Component-based Operating System APIs: A Versioning and Distributed Resource Solution

Robert J. Stets[†]
Galen C. Hunt
Michael L. Scott[†]

July 1999

Technical Report
MSR-TR-99-24

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

[†]Department of Computer Science
University of Rochester
Rochester, NY 14627

Component-based Operating System APIs: A Versioning and Distributed Resource Solution

Robert J. Stets[†], Galen C. Hunt, and Michael L. Scott[†]

Microsoft Research
One Microsoft Way
Redmond, WA 98052
galenh@microsoft.com

[†]Department of Computer Science
University of Rochester
Rochester, NY 14627
{stets, scott}@cs.rochester.edu

Abstract

Component software techniques have been developed to facilitate software reuse. State and functionality are encapsulated inside components with the goal of limiting program errors due to implicit interactions between components. Late binding of components allows implementations to be chosen at run-time, thereby increasing opportunities for reuse. Current component infrastructures also provide version management capabilities to control the evolutionary development of components. In addition to the general goal of reuse, component software has also focused on enabling distributed computing. Current component infrastructures have strong support for distributed applications.

By leveraging these strengths of component software, a component-based operating system (OS) application programmer interface (API) can remedy two weaknesses of current monolithic, procedural APIs. Current APIs are typically very rigid; they can not be modified without jeopardizing legacy applications. This rigidity results in bloat in both API complexity and support code. Also current APIs focus primarily on the single host machine. They lack the ability to name and manipulate OS resources on remote machines. An API constructed entirely of components can leverage version management and distributed computing facilities. Version management can be used to identify legacy APIs, which can then be dynamically loaded. OS resources modeled as components can be instantiated on remote machines and then manipulated with the natural access semantics.

We have developed the COP system as prototype component-based API for Windows NT. The system provides an API with version management capabilities and with a method for naming and manipulating remote OS resources. The advantages are gained with a minimum of overhead and without sacrificing legacy compatibility.

1. Introduction

Component software methodology has primary been motivated by the desire for software re-use. As described by Szyperski [1998], software components are “binary units of independent production, acquisition, and deployment that interact to form a functioning system.” The methodology itself focuses on independence by establishing a strict encapsulation of state and functionality inside each component. This encapsulation helps facilitate reuse. A significant obstacle to effective reuse is the natural evolution of software. Evolution creates multiple versions of the component, a number of which may be actively used by clients. The ability to manage multiple versions of code is generally called *versioning* and is addressed by most current component infrastructures. Also, as component software designers have always considered the distributed application domain important, infrastructures have extensive support for the operation of distributed components.

These advantages of software components can be leveraged to eliminate shortcomings present in current operating system (OS) application programmer interface (API) designs. OS APIs are typically monolithic procedural interfaces addressing single-machine requirements. Their design limits options for evolutionary development and also complicates application development for distributed systems.

During an operating system's lifetime, its functionality will change, and these changes must be reflected in the API. A set of API calls may become obsolete or their semantics may change. In an ideal world, obsolete calls would be deleted and calls with modified semantics (but unmodified parameters and return values) would remain the same. Unfortunately, calls can neither be deleted nor can their semantics change. Such API modifications would jeopardize the operation of legacy applications.

Legacy applications are an important concern for today's operating systems. Installation of a new operating system version is already expensive (in time and money). If new application versions are also required, the expense is only compounded. (In some cases, new versions may not even be feasible.) Operating system evolution must be designed to support legacy applications. Since any changes to the API can break legacy applications, API calls typically become fixed once published. Obsolete calls can never be deleted, and new call semantics must always be introduced through new calls. Backward compatibility thus leads to bloat in both the API and the supporting code.

For example, the UNIX 98 specification (endorsed by IBM, Sun, and NCR) lists 21 calls reserved for legacy support. Many of these calls have been superceded by new, more powerful calls (e.g. the signal management function, `signal()`, has been replaced with the more powerful `sigaction()`). Apple’s Carbon implementation of the Macintosh OS API deprecates over 2100 functions for the earlier MacOS 8.5 implementation. Win32, the primary API for Microsoft's family of 32-bit operating systems, contains over 1700 legacy API calls, including 146 calls providing support for its predecessor, Windows 3.1.

Also the distributed computing paradigm is not well supported by typical operating systems APIs. Virtually all APIs do of course have support for inter-machine communication, but high-level support for accessing remote OS resources is lacking. The primary omission is a uniform method of naming remote resources, for example windows, files, and synchronization objects. This omission prevents an application from easily using resources scattered throughout a distributed system.

A multi-user game serves as a good example. This class of applications needs to open windows, sound channels, and input devices (e.g. joysticks) on numerous machines throughout a distributed system. With typical OS APIs, these applications must rely almost entirely on ad-hoc mechanisms to access the necessary remote resources.

The above two weaknesses in modern OS APIs can be eliminated by the application of component software methodology. A *component-based* API is constructed entirely of software components, with each component modeling an OS resource. As components encapsulate their state and functionality, all access and manipulation functions for a particular resource type are contained in its component. The factoring inherent in a component-based API allows for efficient versioning, and the state and access encapsulation allow OS resources to be instantiated on remote machines.

To clarify, we only propose to componentize the API. The underlying OS can be monolithic, micro-kernel, or component-based. By componentizing the API, we are controlling the access to the OS. Control at this point is sufficient to provide API versioning and also to expose OS resources outside of the host machine. The process of making resources available remotely is called *remoting.*

In this paper, we describe COP (Component-based Operating system Proxy), a prototype of a componentized API. The COP system acts a “traffic cop” that directs OS requests to the appropriate version or resource location. The system currently targets the Win32 API and is implemented on top of Windows NT 4.0. Our implementation currently covers approximately 350 Win32 calls, enough to provide needed development support for a separate project in distributed component applications. We have found that COP only introduces a minimum of overhead in the local case, while providing outstanding OS support for evolutionary development and distributed applications.

2. Component Software Overview

In this section, we will provide a brief overview of the component software methodology and two popular infrastructures. Components have been an extremely rich area of ongoing work during the last ten years. Necessarily, we will only focus on aspects directly related to this paper. To begin, we will provide definitions for some important terms used in this paper.

The term component was specifically defined in the previous section. Roughly speaking, a component provides functional building blocks for a complex application. An *interface* is a well-known contract specifying how a component's functionality is accessed. Interfaces take the form of a set of function or method calls, including parameter and return types. A *component instance* refers to a component that has been loaded into memory and is accessible by a client. All communication between component instances occurs through interfaces. Component software fundamentally maintains a strict separation between the interface and the implementation. This separation is a key requirement for enforcing components to encapsulate their functionality and for guaranteeing component independence.

Independence allows components to be *composed* without introducing implicit interactions that may lead to subtle program errors. The ability to compose is also enhanced by allowing one component to be substituted for another, so long as the substitute provides the same, or an extension of, the functionality of the original. Through *polymorphism* components with differing implementations of the same interface may be interchanged transparently. A final issue in composition is the point in time at which component choices are bound. *Late binding* allows an application to choose components dynamically.

Independence, polymorphism, and late binding are methodological concepts that facilitate reuse in component software. Component infrastructures also address related implementation issues, namely mixed development languages and execution platforms. All popular infrastructures provide mechanisms that allow development in multiple languages and execution across multiple hardware platforms.

Two of the more popular component infrastructures are Microsoft's Component Object Model (COM) [Microsoft, 1995] and the Object Management Group's Common Object Request Broker Architecture (CORBA) [Object Management Group, 1996]. Although originally motivated by different goals, they have largely converged to promote software reuse independent of development language in both a single-machine and distributed computing environment. COP is built on top of COM, and so the next subsection will provide an overview of COM. The following subsection will then contrast the differences between COM and CORBA, focusing especially on the effects on a system such as COP.

2.1. Component Object Model (COM)

COM was developed by Microsoft to address the need for cross-application interaction. As the work evolved, the Distributed COM (DCOM) extensions [Microsoft, 1998] were introduced to support distributed computing. COM provides language independence by employing a binary standard. Component interfaces are implemented as a table of function pointers, which are called *vtables* because they mimic the format of C++ virtual function tables. References to component instances are referred to as *interface pointers*. These are actually double-indirect pointers to the vtable. The extra level of indirection is provided as an implementation convenience. For example, an implementation can attach useful information to the interface pointer, information that will then be shared by all references to the interface.

In keeping with component software methodology, COM maintains a strict separation between a component interface and implementation. COM in fact says nothing about the implementation, only about the interfaces. Interfaces can be defined through single inheritance. (Note only the interface is inherited; implementation is entirely separate.) The lack of multiple inheritance is not a limitation. COM components can implement multiple interfaces regardless of inheritance hierarchy. This provides much the same power as multiple interface inheritance.

All COM interfaces must inherit from the `IUnknown` interface. `IUnknown` contains a `QueryInterface()` method and two methods for memory management. For our discussion, `QueryInterface()` is the most important. A client must use this method to obtain a specific interface pointer from a component instance.

COM components are identified by a globally unique class ID (CLSID). Similarly, all interfaces are specified by a global unique interface ID (IID). A client instantiates a component instance by calling the COM `CoCreateInstance()` function and specifying the desired CLSID and

IID. A pointer to the desired interface is returned. Given an interface pointer, the client can use `QueryInterface()` to determine if the component also supports other interfaces.

By convention, COM holds that all published interfaces are immutable in terms of both syntax (interface method names and method parameters) and semantics. If a change is made to an interface, then a new interface, complete with a new IID, must be created. Immutable interfaces provide for a very effective versioning mechanism. A client can request a specific interface (through its published IID) and be assured of the desired syntax and semantics.

Under COM, components can be instantiated in three different execution contexts. Components can be instantiated directly in the application's process (*in-process*), in another process on the same machine (*local*), or on another machine (*remote*). The ability to access instances regardless of execution context is called *location transparency*. COM provides location transparency by requiring that all instances are accessed through the vtable.

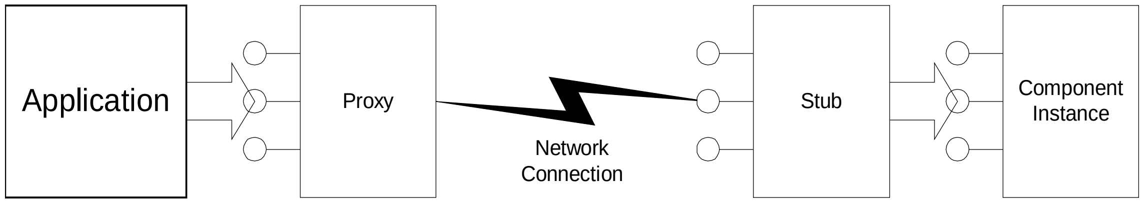


Figure 1: For a call to a remote component instance, the proxy first marshals data arguments into a suitable transmission format. The request and data are then sent across the network by the transport mechanism. (The default mechanism is an object-oriented extension of DCE RPC.) At the server, the stub receives the request, unmarshals the data, and invokes the requested interface function. The process is reversed for the function return values.

For in-process instances, the component implementation is usually held in a dynamically linked library (DLL) and is loaded directly into the process' address space. The vtable then points directly to the component implementation. For local or remote components, the component implementation is loaded into another process and the application must engage in some type of inter-process communication (IPC). To handle these cases, COM instantiates a *proxy* and *stub* pair to perform the communication (see Figure 1). The vtable is set to point directly to the proxy.

