<http://msdn.microsoft.com/en-us/library/ms973872.aspx>

.NET Development (General) Technical Articles

**An Overview of Managed/Unmanaged Code Interoperability**

Sonja Keserovic, Program Manager  
David Mortenson, Lead Software Design Engineer  
Adam Nathan, Lead Software Design Engineer in Test

Microsoft Corporation

October 2003

Applies to:  
   Microsoft® .NET Framework  
   COM Interop

**Summary:** This article provides basic facts about interoperability between managed and unmanaged code, and guidelines and common practices for accessing and wrapping unmanaged API from managed code, and for exposing managed APIs to unmanaged callers. Security and reliability considerations, performance data, and general practices for development processes are also highlighted. (14 printed pages)

**Prerequisites:** The target audience for this document includes developers and managers who need to make high-level decisions about where to use managed code. In order to do that, it is helpful to understand how the interaction between managed and unmanaged code works, and how the current guidelines apply to specific scenarios.

**Contents**

[Introduction to Interoperability](http://msdn.microsoft.com/en-us/library/ms973872.aspx#manunman_topic1)  
[Interoperability Guidelines](http://msdn.microsoft.com/en-us/library/ms973872.aspx#manunman_topic2)  
[Security](http://msdn.microsoft.com/en-us/library/ms973872.aspx#manunman_topic3)  
[Reliability](http://msdn.microsoft.com/en-us/library/ms973872.aspx#manunman_topic4)  
[Performance](http://msdn.microsoft.com/en-us/library/ms973872.aspx#manunman_topic5)  
[Appendix 1: Crossing the Interoperability Boundary](http://msdn.microsoft.com/en-us/library/ms973872.aspx#manunman_topic6)  
[Appendix 2: Resources](http://msdn.microsoft.com/en-us/library/ms973872.aspx#manunman_topic7)  
[Appendix 3: Glossary of Terms](http://msdn.microsoft.com/en-us/library/ms973872.aspx#manunman_topic8)

**Introduction to Interoperability**

The [common language runtime](http://msdn.microsoft.com/en-us/library/ms973872.aspx#manunman_clr) (CLR) promotes the interaction of [managed code](http://msdn.microsoft.com/en-us/library/ms973872.aspx#manunman_mancode) with COM components, COM+ services, the Win32® API, and other types of [unmanaged code](http://msdn.microsoft.com/en-us/library/ms973872.aspx#manunman_rcw). Data types, error-handling mechanisms, creation and destruction rules, and design guidelines vary between managed and unmanaged object models. To simplify interoperation between managed and unmanaged code and to ease the migration path, the CLR interop layer conceals the differences between these object models from both clients and servers.

Interoperability ("[interop](http://msdn.microsoft.com/en-us/library/ms973872.aspx" \l "manunman_interop)") is bidirectional, which makes it possible to:

* **Call into unmanaged APIs from managed code**

This can be done for both flat APIs (static DLL exports, such as the Win32 API, which is exposed from DLLs like kernel32.dll and user32.dll) and COM APIs (object models such as those exposed by Microsoft® Word, Excel, Internet Explorer, ActiveX® Data Objects (ADO) and so forth).

* **Expose managed APIs to unmanaged code**

Examples of doing this include creating an add-in for a COM-based application like Windows Media® Player or embedding a managed Windows Forms control on an MFC form.

Three complementary technologies enable these managed/unmanaged interactions:

* [Platform Invoke](http://msdn.microsoft.com/en-us/library/ms973872.aspx#manunman_pinvoke) (sometimes referred to as P/Invoke) enables calling any function in any unmanaged language as long as its signature is redeclared in managed source code. This is similar to the functionality that was provided by the Declare statement in Visual Basic® 6.0.
* [COM interop](http://msdn.microsoft.com/en-us/library/ms973872.aspx#manunman_cominterop) enables calling into COM components in any managed language in a manner similar to using normal managed components, and vice versa. COM interop is comprised of core services provided by the CLR, plus some tools and APIs in the**System.Runtime.InteropServices** namespace.
* C++ interop (sometimes referred to as It Just Works (IJW)) is a C++-specific feature, which enables flat APIs and COM APIs to be used directly, as they have always been used. This is more powerful than COM interop, but it requires much more care. Make sure that you check the C++ resources before you use this technology.

