SD10670

Exposing Hidden Functionality for AutoCAD .NET Application Development

James Johnson,
Synergis Technologies, LLC, Sr. Application Developer

Learning Objectives

- Learn how to extend the AutoCAD .NET managed capability
- Discover exported functions in AutoCAD files that can be imported to managed code
- Learn how to get the structure of the exported functions and import them into managed code
- Discover Platform Invoke (P/Invoke)

Description

This instructional demonstration is an introduction for AutoCAD.NET API software developers to learn how to extend the managed ObjectARX libraries to use hidden functionality. Although the standard AutoCAD .NET managed libraries provide extensive support, there is some functionality that is not wrapped into the managed libraries. In this instructional demonstration we will discuss ways to find that hidden functionality in AutoCAD software files and ObjectARX libraries. The instructional demonstration will then show how to make that functionality available to managed .NET applications. The instructional demonstration will provide sample code that you can modify and use in your applications.

Speaker

***James Johnson** has worked with CAD products for more than 25 years in many positions, from being a CAD drafter to writing automation applications. In his current position he is doing CAD integration for adept document management system. In previous positions he used RealDWG® to write custom automation to create AutoCAD software drawings of industrial kitchen equipment, and he worked at Autodesk, Inc., resellers in software development groups doing custom applications for Inventor software and AutoCAD software. He has taught AutoCAD software and AutoCAD Microsoft Visual Basic software classes while working for resellers, and he was a CAD instructor at 2 different community colleges.*

Introduction

The .NET (Microsoft® .net Framework) and the ObjectARX® programming environment libraries, provides object-oriented C++, C# and VB .NET application programming interfaces for developers to customize and extend AutoCAD®. These libraries provide direct access to the AutoCAD® database, graphics system, and native command definition. Some capabilities are not exposed with .NET ObjectARX® and can be by using Platform Invoke (P/Invoke) to make exported capabilities in AutoCAD® and native ObjectARX® available in .NET applications. This document will show some ways as to how to find exported functionality and then make it available.

The requirements for the projects discussed in this class and document require Visual Studio, AutoCAD and ObjectARX libraries. Additionally briefly discussed will be using the AutoCAD ObjectARX wizards that create projects from predefined templates.

Note: This is an intermediate class, assumption is made that attendees have a basic understanding of basic programming concepts (including some .NET programming), and have experience with customizing AutoCAD®. Knowing how to create a .NET project that creates a DLL to be load into AutoCAD® is helpful, but is briefly covered later in this handout.

Learn how to extend the AutoCAD® .NET managed capability

There are several ways to extend managed code by building your own API Class Library DLL's that can then be referenced into your applications. These DLL's can be built from C#, VB.NET and or C++/CLI projects. The classes and methods can then be exposed and used in the same way that the AutoCAD DLL's are referenced and used.

Classes and Methods

To expose classes and methods in a Class Library DLL they need to have public availability and should be built using standard framework design guidelines.

Classes, Abstract Classes and Interfaces for Inheritance

The class library can define classes, abstract classes and interfaces that are predefined that when the class library DLL is referenced into the new application can be inhereited. The classes can contain abtract methods that can then be implemented or overridden.

Platform Invoke (P/Invoke)

Using Platform Invoke allows adding functionality that is exported in executables and DLL's that cannot be referenced into managed code. In further sections of this document finding the exported methods and several different ways of importing them will be shown.

Extension Methods

Extension methods allow the creation of methods on an existing "Type" creating the illusion of new methods, even when a class is compiled outside of the current assembly and can not be changed. With extension methods developers can augment the "Type" with new methods to provide their own specialized implementation. Extension methods are defined in static classes as static methods. In C#, extension methods are indicated by the "this" modifier which must be applied to the first parameter of the extension method.

```
public static class extendAcad
{
  public static BlockTableRecord GetBlockTableRecord(this Database _db, bool mspace)
  {
    string space = mspace ? BlockTableRecord.ModelSpace : BlockTableRecord.PaperSpace;
    BlockTable bt = (BlockTable)_db.TransactionManager
                                .TopTransaction
                                .GetObject(_db.BlockTableId, OpenMode.ForRead);
    BlockTableRecord btr =
                (BlockTableRecord)_db.TransactionManager
                                  .TopTransaction
                                  .GetObject(bt[space], OpenMode.ForRead);
    return btr;
  }

  public static IEnumerable<DBText> GetDBTextObjects(this Database _db, bool mspace)
  {
    BlockTableRecord btr = _db.GetBlockTableRecord(mspace);
    IEnumerable<DBText> dbTxt = btr.Cast<ObjectId>()
            .ToList<ObjectId>()
            .ConvertAll<DBObject>(oid =>
              { return  _db.TransactionManager
                          .TopTransaction
                          .GetObject(oid, OpenMode.ForRead, false);
              })
            .Where<DBObject>(obj => obj.GetType() == typeof(DBText))
            .Cast<DBText>().AsEnumerable();
    List<DBText> dbtxt = btr.Cast<DBText>().Select(dbo => (DBText)dbo).ToList();
    return dbTxt;
  }
}
```

