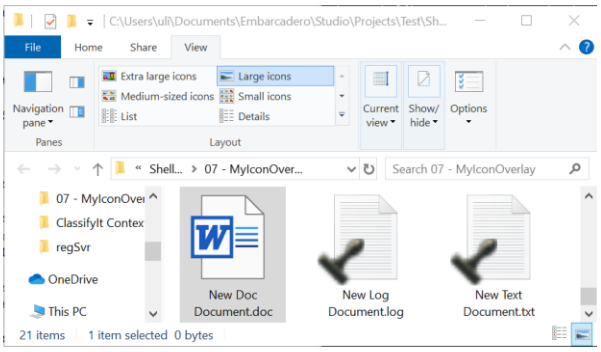
Delphi Tutorial – Step-by-Step
Windows Shell Extension – Icon Overlay

Scope

This is a Delphi tutorial for implementation of a Windows Shell (Explorer) Extension in form of an Icon Overlay, which allows the display of a small graphical indication to the user if a file meets a certain condition. The figure to the right shows the overlay of a stamp symbol for all .log and .txt files. Although the conditions to display the overlay or not can be programmed very detailed and is not only dependent on the file-name extension, the overlay-icon is very limited … it cannot be changed dynamically ☹ … in other words “only one icon per icon-overlay.



The tutorial provides a full step-by-step guide building a Delphi project from scratch to achieve the additional context menu functionality on a Windows Explorer, as shown on the figure.

Background

A Windows Shell Extension is expanding the function of the Windows Explorer and adds additional functionality, like a context menu when right-clicking on a file or a selection of files.

This tutorial provides a step-by-step (idiot) guide with screenshots and code snippets you can copy and paste.

Prerequisite

You need a Delphi Compiler - for this project I used Delphi 10 Seattle.

You need Windows Operating System.

Familiarise with the Tutorial - Delphi - Shell Extension - Context Menu Part 1

Consider using a context menu for registering and unregistering DLLs:
https://ugarbe.de/useful/25-shell-extension-register-server-regsvr32

Feedback- Help

Friendly feedback is always welcome: delphi@ugarbe.de

What will you Learn

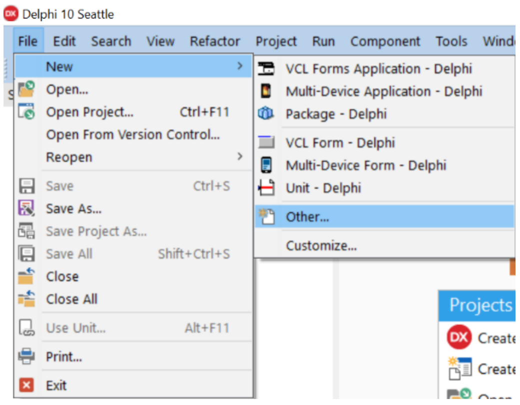
- Delphi
- Active-X COM Object
- Register DLL
- Shell Extension Context Menu

Create an Active-X COM Object

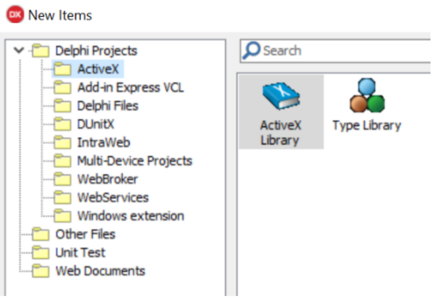
Lets go for it – start your Delphi IDE.

Create an Active-X Library

File -> New -> Other



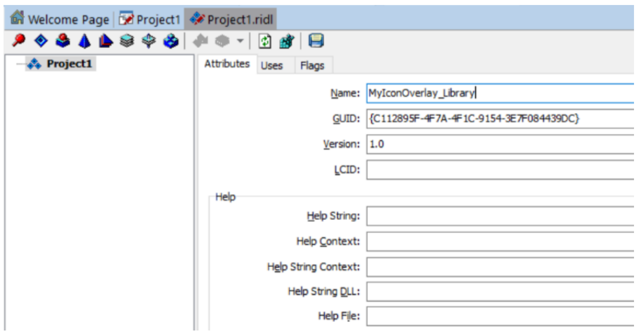
ActiveX -> ActiveX Library



-> a project is created.

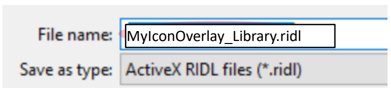
Rename the Attribute Name from Project1 to your plugin name with the _Library. For instance to: MyIconOverlay_Library:

Note the GUID will be different in your code, as this is a global unique identifier.