**Interoperability Guidelines**

**Calling Unmanaged APIs from Managed Code**

There are several types of unmanaged APIs and several types of interop technologies available for calling into them. Suggestions about how and when to use these technologies are described in this section. Please note that these suggestions are very general and do not cover every scenario. You should evaluate your scenarios carefully and apply development practices and/or solutions that make sense for your scenario.

**Calling Unmanaged Flat APIs**

There are two mechanisms for calling into unmanaged flat APIs from managed code: through [Platform Invoke](http://msdn.microsoft.com/en-us/library/ms973872.aspx#manunman_pinvoke) (available in all managed languages) or through C++ interop (available in C++).

Before you decide to call a flat API using either of these interop technologies, you should determine whether there is equivalent functionality available in the .NET Framework. It is suggested that, whenever possible, you use .NET Framework functionality instead of calling unmanaged APIs.

For calling just a few unmanaged methods or for calling simple flat APIs, the suggestion is to use platform invoke instead of C++ interop. Writing platform invoke declarations for simple flat APIs is straightforward. The CLR will take care of DLL loading and all parameter marshaling. Even the work of writing a few platform invoke declarations for [complex flat APIs](http://msdn.microsoft.com/en-us/library/ms973872.aspx#manunman_complex) is negligible compared with the cost of using C++ interop and introducing a whole new module written in C++.

For wrapping complex unmanaged flat APIs, or for wrapping unmanaged flat APIs that are changing while managed code is under development, the suggestion is to use C++ interop instead of platform invoke. The C++ layer can be very thin and the rest of the managed code can be written in any other managed language of choice. Using platform invoke in these scenarios would require a lot of effort to re-declare complex parts of API in managed code and keep them in sync with the unmanaged APIs. Using C++ interop solves this problem by allowing direct access to unmanaged APIs—which requires no rewriting, just the inclusion of a header file.

**Calling COM APIs**

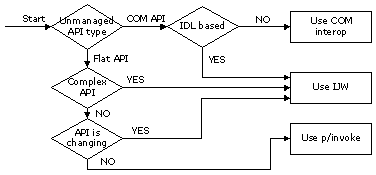
There are two ways to call COM components from managed code: through COM interop (available in all managed languages) or through C++ interop (available in C++).

For calling OLE Automation-compatible COM components, the suggestion is to use COM interop. The CLR will take care of COM component activation and parameter marshaling.

For calling COM components based on Interface Definition Language (IDL), the suggestion is to use C++ interop. The C++ layer can be very thin and the rest of the managed code can be written in any managed language. COM interop relies on information from type libraries to make correct interop calls, but type libraries typically do not contain all the information present in IDL files. Using C++ interop solves this problem by allowing direct access to these COM APIs.

For companies that own COM APIs that have already shipped, it is important to consider shipping primary interop assemblies (PIAs) for these APIs, thus making them easy to consume for managed clients.

**Decision Tree for Calling Unmanaged APIs**



**Figure 1. Calling Unmanaged APIs decision tree**

**Exposing Managed APIs to Unmanaged Code**

There are two main ways to expose a managed API to purely unmanaged callers: as a COM API or as a flat API. For C++ unmanaged clients that are willing to recompile their code with Visual Studio® .NET, there is a third option: directly accessing managed functionality through C++ interop. Suggestions for how and when to use these options are described in this section.

**Directly Accessing a Managed API**

If an unmanaged client is written in C++, it can be compiled with the Visual Studio .NET C++ compiler as a "mixed mode image." After this is done, the unmanaged client can directly access any managed API. However, some coding rules do apply to accessing managed objects from unmanaged code; check the C++ documentation for more details.

Direct access is the preferred option since it does not require any special consideration from managed API developers. They can design their managed API according to managed API design guidelines (DG) and be confident that the API will still be accessible to unmanaged callers.