Before an IPC mechanism can be used, data must be packaged into a suitable transmission format. This step is called *marshaling*. The proxy is responsible for marshaling data and then sending the data and the request to the component instance. At the component instance, the stub receives the request, unmarshals the data, and invokes the appropriate method on the instance. The process is reversed for any return values.

A system programmer can customize the IPC mechanism. Otherwise COM defaults to using shared memory for the Local case and an extension of the Open Group's Distributed Computing Environment remote procedure call facility (DCE RPC) [Hartman, 1992] for the Remote case.

2.2 COM, CORBA, and a Component-based API

Both COM and CORBA share many fundamental similarities, especially in the area of distributed computing. For remote communication, CORBA uses an architecture that is very similar to COM. In essence, both architectures offer the same capabilities for remote component instances.

The two systems however differ greatly in their versioning capabilities. Of current CORBA implementations, IBM's System Object Model (SOM) builds interface specifications at run-time [Forman, 1995], and so interface methods can be added or re-ordered, but not removed. SOM's strategy does not address semantic changes. To address semantic changes, CORBA *repository IDs* could be used to uniquely identify interfaces in much the same manner as COM IIDs. However, repository IDs are only checked when an instance is created and not when an instance reference is obtained directly from another component instance. A more fundamental problem is that CORBA's conventional object model merges all inherited interfaces into the same namespace, so it is impossible to simultaneously support multiple interface versions unless all method signatures are different. A component-based API built on top of CORBA would therefore not be able to offer very robust versioning capabilities.

This work focuses on component software support for evolutionary development and distributed resources in operating systems. Component software infrastructures provide a plethora of other interesting application support, such as transactions, licensing, and persistence. These areas are beyond the scope of our current work.

3. COP Implementation

In this section, we describe the COP implementation. The first subsection describes how the monolithic WIN32 API was factored into a set of interfaces. The second subsection then discusses the COP run-time system, including its support for versioning, distributed computing, and legacy applications.

3.1 Factoring a Monolithic API

The first step in constructing a component-based API is to split, or *factor*, the monolithic API into a set of interfaces. After factoring, the entire API should be modeled by the set of interfaces, with individual and independent OS resources and services modeled by independent interfaces. A good factoring scheme produces interfaces that are appropriately independent and provides the benefits of clarity, effective versioning, and clean remoting of resources.

Our discussion here applies our factoring strategy to the Win32 API. (Our factoring of a 1000+ subset of Win32 is listed in Appendix A.) However, our strategy and techniques should be generally applicable to monolithic, procedural APIs.

API Subset:

```
BOOL AdjustWindowRect(RECT *, DWORD, BOOL);
HANDLE CreateWindow(...);
int DialogBoxParam(...,HANDLE, ...);
int FlashWindow(HANDLE, ...);
HANDLE GetProp(HANDLE, ...);
int GetWindowText(HANDLE,...);
```

Final Factorization:

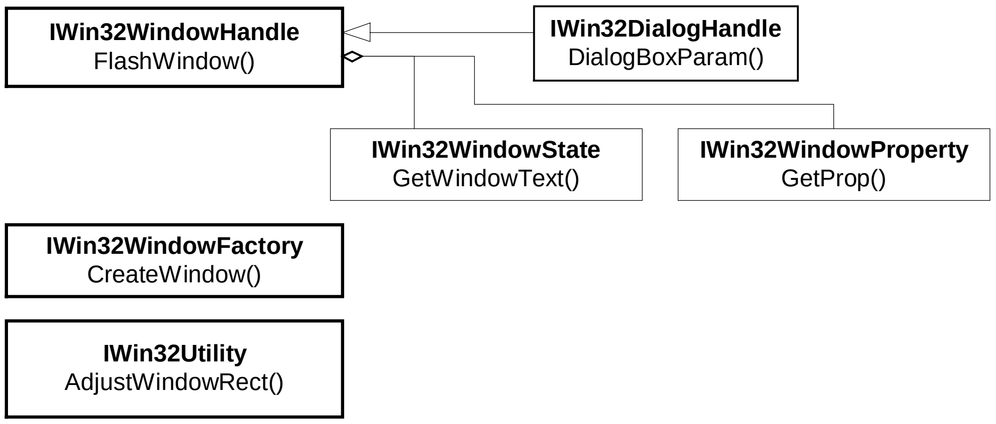


Figure 2: The factoring of a simple subset of the Win32 API. Proposed interfaces are listed in bold and prefixed with “IWin32”. IWin32WindowHandle aggregates the IWin32WindowState and IWin32WindowProperty interfaces. IWin32DialogHandle inherits from IWin32WindowHandle, since dialogs extend the functionality of plain windows.

Our factoring strategy involves three steps. First, the monolithic API calls are factored into groups based on functionality. For example, all graphical window calls are placed in a IWin32Window[1] group. Second, the calls in each group are factored into three sub-groups according to their effect on OS resources. The effect is easily identifiable through the call parameters and return value. A loaded OS resource is exported to the application as an opaque value called a *kernel handle*. Calls that create kernel handles (i.e. OS resources) are moved to a *factory* interface, and calls that then query or manipulate the these kernel handles are moved to a *handle* interface. Any other calls that do not directly involve kernel handles (but may instead manipulate OS settings or provide generic services) are moved to a *utility* interface.

In the third step, we further refine the factorization. In many cases, a monolithic API may contain a set of calls that acts on a number of different OS resources. For example, Win32 has several calls that synchronization on a specified handle. The specified handle can represent a standard synchronization resource, such as a mutual exclusion lock, or less common synchronization resources such as processes or files. Our first two steps in factoring will not capture this relationship. Continuing the example, the synchronization calls will be placed in a IWin32SyncHandle interface, while the process and file calls will be placed in IWin32ProcessHandle and IWin32FileHandle interfaces, respectively. For correctness though, the process and file interfaces should also include the synchronization calls. Since the process and file handles can be thought of as logically extending the functionality of the synchronization

[1] The IWin32 prefix denotes an interface to a Win32 API component.

handle, we can model this relationship through interface inheritance. Both IWin32ProcessHandle and IWin32FileHandle will inherit from the IWin32SyncHandle interface.

Figure 2 is an example of our factoring of the Win32 window functions. The example necessarily focuses on a small, but representative, subset (six calls) of the 130+ window calls. The `AdjustWindowRect()` call determines the necessary size of a window given specific settings. The second call, `CreateWindowEx()`, creates a window, and the remaining calls create a window, execute a dialog box, flash the window's title bar, query various window properties, and return the current text in the window title bar.

These calls all operate on windows and so are first factored to a windows group. Next the calls are further factored depending on the use of a kernel handles (denoted by HANDLE in Figure 2). In the third step, we have further factored the IWin32WindowHandle into IWin32WindowState and IWin32Property interfaces. The State and Property interfaces simply help to make the API easier to read. These interfaces do not extend the IWin32WindowHandle interface, but instead compose the interface. We model this relationship through interface aggregation. Also, we have factored the dialog calls into their own interface, since the dialogs are logically extensions of plain windows. Again this relationship is modeled through interface inheritance.

Properly applied, this factorization strategy will produce a set of interfaces, each with a tightly defined set of calls to access the appropriate underlying OS resource. The factorization will improve API clarity by clearly defining the specific methods for accessing each OS resource and also the relationship between API calls. Versioning capabilities will also be improved since modifications can be isolated within the affected interfaces. Finally, a good factorization inherently encapsulates functionality (and the associated state), which facilitates the remoting of OS resources.

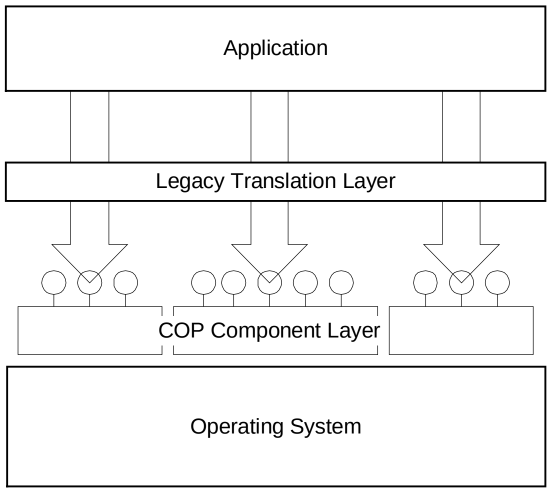


Figure 3: The COP Runtime system consists of a component layer that presents the OS API and an optional Legacy Translation layer available for Win32 applications.

3.2 Run-time System

Overview

At run-time, the application accesses the OS through the COP component layer (see Figure 3). These components implement the interfaces described in the previous subsection. As with the interfaces they implement, the components can also be roughly classified as factories, handles, or utilities.

Most applications will instantiate factory components during initialization and then use the factories to create OS resources during execution. A basic implementation of a factory component first invokes the OS to create the desired resource. The OS will return a kernel handle to identify the resource. This handle however is only valid on the local machine. To enable remote access to the resource, the factory also creates an instance of the associated handle component and stores the kernel handle in the instance's private state. Then rather than returning the kernel handle, the factory returns a pointer to the instance of the handle component. The application makes subsequent accesses to the resource through the instance pointer.

Utility components do not directly manipulate loaded kernel resources, but instead provide generic services such as conversion between time formats or calculating the necessary window rectangle to contain a specified client rectangle and the general window elements. These components can instantiated whenever necessary, anywhere throughout the system. Again once instantiated, all accesses will occur through the instance pointer.

On a simple level then, the instance pointer provides COP with one of its main advantages over typical modern OS APIs. The instance pointer uniquely names the loaded resource throughout the system and also acts as a gateway to the underlying remoting mechanism (COP/DCOM). With COP, applications can create resources throughout the system and subsequently use the instance pointer to access them in a location transparent manner.

Versioning

COP's other main advantage over modern OS APIs is its versioning capabilities. These capabilities follow directly from our factoring strategy and COM's robust versioning mechanism. As described above, published COM interfaces are immutable and are named by a globally unique ID. Clients can request specific interfaces and be assured of desired call syntax and semantics.

To mark the specific interfaces, an application can store the appropriate IDs in its data segment. Alternatively, the OS binary format could be extended to support static binding to a dynamic interface in the same way that current operating systems support static binding to DLLs (or shared libraries). With such an extension, an application binary would declare a set of interfaces to which it should bind instead of a set of DLLs. Of course, COP-aware applications can query dynamically for special interfaces.