The type of the first parameter of an extension method indicates what type the extension applies to.

Extension methods are resolved at compile-time based on which extension methods are in scope. When a namespace is imported with C#'s using statement or VB's Import statement, all extension methods that are defined by static classes for that namespace are brought into scope.

Many of the LINQ standard query operators are extension methods in the System.Linq namespace, these extension methods extend IEnumerable<T> and IQueryable<T>. All but a few (e.g. OfType) of the standard query operators extend from the IEnumerable<T> interface, which means that all IEnumerable<T> types get the standard query operators by adding the using statement in C#: using System.Linq;

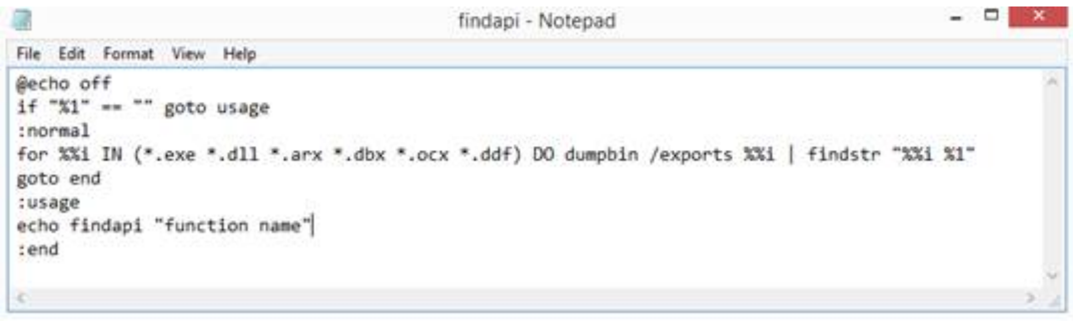
Discover exported functions in AutoCAD® files that can be imported to managed code

The inspiration behind submitting this class was due to that in AutoCAD 2016 a new undelay type was added to make references to Navisworks files and the application that I was working on at the time needed a way to get the full path and name of the new Navisworks object. The problem is that the object was not exposed in the AutoCAD/RealDWG .NET ObjectARX API. Since there was minimum documentation on this new object or any objects for it in the "Object Browser" of the referenced DLL's, decided to look into if it could be found in a different way and found mangled functions exported in the new ObjectDBX vertical custom entity file "AcBIMUnderlayDbx.dbx". Continued looking into it and found a way to get that path which will be used in this document for reference.

The following is a way to find the object and its full path and name using P/Invoke.

Created a .bat file using "dumpbin.exe"

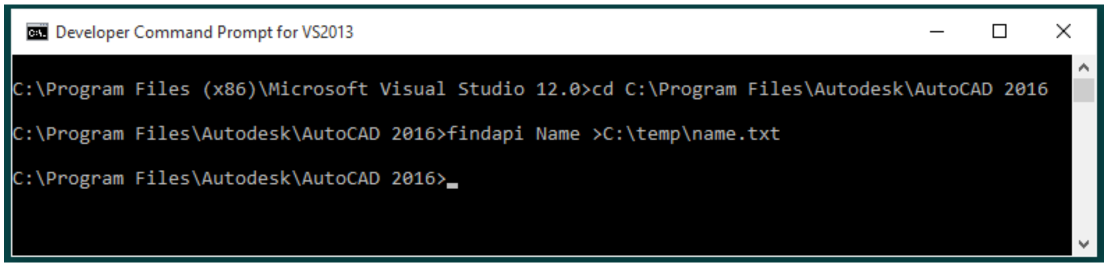
The basic structure of the findapi.bat is documented on several websites explaining how to use P/Invoke. The differences here are that typically .dbx and other extensions are not included.



```
@echo off
if "%1" == "" goto usage
:normal
for %%i IN (*.exe *.dll *.arx *.dbx *.ocx *.ddf) DO dumpbin /exports %%i | findstr "%%i %1"
goto end
:usage
echo findapi "function name"
:end
```

Copied the above findapi.bat to the AutoCAD folder and ran in a command window

After copying the findapi.bat file to the AutoCAD folder it was now time to run it and collect the output into a text file. Start VS2013 developer command window which has the path to dumpbin preset, changed to the AutoCAD folder and then ran the findapi.bat with a search string and output file name.



