



Platforms, Frameworks & Libraries » COM / COM+ » Beginners **Beginner**

VC6Win2K, Visual-Studio, MFC, ATL, COM, Dev

Introduction to COM - What It Is and How to Use It.

By [Michael Dunn](#)

Posted: **1 Jul 2000**
 Updated: **27 Jul 2000**
 Views: **778,507**
 Bookmarked: **653 times**

A tutorial for programmers new to COM that explains how to reuse existing COM components, for example, components in the Windows shell.

542 votes for this article.

Popularity: 13.08 Rating: **4.78** out of 5



[Download demo source code - 7 Kb](#)

Purpose of this Article

I have written this tutorial for programmers who are just starting out in COM and need some help in understanding the basics. The article briefly covers the COM specification, and then explains some COM terminology and describes how to reuse existing COM components. This article does *not* cover writing your own COM objects or interfaces.

Updates:

July 22, 2000: Added a [couple of more paragraphs](#) about using `Release()` and unloading DLLs. Also added the [section on HRESULTs](#).

Introduction

COM (Component Object Model) is the popular TLA (three-letter acronym) that seems to be everywhere in the Windows world these days. There are tons of new technologies coming out all the time, all based on COM. The documentation throws around lots of terms like *COM object*, *interface*, *server*, and so on, but it all assumes you're familiar with how COM works and how to use it.

This article introduces COM from the beginning, describes the underlying mechanisms involved, and shows how to use COM objects provided by others (specifically, the Windows shell). By the end of the article, you will be able to use the COM objects built-in to Windows and provided by third parties.

This article assumes you are proficient in C++. I use a little bit of MFC and ATL in the sample code, but I will explain the code thoroughly, so you should be able to follow along if you are not familiar with MFC or ATL. The sections in this article are:

COM - What Exactly Is It? - A quick introduction to the COM standard, and the problems it was created to solve. You don't need to know this to use COM, but I'd still recommend reading it to get an understanding of why things are done the way they are in COM.

Definitions of the Basic Elements - COM terminology and descriptions of what those terms represent.

Working with COM Objects - An overview of how to create, use, and destroy COM objects.

The Base Interface - IUnknown - A description of the methods in the base interface, `IUnknown`.

Pay Close Attention - String Handling - How to handle strings in COM code.

Bringing it All Together - Sample Code - Two sets of sample code that illustrate all the concepts discussed in the article.

Handling HRESULTs - A description of the `HRESULT` type and how to test for error and success codes.

References - Books you should expense if your employer will let you. :)

COM - What exactly is it?

COM is, simply put, a method for sharing binary code across different applications and languages. This is unlike the C++ approach, which promotes reuse of source code. ATL is a perfect example of this. While source-level reuse works fine, it only works for C++. It also introduces the possibility of name collisions, not to mention bloat from having multiple copies of the code in your projects.

Windows lets you share code at the binary level using DLLs. After all, that's how Windows apps function - reusing kernel32.dll, user32.dll, etc. But since the DLLs are written to a C interface, they can only be used by C or languages that understand the C calling convention. This puts the burden of sharing on the programming language implementer, instead of on the DLL itself.

MFC introduced another binary sharing mechanism with MFC extension DLLs. But these are even more restrictive - you can only use them from an MFC app.

COM solves all these problems by defining a *binary standard*, meaning that COM specifies that the binary modules (the DLLs and EXEs) must be compiled to match a specific structure. The standard also specifies exactly how COM objects must be organized in memory. The binaries must also not depend on any feature of any programming language (such as name decoration in C++). Once that's done, the modules can be accessed easily from any programming language. A binary standard puts the burden of compatibility on the compiler that produces the binaries, which makes it much easier for the folks who come along later and need to use those binaries.

The structure of COM objects in memory just happens to use the same structure that is used by C++ virtual functions, so that's why a lot of COM code uses C++. But remember, the language that the module is written in is irrelevant, because the resulting binary is usable by all languages.

Incidentally, COM is not Win32-specific. It could, in theory, be ported to Unix or any other OS. However, I have never seen COM mentioned outside of the Windows world.