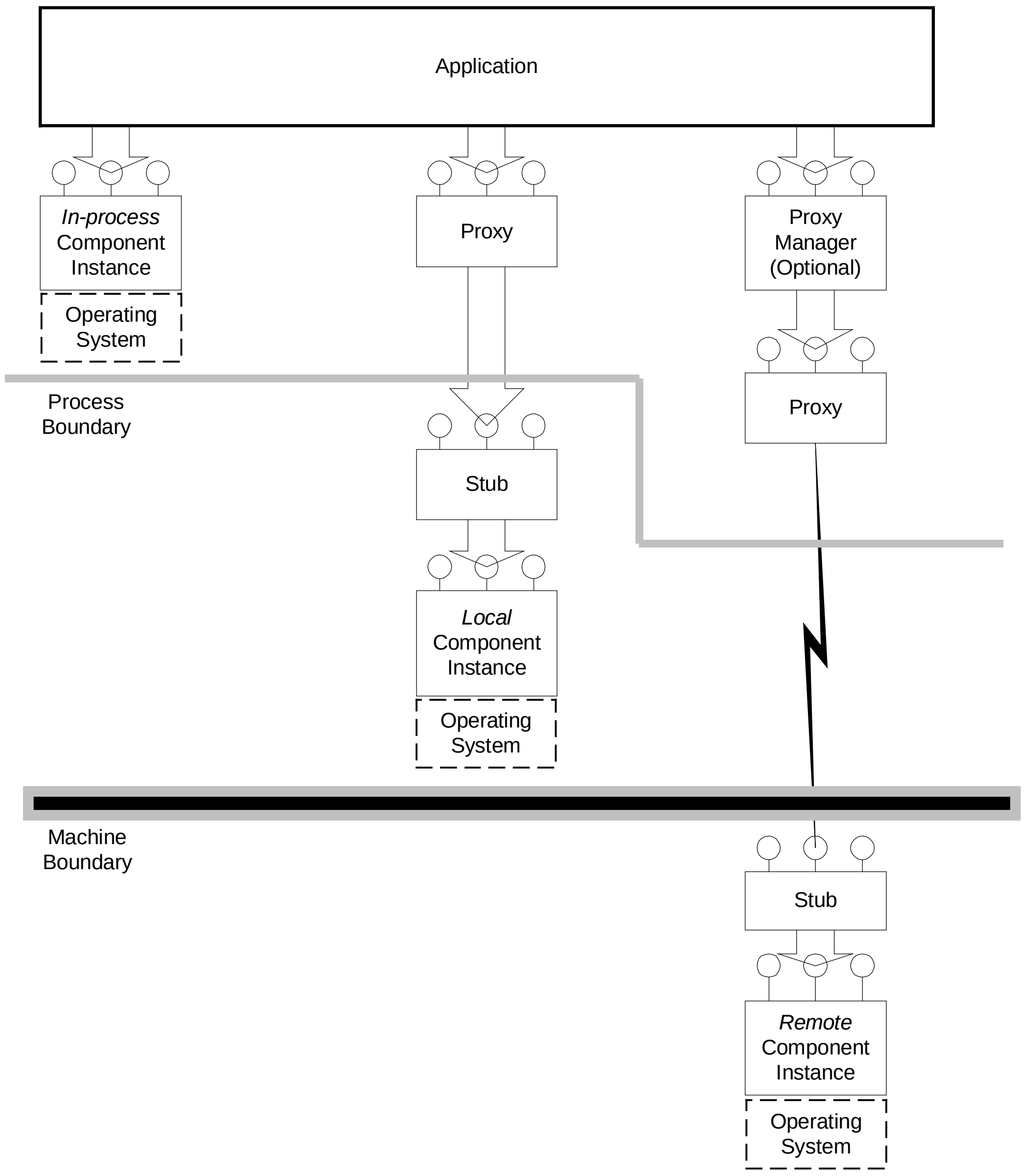


Figure 4: COP is able to instantiate OS resources in a number of locations: inside the client process (in-process), in another process on the same machine (local), or on another machine (remote). The client application can still access the resources in a *location transparent manner* by virtue of the proxy manager, proxy, and stub components.

Location Transparency

One of the main contributions of COP is the ability to instantiate OS resources anywhere throughout a distributed system (see Figure 4). COP components can be instantiated inside the application's process (in-process), in another process on the same machine (local), or on another machine (remote). As described in Section 2, in-process components only experience the added overhead of an indirect function call. In the local case, a COM proxy/stub pair is used to marshal data across the process boundaries. The local case is less efficient than the in-process case, however it provides better fault isolation. The remote case also uses the same general proxy/stub architecture. However, in the remote case, COP also includes an optional Proxy Manager that can be used to optimize remote communication. A common Proxy Manager task is to cache remote data in the hopes of avoiding unnecessary communication. For example, COP currently caches information to improve the re-drawing of remote windows. The Win32 call `BeginPaint()` signals the beginning of a re-draw operation by creating a new drawing context resource. In order to be available remotely, this resource must be wrapped by a COP component. Rather than creating a new component instance on each re-draw operation, COP currently caches a component instance (in the Proxy Manager) and re-uses the instance for the re-draw wrapper.

Although hidden from the application, extra state is obviously required to maintain the location transparency. For instance, the system must keep track the location of component instances and data concerning the network connection. COM maintains this state automatically. COP components often have little extra state to maintain. As the only common example, handle components need to store the value of their associated kernel handle. Optional Proxy Manager implementations may also require extra state, for instance the cache of remote data mentioned above.

In a less common case, some components need extra state to maintain location transparent results. The different execution context – in-process, local, or remote – may cause some calls to execute differently. (We of course try to maintain the same operation as the normal Win32 API.) For example, the call `RegisterClass()` registers a window class[2] for use within a process. The call returns an error if the class is already registered within the process. A naïve component implementation could report this error incorrectly in some cases. In COP, this call falls under the IWin32WindowUtility interface (since it does not target kernel handles). Consider the case where two applications try to register the same class on the same remote machine. To access `RegisterClass()`, both applications would create an instance of IWin32WindowUtility. Since these instance will both be remote and on the same machine, COM creates the instances inside the same process to optimize performance. Note that the instances are separate COM instances, but they do share the same process. The first application to register the class will succeed, but the second application will fail since the class has already been registered inside the COM process. In attempting to mimic standard Win32 operation, this error would be incorrect since the application processes are separate. In COP, the IWin32WindowUtility implementation maintains a list of classes each process has registered. The implementation can then determine if the caller has already registered the specified class and avoid any spurious errors.

[2] A window class specifies various window settings, such as the default cursor and background. Windows are created based on registered window classes.

Obstacles to Remoting OS Resources

Apart from state problems, there are other OS aspects that do pose remoting problems. OS callback functions are a significant obstacle to remote execution. Numerous API functions contain callback functions that the caller specifies and the OS invokes on certain events. For example, the Win32 call `EnumWindows()` calls a specified callback function for each top-level window on the screen. Callbacks are a problem when the caller (i.e. the location of the callback function) is on a different machine from the OS invoking the callback. COP solves this problem in the same way that it remotes OS resources. COP wraps all callback functions with components. Instead of passing the address of the callback function, COP passes a pointer to the component instance wrapping the callback. The OS can then simply use the instance pointer to invoke the callback function in a location transparent manner.

Asynchronous events are the other main obstacle to remote execution. Some OS resources, such as windows, synchronization objects, and asynchronous I/O, must respond to asynchronous events. Windows must receive events such as mouse clicks, key strokes, and re-draw messages and send them to the user-specified window procedure for processing. The OS must ensure synchronization objects are given to requestors as semantics dictate. In asynchronous I/O, the OS must notify the caller when an I/O operation is complete. In all these cases, the OS assumes all involved parties reside on the same machine. COP therefore needs to provide extra support to remote these types of resources.

COP remotes these resources by creating a special *event agent* on the remote machine. This agent is responsible for fielding asynchronous events and forwarding them to the client application. COP currently has support for remote windows. A window procedure is simply a special case of a callback routine. The OS calls the window procedure on every window event. At window creation time, COP creates a component instance to wrap the specified window procedure. COP then invokes the `CreateWindowEx()` method of the IWin32WindowFactory instance on the remote machine.

The IWin32WindowFactory instance creates an IWin32WindowHandle instance, which will manage the actual window. The IWin32WindowHandle instance creates the window as part of initialization. Instead of specifying the application's window procedure though, IWin32WindowHandle specifies its own procedure. In addition stores the pointer to the instance of the application's window procedure, which was provided through a hook in `CreateWindowEx()`.

COM actually delivers remote function call requests to COP components through a standard message queue. An idle component instance simply spins on the message queue, waiting for function call requests. Fortuitously, window events are also delivered through the same message queue. In the course of polling for incoming requests, the IWin32WindowHandle instance will also discover pending window events. The instance can then use the stored instance pointer to send the messages to the application's window procedure for processing.

Synchronization and asynchronous I/O can be handled in the same manner – an event agent can be instantiated on the remote machine. The agent will wait for the desired event and then forward notification to the application via a callback component.

Legacy Translation Layer

Our ultimate intention is for applications to write directly to the COP API. To ease the transition and to support legacy applications that can not be re-written, we have also built an optional COP Translation layer (see Figure 3). This layer is responsible for intercepting the procedural Win32 calls and translating them to COP. To help minimize translation overhead, we have purposely designed the COP interface methods to use the same parameters as their Win32 counterparts.

Run-time interception is performed with the Detours package [Hunt, 1998]. One of this package's many features is the ability to instrument an application's binary file and add a specified DLL to the start of the list loaded at program initialization. This ensures that the specified DLL is the first loaded by the application. We use Detours place our COP startup DLL at the start of the list. The startup DLL then uses the Detours package to intercept and re-route Win32 calls to the Legacy Translation layer. Detours performs the interception by re-writing the first few instructions of a subroutine so that upon entrance, control is automatically transferred to a user-defined detour function. The replaced instructions are combined with a jump instruction to form a *trampoline*. The detour function can call the trampoline code to invoke the original subroutine, in our case the original Win32 call.

The Legacy Translation layer is then responsible for creating the COP factory and utility instances as necessary. (The handle instances are created by the factory instances.) The layer of course caches pointers to interfaces to avoid unnecessary overhead. This approach works well for existing, single-machine Win32 applications, and also even allows the functionality of these applications to be transparently extended. The Translation Layer can be configured to automatically create resources on remote machines. For example, all window resources can be started on a remote machine, very similar to X-Window [Scheifler, 1986] remote displays. We have used this feature to remote the display of several existing Win32 applications. A remote display however only leverages a small amount of COP's most power feature – the ability to trivially connect to resources scattered throughout a distributed system.

The design of the Translation layer is relatively straightforward, but one significant problem did arise. Our translation layer intercepts all invocations of a specified call, even if the call is invoked from within another Win32 call. Re-entrancy problems can result. For example, COP allows applications to access the Win32 registry[3] on remote machines, however COP must do so by instantiating a registry (IWin32Registry) component on the remote machine. The component is instantiated through the Win32 `CoCreateInstanceEx()` call, which itself accesses the registry. If COP intercepted and handled the registry call from `CoCreateInstanceEx()`, an infinite recursion would result. The Legacy Translation Layer tracks when an application is inside a Win32 call and avoids COP handling if an infinite recursion would start. This problem does not arise outside of legacy support, since all clients explicitly specify the execution context when attaching to the COP API.

[3] The Win32 registry is a database of application configuration information.

4. Results

The initial goal for COP is to support the development of the Millennium system. Millennium will be a thin software layer that monitors the execution of a distributed component-based application and intelligently distributes the component instances to maximize performance. As components are distributed throughout the system, they still must be able to access remote OS resources. COP provides that capability.

To this end, we have currently remoted the registry, windows, graphic device interface (the low-level drawing routines), and file APIs. This subset consists of approximately 350 calls and is enough to support the development of Millennium. This also includes the appropriate support in the Legacy Translation Layer.

The primary advantage of COP is enhanced functionality – better versioning support and the ability to instantiate OS resources throughout a distributed system. To gauge the overhead introduced by COP, we have performed two benchmark tests. Our tests were performed on a Gateway 2000 machine with a Pentium II processor running at 266MHz. The machine has a 512Kbytes off-chip cache and 64Mbytes of RAM. Our benchmark timings were calculated based on the Win32 `QueryPerformanceCounter()` call, which has a resolution of approximately 1 microsecond on our machine.