Name: MyIconOverlay_Library
GUID: {C112895F-4F7A-4F1C-9154-3E7F084439DC}
Version: 1.0
LCID:
Help String:
Help Context:
Help String Context:
Help String DLL:
Help File:

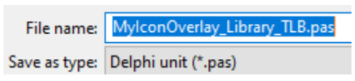
-> Save -> MyIconOverlay_Library.ridl



File name: MyIconOverlay_Library.ridl
Save as type: ActiveX RIDL files (*.ridl)

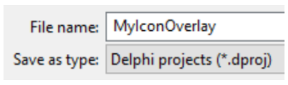
-> Save All ->

To save the unit



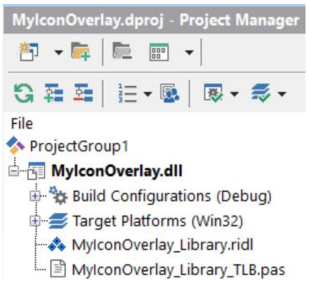
File name: MyIconOverlay_Library_TLB.pas
Save as type: Delphi unit (*.pas)

and the project.



File name: MyIconOverlay
Save as type: Delphi projects (*.dproj)

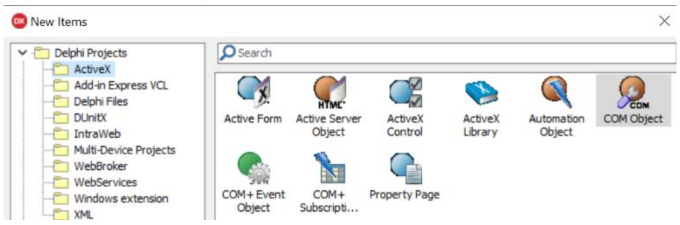
The project should look like this:


MyIconOverlay.dproj - Project Manager

File
ProjectGroup1
MyIconOverlay.dll
Build Configurations (Debug)
Target Platforms (Win32)
MyIconOverlay_Library.ridl
MyIconOverlay_Library_TLB.pas

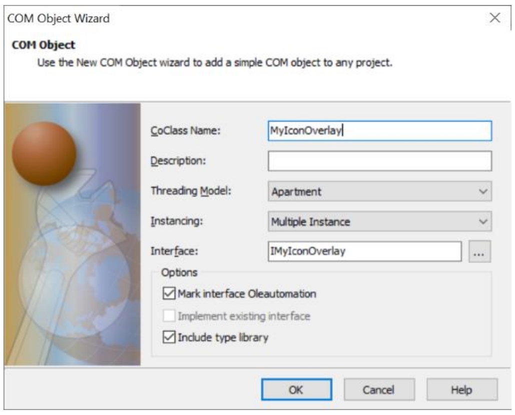
Add a COM Object

-> File -> New -> Other -> ActiveX -> COM Object



-> give the CoClass Name to: MyIconOverlay

The rest should be fine

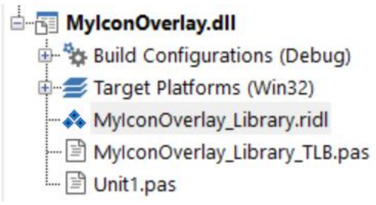

COM Object Wizard

COM Object
Use the New COM Object wizard to add a simple COM object to any project.

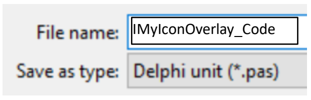
CoClass Name: MyIconOverlay
Description:
Threading Model: Apartment
Instancing: Multiple Instance
Interface: IMyIconOverlay
Options
☑ Mark interface Oleautomation
☐ Implement existing interface
☑ Include type library

OK Cancel Help

There will be a new unit: Unit1

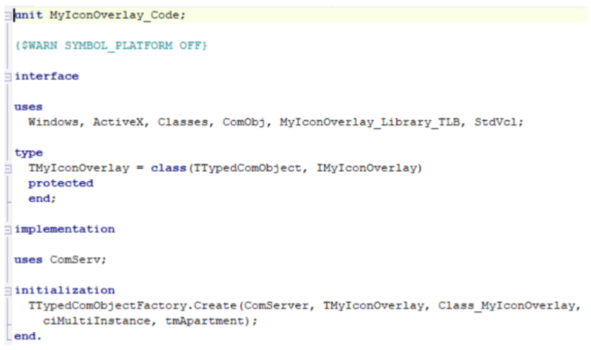

MyIconOverlay.dll
Build Configurations (Debug)
Target Platforms (Win32)
MyIconOverlay_Library.ridl
MyIconOverlay_Library_TLB.pas
Unit1.pas

