# Microsoft.DwayneNeed.Win32 Interop Library

## Introduction

The goal of this project is to provide a useful .Net library for interop with traditional Win32 APIs. In particular, the emphasis is placed on the user-interface aspects of the Win32 API.

This library is intended to be a thin and straight-forward layer. In general, it does not implement additional logic. This library should replace most usages of PInvoke in your projects.

This library is written in C# with traditional PInvoke calls. The general pattern is to offer classes based on the handle types in Win32, and to expose the Win32 flat API as members of the most relevant class. The members are as strongly-typed as possible while not contorting the API too much.

## Classes

While the Win32 API is a flat API, the functions are logically grouped by the primary object they operate on. For example, there are sets of methods that operate on HWND, HDC, HBRUSH, etc. instances. In this library we define classes for each type, and the API functions are exposed as members of the classes. The actual PInvoke methods are not exposed publicly.

The sanctioned method for passing a handle to a PInvoke method is via a SafeHandle. In this library we derive all of our handle classes from SafeHandle – even when it may not be fully justified. For example, HWND is a handle type that doesn’t benefit much from being a SafeHandle. The semantics for an HWND simply don’t follow the typical handle pattern that requires a thread to acquire and release a handle. Deriving our objects from SafeHandle makes it easy to pass them to your own PInvoke definitions. SafeHandle classes are not supposed to contain additional state because they should not become the root of a graph of objects that need to be garbage collected and finalized. Our library exposes the relevant methods to expose the Win32 API functions, but it does not store additional state in our methods.

By being a SafeHandle, the types themselves normally take care of destroying the instance. For some types, this can be done by either disposing or finalizing the instance. For some other types, thread affinity prevents the use of the finalizer. For those cases you must explicitly dispose the SafeHandle from the appropriate thread.

Creating an instance is normally done by calling one of the constructors. There are other cases where an instance is returned from a method call. Pay attention to the ownership semantics, to know when you must keep the instance alive or when you must explicitly dispose it.

Examples:

* HWND is a handle type for a User32 window. You can create one directly via the constructor, or get one from various methods like GetWindow. In these cases, the actual HWND handle never has strong ownership semantics. Disposing the HWND will not destroy the window. You must explicitly call HWND.Destroy() to destroy a window. Child windows are implicitly destroyed when the parent window is destroyed.
* HPEN is a handle to a GDI pen. You can create one via the constructor, in which case the instance has strong ownership semantics and the object will be destroyed when disposed via a call to DeleteObject(). You can also get an instance of HPEN from HDC.SelectPen(pen), in which case the instance has weak ownership semantics, meaning the returned HPEN instance will not be destroyed, even if you call Dispose(). To explicitly destroy a pen you must call

## Static Methods

Some Win32 APIs don’t have an obvious parameter that identifies the primary type that we should expose the corresponding method on. In such cases, we expose the method as a static member of a suggestive type. Example:

* GetMessage() is an API to return the next message from the message queue. It takes a few parameters, including an HWND handle. However, this HWND parameter is actually more of a filter rather than the “primary” object that we could assume owns the API. Further, this API has thread-affinity such that it returns the next message for the calling thread. There is no way for one thread to get the message for another thread. This library exposes this API as a static method on a suggestive class: ThreadMessageQueue.GetMessage.

## Method Overloads

The Win32 API often has functions that take parameters that can be of multiple types. In some cases a simple void type is used for the parameter, in other cases a union is used to combine multiple types. In this library, we use overloads or alternative methods to express the various options. Examples:

* SelectObject is a Win32 GDI method for specifying the active objects of a device context. Each object type has an active instance (pen, brush, bitmap, etc). This API returns the previously active object, so it would be nice to return the correct type. A simple overload is not really possible because method overloads cannot change the return type. The pattern we use here is to expose additional alternative methods that are more strongly typed (SelectPen, SelectBrush, SelectBitmap, etc.) in addition to the SelectObject method.

## Arrays

Passing arrays to unmanaged code through PInvoke calls often requires passing the length of the buffer. This feels unnatural to managed code, so the corresponding methods in this library simply accept regular arrays. Example:

* SendInput in an API for injecting input for processing by the system. The actual PInvoke call takes 3 parameters (a count of inputs, a pointer to an array of input structures, and the size of the input structure). In this library, it simply takes an array of Input structures. In keeping with the other guidelines of this library, we also expose more strongly typed alternative methods (SendMouseInput, SendKeyboardInput, and SendHardwareInput). We also expose overloads that can accept a single instance, rather than an array.

## Enums

Win32 APIs pass various flags and options to functions; the supported values for these flags are primarily exposed as constants, often of type DWORD. The Win32 API essentially relies on naming patterns to indicate which constants are used in a particular context; often with a common prefix or postfix. This library exposes these constants as members of enums, where the name of the enum is generally derived from the naming pattern of the constants. There are some examples where Win32 constants do not follow a naming pattern that unambiguously indicates an appropriate enum name, in which case the enum is named something suggestive. The enum member names retain the capitalization, abbreviation, and other spelling choices of the constants. The values of the enum members are the same as the Win32 constants in order to avoid introducing a translation layer. Examples:

* DT\_TOP, DT\_BOTTOM, DT\_CENTER, etc. are constants used to indicate how text is formatted. In this library, these are exposed as members of the DT enum: DT.TOP, DT.BOTTOM, DT.CENTER, etc.
* ANSI\_CHARSET, GREEK\_CHARSET, OEM\_CHARSET, etc. are constants used to indicate a character set. In this library, these are exposed as members of the CHARSET enum: CHARSET.ANSI, CHARSET.GREEK, CHARSET.OEM, etc.
* OPAQUE and TRANSPARENT are constants used to specify the background mode for text. The constants don’t follow a prefix or postfix pattern to base the enum name on. We could consider expressing this as a Boolean, but I think an enumeration is more appropriate. A method using these constants is SetBkMode, so a suggestive name that retains the spelling choices of the API would be BKMODE. In this library, these constants are exposed as members of the BKMODE enum: BKMODE.OPAQUE and BKMODE.TRANSPARENT.

## Errors

Win32 APIs typically return a value that indicates the success or failure of the call. Sometimes a simple value (such as FALSE or NULL) is returned to indicate the call failed, and another call must be made to fetch the failure code. One example is GetLastError. Such patterns are not well suited for managed code because the CLR can call native code itself and the error code may be overwritten. The PInvoke technology can handle this situation in many cases by storing the last error as part of the PInvoke call. For this reason, we don’t expose GetLastError directly, but rather require calling Marshal.GetLastWin32Error.

## API coverage

The Win32 API is huge, and this project will incrementally expose APIs over time as the needs of other projects demand it. An inventory of the Win32 APIs is maintained in the API.xlsx file that maps how the Win32 APIs are exposed in the Microsoft.DwayneNeed.Win32 library.