Our first benchmark focused on estimating the overhead of our Legacy Translation layer. Our test measured the amount of time to make a "null" Win32 call. (The call actually passes one `integer` parameter and returns an `integer` value.) Our benchmark application simply calls a generic Win32 function, which COP intercepts and routes to the Translation Layer. The Translation Layer then invokes the associated component instance. The component instance immediately returns a success value, which the Translation Layer returns to the application.

As expected, an in-process component instance adds very little overhead in this case. The Win32 "null" call can be executed in 1.3 microseconds. If the component instance is instantiated as a Local server (in another process), the Win32 "null" call time jumps to 200 microseconds. This jump in time is due to the crossing of procedure boundaries.

The second test we performed was to examine the full overhead on an existing Win32 application. We chose RegEdt32, a tool for editing the Win32 registry. At startup, the application reads the entire registry and displays the contents on screen for editing. We measured the time required to start the application and read all elements from the local registry. We feel this is an interesting benchmark because it includes not only the time to make COP calls, but also the time to instantiate COP components. Our COP implementation patched all the involved registry calls, and the startup phase involved a little over 9,500 registry calls, all handled by COP. We report the average of three runs. Our machine was rebooted in between each run in order to remove effects from the Window NT (file) cache.

The plain application (with no COP overhead) starts up in 0.833 seconds. The application using COP in-process components starts in 1.118 seconds, a 34% increase. A large amount of this overhead is due to the cost of instantiating the components. In a normal situation, this overhead would be amortized. The application using COP Local components starts in 5.296 seconds, with the increase due to the frequent process boundary crossings.

We did not benchmark COP with remote components, since the choice of network will have such a strong influence on the results. We feel that these results show that in-process COP components add only a minimal amount of overhead, while providing benefits in versioning management. When COP components are moved to remote machines, the overhead will be much higher, but network transmission time will still be the dominant concern. Regardless, the functionality of the system will be much greater – an application can easily access scattered, remote OS resources.

5. Related Work in Operating Systems

Kernel call *interposition* is the process of intercepting kernel calls and re-routing them to pieces of extension code. There has been a large amount of work, published and unpublished, in this area. Interposition Agents [Jones, 1993] in particular was highly influential to our work. This work demonstrated that a kernel call interface (Berkeley UNIX 4.3) could be factored into a small set of abstractions, which were then used as the basis for an object oriented interposition toolkit. Another recent system of note is SLIC [Ghormley, 1998]. This system allows multiple interposition extensions to be composed at run-time, but the system is not object or component-based. SLIC and Interposition Agents can be considered full-featured interposition systems. COP uses interposition techniques, but our goal is not a general interposition system. Our goal is a new style of API that provides versioning and distributing computing benefits. A general interposition system should be built on top of our component-based API.

As we consider a component-based OS API here, other research efforts are considering building an entirely component-based OS. The OS could then be assembled dynamically in order to reflect the execution environment. Two such examples are MMLite [Helander, 1998] and Jbed [Oberon, 1998]. Both of these operating systems can drop unnecessary components, such as virtual memory or network communication, when running on a slim embedded processor platform. To our knowledge, none of this work addresses API versioning or the naming of remote OS resources. Also, importantly this work requires building a kernel from scratch, whereas our work can be easily applied to existing commercial operating systems.

The work closest to our own is the Inferno distributed operating system [Dorward, 1997]. In this system, all OS resources are treated as files – that is named and manipulated like files. This unique approach provides the advantage of a global, hierarchical namespace for all resources, but also the disadvantage of a rather limited access interface. In contrast, our approach in COP retains the natural semantics for manipulating remote resources.

There have been numerous projects that have focused on remoting small subsets of OS functionality. X Windows [Scheifler, 1986] provides remote access to a system's graphical user interface. Microsoft's Terminal Server [Microsoft, 1997] does the same for Windows NT platforms. Distributed file systems like NFS [Lyon, 1985] provide remote access to files. Unlike these systems, a component-based API targets the remoting of all OS resources.

6. Conclusions

Component software provides excellent support for the evolutionary development of software and for distributed computing. By basing an OS API on components, a system can gain considerable leverage in these two areas. The OS can export different versions of the API, allowing the API to be modified without jeopardizing legacy applications. Instead the support for legacy applications can be dynamically loaded. By modeling the

OS resources as components in the API, a global namespace is created. An application can instantiate and manipulate any number of resources scattered throughout a distributed system. Natural access semantics for the remote resources is maintained by virtue of the encapsulation of functionality inherent in components. Applications will no longer have to rely on ad-hoc methods to access remote resources.

Future work on COP will focus on increasing coverage of the Win32 API. (There are thousands of calls in the API.) Also we are interested in researching methods to provide consistent, global view and management of resources throughout a cluster and also for providing fault tolerance and security throughout the system.

7. Acknowledgements

Our factoring strategy borrows heavily from previous work by Microsoft Research's Component Applications group. The authors would like to thank Craig Wittenberg and Crispin Goswell for their advice on factoring. Also Rich Draves, Mike Jones, Johannes Helander, and Rick Rashid provided much useful feedback on factoring and design of the COP run-time.

Bibliography

Dorward S., Pike R., Presotto D., Ritchie D., Trickey H. and Winterbottom P. (1997). *Bell Labs Technical Journal*. Lucent Technologies, Inc.

Forman I.R., Conner M.H., Danforth S.H. and Raper L.K. (1995) Release-to-Release Binary Compatibility in SOM. *Proceedings of the Tenth Annual Conference on Object Oriented Programming Systems, Languages, and Applications*. Austin, Texas.

Ghormley D., Petrou D. Rodrigues S. and Anderson T. (1998) SLIC: An Extensibility System for Commodity Operating Systems. *Proceedings of the USENIX Annual Technical Conference*. New Orleans, Louisiana.

Hartman D. (1992) Unclogging Distributed Computing. *IEEE Spectrum*, **29(5)**, pp. 36-39.

Helander J. and Forin A. (1998) MMLite: A Highly Componentized System Architecture. *Proceedings of the Eighth ACM SIGOPS European Workshop*. Sintra, Portugal.

Hunt G. (1998) Detours: Binary Interception of Win32 Functions. *Technical Report* MSR-TR-98-33. Microsoft Research, Redmond, Washington.

Jones M. (1992) Interposition Agents: Transparently Interposing User Code at the System Interface. *Proceedings of the Fourteenth Symposium on Operating Systems Principles. Asheville,* North Carolina.

Lyon B., Sager G., Chang A. J., Goldberg D., Kleinman S., Lyon A. T., Sandberg A. R., Walsh A. D. and Weiss A. P. (1985) Overview of the Sun Network File System, Sun Microsystems, Inc.

Microsoft Corporation and Digital Equipment Corporation (1995) *The Component Object model Specification*. Redmond, Washington.

Microsoft Corporation (1997) *Windows-Based Terminal Server*. Beta 1, Redmond, Washington.

Microsoft Corporation (1998) *Distributed Component Object Model Protocol, version 1.0*. Redmond, Washington.

Oberon Microsystems (1998) Jbed Whitepaper: Component Software and Real-time Computing. *Technical Report*. Zürich, Switzerland.

Object Management Group (1996) *The Common Object Request Broker: Architecture and Specification, Revision 2.0*. Framingham, Massachusetts.

Scheifler R. and Gettys J. (1986) The X Window System. *ACM Transactions on Graphics.* **5(2)**, 79-109.

Szyperski C. (1998) *Component Software: Beyond Object-Oriented Programming*. ACM Press, New York, New York.

Appendix A: Proposed Factoring of a 1000+ subset of Win32

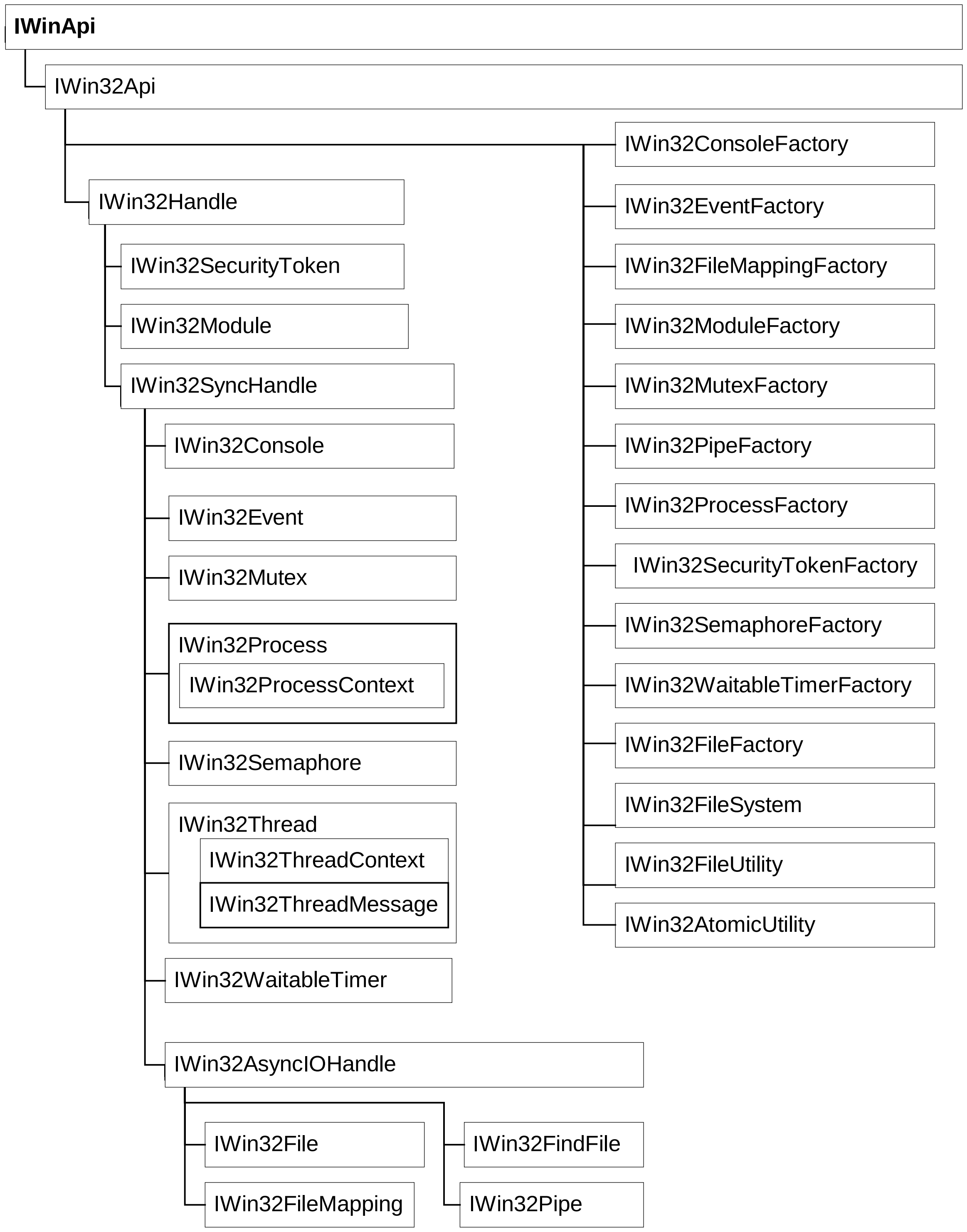
This Appendix lists the interface hierarchy and factoring of a 1000+ subset of Win32. The subset contains the necessary Win32 calls to support three OS-intensive applications: Microsoft PictureIt!, the Microsoft Developers' Network Corporate Benefits sample, and Microsoft Research's Octarine. The first is a commercial image manipulation package, the second is a widely distributed sample three-tiered, client-server application, and the third is a prototype COM-based integrated productivity application. This subset does not cover DirectX or ODBC, but we feel it does cover many of the major areas of functionality in Win32.