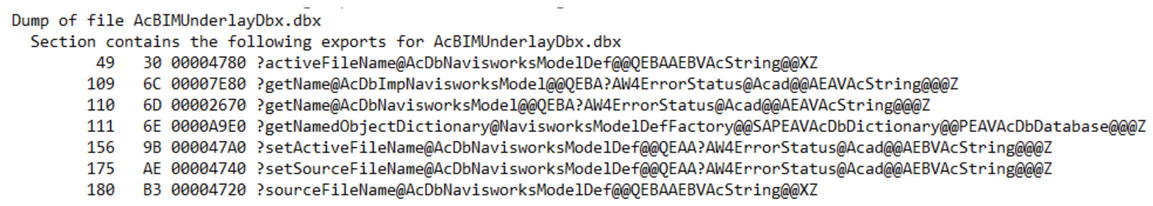
```
C:\Program Files (x86)\Microsoft Visual Studio 12.0>cd C:\Program Files\Autodesk\AutoCAD 2016

C:\Program Files\Autodesk\AutoCAD 2016>findapi Name >C:\temp\name.txt

C:\Program Files\Autodesk\AutoCAD 2016>_
```

In the Name.txt file created from running findapi.txt, found the following

The text file contains the mangled representation of the exported functions in the files. The ones of interest here were the 'getName' functions that are exported in the 'AcBIMUnderlatDbx.dbx' file.



```
 Dump of file AcBIMUnderlayDbx.dbx
   Section contains the following exports for AcBIMUnderlayDbx.dbx
          49   30 00004780 ?activeFileName@AcDbNavisworksModelDef@@QEBAAEBVAcString@@XZ
         109   6C 00007E80 ?getName@AcDbImpNavisworksModel@@QEBA?AW4ErrorStatus@Acad@@AEAVAcString@@@Z
         110   6D 00002670 ?getName@AcDbNavisworksModel@@QEBA?AW4ErrorStatus@Acad@@AEAVAcString@@@Z
         111   6E 0000A9E0 ?getNamedObjectDictionary@NavisworksModelDefFactory@@SAPEAVAcDbDictionary@@PEAVAcDbDatabase@@@Z
         156   9B 000047A0 ?setActiveFileName@AcDbNavisworksModelDef@@QEAA?AW4ErrorStatus@Acad@@AEBVAcString@@@Z
         175   AE 00004740 ?setSourceFileName@AcDbNavisworksModelDef@@QEAA?AW4ErrorStatus@Acad@@AEBVAcString@@@Z
         180   B3 00004720 ?sourceFileName@AcDbNavisworksModelDef@@QEBAAEBVAcString@@XZ
```

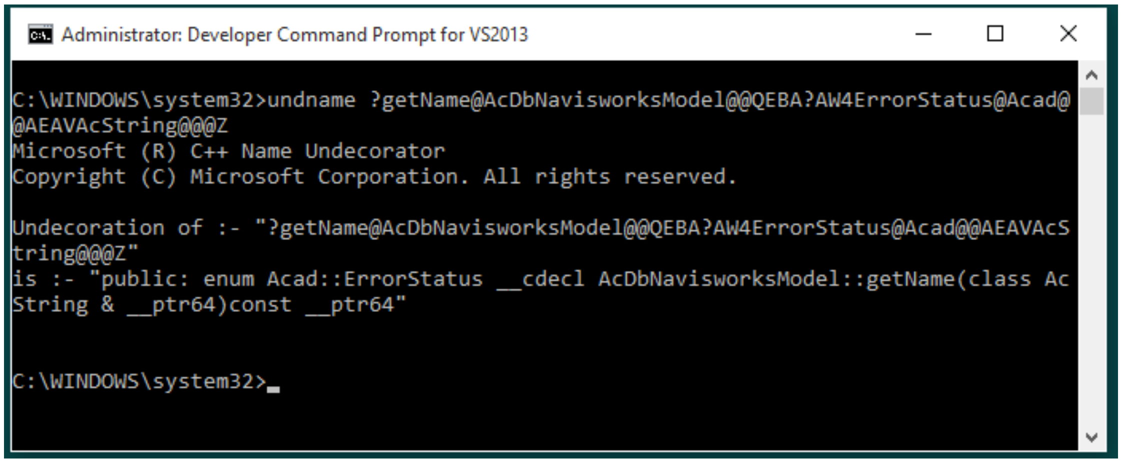
?getName@AcDbNavisworksModel@@QEBA?AW4ErrorStatus@Acad@@AEAVAcString@@@Z