Definitions of the Basic Elements

Let's go from the bottom up. An *interface* is simply a group of functions. Those functions are called *methods*. Interface names start with *I*, for example `IShellLink`. In C++, an interface is written as an abstract base class that has only pure virtual functions.

Interfaces may *inherit* from other interfaces. Inheritance works just like single inheritance in C++. Multiple inheritance is not allowed with interfaces.

A *coclass* (short for **component object class**) is contained in a DLL or EXE, and contains the code behind one or more interfaces. The coclass is said to *implement* those interfaces. A *COM object* is an instance of a coclass in memory. Note that a COM "class" is not the same as a C++ "class", although it is often the case that the implementation of a COM class is a C++ class.

A *COM server* is a binary (DLL or EXE) that contains one or more coclasses.

Registration is the process of creating registry entries that tell Windows where a COM server is located. *Unregistration* is the opposite - removing those registry entries.

A *GUID* (rhymes with "fluid", stands for **globally unique identifier**) is a 128-bit number. GUIDs are COM's language-independent way of identifying things. Each interface and coclass has a GUID. Since GUIDs are unique throughout the world, name collisions are avoided (as long as you use the COM API to create them). You will also see the term UUID (which stands for **universally unique identifier**) at times. UUIDs and GUIDs are, for all practical purposes, the same.

A *class ID*, or *CLSID*, is a GUID that names a coclass. An *interface ID*, or *IID*, is a GUID that names an interface.

There are two reasons GUIDs are used so extensively in COM:

1. GUIDs are just numbers under the hood, and any programming language can handle them.
2. Every GUID created, by anyone on any machine, is unique when created properly. Therefore, COM developers can create GUIDs on their own with no chance of two developers choosing the same GUID. This eliminates the need for a central authority to issue GUIDs.

An *HRESULT* is an integral type used by COM to return error and success codes. It is not a "handle" to anything, despite the *H* prefix. I'll have more to say about HRESULTs and how to test them later on.

Finally, the *COM library* is the part of the OS that you interact with when doing COM-related stuff. Often, the COM library is referred to as just "COM," but I will not do that here, to avoid confusion.

Working with COM Objects

Every language has its own way of dealing with objects. For example, in C++ you create them on the stack, or use `new` to dynamically allocate them. Since COM must be language-neutral, the COM library provides its own object-management routines. A comparison of COM and C++ object management is listed below:

Creating a new object

- In C++, use `operator new` or create an object on the stack.
- In COM, call an API in the COM library.

Deleting objects

- In C++, use `operator delete` or let a stack object go out of scope.
- In COM, all objects keep their own reference counts. The caller must tell the object when the caller is done using the object. COM objects free themselves from memory when the reference count reaches 0.

Now, in between those two stages of creating and destroying the object, you actually have to *use* it. When you create a COM object, you tell the COM library what interface you need. If the object is created successfully, the COM library returns a pointer to the requested interface. You can then call methods through that pointer, just as if it were a pointer to a regular C++ object.

Creating a COM object

To create a COM object and get an interface from the object, you call the COM library API `CoCreateInstance()`. The prototype for `CoCreateInstance()` is:

```
HRESULT CoCreateInstance (
    REFCLSID rclsid,
    LPUNKNOWN pUnkOuter,
    DWORD dwClsContext,
    REFIID riid,
    LPVOID* ppv );
```

The parameters are:

`rclsid`

The CLSID of the coclass. For example, you can pass `CLSID_ShellLink` to create a COM object used to create shortcuts.

`pUnkOuter`

This is only used when aggregating COM objects, which is a way of taking an existing coclass and adding new methods to it. For our purposes, we can just pass NULL to indicate we're not using aggregation.

`dwClsContext`

Indicates what kind of COM servers we want to use. For this article, we will always be using the simplest kind of server, an in-process DLL, so we'll pass `CLSCTX_INPROC_SERVER`. One caveat: you should not use `CLSCTX_ALL` (which is the default in ATL) because it will fail on Windows 95 systems that do not have DCOM installed.

`riid`

The IID of the interface you want returned. For example, you can pass `IID_IShellLink` to get a pointer to an `IShellLink` interface.