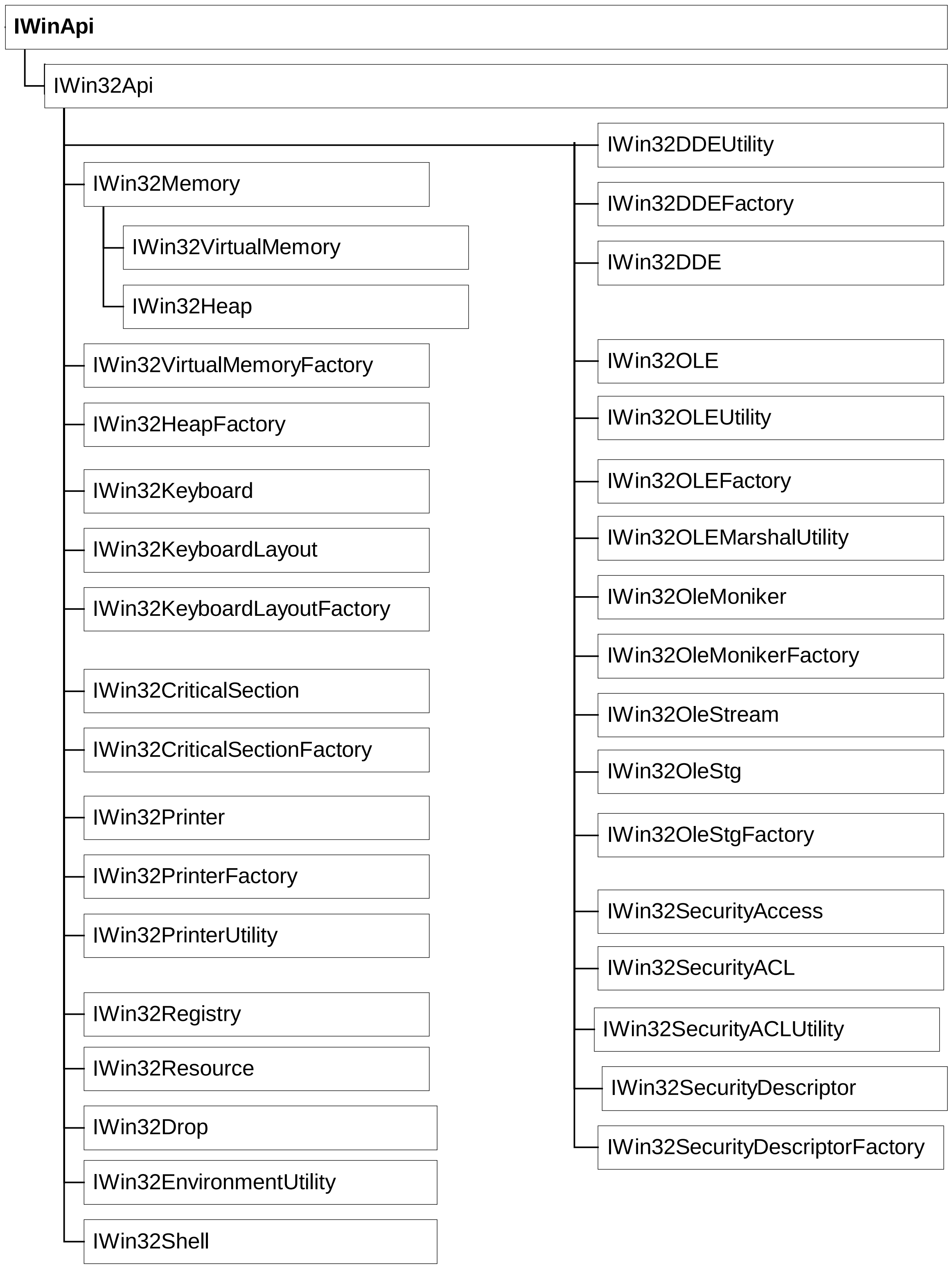
All obsolete Windows 3.1 (16-bit) calls have been placed in IWin16 interfaces. In implementation, the top-level call prototypes will mirror their WIN32 counterparts, with the appropriate parameters replaced by interface pointers. Note that these calls can wrap lower-level methods that implement different parameters. For example, the lower level methods could return descriptive HRESULTs directly and the WIN32 return types as OUT parameters. Also, we expect ANSI API calls to be implemented as wrappers of their UNICODE counterparts. The wrappers will simply perform argument translation and then invoke the counterpart.

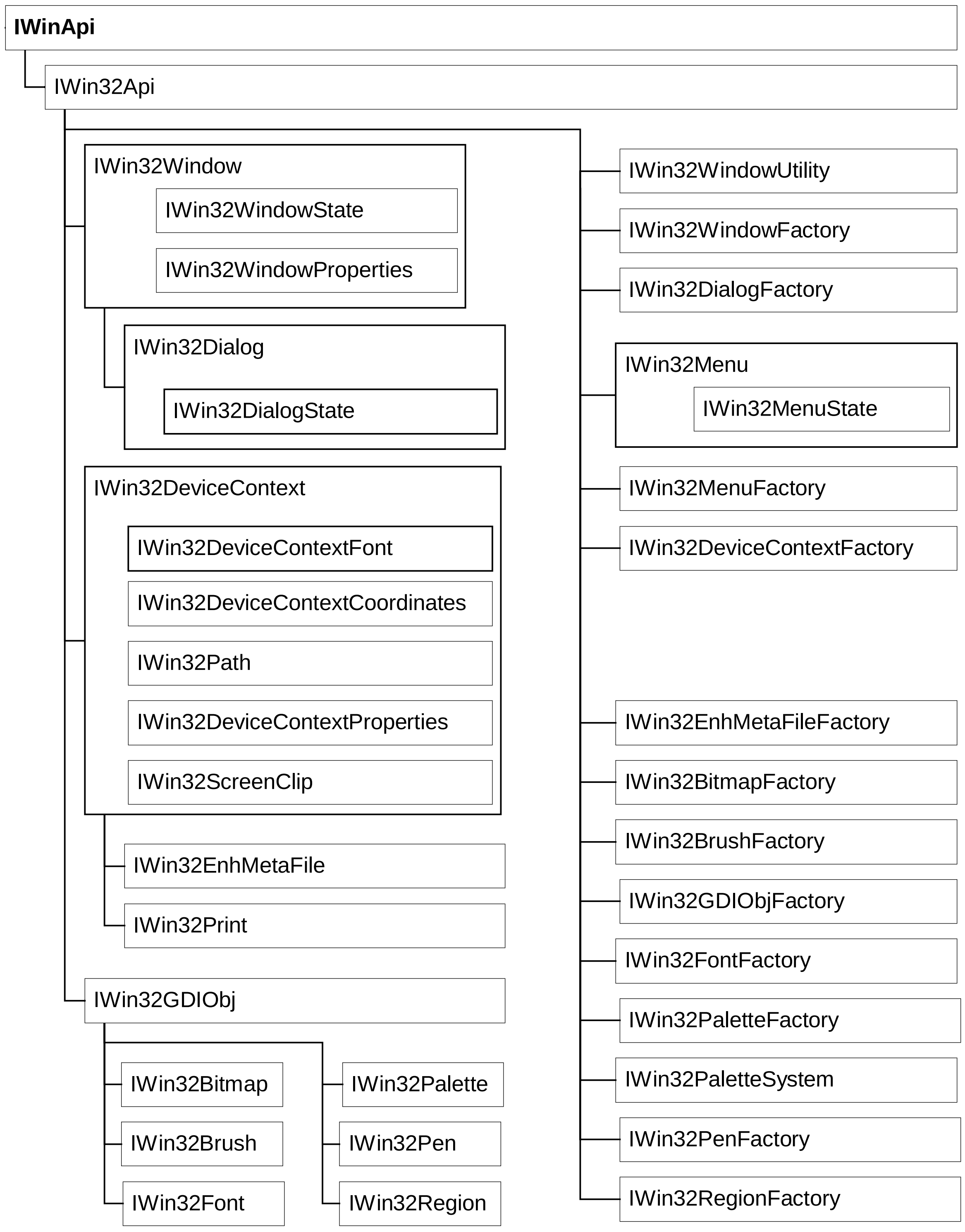
The next subsection lists the interface hierarchy. Inheritance relationships are clearly shown by the connecting lines, while aggregation is pictured by placing one interface block within another.

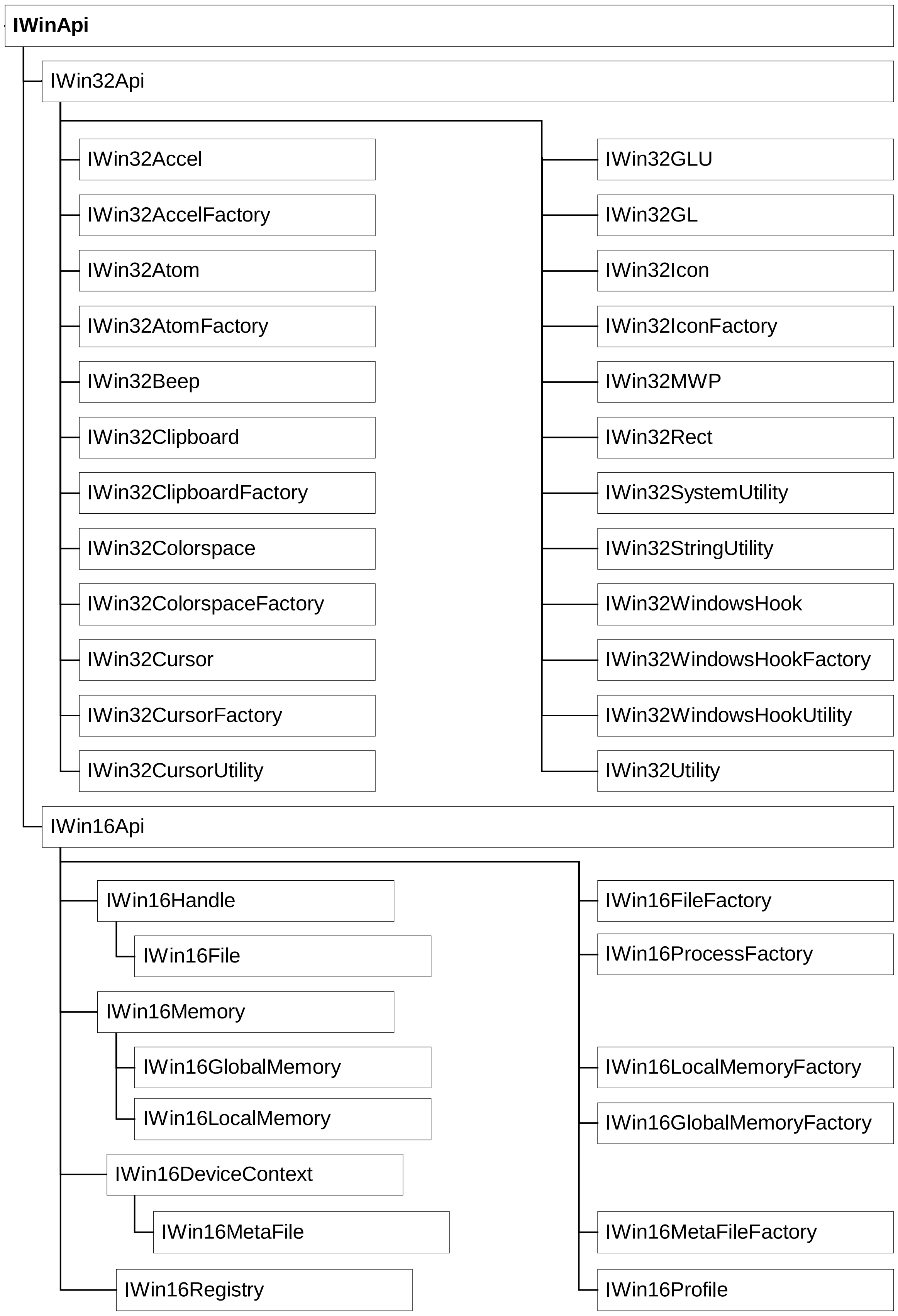
The final subsection then lists the call factorization. In the factorization list, "X : Y" denotes that X inherits from Y, and "Y ← X" denotes that X is aggregated into Y.

