19 to $39 for them and not exposing APIs at all. So the only solution was to create my own library capable of creating and parsing ICO files.

I believe in open-source code and I thought that I could help the developer community by sharing this knowledge. In addition, open-source pushes companies and commercial products to go farther.

After the work was done, I read about ICL files (Icon Libraries). These can contain many Icons inside a file and I decided to support that too. The same happened with EXE/DLLs and last but not the least I decided to support Windows Vista Icons. All this was really hard work and a lot of headache because there is not much information exposed. I ended up spending a lot of time reverse-engineering and researching over the net. I hope it will be useful for you as it is for me.

Note

As in every new project, many things can happen. Not every case can be tested and many things cannot be seen even after they are tested. Since it is a very fresh project, if there is something that doesn't work, or you think should work differently than it does, before you give your vote, write a post and give me the chance to fix it. That way, we both get the benefit of getting more stable code and creating a more complete library, and at the same time you get my thanks if that helps.

I have also included two libraries as samples. I borrowed some icons from Windows Vista, for the 256x256 versions and I put a watermark because the icons has copyright ownership. Hopefully I won't have trouble with that.

Objective

The objective of the library is to create an abstraction layer from the different file formats and to provide an interface to allow icon modification without the hassle of knowing internal file formats.

Current Formats Supported

- ICO Read and write icons with different images size and depths
- ICL Read and write icons inside the icon library
- DLL Read DLLs and export to a new DLL
- EXE Import
- OCX Import
- CPL Import
- SRC Import

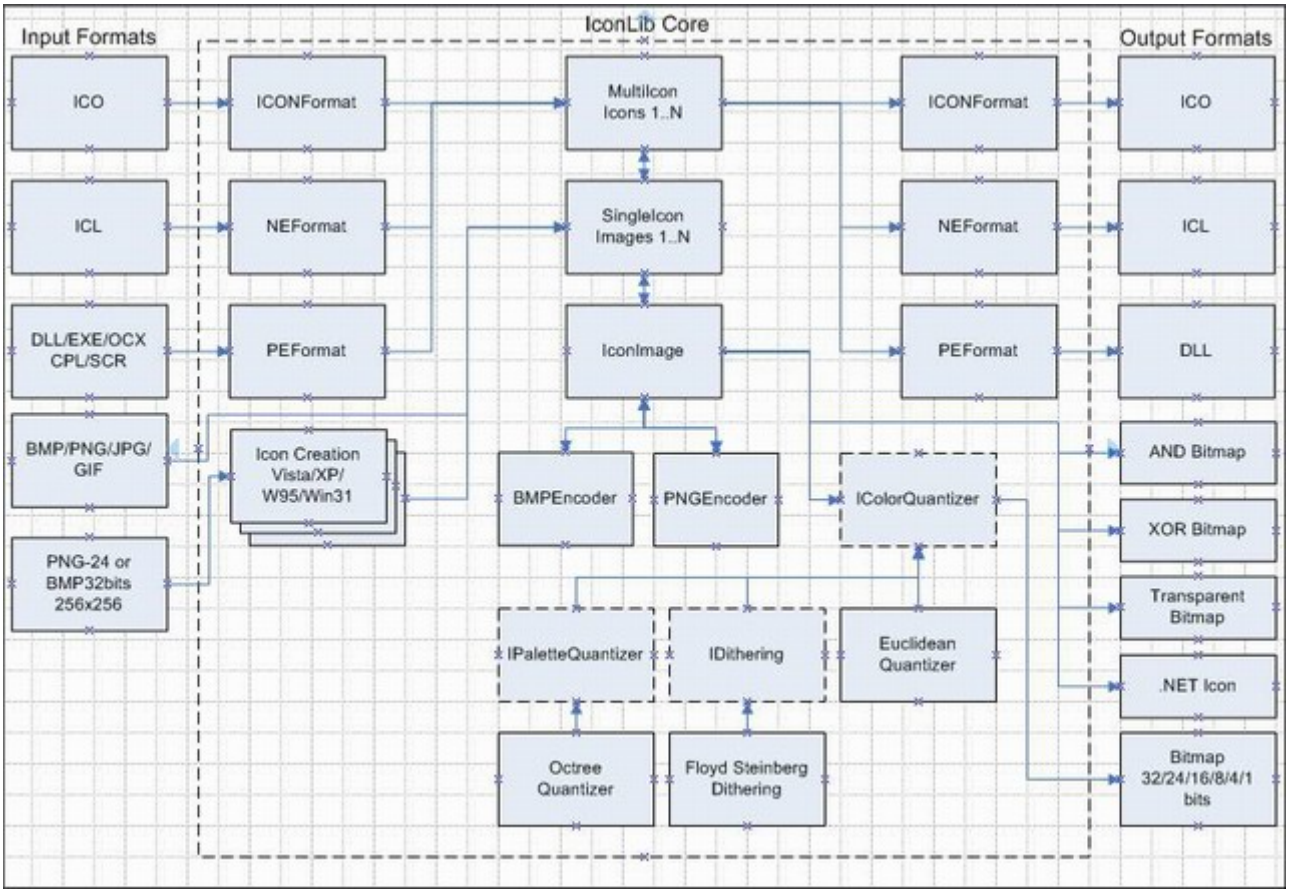
MultiIcon

`Iconlib` exposes three different objects.

1. `MultiIcon`: This is the only object that can be instantiated in the library. Once a `MultiIcon` object is created, it allows APIs to load and save in the file system or streams with standard formats as ICO/ICL/DLL.
2. `SingleIcon`: This represents a single icon inside `MultiIcon` and it allows add/remove images in it.
3. `IconImage`: This represents a single image inside `SingleIcon`. At this point, `IconImage` exposes the icon's lowest resources like the `XOR` (Image) and the `AND` Image (Mask). It also exposes an Icon property that will basically construct a .NET Icon created from the `XOR` and `AND` image from where you can get a `HICON` handle.

As you can see there is a hierarchical structure, basically a `MultiIcon` contains `Icons` and an `Icon` contains `Images`.

Library Objects Diagram



The library contains many classes and structs but only exposes the three important classes. The developer needs to control the complete behavior of the library, the rest are all internal, many classes/structs and methods are not safe to be exposed to the developer. For that reason, I recommend keeping `IconLib` as a separated project because if the developer incorporates the source in his/her project, all the internal classes/structs/methods will become visible and probably will not be used properly.

I cannot give support for the library when it is not used the way it was designed. I'm providing the source code as a nice gesture because I believe in open-source code, and I hope you will make good use of it without ripping of the source from where it belongs.

Icon Format

Before I started `IconLib`, I had no clue how icons work. However, I was not too long in the net before I found the excellent article Icons in Win32

Although this article is outdated with the arrival of Windows Vista icons, it is very precise in explaining how Icons format files are.

Something to take care; in my first version, I followed the icon format details but the library could not load some of the icons I was testing. When I went deep into the bytes, I could notice that much of the information was missing from the directory entry.

I tested those Icons with another product and I could see that, for example one popular product had no problem opening this kind of icon, and that is because every icon directory entry points to a `ICONIMAGE` struct, this `ICONIMAGE` struct has a `BITMAPINFOHEADER` struct that contains more information than the icon directory entry itself. So basically, using the information from the `BITMAPINFOHEADER`, I could reconstruct the information in the directory entry.

The same rule cannot be applied to Windows Vista Icons, because those images don't contain a `BITMAPINFOHEADER` struct anymore. Hence, if some information is missing from the directory entry, the icon image becomes invalid.

Anyway, reconstructing the icon directory entry is a plus and discarding icon image not properly constructed is acceptable, no company should provide icons with missing information in the headers.

NE Format (ICL)

NE Format is the popular format to store icon libraries; this format was originally used for Executables on 16-bit version of Windows.