Learn how to get the structure of the exported functions and import them into managed code.

Once "dumpbin.exe" is ran through the findapi.bat file referenced above the output text file has mangled representations of the functions that are exported in the files that are scanned. These mangled representations are not always easily understood, but there is a system tool that will un-mangle that representation and produce a representation that more resembles something that would represent code, showing the arguments and return types.

Then took the mangled string and used "undname.exe" to get the methods structure

This gives an 'un-mangled' representation that helps with understanding the structure of the function in somewhat of a code representation to assist in creating the P/Invoke entry.



Administrator: Developer Command Prompt for VS2013

```
C:\WINDOWS\system32>undname ?getName@AcDbNavisworksModel@@QEBA?AW4ErrorStatus@Acad@
@AEAVAcString@@@Z
Microsoft (R) C++ Name Undecorator
Copyright (C) Microsoft Corporation. All rights reserved.

Undecoration of :- "?getName@AcDbNavisworksModel@@QEBA?AW4ErrorStatus@Acad@@AEAVAcS
tring@@@Z"
is :- "public: enum Acad::ErrorStatus __cdecl AcDbNavisworksModel::getName(class Ac
String & __ptr64)const __ptr64"


C:\WINDOWS\system32>_
```

This shows that the output type of the ‘getName’ function is of a type ‘AcString’ and in .NET there is no direct conversion of that type to a System.String type. After looking at several different possible ways to convert from ‘AcString’ to System.String it appeared that the easiest would be to do the P/Invoke in a C++/CLI ObjectARX project and then get it into the C# project that required getting the name of the Navisworks underlay object.

Discover Platform Invoke (P/Invoke)

Now to construct this information in a useable form it gets added to the application using Platform Invoke (P/Invoke) by using DLLImport attribute to load and specify the structure of the method.

Then added it to the code to a C++/CLI ObjectARX project

Created a simple C++/CLI ObjectARX project using the ObjectARX wizards. This wizard created the project and files which only required minor changes to have a DLL that could then be referenced into other .NET applications.

In the C++/CLI to do the P/Invoke is accomplished very similar to how it is done in C# projects.