Interface Hierarchy




IWinApi
IWin32Api
IWin32Memory
IWin32VirtualMemory
IWin32Heap
IWin32VirtualMemoryFactory
IWin32HeapFactory
IWin32Keyboard
IWin32KeyboardLayout
IWin32KeyboardLayoutFactory
IWin32CriticalSection
IWin32CriticalSectionFactory
IWin32Printer
IWin32PrinterFactory
IWin32PrinterUtility
IWin32Registry
IWin32Resource
IWin32Drop
IWin32EnvironmentUtility
IWin32Shell
IWin32DDEUtility
IWin32DDEFactory
IWin32DDE
IWin32OLE
IWin32OLEUtility
IWin32OLEFactory
IWin32OLEMarshalUtility
IWin32OleMoniker
IWin32OleMonikerFactory
IWin32OleStream
IWin32OleStg
IWin32OleStgFactory
IWin32SecurityAccess
IWin32SecurityACL
IWin32SecurityACLUtility
IWin32SecurityDescriptor
IWin32SecurityDescriptorFactory


IWinApi
IWin32Api
IWin32Window
IWin32WindowState
IWin32WindowProperties
IWin32Dialog
IWin32DialogState
IWin32DeviceContext
IWin32DeviceContextFont
IWin32DeviceContextCoordinates
IWin32Path
IWin32DeviceContextProperties
IWin32ScreenClip
IWin32EnhMetaFile
IWin32Print
IWin32GDIObj
IWin32Bitmap
IWin32Brush
IWin32Font
IWin32Palette
IWin32Pen
IWin32Region
IWin32WindowUtility
IWin32WindowFactory
IWin32DialogFactory
IWin32Menu
IWin32MenuState
IWin32MenuFactory
IWin32DeviceContextFactory
IWin32EnhMetaFileFactory
IWin32BitmapFactory
IWin32BrushFactory
IWin32GDIObjFactory
IWin32FontFactory
IWin32PaletteFactory
IWin32PaletteSystem
IWin32PenFactory
IWin32RegionFactory


IWinApi
IWin32Api
IWin32Accel
IWin32AccelFactory
IWin32Atom
IWin32AtomFactory
IWin32Beep
IWin32Clipboard
IWin32ClipboardFactory
IWin32Colorspace
IWin32ColorspaceFactory
IWin32Cursor
IWin32CursorFactory
IWin32CursorUtility
IWin32GLU
IWin32GL
IWin32Icon
IWin32IconFactory
IWin32MWP
IWin32Rect
IWin32SystemUtility
IWin32StringUtility
IWin32WindowsHook
IWin32WindowsHookFactory
IWin32WindowsHookUtility
IWin32Utility
IWin16Api
IWin16Handle
IWin16File
IWin16Memory
IWin16GlobalMemory
IWin16LocalMemory
IWin16DeviceContext
IWin16MetaFile
IWin16Registry
IWin16FileFactory
IWin16ProcessFactory
IWin16LocalMemoryFactory
IWin16GlobalMemoryFactory
IWin16MetaFileFactory
IWin16Profile

Call Factorization

Generic Handles

IWin32Handle

CloseHandle

Atoms

IWin32Atom

GlobalDeleteAtom
GlobalGetAtomNameA

IWin32AtomFactory

GlobalAddAtomA

Clipboard

IWin32Clipboard

ChangeClipboardChain
CloseClipboard
GetClipboardData
GetClipboardFormatNameA
GetClipboardFormatNameW
GetClipboardOwner
GetClipboardViewer
GetOpenClipboardWindow
IsClipboardFormatAvailable
SetClipboardData

IWin32ClipboardFactory

RegisterClipboardFormatA
RegisterClipboardFormatW

Console

IWin32Console : IWin32SyncHandle

GetConsoleMode
GetNumberOfConsoleInputEvents
PeekConsoleInputA
ReadConsoleA
ReadConsoleInputA
SetConsoleMode
SetStdHandle
WriteConsoleA

IWin32ConsoleFactory

AllocConsole
GetStdHandle

Drawing

IWin16DeviceContextFont : IWin16DeviceContext

EnumFontFamiliesA
EnumFontsW
GetCharWidthA
GetTextExtentPointA
GetTextExtentPointW

IWin16MetaFile : IWin16DeviceContext

CloseMetaFile
CopyMetaFileA
DeleteMetaFile
EnumMetaFile
GetMetaFileA
GetMetaFileBitsEx
GetWinMetaFileBits
PlayMetaFile
PlayMetaFileRecord

IWin16MetaFileFactory

GetEnhMetaFileA
SetEnhMetaFileBits
SetMetaFileBitsEx

IWin32Bitmap:IWin32GDIObject

CreatePatternBrush
GetBitmapDimensionEx
GetDIBits
SetBitmapDimensionEx
SetDIBits
SetDIBitsToDevice

IWin32BitmapFactory

CreateBitmap
CreateBitmapIndirect
CreateCompatibleBitmap
CreateDIBSection
CreateDIBitmap
CreateDiscardableBitmap

IWin32BrushFactory

CreateBrushIndirect
CreateDIBPatternBrushPt
CreateHatchBrush
CreateSolidBrush

IWin32Colorspace

DeleteColorSpace

IWin32ColorspaceFactory

CreateColorSpaceA

IWin32Cursor

DestroyCursor
SetCursor

IWin32CursorFactory

GetCursor

IWin32CursorUtility

ClipCursor
GetCursorPos
SetCursorPos
ShowCursor

IWin32DeviceContext← IWin32DeviceContextFont,

IWin32DeviceContextCoords, IWin32Path, IWin32DeviceContextProperties, IWin32ScreenClip

AngleArc
Arc
ArcTo
BitBlt
Chord
CreateCompatibleDC
DeleteDC
DrawEdge
DrawEscape
DrawFocusRect
DrawFrameControl
DrawIcon
DrawIconEx
DrawStateA
DrawTextA
DrawTextW
Ellipse
EnumObjects
ExtFloodFill
ExtTextOutA
ExtTextOutW
FillRect
FillRgn
FloodFill
FrameRect
FrameRgn
GdiFlush
GetCurrentObject
GetCurrentPositionEx
GetPixel
GrayStringA
GrayStringW
InvertRect
InvertRgn
LineDDA
LineTo
MaskBlt
MoveToEx
PaintRgn
PatBlt
Pie
PlgBlt
PolyBezier
PolyBezierTo
PolyDraw
PolyPolygon
PolyPolyline
Polygon
Polyline
PolylineTo
Rectangle
ReleaseDC
ResetDCA
RestoreDC
RoundRect
SaveDC
ScrollDC
SetPixel
SetPixelV
StretchBlt
StretchDIBits
TabbedTextOutA
TextOutA
TextOutW
WindowFromDC

IWin32DeviceContextCoordinates

DPtoLP
LPtoDP

IWin32DeviceContextFactory

CreateDCA
CreateDCW
CreateICA
CreateICW
CreateMetaFileA
CreateMetaFileW

IWin32DeviceContextFont

EnumFontFamiliesExA
GetAspectRatioFilterEx
GetCharABCWidthsA
GetCharABCWidthsFloatA
GetCharABCWidthsW
GetCharWidth32A
GetCharWidth32W
GetCharWidthFloatA
GetFontData
GetGlyphOutlineA
GetGlyphOutlineW
GetKerningPairsA
GetOutlineTextMetricsA
GetTabbedTextExtentA
GetTextAlign
GetTextCharacterExtra
GetTextCharsetInfo
GetTextColor
GetTextExtentExPointA
GetTextExtentExPointW
GetTextExtentPoint32A
GetTextExtentPoint32W
GetTextFaceA
GetTextMetricsA
GetTextMetricsW

SetMapperFlags
SetTextAlign
SetTextCharacterExtra
SetTextColor
SetTextJustification

IWin32DeviceContextProperties

GetArcDirection
GetBkColor
GetBkMode
GetBoundsRect
GetBrushOrgEx
GetColorAdjustment
GetColorSpace
GetDeviceCaps
GetMapMode
GetNearestColor
GetPolyFillMode
GetROP2
GetStretchBltMode
GetViewportExtEx
GetViewportOrgEx
GetWindowExtEx
GetWindowOrgEx
OffsetViewportOrgEx
OffsetWindowOrgEx
PtVisible
RectVisible
ScaleViewportExtEx
ScaleWindowExtEx
SetArcDirection
SetBkColor
SetBkMode
SetBoundsRect
SetBrushOrgEx
SetColorAdjustment
SetColorSpace
SetDIBColorTable
SetICMMode
SetMapMode
SetMiterLimit
SetPolyFillMode
SetROP2
SetStretchBltMode
SetViewportExtEx
SetViewportOrgEx
SetWindowExtEx
SetWindowOrgEx
UpdateColors

IWin32EnhMetaFile: IWin32DeviceContext

CloseEnhMetaFile
CopyEnhMetaFileA
CreateEnhMetaFileA
CreateEnhMetaFileW
DeleteEnhMetaFile
EnumEnhMetaFile
GdiComment
GetEnhMetaFileBits
GetEnhMetaFileDescriptionA
GetEnhMetaFileDescriptionW
GetEnhMetaFileHeader
GetEnhMetaFilePaletteEntries
PlayEnhMetaFile
PlayEnhMetaFileRecord

IWin32EnhMetaFileFactory

SetWinMetaFileBits

IWin32FontFactory

CreateFontA
CreateFontIndirectA
CreateFontIndirectW
CreateFontW

IWin32GDIObject

DeleteObject
GetObjectA
GetObjectType
GetObjectW
SelectObject
UnrealizeObject

IWin32GDIObjectFactory

GetStockObject

IWin32Icon

CopyIcon
DestroyIcon
GetIconInfo

IWin32IconFactory

CreateIcon
CreateIconFromResource
CreateIconFromResourceEx
CreateIconIndirect
CreateMenu