You can get more information for NE format from the Microsoft web site at Executable-File Header Format

This was the most challenging part of the project. When I started researching about ICL, I had no clue that these were 16-bit DLLs. I couldn't find any data about this extension and couple of days later, I almost dropped the project. But I read in some place that ICLs are 16-bit with resources inside so my quest started on how to recover resources from a 16-bit DLL. So far my only next objective was trying to load in memory a 16-bit DLL. Of course, at first I tried to load the library with standard Win32API `LoadLibrary`, `LoadLibraryEx` but this failed with:

Hide Copy Code

```
193 - ERROR_BAD_EXE_FORMAT (Is not a valid application.)
```

I'm not an expert in Kernel memory allocation but I guess this is because in Win32, the memory is protected between applications and in 16-bit is not so when trying to allocate memory for 16-bit the OS rejects the operation.

The next step was trying to load the ICL (16-bit DLL) in memory using just 16 bits APIs. If you read the MSDN WIN32 API, the only API left for 16-bit is `Loadmodule`

When I tried, it loaded the library but immediately Windows started giving strange message boxes, as "Not enough memory to run 16-bit applications" or things like that.

I wrote in Microsoft forums and other forums, but found nothing really helpful on how I could get those resources. At that time, it was very clear that I could not load 16-bit DLL in memory and that I needed to create my own NE parser/"linker".

Microsoft article about NE Format (New Executable) is an excellent source and describes in detail every field in the file.

A NE format file start with an `IMAGE_DOS_HEADER`, this header is there to keep compatibility with MS-DOS OS. This header also contains some specific fields to indicate the existence of a new segmented file format. `IMAGE_DOS_HEADER` usually contains a valid executable program to run on MS-DOS. This program is called a stub program and usually it just prints the message on the screen 'This program cannot run on MS-DOS'.

```
                    IMAGE DOS HEADER
ushort e_magic;      // Magic number
ushort e_cblp;       // Bytes on last page of file
ushort e_cp;         // Pages in file
ushort e_crlc;       // Relocations
ushort e_cparhdr;    // Size of header in paragraphs
ushort e_minalloc;   // Minimum extra paragraphs needed
ushort e_maxalloc;   // Maximum extra paragraphs needed
ushort e_ss;         // Initial (relative) SS value
ushort e_sp;         // Initial SP value
ushort e_csum;       // Checksum
ushort e_ip;         // Initial IP value
ushort e_cs;         // Initial (relative) CS value
ushort e_lfarlc;     // File address of relocation table
ushort e_ovno;       // Overlay number
short  e_res[4];     // Reserved words
ushort e_oemid;      // OEM identifier (for e_oeminfo)
ushort e_oeminfo;    // OEM information; e_oemid specific
short  e_res2[10];   // Reserved words
uint   e_lfanew;     // File address of new exe header
```

After we read the `IMAGE_DOS_HEADER`, the first thing to do is to know if this is a valid header. Usually every file contains what is called a magic number. It is called a magic number because the data stored in that field is not relevant to the program, but it contains a signature to describe the type of the file.

You can find Magic Number almost everywhere. The magic number for the IMAGE_DOS_HEADER is `0x5A4D`, this represents the chars 'MZ', and it stands by "Mark Zbikowski" who is a Microsoft Architect and started working with Microsoft a few years after its inception. Probably he could never have thought that his signature was going to be used thousands of times in almost every personal computer in the world.

If the magic number is 'MZ' then the only extra field we care about is the `e_lfanew`, this header is the offset to the new exe header, NE Header.

We search in the file for this offset, and then at this point we read a new header. This header is `IMAGE_OS2_HEADER` and it contains all information about the program to be loaded in memory.

```
                        IMAGE OS2 HEADER
ushort ne_magic;         // Magic number
sbyte  ne_ver;           // Version number
sbyte  ne_rev;           // Revision number
ushort ne_enttab;        // Offset of Entry Table
ushort ne_cbenttab;      // Number of bytes in Entry Table
uint   ne_crc;           // Checksum of whole file
ushort ne_flags;         // Flag word
ushort ne_autodata;      // Automatic data segment number
ushort ne_heap;          // Initial heap allocation
ushort ne_stack;         // Initial stack allocation
uint   ne_csip;          // Initial CS:IP setting
uint   ne_sssp;          // Initial SS:SP setting
ushort ne_cseg;          // Count of file segments
ushort ne_cmod;          // Entries in Module Reference Table
ushort ne_cbnrestab;     // Size of non-resident name table
ushort ne_segtab;        // Offset of Segment Table
ushort ne_rsrctab;       // Offset of Resource Table
ushort ne_restab;        // Offset of resident name table
ushort ne_modtab;        // Offset of Module Reference Table
ushort ne_imptab;        // Offset of Imported Names Table
uint   ne_nrestab;       // Offset of Non-resident Names Table
ushort ne_cmovent;       // Count of movable entries
ushort ne_align;         // Segment alignment shift count
ushort ne_cres;          // Count of resource segments
byte   ne_exetyp;        // Target Operating system
byte   ne_flagsothers;   // Other .EXE flags
ushort ne_pretthunks;    // offset to return thunks
ushort ne_psegrefbytes;  // offset to segment ref. bytes
ushort ne_swaparea;      // Minimum code swap area size
ushort ne_expver;        // Expected Windows version number
```

The first thing to do is to load the magic number again, but this time the magic number must be `0x454E` and it means 'NE'. If the signatures match, then we can continue analyzing the rest of the headers. At this point the more important field is `ne_rsrctab` as this field contains the offset of the resource table. From this offset, we get the number of bytes we have to jump from the beginning of this header to be in position to read the resource table.

If everything went well, we are ready to read the resource table.

```
        RESOURCE TABLE
ushort        rscAlignShift;
TYPEINFO[]    rscTypes;
ushort        rscEndTypes;
byte[]        rscResourceNames;
byte          rscEndNames;
```

The first field of the Resource Table is the align shift, usually you find the explanation as "The alignment shift count for resource data. When the shift count is used as an exponent of 2, the esulting value specifies the factor, in bytes, for computing the location of a resource in the executable file."

In my own words, the working of this field was tricky to understand. It was created for compatibility with MS-DOS, and it will contain the multiply factor necessary to reach the resource.

As you will see, the resource offset is a variable of type `ushort`, which means that it can only address 64Kb (65536). Actually almost every file is bigger than that, and here is where the 'alignment shift' field comes to play.

Alignment shift is a `ushort` and "usually" it is in the range of 2 to 10. This number is the number of times we have to shift the number 1 to the left. For example:

- Alignments shift of 5 means 1 << 5 which is equal to 32.
- Alignments shift of 10 means 1 << 10 = 1024

Now with the virtual offset address from the resource table we multiply for the result shift value and we get the real offset address in the file.

For Example

The resource located at the virtual address 0x2000 and the alignment shift is 5 then we get:

Hide Copy Code

```
Realoffset = (1 << 5) * 0x2000
Realoffset = 32 * 0x2000
Realoffset = 0x40000
```

The real offset of this resource is at 262144 (0x40000).

Wow, this is cool right? Because we just use an `ushort` and we can locate a resource at any position. Now you will wonder where the trick lies?

The trick is for example if you use a shift alignment of 5 that means the minimum addressable space is 32 bytes (1 << 5), which means if you want to allocate 10 bytes with this method 32 bytes will be allocated and just the first 10 will be used, another 22 bytes will be wasted.

Now you might wonder, ok then let's take the shift alignment as 0, then e won't waste space because the virtual address will match with the real address. It is not so easy, and that works only if the resource is located in the range of the first 64Kb space.

So to make it clear, this shift alignment is directly proportional to the file size.