**Exposing a Managed API as a COM API**

Every public managed class can be exposed to unmanaged clients through COM interop. This process is very easy to implement, because the COM interop layer takes care of all the COM plumbing. Thus, for example, every managed class appears to implement **IUnknown**,**IDispatch**, **ISupportErrorInfo**, and a few other standard COM interfaces.

Despite the fact that exposing managed APIs as COM APIs is easy, managed and COM object models are very different. Therefore, exposing managed API to COM should always be an explicit design decision. Some features available in the managed world have no equivalent in the COM world and will not be usable from COM clients. Because of this, there is often tension between managed API design guidelines (DG) and compatibility with COM.

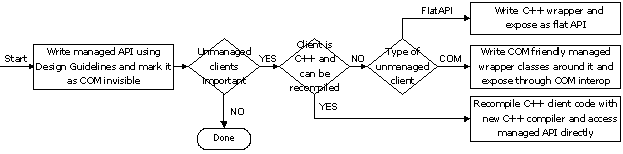
If COM clients are important, write your managed API according to the managed API design guidelines, and then write a thin COM-friendly managed [wrapper](http://msdn.microsoft.com/en-us/library/ms973872.aspx#manunman_rcw) around your managed API that will be exposed to COM.

**Exposing a Managed API as a Flat API**

Sometimes unmanaged clients cannot use COM. For example, they might already be written to use flat APIs and cannot be changed or recompiled. C++ is the only high-level language that allows you to expose managed APIs as flat APIs. Doing this is not as straightforward as exposing a managed API as a COM API. It is a very advanced technique that requires advanced knowledge of C++ interop and the differences between the managed and unmanaged worlds.

Expose your managed API as a flat API only if absolutely necessary. If you have no choice, be sure to check the C++ documentation and be fully aware of all the limitations.

**Decision Tree for Exposing Managed APIs**



**Figure 2. Exposing Managed APIs decision tree**

**Security**

The common language runtime ships with a security system, [Code Access Security](http://msdn.microsoft.com/en-us/library/c5tk9z76.aspx) (CAS), which regulates access to protected resources based on information about the origin of an [assembly](http://msdn.microsoft.com/en-us/library/ms973872.aspx#manunman_interopassembly). Calling unmanaged code presents a major security risk. Without appropriate security checks, unmanaged code could manipulate any state of any managed application in the CLR process. It is also possible to call resources in unmanaged code directly, without these resources being subject to any CAS permission checks. For that reason, any transition into unmanaged code is regarded as a highly protected operation and should include a security check. This security check looks for the unmanaged code permission that requires the assembly containing the unmanaged code transition, as well as all assemblies calling into it, to have the right to actually invoke unmanaged code.

There are some limited interop scenarios where full security checks are unnecessary and would unduly limit the performance or scope of the component. This is the case if a resource exposed from unmanaged code has no security relevance (system time, window coordinates, and so forth), or the resource is only used internally in the assembly and is not exposed publicly to arbitrary callers. In such cases, you can suppress the full security check for unmanaged code permission against all callers of the relevant APIs. You do this by applying the**SuppressUnmanagedCodeSecurity** custom attribute to the respective interop method or class. Note that this assumes a careful security review where you have determined that no partially trusted code could exploit such APIs.

**Reliability**

Managed code is designed to be more reliable and robust than unmanaged code. One example of a CLR feature that promotes these qualities is garbage collection, which takes care of freeing unused memory in order to prevent memory leaks. Another example is managed type safety, which is used to prevent buffer overrun mistakes and other type-related errors.

When you use any type of interop technology, your code might not be as reliable or robust as pure managed code. For example, you might need to allocate unmanaged memory manually and remember to free it when you are done with it.

Writing any non-trivial interop code requires the same attention to reliability and robustness as writing unmanaged code. Even when all interop code is written correctly, your system will only be as reliable as its unmanaged parts.

**Performance**

With every transition from managed code to unmanaged code (and vice versa), there is some performance overhead. The amount of overhead depends on the types of parameters used. The CLR interop layer uses three levels of interop call optimization based on transition type and parameter types: just-in-time (JIT) inlining, compiled assembly stubs, and interpreted marshaling stubs (in order of fastest to slowest type of call).