Open Unit1 and save it as: MyIconOverlay_Code


File name: IMyIconOverlay_Code
Save as type: Delphi unit (*.pas)

Result of COM Object Creation

Switch to the MyIconOverlay_Code. The unit code looks like this:



```
unit MyIconOverlay_Code;

{$WARN SYMBOL_PLATFORM OFF}

interface

uses
  Windows, ActiveX, Classes, ComObj, MyIconOverlay_Library_TLB, StdVcl;

type
  TMyIconOverlay = class(TTypedComObject, IMyIconOverlay)
  protected
  end;

implementation

uses ComServ;

initialization
  TTypedComObjectFactory.Create(ComServer, TMyIconOverlay, Class_MyIconOverlay,
    ciMultiInstance, tmApartment);
end.
```

Build the Code for the Shell Extension

Create an ObjectFactory for initialising the COM Object

Now we adopt the code to write the initialisation handler which must register our COM object (the Class we defined: Class_MyIconOverlay) and add a procedure to registry keys in the Windows Registry to make the shell extension known to the Explorer.

For this we define a new type: TMyIconOverlay_Factory with an UpdateRegistry procedure and adopt the code in the initializsation section to create and instance of the COM object. Lets also add a finalization section for later.

Here the code for copy pasting:

```
unit MyIconOverlay_Code;

{$WARN SYMBOL_PLATFORM OFF}

interface

uses
  Windows, ActiveX, Classes, ComObj, MyIconOverlay_Library_TLB, StdVcl;

type
  TMyIconOverlay = class(TTypedComObject, IMyIconOverlay)
  protected
  end;

  TMyIconOverlay_Factory = class (TComObjectFactory)
  public
    procedure UpdateRegistry (Register: Boolean); override;
  end;

implementation

uses ComServ;

initialization
  TMyIconOverlay_Factory.Create(ComServer, TMyIconOverlay, Class_MyIconOverlay,
'testing', ciMultiInstance, tmApartment);

finalization

end.
```

Declare the Shell Extension Interfaces

Now we tackle the type definition of TMyIconOverlay to provide the interfaces required for a shell extension. Pending which kind of extension you wish to implement different interfaces need to be provided. The Icon Overlay extension requires:

class(TComObject, IShellIconOverlayIdentifier)

We need to add the *shlObj* in the uses clause, which defines the IShellIconOverlayIdentifier, which requires 3 specific procedures being provided by the object. For more info on there query the Microsoft webpages.

```
uses
  Windows, ActiveX, Classes, ComObj, MyIconOverlay_Library_TLB, StdVcl, ShlObj;

type
  TMyIconOverlay = class(TComObject, IShellIconOverlayIdentifier)
  private

  public
    function IsMemberOf(pwszPath: LPCWSTR; dwAttrib: DWORD): HResult; stdcall;
    function GetOverlayInfo(pwszIconFile: LPWSTR; cchMax: Integer; var pIndex:
Integer; var pdwFlags: DWORD): HResult; stdcall;
    function GetPriority(out pPriority: Integer): HResult; stdcall;
  end;
```

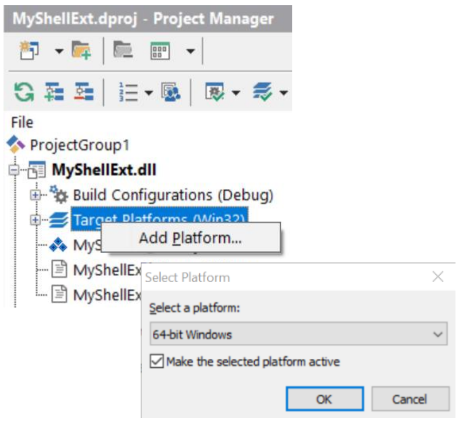
Remember that the Icon Overlay shell extension can only provide a specific (one) icon overay to a file-icon or not. If a file in Explorer is displayed with that overlay icon is decided in the IsMemberOf function. GetOverlayInfo links to the icon which will be the overlay and GetPriority can set a priority value, should there my multiple overlays for the same file.