The next table tells what the maximum file sizes are that you can get with different shift alignments:

Hide Copy Code

```
(1 << 0) * (2 ^ 16) = 64KB
(1 << 1) * (2 ^ 16) = 128KB
(1 << 2) * (2 ^ 16) = 256KB
(1 << 3) * (2 ^ 16) = 512KB
(1 << 4) * (2 ^ 16) = 1MB
(1 << 5) * (2 ^ 16) = 2MB
(1 << 6) * (2 ^ 16) = 4MB
(1 << 7) * (2 ^ 16) = 8MB
(1 << 8) * (2 ^ 16) = 16MB
(1 << 9) * (2 ^ 16) = 32MB
(1 << 10) * (2 ^ 16) = 64MB
```

Calculating this value is not so easy. `IconLib` at first uses a shift factor of 9 because I thought that 32MB was more than enough for an `Icon` library. But great was my surprise when I extracted Windows Vista DLLs and `IconLib` got out of range for some files. I then incremented the shift factor to 10 which enabled me to dump the content of the Windows Vista DLL in an ICL file, but it took 63MB.

A factor of ten allows to us to create an ICL library up to 64MB but every resource will address at minimum 1024 bytes. If you think that's not bad because all resources will be bigger than 1024, it is not so easy. A factor of ten means it can address in multiples of 1024, then if the resource is 1025 then it will allocate 2048 bytes in the file system.

In conclusion, with a factor of 10, `IconLib` is wasting an average of (1024 / 2) 512 bytes by resource allocated, but at the same time it lets us create an `Icon` Library with 64MB.

My next release will predict the max file size and will adjust the shift factor dynamically; it is not an easy task if you want to predict the number without scanning memory to know the max space to be addressed, especially for PNG images where this value is dynamic too.

Hopefully shift alignment field is clear now and we come back to the resource table.

The next field is an array of `TYPEINFO`

```
          TYPEINFO
ushort       rtTypeID;
ushort       rtResourceCount;
uint         rtReserved;
TNAMEINFO[]  rtNameInfo;
```

`TYPEINFO` is a struct that gives us information about the resource; there are many types of resources that can be allocated, but `IconLib` is just interested in two types `RT_GROUP_ICON` and `RT_ICON`.

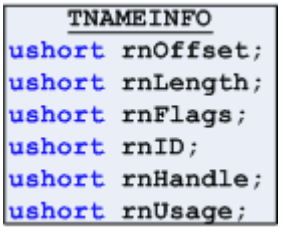
When `IconLib` reads the `TYPEINFO` array, it discards all structs where `rtTypeID` is not `RT_GROUP_ICON` or `RT_ICON`

`RT_GROUP_ICON` type gives us information about a icon.

`RT_ICON` type gives us information about a single image inside the icon

`rtResourceCount` is the number of resources of this type in the executable.

`rtNameInfo` is an array of `TNAMEINFO` containing the information about every resource of this type. The length of this array is equal to `rtResourceCount`.



Here is where we have the information about the resource itself; the `rnOffset` is the virtual address where the physical resource is located. To know the real address, see how alignment shift works above.

The `rnLength` is the length of the resource on a virtual address space. This means if for example the resource has a length of 1500 bytes and the alignment shift is 10, then the value on this field will be 2.

The way to calculate the length is:

Hide Copy Code

```
rnLenght = Ceiling(ralresourcesize / (1 << resource_table.rscAlignShift));
rnFlags tell us if the resource is fixed, preloaded, or shareable
rnID is the ID of the resource.
rnHandle is reserved.
rnUsage is reserved.
```

Going back to `TYPEINFO`, if the `TYPEINFO` struct type is `<a name=""OLE_LINK6""></a>RT_GROUP_ICON` then we read the array of `TNAMEINFO` which gives us information about every icon in the resource.

The offset in every `TNAMEINFO` will contain a pointer to a `GRPICONDIR` struct, this struct will give information about a single icon, like how many images it contains and one array of `GRPICONDIRENTRY`, whenever `GRPICONDIRENTRY` contains information about the image, like width, height, colorcount, etc.

Now if the `TYPEINFO` struct type is `RT_ICON` then we read the array of `TNAMEINFO` which gives us information about every single image inside the resource.

Going back to the Resource Table we have another three fields, `rscEndTypes`, `rscResourcesNames` and `rscEndNames`.

`rscEndTypes` is a `ushort` value that tell us when to stop reading for `TYPEINFO` structs. The resource table struct doesn't tell how many TYPEINFO structs it contains, so the only way to know it is with a stopper flag. This flag is `rscEndTypes`. If when we read `TYPEINFO` the first two bytes are zero, then it means that we reached the end of the `TYPEINFO` array.

`rscResourceNames` is an array of bytes with the names of every resource in `TYPEINFO` struct. The names (if any) are associated with the resources in this table. Each name is stored as consecutive bytes; the first byte specifies the number of characters in the name.

For example if the array is `[5, 73, 67, 79, 78, 48, 5, 73, 67, 79, 78, 49]`

This is translated like an array of two strings "`Icon1`", "`Icon2`".

Hide Copy Code

```
[5, 73, 67, 79, 78, 48, 5, 73, 67, 79, 78, 49]
[73, 67, 79, 78, 48] = "Icon1"
[73, 67, 79, 78, 49] = "Icon2"
```

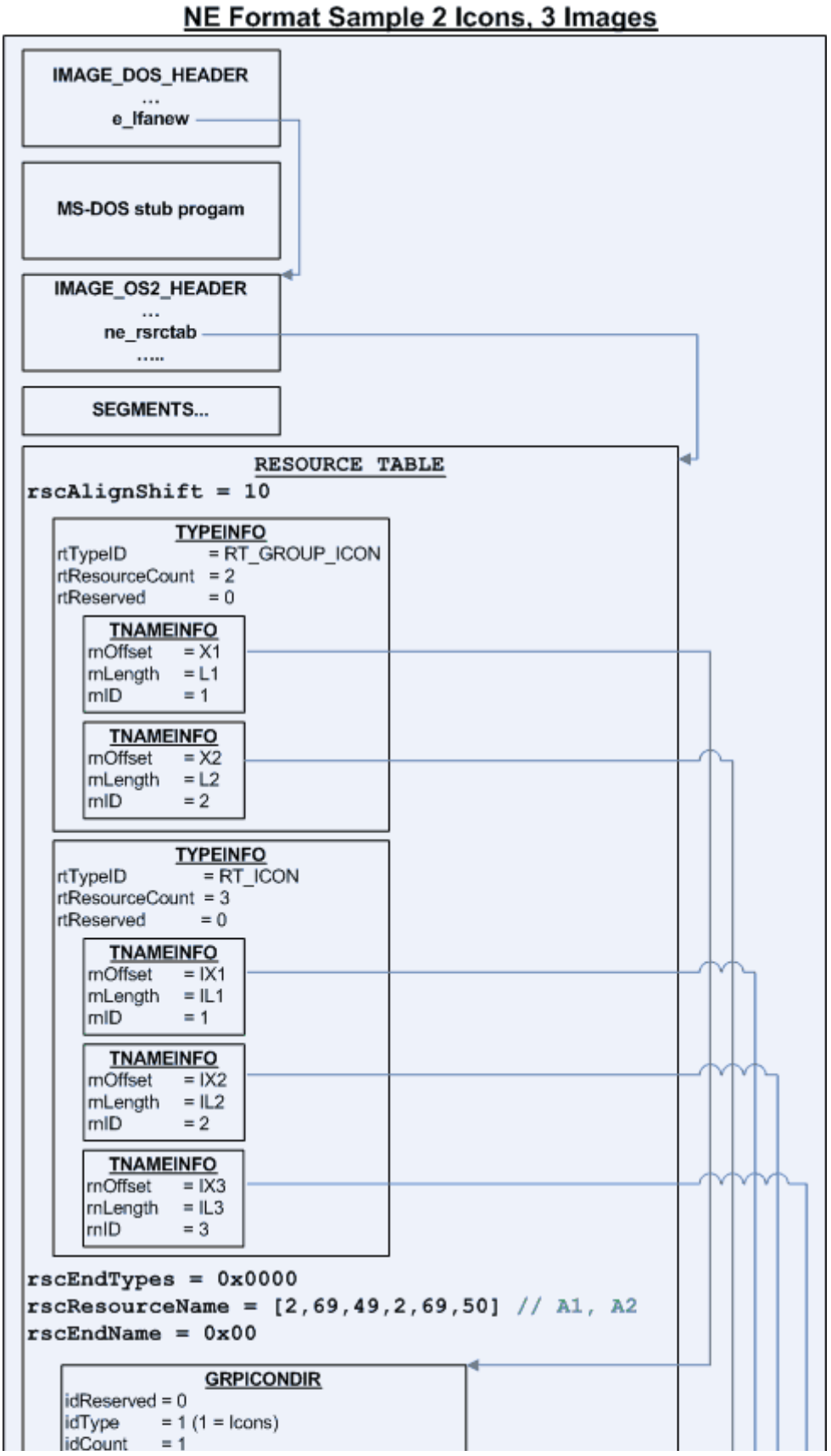
If you wonder when you have to stop reading for bytes in the array, there exists another stopper flag `rscEndNames`with a value of zero. When the bytes are being read, if a null ('\x0') character is detected, then the process must stop reading the names, and they are ready to be translated as ANSI strings.