Approximate overhead for a platform invoke call: **10 machine instructions** (on an x86 processor)

Approximate overhead for a COM interop call: **50 machine instructions** (on an x86 processor)

The work done by these instructions is shown in the appendix sections [Calling a Flat API: Step by Step](http://msdn.microsoft.com/en-us/library/ms973872.aspx#manunman_flatapicall) and [Calling a COM API: Step by Step](http://msdn.microsoft.com/en-us/library/ms973872.aspx#manunman_comapi). Besides ensuring that the garbage collector will not block unmanaged threads during the call, and handling calling conventions and unmanaged exceptions, COM interop does extra work to convert the call on the current runtime callable wrapper (RCW) to a COM interface pointer appropriate to the current context.

Every interop call introduces some overhead. Depending on how often these calls occur and the significance of the work happening inside the method implementation, the per-call overhead can range from negligible to very noticeable.

Based on these considerations, the following list provides some general performance suggestions that you might find useful:

* If you control the interface between managed and unmanaged code, make it "chunky" rather than "chatty," to reduce the total number of transitions made.

Chatty interfaces are interfaces that make a lot of transitions without performing any significant work on the other side of the interop boundary. For example, property setters and getters are chatty. Chunky interfaces are interfaces that make only a few transitions, and the amount of work done on the other side of the boundary is significant. For example, a method that opens a database connection and retrieves some data is chunky. Chunky interfaces involve fewer interop transitions, so you eliminate some performance overhead.

* Avoid Unicode/ANSI conversions if possible.

Converting strings from Unicode to ANSI and vice versa is an expensive operation. For example, if strings need to be passed, but their content is not important, you can declare a string parameter as an **IntPtr** and the interop marshaler will not do any conversions.

* For high-performance scenarios, declaring parameters and fields as **IntPtr** can boost performance, albeit at the expense of ease-of-use and maintainability.

Sometimes it is faster to do manual marshaling using methods available on the **Marshal** class, rather than rely on default interop marshaling. For example, if large arrays of strings must be passed across an interop boundary, but only a few elements will be needed, declaring the array as an **IntPtr** and accessing only those few elements manually will be much faster.

* Use **InAttribute** and **OutAttribute** wisely to reduce unnecessary marshaling.

The interop marshaler uses default rules when deciding if a certain parameter needs to be marshaled in before the call and marshaled out after the call. These rules are based on the level of indirection and the parameter type. Some of these operations might not be necessary depending on the method's semantics.

* Use **SetLastError=false** on platform invoke signatures only if you will call **Marshal.GetLastWin32Error** afterwards.

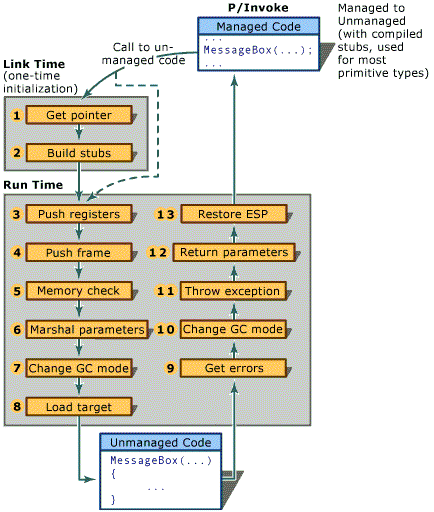
Setting **SetLastError=true** on platform invoke signatures requires additional work from the interop layer to preserve the last error code. Use this feature only when you rely on this information and will use it after you make the call.

* If, and only if, unmanaged calls are exposed in a non-exploitable fashion, use **SuppressUnmanagedCodeSecurityAttribute** to reduce the number of security checks.

Security checks are very important. If your API does not expose any protected resources or sensitive information, or they are well protected, extensive security checks might introduce unnecessary overhead. However, the cost of not doing any security check is very high.