Adjust Delphi Compiler Options

Before compiling the code adjust the Target Platform to Win64 which is the 64-bit code which most of the Windows installations require today.

-> right click on Target Platforms (Win32) -> Add Platform

-> select 64-bit Windows

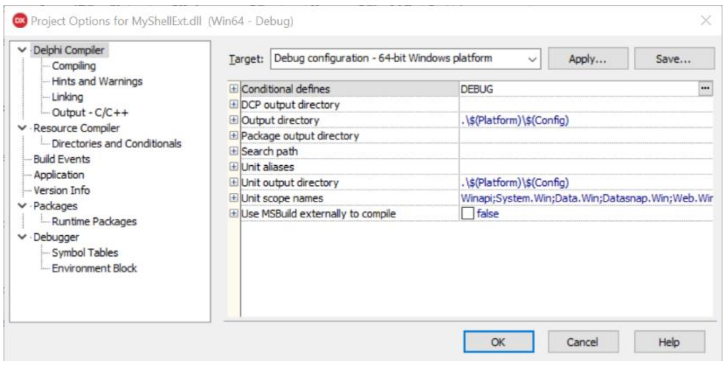


Select a platform: 64-bit Windows

☑ Make the selected platform active

A proposed adjustment is the location where Delphi stores the complied code, the MyShellExt.dll in our case. This step is not important and is purely to support my style of working 😊

-> Project -> Options

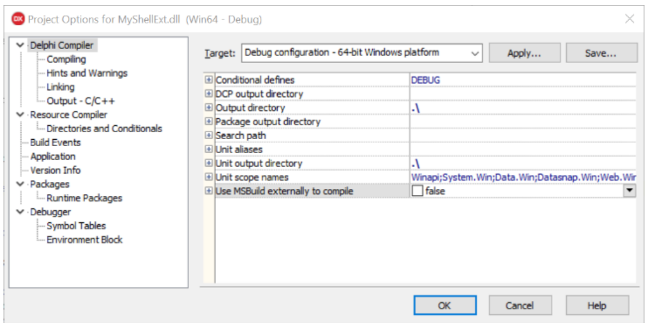


Target: Debug configuration - 64-bit Windows platform

Conditional defines	DEBUG
DCP output directory	
Output directory	.\$(Platform)\$(Config)
Package output directory	
Search path	
Unit aliases	
Unit output directory	.\$(Platform)\$(Config)
Unit scope names	Winapi;System.Win;Data.Win;Datasnap.Win;Web.Wir
Use MSBuild externally to compile	☐ false

Then you can change the Output Directory and Unit Output Directory entries to .\

After this the .dll will be created in the same directory as where the project is saved. (otherwise the .dll will be in sub-directories – this).



Target: Debug configuration - 64-bit Windows platform

Conditional defines	DEBUG
DCP output directory	
Output directory	.\
Package output directory	
Search path	
Unit aliases	
Unit output directory	.\
Unit scope names	Winapi;System.Win;Data.Win;Datasnap.Win;Web.Wir
Use MSBuild externally to compile	☐ false

Prepare the Interface Functions Implementation for Compiler Test

Now we prepare the implementation of all functions which have been defined for the defined 2 types: TMyIconOverlay and TMyIconOverlay_Factory.

After this step we should be able to compile the code and receive the .dll in the code directory.

Please ensure this works. There will be warnings but there must be no errors.

```
implementation

uses ComServ, Registry, SysUtils;

function TMyIconOverlay.IsMemberOf(pwszPath: LPCWSTR; dwAttrib: DWORD): HResult;
begin
end;

function TMyIconOverlay.GetOverlayInfo(pwszIconFile: LPWSTR; cchMax: Integer; var
pIndex: Integer; var pdwFlags: DWORD): HResult;
begin
end;

function TMyIconOverlay.GetPriority(out pPriority: Integer): HResult;
begin
end;


procedure TMyIconOverlay_Factory.UpdateRegistry (Register: Boolean);
begin
end;
```

Add the functional code

To see that we are on the right track we need to provide code to the following 2 functions: we need to update the registry, so the Explorer knows which object is serving an Explorer event – in our case a right mouse click.