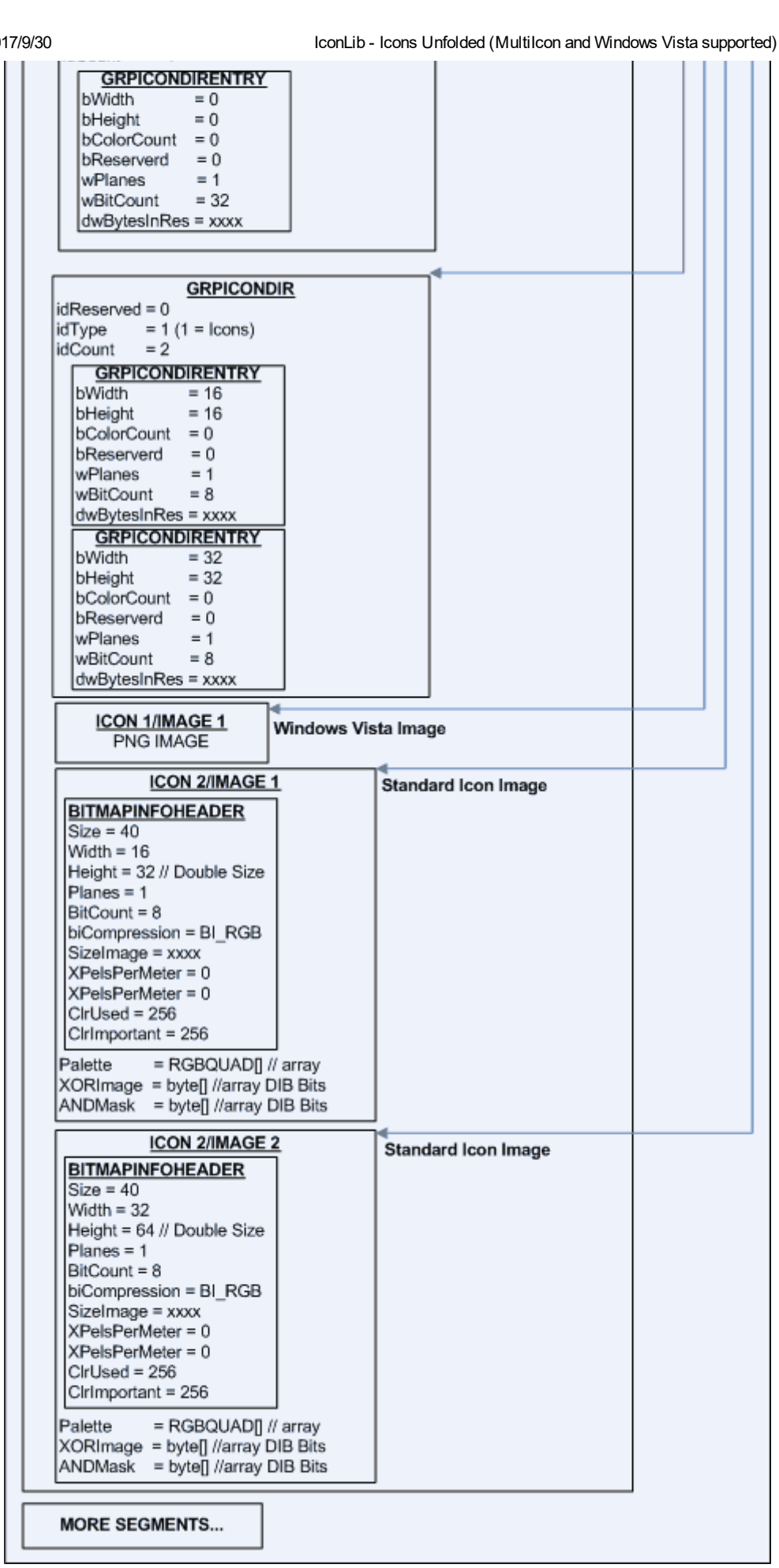
At this point we already have all the information and binary data for the `Icon`s and the images inside the `Icon`.

`IconLib` loads all the `Icon`s and `Icon` images in memory to obtain a good performance while working with them. In addition, it does not need to lock the file on the file system.

Creating an ICL file is not so complex after all. Because `IconLib` creates an ICL from scratch, it doesn't have to care about the other segments in the NE Format, so the process is relatively simple. We write an `IMAGE_DOS_HEADER`, write the `MSDOS` stub program, write an `IMAGE_OS2_HEADER`, where we choose the right alignment factor and we write the resource table at the location specified by the field `ne_rsrctab` in the `os2_header`.

When the resource table is written it has to apply the same rules when loading. This means write a partial resource table struct, the two `TYPEINFO` structs (`RT_GROUP_ICON` and `RT_ICON` and inside the `TYPEINFO</ode>`, write the `TNAMEINFO` information)





The following table shows a NE format that stores 2 Icons, the first icon contains one image, the second icon contains 2 images.

That is something that I would like to mention. As I mentioned before, I redesigned the core 3 times, the first time I followed every known specification on how the Icon file has to be read and written from Icons and DLLs. When I exported icons from DLLs, I kept all information about the icon, as the Icon names, group ID and icon ID. When I saved them on the file system, I saved them in the same

way I read them, so basically I could export the icons from a DLL, export it to a ICL file, then load the ICL and export to a DLL, and I would keep the same IDs for the groups and Icons.

So far I tested two popular commercial products and they could open them without problems, but for example I started to have problems when I exported some DLLs or EXEs to ICL files. For example if you open *explorer.exe* from Windows folder in Visual Studio, the first thing you will notice is that the icons IDs are not consecutive, they start with ID 100, 101, 102, 103, 104 and jump to 107, and continue.

`IconLib` exported *explorer.exe* to an ICL file, and import it on Visual Studio and there was no problem at all. But to my surprise, when I tried to open it with a popular Icon Editor, the icon library showed images with icons mismatched and mixed between the icons. I spent many days trying to figure why it was happening.

Basically after many tests of different applications, I could notice that those applications write the ICL files and discard the Icons and Group IDs and they expect consecutive IDs.

For ICL files there is a header to be written `TNAMEINFO`, this header contains a field which is the ID. This ID can be a `GRPICONDIRENTRY` (Icon itself) or an `ICONDIRENTRY` ID (single image ID inside the icon). When those applications write ICL files, they do it in a consecutive way, basically they discard the IDs when they imported from the DLL and write groups id as 1, 2, 3, 4, same for the icons id, they do 1,2,3,4. etc.

So basically I noticed that some applications are not prepared to handle ICL files properly for all cases. Another not so popular application passed it and basically it could read ICL files where the IDs were not consecutive, but still when it saved the ICL file it discarded the source ID and put its own.