IWin32Palette : IWin32GDIObject

AnimatePalette
GetNearestPaletteIndex
GetPaletteEntries
ResizePalette
SelectPalette
SetPaletteEntries

IWin32PaletteFactory

CreateHalftonePalette
CreatePalette

IWin32PaletteSystem

GetSystemPaletteEntries
GetSystemPaletteUse
RealizePalette

IWin32Path

AbortPath
BeginPath
CloseFigure
EndPath
FillPath
FlattenPath
GetMiterLimit
GetPath
PathToRegion
StrokeAndFillPath
StrokePath
WidenPath

IWin32PenFactory

CreatePen
CreatePenIndirect
ExtCreatePen

IWin32Print : IWin32DeviceContext

AbortDoc
EndDoc
EndPage
Escape
ExtEscape
SetAbortProc
StartDocA
StartDocW
StartPage

IWin32Rect

CopyRect
EqualRect
InflateRect
IntersectRect
IsRectEmpty
OffsetRect
PtInRect
SetRect
SetRectEmpty
SubtractRect
UnionRect

IWin32Region : IWin32GDIObject

CombineRgn
EqualRgn
GetRegionData
GetRgnBox
OffsetRgn
PtInRegion
RectInRegion
SetRectRgn

IWin32RegionFactory

CreateEllipticRgn
CreateEllipticRgnIndirect
CreatePolyPolygonRgn
CreatePolygonRgn
CreateRectRgn
CreateRectRgnIndirect
CreateRoundRectRgn
ExtCreateRegion

IWin32ScreenClip : IWin32DeviceContext

ExcludeClipRect
ExcludeUpdateRgn
ExtSelectClipRgn
GetClipBox
GetClipRgn
IntersectClipRect
OffsetClipRgn
SelectClipPath
SelectClipRgn

Environment

IWin32EnvironmentUtility

FreeEnvironmentStringsA
FreeEnvironmentStringsW
GetEnvironmentStrings
GetEnvironmentStringsW
GetEnvironmentVariableW
SetEnvironmentVariableA
SetEnvironmentVariableW

File

IWin16File : IWin16Handle

_hread
_hwrite
_lclose
_llseek
_lopen
_lwrite

IWin16FileFactory

OpenFile
_lcreat
_lread

IWin32File : IWin32AsyncIOHandle

FlushFileBuffers
GetFileInformationByHandle
GetFileSize
GetFileTime
GetFileType
LockFile
LockFileEx
ReadFile
ReadFileEx

SetEndOfFile
SetFilePointer
SetFileTime
UnlockFile
WriteFile
WriteFileEx

IWin32FileFactory

CreateFileA
CreateFileW
OpenFileMappingA

IWin32FileMapping: IWin32ASyncIOHandle

MapViewOfFile
UnmapViewOfFile

IWin32FileMappingFactory

CreateFileMappingA

IWin32FileSystem

CopyFileA
CopyFileEx
CopyFileW
CreateDirectoryA
CreateDirectoryExA
CreateDirectoryExW
CreateDirectoryW
DeleteFileA
DeleteFileW
GetDiskFreeSpaceA
GetDiskFreeSpaceEx
GetDriveTypeA
GetDriveTypeW
GetFileAttributesA
GetFileAttributesW
GetFileVersionInfoA
GetFileVersionInfoSizeA
GetLogicalDriveStringsA
GetLogicalDrives
GetVolumeInformationA
GetVolumeInformationW
MoveFileA
MoveFileEx
MoveFileW
RemoveDirectoryA
RemoveDirectoryW
SetFileAttributesA
SetFileAttributesW
UnlockFileEx
VerQueryValueA

IWin32FileUtility

AreFileApisANSI
CompareFileTime
DosDateTimeToFileTime
FileTimeToDosDateTime
FileTimeToLocalFileTime
FileTimeToSystemTime
GetFullPathNameA
GetFullPathNameW
GetShortPathNameA
GetShortPathNameW
GetTempFileNameA
GetTempFileNameW
GetTempPathA
GetTempPathW
LocalFileTimeToFileTime
SearchPathA
SystemTimeToFileTime

IWin32FindFile : IWin32ASyncIOHandle

FindClose
FindCloseChangeNotification
FindFirstFileEx
FindNextChangeNotification
FindNextFileA
FindNextFileW

IWin32FindFileFactory

FindFirstChangeNotificationA
FindFirstChangeNotificationW
FindFirstFileA
FindFirstFileW

Interprocess Communication

IWin32DDE

DdeAccessData
DdeDisconnect
DdeFreeDataHandle
DdeFreeStringHandle
DdeUnaccessData

IWin32DDEFactory

DdeClientTransaction
DdeConnect
DdeCreateStringHandleA

IWin32DDEUtility

DdeGetLastError
DdeInitializeA
ReuseDDElParam
UnpackDDElParam

IWin32Pipe : IWin32AsyncIOHandle

PeekNamedPipe

IWin32PipeFactory

CreatePipe

Keyboard

IWin32Keyboard

GetAsyncKeyState

GetKeyState
GetKeyboardState
MapVirtualKeyA
SetKeyboardState
VkKeyScanA
keybd_event

IWin32KeyboardLayout

ActivateKeyboardLayout

IWin32KeyboardLayoutFactory

GetKeyboardLayout

Memory

IWin16GlobalMemory : IWin16Memory

GlobalFlags
GlobalFree
GlobalLock
GlobalReAlloc
GlobalSize
GlobalUnlock

IWin16GlobalMemoryFactory

GlobalAlloc
GlobalHandle

IWin32Heap : IWin32Memory

HeapAlloc
HeapCompact
HeapDestroy
HeapFree
HeapReAlloc
HeapSize
HeapValidate
HeapWalk

IWin32HeapFactory

GetProcessHeap
HeapCreate

IWin16LocalMemory : IWin16Memory

LocalFree
LocalLock
LocalReAlloc
LocalUnlock

IWin32LocalMemoryFactory

LocalAlloc

IWin16Memory

IsBadCodePtr
IsBadReadPtr
IsBadStringPtrA
IsBadStringPtrW
IsBadWritePtr

IWin32Memory

IsBadCodePtr
IsBadReadPtr
IsBadStringPtrA
IsBadStringPtrW
IsBadWritePtr

IWin32VirtualMemory : IWin32Memory

VirtualFree
VirtualLock
VirtualProtect
VirtualQuery
VirtualUnlock

IWin32VirtualMemoryFactory

VirtualAlloc

Module

IWin32Module : IWin32Handle

DisableThreadLibraryCalls
EnumResourceNamesA
FindResourceA
FreeLibrary
GetModuleFileNameA
GetModuleFileNameW
GetProcAddress
LoadBitmapA
LoadBitmapW
LoadCursorA
LoadCursorW
LoadIconA
LoadIconW
LoadImageA
LoadMenuA
LoadMenuIndirectA
LoadStringA
SizeofResource

IWin32ModuleFactory

GetModuleHandleA
GetModuleHandleW
LoadLibraryA
LoadLibraryExA
LoadLibraryW

Multiple Window Position

IWin32MWP

BeginDeferWindowPos
DeferWindowPos
EndDeferWindowPos

Ole

IWin32Ole

CoDisconnectObject
CoLockObjectExternal
CoRegisterClassObject
CoRevokeClassObject

IWin32OleFactory

- BindMoniker
- CoCreateInstance
- CoGetClassObject
- CoGetInstanceFromFile
- CreateDataAdviseHolder
- CreateDataCache
- CreateILockBytesOnHGlobal
- CreateOleAdviseHolder
- CreateStreamOnHGlobal
- OleCreate
- OleCreateDefaultHandler
- OleCreateFromData
- OleCreateFromFile
- OleCreateLink
- OleCreateLinkFromData
- OleCreateLinkToFile
- OleGetClipboard
- OleLoad

IWin32OleMarshalUtility

- CoMarshalInterface
- CoReleaseMarshalData
- CoUnmarshalInterface

IWin32OleMoniker

- CreateGenericComposite
- CreateItemMoniker
- CreatePointerMoniker
- CreateURLMoniker
- MkParseDisplayName
- MonikerCommonPrefixWith
- MonikerRelativePathTo

IWin32OleMonikerFactory

- CreateBindCtx
- CreateFileMoniker
- GetRunningObjectTable

IWin32OleStg

- OleConvertIStorageToOLESTREAM
- OleSave
- ReadClassStg
- ReleaseStgMedium
- WriteClassStg
- WriteFmtUserTypeStg

IWin32OleStgFactory

- StgCreateDocfile
- StgCreateDocfileOnILockBytes
- StgIsStorageFile
- StgOpenStorage

IWin32OleStream

- GetHGlobalFromStream
- OleConvertOLESTREAMToIStorage
- OleLoadFromStream
- OleSaveToStream
- ReadClassStm
- WriteClassStm

IWin32OleUtility

- CLSIDFromProgID
- CLSIDFromString
- CoCreateGuid
- CoFileTimeNow
- CoFreeUnusedLibraries
- CoGetMalloc
- CoInitialize
- CoRegisterMessageFilter
- CoTaskMemAlloc
- CoTaskMemFree
- CoTaskMemRealloc
- CoUninitialize
- GetClassFile
- GetHGlobalFromILockBytes
- IIDFromString
- OleGetIconOfClass
- OleInitialize
- OleIsRunning
- OleRegEnumVerbs
- OleRegGetMiscStatus
- OleRegGetUserType
- OleSetClipboard
- OleUninitialize
- ProgIDFromCLSID
- PropVariantClear
- RegisterDragDrop
- RevokeDragDrop
- StringFromCLSID
- StringFromGUID2
- StringFromIID

OpenGL

IWin32GL

- glBegin
- glClear
- glClearColor
- glClearDepth
- glColor3d
- glEnable
- glEnd
- glFinish
- glMatrixMode
- glNormal3d
- glPolygonMode
- glPopMatrix
- glPushMatrix
- glRotated
- glScaled

glTranslated
glVertex3d
glViewport
wglCreateContext
wglGetCurrentDC
wglMakeCurrent

IWin32GLU

gluCylinder
gluDeleteQuadric
gluNewQuadric
gluPerspective
gluQuadricDrawStyle
gluQuadricNormals

Printer

IWin32Printer

ClosePrinter
DocumentPropertiesA
GetPrinterA

IWin32PrinterFactory

OpenPrinterA
OpenPrinterW

IWin32PrinterUtility

DeviceCapabilitiesA
EnumPrintersA

Process

IWin16ProcessFactory

WinExec

IWin32Process : IWin32SyncHandle ← IWin32ProcessContext

DebugBreak
ExitProcess
FatalAppExitA
FatalExit
GetExitCodeProcess
GetCurrentProcessId
GetProcessVersion
GetProcessWorkingSetSize
OpenProcessToken
SetProcessWorkingSetSize
TerminateProcess
UnhandledExceptionFilter

IWin32ProcessContext

GetCommandLineA
GetCommandLineW
GetCurrentDirectoryA
GetCurrentDirectoryW
GetStartupInfoA
SetConsoleCtrlHandler
SetCurrentDirectoryA
SetCurrentDirectoryW
SetHandleCount
SetUnhandledExceptionFilter