Registry Entries for Context Menu Handlers

First we link our shell extension class (Class_MyIconOverlay) to the icon overlay handlers. This is done through the registry with this code. When later we will register our .dll this procedure is called with the value True, if we unregister it will be called with the value False.

```
procedure TMyIconOverlay_Factory.UpdateRegistry (Register: Boolean);
var
  Reg: TRegistry;
begin
  inherited UpdateRegistry (Register);
  Reg := TRegistry.Create;
  Reg.RootKey := HKEY_LOCAL_MACHINE;
  try
    if Register then
      if  Reg.OpenKey('\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\ShellIconOverlayIdentifiers\MyIconOverlay', True) then
        Reg.WriteString('', GUIDToString(Class_MyIconOverlay));
    if not Register then
      if  Reg.OpenKey('\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\ShellIconOverlayIdentifiers\MyIconOverlay', False) then
          Reg.DeleteKey('\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\ShellIconOverlayIdentifiers\MyIconOverlay');
  finally
    Reg.CloseKey;
    Reg.Free;
  end;
end;
```

If Register the registry key is created and under the default value the GUID (Globally Unique Identifier) to our object is provided. If Register is false the key will be deleted from the Registry.

Decide if the Icon Overview is Displayed

Now we provide the most important code to decide if the overlay icon is displayed or not. Whenever explorer is to display an icon for a file our ***IsMemberOf*** function is called. If we return the result S_OK, the overlay-icon will be shown, if we return S_FALSE then the overlay-icon is not shown. The below example will display the overlay icon for all files ending with '.txt' and all files starting with the character 'z'. (Note: textfiles and files ending with '.txt' is not the same – although '.txt' files are textfiles also '.TXT' files are textfiles, which this examples will not catch).

pwszPath provides a pointer to the full full path and name of the file. We can convert this easy to a string holding the full file name, including its path. The code then examines this string to either leave the *Result* at S_FALSE, or for the 2 conditions, ending with '.txt' or starting with 'z', returning S_OK.

```
function TMyIconOverlay.IsMemberOf(pwszPath: LPCWSTR; dwAttrib: DWORD):
HResult;
var
  fullFileName: string;
begin
  Result := S_FALSE;            // default no overlay-icon display
  fullFileName := pwszPath;     // convert filename and path to string
  if fullFileName.EndsWith('.txt') then
    Result := S_OK;  // all .txt files with overlay-icon
  if extractFileName(fullFileName).StartsWith('z') then
    Result := S_OK;  // all files starting with z, with overlay-icon
end;
```

Of coarse the *IsMemberOf* function can also read the file or make any other checks to trigger the result. But remember Explorer will trigger this function for each file before displaying it!! Slow code in this function will slow down Explorer.

Set the Icon Overlay-Icon

The GetOverlayInfo function is to provide the full path to the Overlay-Icon. There are 2 ways of doing this, either providing the icon-file, or providing a container file (e.g. a .dll) and the index of the icon to be used. I was not able to make the code running using the container file – if you identify a way, please contact under delphi@ugarbe.de. So in this example we will provide a single icon-file, by casting the name to the ***pwszIconFile*** variable – set the highlighted PATH to your directory. The icon file should be small, e.g. 16x16 pixels.

Important Note: this function is only called once, when Explorer starts or restarts. So any dynamic use to specify different overlay icons will not work! Also, when we register or un-register our DLL later, we need to enforce Explorer restarts to see the desired effects (more details further down).

Set Overlay Priority

In Explorer up to 15 overlay handlers can be impleted. If there is conflict Explorer will first try to resolve by comparing priority values each overlay handler can set. Values between 0 (highest priority) to 100 (lowest priority) can be set.

The priority value is set by the ***GetPriority*** function. In this example we set to highest priority.

```
function TMyIconOverlay.GetPriority(out pPriority: Integer): HResult;
begin
  pPriority := 0; // highest priority
  Result := S_OK;
end;
```

We are done 😊 - lets test the code.

First testing of code

Now is the time to test and validate that our code works 😊

In principle there are 3 steps: Compile, register the DLL, make Explorer aware of the DLL.

Compile Code

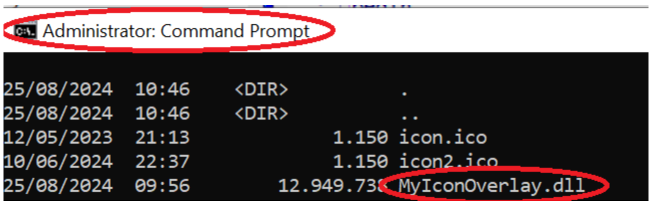
Compile the code and ensure there are no errors. After compilation your project directory should have the MyIconOverlay.dll file.