So I had a big dilemma. Should I keep all the information and write that information as it is coming from the EXE/DLLs in the ICL files; that would make my ICL files properly constructed but incompatible with some applications out there. Or should I discard the original IDs in the importation and create consecutive IDs which means discard part of the original information and put my own (I was not keen on this solution), but small fishes can swim in a pool with big fishes unless they behave like one.

So I didn't have another choice than to redesign my core to produce those results using consecutive IDs. After I redesigned I reduced the source code because now I didn't need to keep all the information that was generated on the fly, but when icons are exported from the DLLs the original Icons IDs are lost. Anyway, a regular developer will rarely use those IDs.

I still wonder if it is a mis-implementation of those products to fully support ICLs, or if there is a rule in the NE Format files that says you can't store a resource with a "random" ID. So far, all my research concludes that you can use any ID for the `ICONIMAGES` inside NE Format.

PE Format (DLL, EXE, OCX, CPL, SRC)

PE Format means Portable Executable; this format was created by Microsoft to supports 32-bit and 64-bit version of Windows in NE Format replacement used for 16-bit version of Windows.

Basically files format like EXE, DLL, OCX, CPL, SCR don't differ too much amongst them. For example, think of an EXE like a DLL with an entry point. When working with resources, all those files are identical. This means if the library supports PE Format then it supports all the above extensions.

Because Win32 API already supports resources handling for PE format, then it was not necessary to support this file format natively, instead IconLib makes use of Win32 APIs to gain access to the icons resources.

The only native functionality was to read the first set of headers from the PE file to detect whether the file to be loaded is a PE format or not.

If we want to access just the resources then the best way to do it is to load the library as a `DATAFILE`. This means no code at all will be executed from the library, instead Win32 API will access just the resources data.

Hide Copy Code

```
hLib = Win32.LoadLibraryEx(fileName, IntPtr.Zero, LoadLibraryFlags.LOAD_LIBRARY_AS_DATAFILE);
```

`IconLib` core just supports reading and writing from and to a stream. `MultiIcon` overloads some functions as `Load`/`Save` and creates a `FileStream` from a file in the file system before calling the `Load(stream)`. Win32 API `LoadLibrary` can only load libraries from the file system; therefore the stream will be saved in a temporary file before Win32 API `LoadLibraryEX` is called.

Access to the resources is an easy task when the resources are accessed in the proper order.

The first thing that `IconLib` does is call `Win32.EnumResourceNames` sending as parameter `GROUP_ICONS`, which gives us back the ID of every icon. This ID can be a number or a pointer to a string. If the value returned is less than 65535 then it is a number, if the value if bigger than 65535 then it is a pointer to one string.

Once we have all the IDs for the icons, we call the function `Win32.FindResource`. For every ID found, this gives us a handle to the resource and then we can proceed to load and lock the resource to access the resources entries. Those entries contain the IDs of every image inside the icon just loaded/locked. Now we repeat the steps that we did before, but instead of using the constant code>**GROUP_ICONS** we use `RT_ICON`. This tells the Win32 API that we want to access the image inside the icons.

Here is the critical step that needs to be done. Under Windows XP or previous OS, after we lock the resource for the icon image we will have a pointer to an `ICONIMAGE`. This icon image will contain the `BITMAPINFOHEADER`, `Palette`, `XORImage`, and the `AND`Image (mask), but in Windows Vista instead, it returns a pointer to a `PNG` image, basically this is the main reason why current Icon Editors including the more popular ones will crash. They will allocate huge amount of memory or just drop the image because they will be parsing a `PNG` image like a `BMP` image.

`IconLib`resolves that issue by reading the first bytes of the Image and detecting the signature of the image creating the proper encoder instance before reading and parsing the image.

When `IconLib`has to create a `DLL`, the best way so far was to use an empty `DLL` as a template, and add the resources to it.

Win32 API offers three APIs that will do the job for us.

1. `BeginUpdateResources`
2. `UpdateResources`
3. `EndUpdateResources`

MSDN tells us that you can call `BeginUpdateResources`, and then call `UpdateResources`as many times as you want, the file won't be written yet. At last you call `EndUpdateResources`and the changes are committed to the `DLL`.

That methodology worked pretty good for small `DLL` files. When `IconLib`was creating libraries with more than 80 images, everything was OK but the call to `EndUpdateResources`always failed. After a lot of unsuccessful tries, the only thing I could think of was that the API to update resources has an internal buffer. When that buffer is full, calls to `EndUpdateResoruces`fail to commit the changes into the `DLL`.

The workaround that I found was to commit every time on an average of 70 updates, this worked pretty well but enormously increased the time to update the `DLL`. For that reason, unless I can find why Win32 is doing that, I'll try to come up with my own PE Format implementation, and not use the Win32 at all. That will speed up the process a lot.

You can get more information for PE format from Microsoft web site at Microsoft Portable Executable and Common Object File Format Specification

Windows Vista Icons Support

I wanted to create a library to work with icons without having any limitations, so support for Windows Vista was a must.

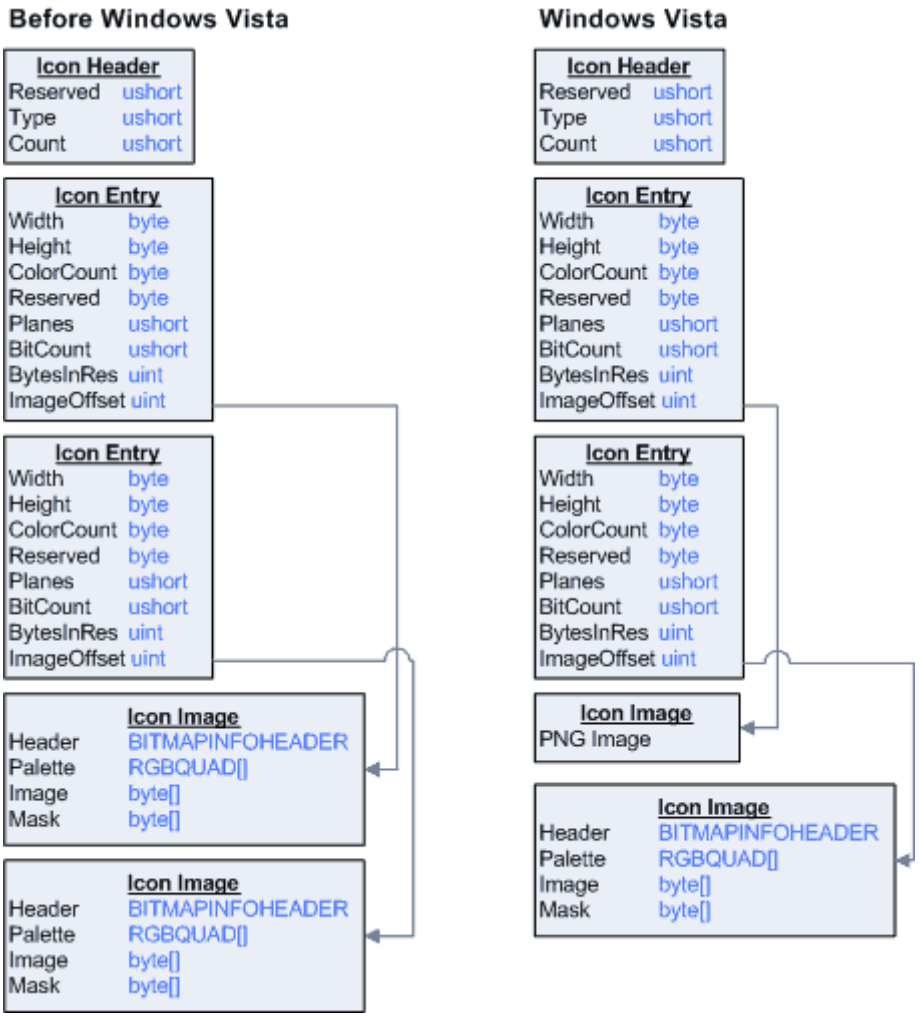
In Windows XP, they introduced icons with an alpha channel and 48x48 pixels. In Windows Vista, Microsoft introduced icons images with a size of 256x256 pixels. This image inside the icon can take 256Kbytes for the image and another 8Kbytes for the mask in uncompress format. That increases the size of icons library substantially and basically resolves this issue of storing the image in a compressed format.