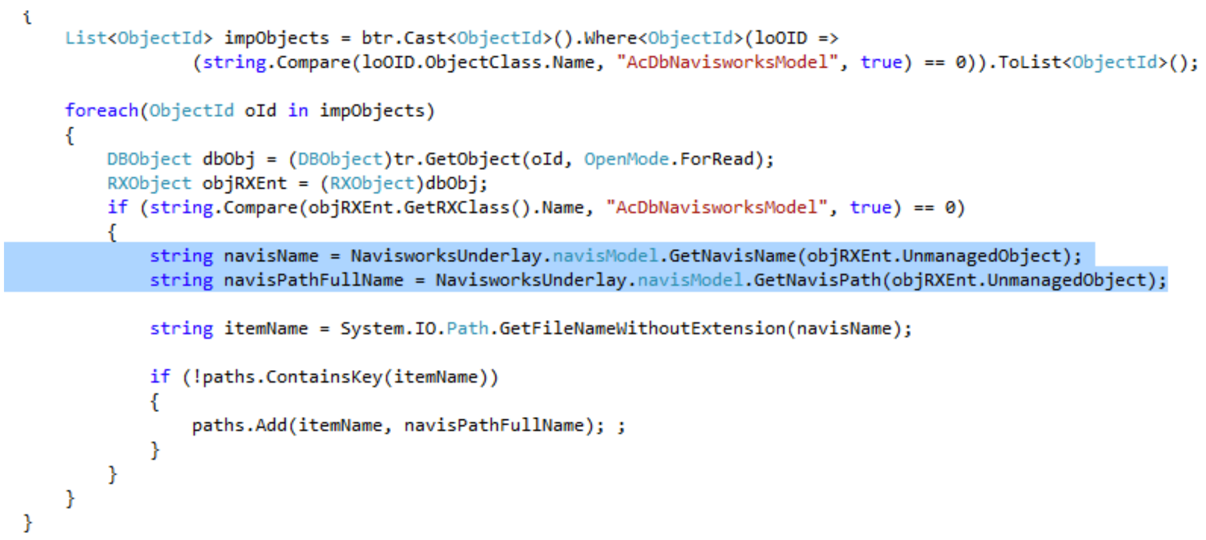
The ‘AcString’ object was easily converted to ‘System::String’ object in the C++/CLI project.

```
// P/Invoke Exported ObjectDBX classes/methods
[DllImport("AcBIMUnderlayDbx.dbx",CallingConvention =
CallingConvention::ThisCall,  CharSet = CharSet::Unicode, EntryPoint =
"?getName@AcDbNavisworksModel@@QEBA?AW4ErrorStatus@Acad@@AEAVAcString@@@Z")]
extern Acad::ErrorStatus getName(IntPtr AcDbNavisworksModel, AcString *str);

// Get Navis Works Overlay objects
String ^ AcadDBXhost::DbxExtender::DWGutil::GetNavisName(IntPtr navisModel)
{
       AcString str;
       getName(navisModel, &str);
       const char *pStrA1 = str.ansiPtr();
       String ^ strName = (gcnew String(pStrA1));
       return strName;
}
```

Then called from a C# assembly that references the C++/CLI DLL

Looking at objects in the drawing and getting their type allows filtering the Navisworks undelay object. The object is by default an “ImpDBObject” and then further its RXClass type is “AcDbNavisworksModel”.



```
{
    List<ObjectId> impObjects = btr.Cast<ObjectId>().Where<ObjectId>(loOID =>
                (string.Compare(loOID.ObjectClass.Name, "AcDbNavisworksModel", true) == 0)).ToList<ObjectId>();

    foreach(ObjectId oId in impObjects)
    {
        DBObject dbObj = (DBObject)tr.GetObject(oId, OpenMode.ForRead);
        RXObject objRXEnt = (RXObject)dbObj;
        if (string.Compare(objRXEnt.GetRXClass().Name, "AcDbNavisworksModel", true) == 0)
        {
            string navisName = NavisworksUnderlay.navisModel.GetNavisName(objRXEnt.UnmanagedObject);
            string navisPathFullName = NavisworksUnderlay.navisModel.GetNavisPath(objRXEnt.UnmanagedObject);

            string itemName = System.IO.Path.GetFileNameWithoutExtension(navisName);

            if (!paths.ContainsKey(itemName))
            {
                paths.Add(itemName, navisPathFullName); ;
            }
        }
    }
}
```

Once the type is found then the call is made to the C++/CLI referenced DLL to return the value.

```
NavisworksUnderlay.navisModel.GetNavisName(objRXEnt.UnmanagedObject);
NavisworksUnderlay.navisModel.GetNavisPath(objRXEnt.UnmanagedObject);
```

Note:

Using the C++/CLI ObjectARX to convert the return values of AcString to System::String is just an example of how creating ObjectARX classes can be used to extend the .NET API.

In addition to exposing simple P/Invoke functions to .NET, a C++/CLI ObjectARX DLL can be used to expose other native ObjectARX code that you may have by exposing those methods to .NET applications.

Additional uses of Platform Invoke (P/Invoke)

Read and write INI files…

```
[DllImport("KERNEL32.DLL",   EntryPoint = "GetPrivateProfileStringW",
SetLastError=true,
CharSet=CharSet.Unicode, ExactSpelling=true,
CallingConvention=CallingConvention.StdCall)]
private static extern int GetPrivateProfileString(
                string lpAppName,
                string lpKeyName,
                string lpDefault,
                string lpReturnString,
                int nSize,
                string lpFilename);

[DllImport("KERNEL32.DLL", EntryPoint = "WritePrivateProfileStringW",
SetLastError = true,
CharSet = CharSet.Unicode, ExactSpelling = true,
CallingConvention = CallingConvention.StdCall)]
private static extern int WritePrivateProfileString(
                string lpAppName,
                string lpKeyName,
                string lpString,
                string lpFilename);

private static string GetIniFileString(string iniFile, string category, string key,int
strSize, string defaultValue)
{
       string returnString = new string(' ', strSize);
       GetPrivateProfileString(category, key, defaultValue, returnString, strSize,
       iniFile);
       return returnString.Split('\0')[0];
}

private static List<string> GetCategories(string iniFile)
{
      string returnString = new string(' ', 65536);
      GetPrivateProfileString(null, null, null, returnString, 65536, iniFile);
      List<string> result = new List<string>(returnString.Split('\0'));
      result.RemoveRange(result.Count - 2, 2);
      return result;
}

public static void WriteIniFileString(string iniFile, string category, string key, string
defaultValue)
{
    WritePrivateProfileString(category, key, defaultValue, iniFile);
}
```