**Appendix 1: Crossing the Interoperability Boundary**

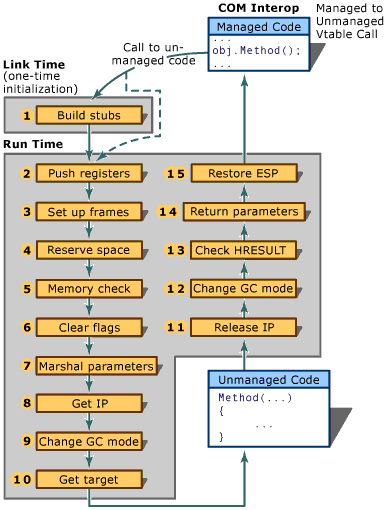
**Calling a Flat API: Step by Step**



**Figure 3. Calling a Flat API**

1. Get **LoadLibrary** and **GetProcAddress**.
2. Build a **DllImport** stub from the signature containing the target address.
3. Push callee-saved registers.
4. Set up a DllImport frame, and push it onto the stack of frames.
5. If temporary memory is allocated, initialize a clean-up list for rapid freeing when the call completes.
6. Marshal parameters. (This could allocate memory.)
7. Change Garbage Collection mode from cooperative to preemptive, so a Garbage Collection can occur at any time.
8. Load the target address and call it.
9. If **SetLastError** bit is set, call **GetLastError** and store the result in a thread abstraction stored in Thread Local Storage.
10. Change back to cooperative Garbage Collection mode.
11. If **PreserveSig=false** and the method returned a failure HRESULT, throw an exception.
12. If no exception was thrown, back-propagate **out** and **by-ref** parameters.
13. Restore the Extended Stack Pointer to its original value to account for caller-popped arguments.

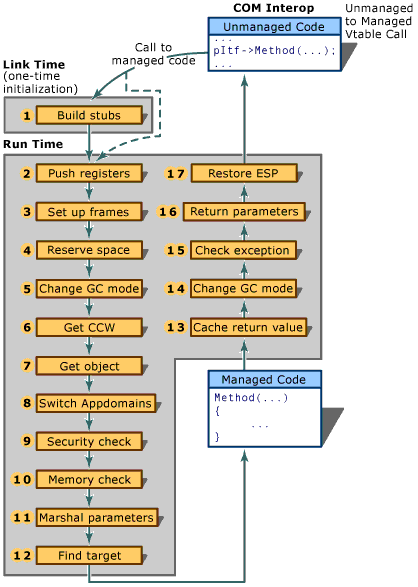
**Calling a COM API: Step by Step**



**Figure 4. Calling a COM API**

1. Build a managed-to-unmanaged stub from the signature.
2. Push callee-saved registers.
3. Set up a managed-to-unmanaged COM Interop frame, and push it onto the stack of frames.
4. Reserve space for temporary data used during the transition.
5. If temporary memory is allocated, initialize a clean-up list for rapid freeing when the call completes.
6. Clear floating-point exception flags (x86 only).
7. Marshal parameters. (This could allocate memory.)
8. Retrieve the correct interface pointer for the current context inside the Runtime Callable Wrapper. If the cached pointer cannot be used, call QueryInterface on the COM component to obtain it.
9. Change Garbage Collection mode from cooperative to preemptive, so a Garbage Collection can occur at any time.
10. From the vtable pointer, index by the slot number, get the target address, and call it.
11. Call **Release** on the interface pointer if **QueryInterface** was previously called.
12. Change back to cooperative Garbage Collection mode.
13. If the signature is not marked **PreserveSig**, check for failure HRESULT and throw an exception (potentially filled with **IErrorInfo**information).
14. If no exception was thrown, back-propagate **out** and **by-ref** parameters.
15. Restore the Extended Stack Pointer to the original value to account for caller-popped arguments.