The compression used was PNG (Portable Network Graphic) because it is free of patents, supports transparency (Alpha Channel) and employs lossless data compression.

The factor is on an average between 3 to 5 times smaller than uncompress bitmaps.

If you think there is not much difference, then load the file *imageres.dll* from *Windows\System32* in Windows Vista (11MB), do a for loop for all images and set the encoder to be BMP instead PNG, then save it to a DLL or a ICL file. You will notice that the DLL is about 45MB and the ICL about 54MB. This is where you can see that PNG really makes the difference.

To store the compress image, they could come up with a way to keep some backward compatibility and this was setting the field `biCompression`in `BITMAPINFOHEADER`to `BI_PNG` instead of `BI_RGB`. This header is already supported from Windows 3.1 and the filed `BI_PNG`from Windows 95, but instead they broke compatibility and they store the image alone. (see 'Ohh Microsoft policy about compatibility is changing?' below)



The sample only contains two images but there can be up to 65535.

Although in all my research, I saw only 256x256 images in PNG format, that doesn't mean it could not store all images as PNG. This was only a decision to have compatibility with previous version of Windows.

Personally I think Icons editors should support PNG at any size and bits depth. Icons not only are used by Windows OS, it is the same why Windows icons allow introducing non-standard images like 128x96x24 when Windows will never make use of it.

If you are creating an icon that Windows Vista will make use of, only store PNG compression for 256x256 images.

Smart Classes/Structs

The more difficult stuff was how to come up with a clean code and APIs capable of understanding different icons formats and icons libraries and also different image compressions without creating a chaos of switch/if/else.

In my journey of creating the library, I redesigned the core from scratch 3 times, and still there is a TODO changes to avoid `IconImages` objects from knowing about different compression methods. Basically an `IconImage` should not be responsible for knowing the format of the image to be read/written. Instead it should depend on the different encoders to know this information.

Right now `IconImage` has a reference to an `ImageEncoder` object (base class), but still `IconImage` object is responsible for discovering the signature of the image to know if it has to create a `BMPEncoder` or a `PNGEncoder` instance.

Also there are a couple of changes to manage the memory allocations more efficiently, but that won't change the core design.

Coming back to what I called Smart Class/Structs: Basically an Icon is a hierarchical structure and Icon libraries are the same but contain one more level of information.

The objective of this smart classes/structs was to avoid interchange data between the different objects; instead every class/struct should be capable of reading and writing itself. If a class of struct contains more classes or structs inside, then it should ask the child to read/write that portion of information and so on.

If you open the source code, immediately you will notice that the parameter '`Stream stream`' is everywhere. This allows the object that receives this parameter to read/write itself in the stream at the current position.

For example, when a `Icon` file has to be created, the `MultiIcon` object will open a `FileStream` and will call `ImageFormat.Save(stream)`, sending as a parameter the stream just opened.

`ImageFormat` object will contain only the logic to write itself and rely on the different classes/struct to write the rest of the information.

Hide Copy Code

```
ImageFormat.Save(stream)
{
 ICONDIR.write(stream)
 {
   Write iconDir header
 }

 Loop for each IconImage
 {
   ImageEntry.Write(stream)
   {
     Write iconEntry header
   }

   Image.Write(stream)
   {
     BitmapInfoHeader.Write(stream)
     {
       Write bitmap info header
     }
     Write Color Palette

     Write XOR image

     Write AND image
   }
 }
}
```

This is a simple case, but more complex cases like reading ICL (Icon Libraries) follow the same behavior.

So, following this model writing and reading different formats was really easy. It also produced a super, cleaner code.

Image Encoder

I wanted to provide a library easy to understand and flexible enough to adapt to any kind of image format. The ideal case was to create a class with basic functionality but to leave the specific format implementation to other classes.

`ImageEncoder` object keeps all the information about one icon image and it contains information like image properties, palette, icon image, and icon mask.

This class is an abstract class that cannot be instantiated.

BMP Encoder

`BMPEncoder` class has the logic to read and write Icon entries when `biCompression` is `BI_RGB` (BMP)

PNG Encoder

`PNGEncoder` class has the logic to read and write Icon entries when image is PNG format.

I followed the information I could get from different sources to create Icons with PNG compression. So far the implementation doesn't have problems and Icons Images in PNG format can be opened with all Icons editors that support Windows Vista.

`Icon` libraries are a different subject. So far I did not find a single open source or commercial icon editor, including the more popular ones that allow opening or writing icon libraries like ICL or DLL with PNG compression, In some commercial products you

will notice that the PNG icons are not loaded and also if you create PNG icons they are uncompressed before getting saved on a DLL or ICL. I think that is because Microsoft still didn't release any information about it and companies are waiting for the final Windows Vista to come out.

I based my work for creating compressed icon libraries (ICL, DLL) on reverse engineering in Windows Vista RC2 and following the same logic that the Microsoft boys used for icon files.

`IconLib` is capable of loading all icons from Windows Vista DLLs/EXEs and CPL files (PNG format inclusive). It also allows writing ICL/DLL icon libraries with PNG compressions.

The bad news is that only you can load them with `IconLib`for now. If you try to load an ICL or DLL with PNG images generated with `IconLib`and try to open it with third party icon editor, you will see that the PNG icons are gone, and also the icons contain images from other icons. So far, all my research concludes that there is a mis-implementation of PNG format for ICL libraries in those products and has nothing to do with `IconLib`.

Now if you wonder how I can be sure that `IconLib`generates ICL or DLL properly?.

Basically if you try to open Windows Vista icons that contains 26x256 PNG icons in any VisualSudio version (Orcas inclusive if you wonder about VS2006 so far), it will show an image with a size about 2573x1293 with XP format. Of course that image doesn't exist and you can't edit it, but that's how Visual Studio sees it.

Now if you load a DLL with 256x256 PNG files, generated with `IconLib`and save the icon that contains the PNG image to the file system and open the icon image with Visual Studio, you will notice the same behavior as the DLLs from Windows Vista.

Anyway, the entire work is based on suppositions, and I can't be really sure yet as long as any Windows Vista Libraries Icons Editors hit the market or Microsoft releases more information about it.

Color Reduction and Palette Optimization

Usually there are programs that allow creating icons from a bitmap and they produce an Icon with alpha channel (transparency) compatible with Windows XP, those icons lacks of the support of low resolution images, Iconlib allows to add a low resolution image and also incorporate a whole namespace to produce a low resolution image from a high resolution image.

The techniques used by IconLib are:

- Palette Optimization
- Color Reduction
- Dithering

Palette Optimization

A palette is an array of RGB colors, most of the times the length of the palette is the amount of colors supported, a palette can contains any length but in most of the cases the palettes are 256 or 16 indexes.

An optimized palette is created on base to the bitmap to be processed; it will analyze the input image and will create a new palette with the most used colors from the input image, many ways might be used to create an optimized palette.

Why use a palette on a Bitmap?

Every index in the palette is a RGB color, 3 bytes are necessary to create the color (1 byte for Red, 1 byte for Green, 1 byte for Blue), this allow to create a combination of 16 million colors because each channel can produce a 256 color gradient, then 256R * 256G * 256B = 16777216 color combinations.

If on the bitmap data we store the RGB information then at least we require 3 bytes to store every pixel color.

Instead, indexed bitmaps will store just one index to an array of colors; this means that the bitmap data does not contain color information but an index to an array (palette).

This can save a lot of space but the image quality may suffer considerably because very similar colors on non-indexed image will be converted to the same color (index) on an indexed image.

There are many more data store in a Bitmap but just for example let's compare the size of 3 bitmaps.