Register the DLL

There are many was of doing this. For the beginning just use the command line interface. Although this is quite cumbersome after a while, you need to know the basiscs (there are many articles in the web):

-> open CMD with administrator privileges!
… and go to the directory of your project.

When you list the directory (dir) you will find our MyIconOverlay.dll file.



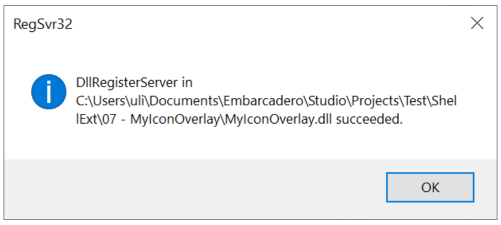
```
25/08/2024  10:46    <DIR>          .
25/08/2024  10:46    <DIR>          ..
12/05/2023  21:13             1.150 icon.ico
10/06/2024  22:37             1.150 icon2.ico
25/08/2024  09:56        12.949.738 MyIconOverlay.dll
```

Ensure the command prompt runs in Administrator mode (top of screenshot).

Now you can register the dll with the following command:

```
regsvr32 MyIconOverlay.dll
```

The registration will be confirmed of being successful.



DllRegisterServer in C:\Users\uli\Documents\Embarcadero\Studio\Projects\Test\ShellExt\07 - MyIconOverlay\MyIconOverlay.dll succeeded.

Note: at this moment the overlay is very likely not visible in Explorer.

Tip: if you work regulary with Shell Exentsions or .DLLs you could use the freeware tool, RegSvr, which is a Shell Extension allowing registration and de-registration through Explorer's context menu (right click): https://ugarbe.de/useful/25-shell-extension-register-server-regsvr32

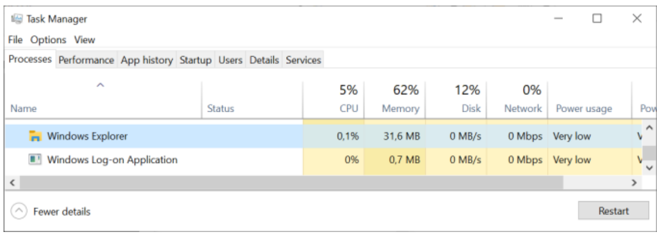
Restart Explorer

In order to activate our overlay handler we need to restart Explorer. Although there are many ways of doing this, I recommend to use the **Task Manager** for this.

Start the Task Manager. You can let it run in the background during the testing activities.

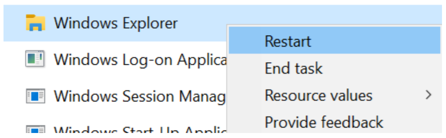
Close all all Explorer windows you might have open thought the Windows GUI.

Go to Task Manager and find the Windows Explorer process – you might have to refresh the window by pressing F5. The Windows Explorer process is likely in the lower part of the Task Manager listed processes.

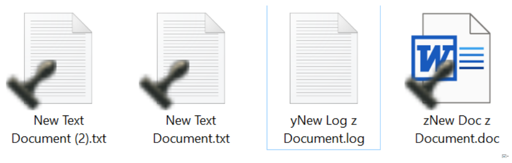

Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fewer details
Restart

Name	Status	5% CPU	62% Memory	12% Disk	0% Network	Power usage	Pow
Windows Explorer		0,1%	31,6 MB	0 MB/s	0 Mbps	Very low	V
Windows Log-on Application		0%	0,7 MB	0 MB/s	0 Mbps	Very low	V

Right click on the icon and select Restart.



If you open now an Explorer window and look on a file which ends with .txt, or starts with a z, then you should see the icon-overlay 😊



Disable the Shell Extension

If you want to change anything on the Shell Extension, the shell extension needs to be un-registered and Explorer needs to be restarted. Otherwise you cannot delete the DLL file, nor can you compile a new version.

On the command line (CMD) in admin mode, go the directory where the DLL file is stored. Use the command:

```
regsvr32 -u MyIconOverlay.dll
```

Afterwards close all Explorer windows and restart Explorer through the Task Manager.

In most cases you should be able to remove or overwrite the MyIconOverlay.dll file. Note: sometimes Windows is still not allowing the removal, try restart Explorer a further time – as last resort, restart Windows.

Tip: if you use the freeware tool, https://ugarbe.de/useful/25-shell-extension-register-server-regsvr32, you can configure it to automatically restart Explorer when unregistering a DLL file. It will also open Explorer at the path where the DLL was unregistered.