IWin32ProcessFactory

CreateProcessA
CreateProcessW
OpenProcess

Registry

IWin16Profile

GetPrivateProfileIntA
GetPrivateProfileStringA
GetPrivateProfileStringW
GetProfileIntA
GetProfileIntW
GetProfileStringA
GetProfileStringW
WritePrivateProfileStringA
WritePrivateProfileStringW
WriteProfileStringA
WriteProfileStringW

IWin16Registry

RegCreateKeyExA
RegCreateKeyW
RegEnumKeyA
RegEnumKeyW
RegOpenKeyA
RegOpenKeyW
RegQueryValueA
RegQueryValueW
RegSetValueA
RegSetValueW

IWin32Registry

RegCloseKey
RegCreateKeyA
RegCreateKeyExW
RegDeleteKeyA
RegDeleteKeyW
RegDeleteValueA
RegDeleteValueW
RegEnumKeyExA
RegEnumKeyExW
RegEnumValueA
RegEnumValueW
RegFlushKey
RegNotifyChangeKeyValue
RegOpenKeyExA
RegOpenKeyExW
RegQueryInfoKeyA
RegQueryInfoKeyW
RegQueryValueExA
RegQueryValueExW

RegSetValueExA
RegSetValueExW

Resource

IWin32Resource

LoadResource
LockResource

Security

IWin32SecurityACL

AddAccessAllowedAce
AddAccessDeniedAce
AddAce
DeleteAce
GetAce
GetAclInformation

IWin32SecurityACLUtility

InitializeAcl
IsValidAcl

IWin32SecurityAccess

CopySid
EqualSid
GetLengthSid
IsValidSid
LookupAccountNameA
LookupAccountSid
LookupPrivilegeValueA

IWin32SecurityDescriptor

GetSecurityDescriptorDacl
GetSecurityDescriptorGroup
GetSecurityDescriptorOwner
GetSecurityDescriptorSacl
IsValidSecurityDescriptor
SetSecurityDescriptorDacl
SetSecurityDescriptorGroup
SetSecurityDescriptorOwner
SetSecurityDescriptorSacl

IWin32SecurityDescriptorFactory

InitializeSecurityDescriptor

IWin32SecurityToken : IWin32Handle

AdjustTokenPrivileges
GetTokenInformation

IWin32SecurityToken : IWin32Handle

OpenProcessToken
OpenThreadToken

Shell

IWin32Drop

DragFinish
DragQueryFileW
DragQueryPoint

IWin32Shell

SHGetDesktopFolder
SHGetFileInfoA
ShellExecuteA

Synchronization

IWin32AtomicUtility

InterlockedDecrement
InterlockedExchange
InterlockedIncrement

IWin32CriticalSection

DeleteCriticalSection
EnterCriticalSection
LeaveCriticalSection

IWin32CriticalSectionFactory

InitializeCriticalSection

IWin32Event : IWin32SyncHandle

PulseEvent
ResetEvent
SetEvent

IWin32EventFactory

CreateEventA

IWin32Mutex : IWin32SyncHandle

ReleaseMutex

IWin32MutexFactory

CreateMutexA
OpenMutexA

IWin32Semaphore : IWin32SyncHandle

ReleaseSemaphore

IWin32SemaphoreFactory

CreateSemaphoreA

IWin32SyncHandle : IWin32Handle

MsgWaitForMultipleObjects
SignalObjectAndWait
WaitForMultipleObjects
WaitForSingleObject
WaitForSingleObjectEx

IWin32WaitableTimer : IWin32SyncHandle

CancelWaitableTimer
SetWaitableTimer

IWin32WaitableTimerFactory

CreateWaitableTimer
OpenWaitableTimer

System

IWin32WindowsHook

CallNextHookEx

UnhookWindowsHookEx

IWin32WindowsHookFactory

SetWindowsHookExA
SetWindowsHookExW

IWin32WindowsHookUtility

CallMsgFilterA
CallMsgFilterW

Thread

IWin32Thread : IWin32SyncHandle ← IWin32ThreadContext, IWin32ThreadMessage

DispatchMessageA
DispatchMessageW
ExitThread
GetCurrentThreadId
GetExitCodeThread
GetThreadLocale
GetThreadPriority
OpenThreadToken
ResumeThread
SetThreadPriority
SetThreadToken
Sleep
SuspendThread
TerminateThread
TlsAlloc
TlsFree
TlsGetValue
TlsSetValue

IWin32ThreadContext

EnumThreadWindows
GetActiveWindow

IWin32ThreadFactory

CreateThread

IWin32ThreadMessage

GetMessageA
GetMessagePos
GetMessageTime
GetMessageW
GetQueueStatus
PostQuitMessage
PostThreadMessageA
TranslateMessage
WaitMessage

IWin32ThreadUtility

Timer

IWin32Timer

KillTimer
SetTimer

Utilities

IWin32Beep

Beep
MessageBeep

IWin32StringUtility

CharLowerA
CharLowerBuffA
CharLowerW
CharNextA
CharNextW
CharPrevA
CharToOemA
CharUpperA
CharUpperBuffA
CharUpperBuffW
CharUpperW
CompareStringA
CompareStringW
FormatMessageA
FormatMessageW
GetStringTypeA
GetStringTypeExA
GetStringTypeW
IsCharAlphaA
IsCharAlphaNumericA
IsCharAlphaNumericW
IsCharAlphaW
IsDBCSLeadByte
IsDBCSLeadByteEx
LCMapStringA
LCMapStringW
MultiByteToWideChar
OutputDebugStringA
OutputDebugStringW
ToAscii
WideCharToMultiByte
lstrcatA
lstrcmpA
lstrcmpiA
lstrcpyA
lstrcpyW
lstrcpynA
lstrlenA
lstrlenW
wsprintfA
wsprintfW
wvsprintfA

IWin32SystemUtility

CountClipboardFormats
EmptyClipboard
EnumClipboardFormats

EnumSystemLocalesA
GetACP
GetCPInfo
GetComputerNameW
GetCurrentProcess
GetCurrentProcessId
GetCurrentThread
GetCurrentThreadId
GetDateFormatA
GetDateFormatW
GetDialogBaseUnits
GetDoubleClickTime
GetLastError
GetLocalTime
GetLocaleInfoA
GetLocaleInfoW
GetOEMCP
GetSysColor
GetSysColorBrush
GetSystemDefaultLCID
GetSystemDefaultLangID
GetSystemDirectoryA
GetSystemInfo
GetSystemMetrics
GetSystemTime
GetTickCount
GetTimeFormatA
GetTimeFormatW
GetTimeZoneInformation
GetUserDefaultLCID
GetUserDefaultLangID
GetUserNameA
GetUserNameW
GetVersion
GetVersionExA
GetWindowsDirectoryA
GetWindowsDirectoryW
GlobalMemoryStatus
IsValidCodePage
IsValidLocale
OemToCharA
QueryPerformanceCounter
QueryPerformanceFrequency
RaiseException
RegisterWindowMessageA
SetErrorMode
SetLastError
SetLocalTime
SystemParametersInfoA

IWin32Utility

MulDiv

Window

IWin32Accel

CopyAcceleratorTableA
TranslateAcceleratorA

IWin32AccelFactory

LoadAcceleratorsA

IWin32Dialog : IWin32Window ← IWin32DialogState

ChooseColorA
DialogBoxParamA
DialogBoxParamW
EndDialog
MapDialogRect
SendDlgItemMessageA

IWin32DialogFactory

CreateDialogIndirectParamA
CreateDialogParamA
DialogBoxIndirectParamA

IWin32DialogState

CheckDlgButton
GetDlgCtrlID
GetDlgItem
GetDlgItemInt
GetDlgItemTextA
GetNextDlgGroupItem
GetNextDlgTabItem
IsDlgButtonChecked
SetDlgItemInt
SetDlgItemTextA

IWin32Menu ← IWin32MenuState

DeleteMenu
DestroyMenu
InsertMenuA
InsertMenuW
IsMenu
ModifyMenuA
RemoveMenu
TrackPopupMenu

IWin32MenuFactory

CreatePopupMenu

IWin32MenuState

AppendMenuA
AppendMenuW
ArrangeIconicWindows
BringWindowToTop
CheckMenuItem
CheckMenuRadioItem
CheckRadioButton
EnableMenuItem
GetMenuItemCount
GetMenuItemID

GetMenuItemRect
GetMenuState
GetMenuStringA
GetSubMenu
HiliteMenuItem
SetMenuDefaultItem
SetMenuItemBitmaps

IWin32Window← IWin32WindowProperties, IWin32WindowState

BeginPaint
CallWindowProcA
CallWindowProcW
ChildWindowFromPoint
ChildWindowFromPointEx
ClientToScreen
CloseWindow
CreateCaret
DefFrameProcA
DefMDIChildProcA
DefWindowProcA
DefWindowProcW
DestroyWindow
DlgDirListA
DlgDirListComboBoxA
DlgDirSelectComboBoxExA
DlgDirSelectExA
DrawAnimatedRects
DrawMenuBar
EndPaint
EnumChildWindows
EnumWindows
FindWindow
FlashWindow
MapWindowPoints
MessageBoxA
MessageBoxW
MoveWindow
OpenClipboard
OpenIcon
PeekMessageA
PeekMessageW
PostMessageA
PostMessageW
RedrawWindow
ScreenToClient
ScrollWindow
ScrollWindowEx
SendMessageA
SendMessageW
SendNotifyMessageA
TranslateMDISysAccel
UpdateWindow

IWin32WindowFactory

CreateWindowExA
CreateWindowExW

IWin32WindowProperties

DragAcceptFiles
GetClassLongA
GetClassNameA
GetClassNameW
GetPropA
GetPropW
RemovePropA
RemovePropW
SetClassLongA
SetPropA
SetPropW

IWin32WindowState

EnableScrollBar
EnableWindow
GetClientRect
GetDC
GetDCEx
GetLastActivePopup
GetMenu
GetParent
GetScrollInfo
GetScrollPos
GetScrollRange
GetSystemMenu
GetTopWindow
GetUpdateRect
GetUpdateRgn
GetWindow
GetWindowDC
GetWindowLongA
GetWindowLongW
GetWindowPlacement
GetWindowRect
GetWindowRgn
GetWindowTextA
GetWindowTextLengthA
GetWindowTextW
GetWindowThreadProcessId
HideCaret
InvalidateRect
InvalidateRgn
IsWindowEnabled
IsChild
IsIconic
IsWindow
IsWindowUnicode
IsWindowVisible
IsZoomed

LockWindowUpdate
SetActiveWindow
SetClipboardViewer
SetFocus
SetForegroundWindow
SetMenu
SetParent
SetScrollInfo
SetScrollPos
SetScrollRange
SetWindowLongA
SetWindowLongW
SetWindowPlacement
SetWindowPos
SetWindowRgn
SetWindowTextA
SetWindowTextW
ShowCaret
ShowOwnedPopups
ShowScrollBar
ShowWindow
ValidateRect
ValidateRgn

IWin32WindowUtility

AdjustWindowRect
AdjustWindowRectEx
EnumWindows
FindWindowA
GetActiveWindow
GetCapture
GetCaretPos
GetClassInfoA
GetClassInfoExA
GetClassInfoW
GetDesktopWindow
GetFocus
GetForegroundWindow
InSendMessage
IsDialogMessageA
RegisterClassA
RegisterClassExA
RegisterClass