100x100 pixels 24 bpp image

1 pixel = 3 bytes

100x100x3 = 30000 bytes to store the color information.

100x100 pixels 8bpp indexed image

1 pixel = 1 byte

1 palette = 256 indexes of RGB color = 256x3 = 768

100x100x1 + 768 = 10768 bytes to store the color information.

100x100 pixels 4bpp indexed image

1 pixel = 1/2 byte

1 palette = 16 indexes of RGB color = 16x3 = 48

100x100x1/2 + 48 = 5048 bytes to store the color information.

The key to have a low resolution indexed image and still good looking is to choose the right color for the palette, there are different palettes that can be used.

System palette: this is the default Windows palette and it contains 256 colors, it has a variety of colors in a wide spectrum, IconLib make no use of this palette because if for example the icon to be color reduced has many gradients when those gradient are converted to an index pixel version many of them will have the same index and the quality of the image will be greatly degraded.

Exact: If the image contains less than 256 colors, those are mapped directly to the palette.

Web: Is the intersection between Windows and Mac OS palette, it contains 216 colors that are safe to be used on Windows or Mac OS systems.

Adaptive: This palette reduces the colors in the bitmap based on their frequency; for example, if your image contains mostly skin tones, the adaptive color palette will be mostly skin tones.

Perceptual: This palette is weighted toward reducing the colors in the bitmap to those to which we are the most sensitive.

Selective: The Selective palette will choose the colors from the bitmap to the web-safe colors.

Custom: A custom palette might be provided.

IconLib creates an optimized palette using the Adaptive algorithm with an Octtree structure.

Color Reduction

The idea behind color reduction is take an 32bits (ARGB) or 24bits (RGB) image where the data of every pixel contains the RGB color information and convert this image to a indexed image, they are called indexed because every pixel data **does not** contain the RGB color information instead it contains a index to a palette (Array of colors), this palette store n numbers of colors, 32bits and 24bits images can produce 16 million colors and every pixel is stored as 3 bytes (4^{th} byte for alpha channel). Because indexed just store an index to the palette the store needed depends of the image resolution.

Non-Indexed 32 bits (16M colors plus transparency) = 4 bytes per pixel

Non-Indexed 24 bits (16M colors) = 3 bytes per pixel

Indexed 8 bits (256 colors) = 1 byte per pixel

Indexed 4 bits (16 colors) = 1/2 byte per pixel or 2 pixel per byte

Indexed 1 bit (Black&White) = 1/8 byte per pixel or 8 pixel per byte

In IconLib color reduction algorithm works pretty close with the palette optimization algorithm.

Before a pixel can be converted to an indexed pixel a palette must be available to choose the right color index.

Different palettes can be used in the process of color reduction.

See Palette Optimization above.

The algorithm I have use in the color selection was the Euclidian distance, basically it finds the nearest neighbor color in the palette, it maps the current color in the image with a color in the palette finding the shortest distance between the current color and the neighbor color in a 3D space.

Dithering

Even when an optimized palette is used in the process of color reduction the resulting image may looks not good especially when the input bitmap contains high number of gradient, to improve the looking of image dithering is used.

Dithering is the process of juxtaposing pixels of two colors to create the illusion that a third color is present, basically noise is added in the process, this noise is proportional to the different color gaps between pixels.

There are many algorithm to implement dithering, and the output image vary between them, personally I like Floyd-Steinberg algorithm because the noise generated is spread uniformly creating a nice looking image.

No dithering: no noise is added to the output bitmap.

There are three kinds of dithering:

Noise dither: It is not really acceptable as a production method, but it is very simple to describe and implement. For each value in the image, simply generate a random number 1..256; if it is greater than the image value at that point, plot the point white, otherwise plot it black.

Ordered dither: Ordered dithering adds a noise pattern with specific amplitudes, for every pixel in the image the value of the pattern at the corresponding location is used as a threshold. Different patterns can generate completely different dithering effects.

Error diffusion: diffuses the quantization error to neighboring pixels.

Floyd-Steinberg dither: it is an error diffusion dither algorithm and is which is used in IconLib, it is based on error dispersion. For each point in the image, first find the closest color available. Calculate the difference between the value in the image and the color you have. Now divide up these error values and distribute them over the neighboring pixels which you have not visited yet. When you get to these later pixels, just add the errors distributed from the earlier ones, clip the values to the allowed range if needed, then continue as above.

In the following sample it reduces the image to 8, 4 and 1bpp from a 24bpp source image.

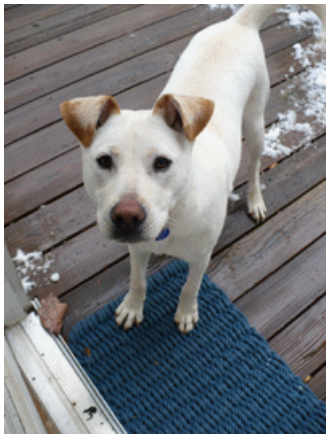
Hide Copy Code

```
IColorQuantizer colorReduction = new EuclideanQuantizer(new OctreeQuantizer(), new
FloydSteinbergDithering());
Bitmap bmp = (Bitmap) Bitmap.FromFile("c:\\Pampero.png");

Bitmap newBmp = colorReduction.Convert(bmp, PixelFormat.Format8bppIndexed);
newBmp.Save("c:\\Pampero 8.png", ImageFormat.Png);

newBmp = colorReduction.Convert(bmp, PixelFormat.Format4bppIndexed);
newBmp.Save("c:\\Pampero 4.png", ImageFormat.Png);

newBmp = colorReduction.Convert(bmp, PixelFormat.Format1bppIndexed);
newBmp.Save("c:\\Pampero 1.png", ImageFormat.Png);
```

<td style="PADDING-RIGHT: 5.4pt; PADDING-LEFT: 5.4pt; PADDING-BOTTOM: 0in; WIDTH: 162.3pt; PADDING-TOP: 0in" valign=removed" width="216">

1bit Black and White

Floyd-Steinberg dither

<table>
<tr>
<td>  </td>
<td>  </td>
<td>  </td>
<td>  </td>
</tr>
<tr>
<td>24bits RGB 16M Colors</td>
<td>8bits 256 colors
Floyd-Steinberg dither</td>
<td>4bits 16 colors
Floyd-Steinberg dither</td>
<td></td>
</tr>
</table>

Extensible ColorProcessing Namespace

For most of the application that use IconLib the ColorProcessing namespace contains all the tools necessary to create a quality icon, but because there are so many algorithm for color reduction then it is implemented with interfaces, this means that the library can be expanded to use different algorithms if it is necessary.

For color reduction there is an interface `IColorQuantizer` and it is implemented for the default class `EuclideanQuantizer`

For palette optimization there is an interface `IPaletteQuantizer` and it is implemented for the default class `OctreeQuantizer`

For dithering there is an interface `IDithering` and it is implemented for the default class `FloydSteinbergDithering`

Any of those interfaces can be implemented and the default can be replaced.

For example, if the developer implemented the noise or random dither algorithm then the color reduction initialization could be something like:

Hide Copy Code

```
IColorQuantizer colorReduction = new EuclideanQuantizer(new OctreeQuantizer(), new
NoiseDithering());
```

Automatic Icon Creation

Even when with a few lines of code IconLib can create an icon with multiple images from a single one, anyway IconLib provideds a special API that will create a full Icon from a single input image.

Hide Copy Code

```
MultiIcon mIcon = new MultiIcon();
SingleIcon sIcon = mIcon.Add("Icon1");
sIcon.CreateFrom("c:\\Clock.png", IconOutputFormat.FromWin95);
```

`CreateFrom` is a method exposed on `SingleIcon` class, this method will take a input image that must be 256x256 pixels and it must be a 32bpp (alpha channel must be included), the perfect candidate for this method are PNG24 images created for PhotoShop or any Image editing software.