ObjectARX® functions that are not included in the AutoCAD® .NET assemblies can be run by using platform invoke (P/Invoke). Several useful functions to use with AutoLISP® are acedInvoke, acedPutSym and acedGetSym. The acedInvoke function allows running an AutoLISP® function from .NET applications.

```
[System.Security.SuppressUnmanagedCodeSecurity]
[DllImport("accore.dll", CallingConvention = CallingConvention.Cdecl)]
extern static private int acedInvoke(IntPtr args, out IntPtr result);

[CommandMethod("runLisp")]
public static void runLisp()
{
    ResultBuffer rbInvoke = new ResultBuffer(new TypedValue((int)(LispDataType.Text),
    "c:testLisp")); IntPtr ip = IntPtr.Zero;
    int iRes = acedInvoke(rbInvoke.UnmanagedObject, out ip);
    rbInvoke.Dispose();
}
```

The acedPutSym and acedGetSym functions allow getting and changing AutoLISP® variable values.

acedPutSym

```
[System.Security.SuppressUnmanagedCodeSecurity]
[DllImport("accore.dll", CharSet = CharSet.Unicode, CallingConvention =
CallingConvention.Cdecl, EntryPoint = "acedPutSym")]

extern static private int acedPutSym(string name, IntPtr result);

[CommandMethod("putLispSym")]
static public void putLispSym()
{
    Editor ed = Application.DocumentManager.MdiActiveDocument.Editor;
    PromptResult pRes = ed.GetString("\nEnter lisp variable name: ");
    if (pRes.Status == PromptStatus.OK)
    {
       PromptStringOptions pOpt = new PromptStringOptions("Enter new value: ");
       pOpt.AllowSpaces = true;
       PromptResult resVal = ed.GetString(pOpt);
       ResultBuffer rb = new ResultBuffer();
       rb.Add(new TypedValue((int)LispDataType.Text, resVal.StringResult));
       acedPutSym(pRes.StringResult, rb.UnmanagedObject);
    }
}
```

acedGetSym

```
[System.Security.SuppressUnmanagedCodeSecurity]
[DllImport("accore.dll", CharSet = CharSet.Unicode, CallingConvention =
CallingConvention.Cdecl, EntryPoint = "acedGetSym")]

extern static private int acedGetSym(string name, out IntPtr result);

[CommandMethod("getLispSym")]
static public void getLispSym()
{
   Editor ed = Application.DocumentManager.MdiActiveDocument.Editor;
   PromptResult pRes = ed.GetString("\nEnter lisp variable name: ");
   if (pRes.Status == PromptStatus.OK)
   {
      IntPtr rb = IntPtr.Zero;
      int stat = acedGetSym(pRes.StringResult, out rb);
      if (stat == (int)PromptStatus.OK && rb != IntPtr.Zero)
           {
           string s = string.Empty;
           foreach (TypedValue val in (IEnumerable)ResultBuffer.Create(rb, true))
           {
              ed.WriteMessage("\n" + pRes.StringResult + " = " + val.Value.ToString());
           }
       }
   }
}
```

Useful Links discussing using P/Invoke

https://msdn.microsoft.com/en-us/library/aa288468%28v=vs.71%29.aspx?f=255&MSPPError=-2147217396

https://msdn.microsoft.com/en-us/library/aa446536.aspx

http://through-the-interface.typepad.com/through_the_interface/2008/08/using-the-pinvo.html

http://adndevblog.typepad.com/autocad/2012/04/synchronously-send-and-wait-for-commands-in-autocad-using-c-net.html

http://www.quuxsoft.com/Programming/PInvoke.aspx