**Calling a Managed API from COM: Step by Step**



**Figure 5. Calling a Managed API from COM**

1. Build an unmanaged-to-managed stub from the signature.
2. Push callee-saved registers.
3. Set up an unmanaged-to-managed COM Interop frame, and push it onto the stack of frames.
4. Reserve space for temporary data used during the transition.
5. Change Garbage Collection mode from cooperative to preemptive so a Garbage Collection can occur at any time.
6. Retrieve the [COM callable wrapper (CCW)](http://msdn.microsoft.com/en-us/library/ms973872.aspx#manunman_ccw) from the interface pointer.
7. Retrieve the managed object inside the CCW.
8. Transition [appdomains](http://msdn.microsoft.com/en-us/library/ms973872.aspx" \l "manunman_appdomain), if required.
9. If an appdomain is without full trust, perform any link demands that the method might have against the target appdomain.
10. If temporary memory is allocated, initialize a clean-up list for rapid freeing when the call completes.
11. Marshal parameters. (This could allocate memory.)
12. Find the target-managed method to call. (This involves mapping interface calls to the target implementation.)
13. Cache the return value. (If it is a floating point return value, get it from floating point register.)
14. Change back to cooperative Garbage Collection mode.
15. If an exception was thrown, extract its HRESULT to return, and call **SetErrorInfo**.
16. If no exception was thrown, back-propagate **out** and **by-ref** parameters.
17. Restore the Extended Stack Pointer to the original value to account for caller-popped arguments.

**Appendix 2: Resources**

***Must read!***[.NET and COM: The Complete Interoperability Guide](http://www.amazon.com/exec/obidos/ASIN/067232170X/102-9869026-1349709) by Adam Nathan

[Interoperating with Unmanaged Code](http://msdn.microsoft.com/en-us/library/sd10k43k.aspx), Microsoft .NET Framework Developer's Guide

[Interop samples](http://msdn.microsoft.com/en-us/library/bb985764.aspx), Microsoft .NET Framework

Adam Nathan's [blog](http://blogs.msdn.com/adam_nathan/)

Chris Brumme's [blog](http://blogs.gotdotnet.com/cbrumme/)

**Appendix 3: Glossary of Terms**

|  |  |
| --- | --- |
| AppDomain (Application Domain) | An application domain can be considered similar to a lightweight OS process, and is managed by the common language runtime. |
| CCW (COM callable wrapper) | A special kind of wrapper created by the CLR interop layer around managed objects activated from COM code. A CCW hides differences between managed and COM object models by providing data marshaling, lifetime management, identity management, error handling, correct apartment and threading transitions, and so forth. CCWs expose managed object functionality in a COM-friendly manner without requiring the managed code implementer to know anything about COM plumbing. |
| CLR | The common language runtime. |
| COM interop | The service provided by the CLR interop layer for using COM APIs from managed code, or exposing managed APIs as COM APIs to unmanaged clients. COM interop is available in all managed languages. |
| C++ interop | The service provided by the C++ language compiler and the CLR, it is used for directly mixing managed and unmanaged code in the same executable file. C++ interop usually involves including header files in unmanaged APIs and following certain coding rules. |
| Complex flat API | APIs that have signatures that are hard to declare in managed language. For example, methods with variable size structure parameters are hard to declare since there is no equivalent concept in the managed type system. |
| Interop | The general term that covers any type of interoperability between managed and unmanaged (also called "native") code. Interop is one of many services provided by the CLR. |
| Interop assembly | A special type of managed assembly that contains managed type equivalents for COM types contained in a type library. Typically produced by running the Type Library Importer tool (Tlbimp.exe) on a type library. |
| Managed code | Code executing under the control of the CLR is called managed code. For example, any code written in C# or Visual Basic .NET is managed code. |
| Platform Invoke | The service provided by the CLR interop layer for calling unmanaged flat APIs from managed code. Platform invoke is available in all managed languages. |
| RCW (runtime callable wapper) | A special kind of wrapper created by the CLR interop layer around COM objects that are activated from managed code. An RCW hides differences between managed and COM object models by providing data marshaling, lifetime management, identity management, error handling, correct apartment and threading transitions, and so forth. |
| Unmanaged code | Code that runs outside the CLR is referred to as "unmanaged code." COM components, ActiveX components, and Win32 API functions are examples of unmanaged code. |