The second parameter in the API is a flag enumeration that target the OS which we want to create the icon, in the previous example it will take the input image and it will create the following IconImage formats.

256x256x32bpp (PNG compression)
48x48x32bpp
48x48x8bpp
48x48x4bpp
32x32x32bpp
32x32x8bpp
16x16x32bpp
16x16x8bpp

There are 14 possible enumerations defined, but they can be combined to get whatever format the developer is looking for.

This method make use of the whole library to provide the best IconImage for each format.

Ohh Microsoft policy about compatibility is changing?

Something I have to comment about because I think it is a breakthrough on how Microsoft usually does things from my point of view.

I have been developing on Windows platform for the last decade from the Windows 3.1 to date, and something that I saw in Microsoft APIs is the amazing compatibility between versions. Personally I think many Win32 APIs are so intrinsic and complicated because they had to keep backward compatibility, and I had so many headaches in the last years because of it.

For example the huge show stopper for Windows future generation was the GDI that imposed a set of rules that could not be broken in any way, GDI+ helped but still ran under the GDI rules, and that is the reason why there are things that Windows could never do until now.

This happened when I decided to implement Windows Vista icons support.

I read that Windows Vista icons are 256x256 and they use PNG compression for them.

At first I was 100% convinced they were going to keep backward compatibility, so I started to think how they did it. The first thing that came to my mind was that Microsoft boys were going to use the field `biCompression` in the header `BITMAPINFOHEADER`and instead set to `BI_RGB` (BMP). They were going to use `BI_PNG`(PNG) that is already supported in the header, the palette was going to be empty and the `XOR` and `AND` Image would contain the PNG data.

I was surprised when that didn't happen. Instead they completely dropped the concept of having a `BITMAPINFOHEADER`, the image (`XOR`) and the mask (`AND`). Instead, the icon directory pointed to a 100% PNG structure.

At first I thought, 'oh my God what have they done!'

This was going to break all Icons Editors out there, also Visual Studio and Resource Editors won't be able to open ICO files anymore, but when I sat and thought about it, it occurred to me that it was the right way to go.

Developers have always complained about how complicated some Win32 APIs are, and this time Microsoft heard that and did things right.

If they could have kept compatibility, it would mean that now ICO and ICON libraries could have 3 places with redundant information about each image.

Like `ICONDIRENTRY`, `BITMAPINFOHEADER`, and `PNGHEADER` usually find those bizarre things in Win32 API.

Instead now they have the Icon directory entry that points to the image itself. That way, they open the way for future implementation for different images or compressions. Still ICO files are limited by a maximum of 256x256 pixels because the Icon directory stores width and height in two byte type fields `bWidth`and `bHeight`. Probably that can be resolved using more than one Plane. But anyway still we are far from use ICONS with more than 256x256 pixels.

So this time I congratulate the boys at Microsoft for thinking "what is the best way to do it" over anything else.

If you wonder if this means VS2005 or any VS won't be able to open properly ICO or DLLs from Windows Vista, then you are right, it WON'T. I also tested ORCAS (VS2006) and it doesn't support it. But that can be easily resolved with a VS patch that hopefully will come out soon, else you will have a product like this library that will support Windows Vista Icons.

Roadmap

`IconLib` is a powerful library to allow icons or icon libraries creation and modifications. I plan to support updates for DLLs and EXE in the next version, allowing to replace/add/delete icons inside them.

`IconLib` alone is only useful from a programming language. So, I also plan to create an advance Icon Editor application to make full use of `IconLib`. That will probably be my next article in the next few months.

If I can get file formats like .icc (Icons collection), Icns, RSC, bin (mac), I'll support them. If you know of some file format and you have the internal file structure, let me know and I'll try it to implement it.

If someone is interested in creating an open-source Icon Extractor & Editor, then he or she is welcome to use `IconLib` as the file formats engine and I can provide support for `IconLib`.

History

IconLib 0.73 (01/31/2008)

- Fixed a small problem with indexed 8bpp images.
- Properly processing when adding PNG24 images.
- Automatic Icon creation from a PNG or BMP32 for Vista, XP, W95 and Win31.
- Added a new namespace "ColorProcessing" which supports
    - Color Reduction
    - Dithering
    - Palette Optimization

- Allow to save an IconImage as PNG or BMP32 with transparency.
- SingleIcon.Add() methods now returns a reference the IconImage just been created.
- Some code and method signatures changes but backward compatible.
- Demo application allows to export XOR, AND and Transparent Image, also now IconImages can be exported as PNG24 or BMP32.

IconLib 0.72 (11/02/2006)

- Change default shift factor from 9 to 10 for ICL libraries. (now it supports 64MB Max ICL file size).
- Re-coded function to make vertical flip over Black&White images using pointers and memcopy (increased performance).
- Different Namespaces for Bitmap Encoders and Library Formats.
- Removed Static classes for Library Formats and replaced for a Interface from which the different formats implement.
- Included `IconLib` license type.

IconLib 0.71 (Initial Release)

License

References

- Icons in Win32
- Wikipedia Icon Image File Format
- New-style EXE Format
- Executable-File Header Format
- Microsoft Portable Executable
- Portable Executable
- Floyd-Steinberg dithering
- Floyd-Steinberg dithering source
- Octree structure
- Octree Quantizer implementation

License

Share

About the Author

CastorTiu
Software Developer Microsoft
United States

I started with programming about 19 years ago as a teenager, from my old Commodore moving to PC/Server environment Windows/UNIX SQLServer/Oracle doing gwBasic, QBasic, Turbo Pascal, Assembler, Turbo C, BC, Summer87, Clipper, Fox, SQL, C/C++, Pro*C, VB3/5/6, Java, and today loving C#.

Currently working as SDE on Failover Clustering team for Microsoft.

Passion for most programming languages and my kids Aidan&Nadia.

You may also be interested in...

SAPrefs - Netscape-like Preferences Dialog

OLE DB - First steps

Generate and add keyword variations using AdWords API

Introduction to D3DImage

Window Tabs (WndTabs) Add-In for DevStudio

Creating alternate GUI using Vector Art

Comments and Discussions

Add a Comment or Question

Search Comments

First Prev Next

Any plans to support .msi format?
Member 13063306 16-Mar-17 21:31

How to save an ico file with images from several pictureboxes?
Mc Gwyn 16-Feb-17 20:11

Great library
StoneFactory 4-Jul-16 23:44

Bug?

gyrtenudre 15-Dec-15 16:46

Re: Bug?
pcbbc 8-Sep-17 21:39

Visual Studio 2012
paolo guccini 15-Jun-15 19:27

Nice
NikolaB 21-Apr-15 19:03

My vote of 5
RedDk 17-Apr-15 1:39

very nice project
zhouhu 26-Dec-14 8:34

My vote of 5
WLDNA 12-Aug-14 10:11

Can we get some documentation?
Will Pittenger 23-Jul-14 20:20

My vote of 5
liruikuan 2-Jul-13 15:39

My vote is 5 of 5
Lil' D 'mBoss 26-May-13 2:06

My vote of 5
Tapirro 6-Feb-13 18:59

Will Saving to a DLL be possible?
lucaso 9-Dec-12 6:01

My vote of 5
malparry 17-Sep-12 13:22

My vote of 5
fenriv 13-Aug-12 20:26

License - CC-BY-SA
CyberKnet 25-Jul-12 21:58

FT Icon Studio 2012
fireHLF 1-Apr-12 18:02

IE8 or IE7 can't Icon files exported from IconLib
CPMJim 26-Mar-12 10:15

My vote of 5
elea30 29-Dec-11 20:58

My vote of 4
leochow 7-Dec-11 14:22

Error with .bmp; works somewhat with.png
Member 8246511 30-Nov-11 11:00

My vote of 5

ZotovBST **28-Sep-11 3:22**

A Suggestion
Rick York **7-Sep-11 5:20**

Refresh

1 2 3 4 5 6 7 8 9 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.