`ppv`

Address of an interface pointer. The COM library returns the requested interface through this parameter.

When you call `CoCreateInstance()`, it handles looking up the CLSID in the registry, reading the location of the server, loading the server into memory, and creating an instance of the coclass you requested.

Here's a sample call, which instantiates a `CLSID_ShellLink` object and requests an `IShellLink` interface pointer to that COM object.

```
HRESULT      hr;
IShellLink* pISL;

hr = CoCreateInstance ( CLSID_ShellLink,          // CLSID of coclass
                       NULL,                    // not used - aggregation
                       CLSCTX_INPROC_SERVER,    // type of server
                       IID_IShellLink,         // IID of interface
                       (void**) &pISL );        // Pointer to our interface pointer

if ( SUCCEEDED ( hr ) )
{
    // Call methods using pISL here.
}
else
{
    // Couldn't create the COM object.  hr holds the error code.
}
```

First we declare an `HRESULT` to hold the return from `CoCreateInstance()` and an `IShellLink` pointer. We call `CoCreateInstance()` to create a new COM object. The `SUCCEEDED` macro returns TRUE if `hr` holds a code indicating success, or FALSE if `hr` indicates failure. There is a corresponding macro `FAILED` that tests for a failure code.

Deleting a COM object

As stated before, you don't free COM objects, you just tell them that you're done using them. The `IUnknown` interface, which every COM object implements, has a method `Release()`. You call this method to tell the COM object that you no longer need it. Once you call `Release()`, you must not use the interface pointer any more, since the COM object may disappear from memory at any time.

If your app uses a lot of different COM objects, it's vitally important to call `Release()` whenever you're done using an interface. If you don't release interfaces, the COM objects (and the DLLs that contain the code) will remain in memory, and will needlessly add to your app's working set. If your app will be running for a long time, you should call the `CoFreeUnusedLibraries()` API during your idle processing. This API unloads any COM servers that have no outstanding references, so this also reduces your app's memory usage.

Continuing the above example, here's how you would use `Release()`:

```
// Create COM object as above.  Then...

if ( SUCCEEDED ( hr ) )
{
    // Call methods using pISL here.

    // Tell the COM object that we're done with it.
    pISL->Release();
}
```

The `IUnknown` interface is explained fully in the next section.

The Base Interface - IUnknown

Every COM interface is derived from `IUnknown`. The name is a bit misleading, in that it's not an unknown interface. The name signifies that if you have an `IUnknown` pointer to a COM object, you don't know what the underlying object is, since every COM object implements `IUnknown`.

`IUnknown` has three methods:

1. `AddRef()` - Tells the COM object to increment its reference count. You would use this method if you made a copy of an interface pointer, and both the original and the copy would still be used. We won't need to use `AddRef()` for our purposes in this article.
2. `Release()` - Tells the COM object to decrement its reference count. See the previous example for a code snippet demonstrating `Release()`.
3. `QueryInterface()` - Requests an interface pointer from a COM object. You use this when a coclass implements more than one interface.

We've already seen `Release()` in action, but what about `QueryInterface()`? When you create a COM object with `CoCreateInstance()`, you get an interface pointer back. If the COM object implements more than one interface (not counting `IUnknown`), you use `QueryInterface()` to get any additional interface pointers that you need. The prototype of `QueryInterface()` is:

```
HRESULT IUnknown::QueryInterface (
    REFIID iid,
    void** ppv );
```

The parameters are:

`iid`

The IID of the interface you're requesting.

`ppv`

Address of an interface pointer. `QueryInterface()` returns the interface through this parameter if it is successful.

Let's continue our shell link example. The coclass for making shell links implements `IShellLink` and `IPersistFile`. If you already have an `IShellLink` pointer, `pISL`, you can request an `IPersistFile` interface from the COM object with code like this:

```
HRESULT hr;
IPersistFile* pIPF;

hr = pISL->QueryInterface ( IID_IPersistFile, (void**) &pIPF );
```

You then test `hr` with the `SUCCEEDED` macro to determine if `QueryInterface()` worked. If it succeeded, you can then use the new interface pointer, `pIPF`, just like any other interface. You must also call `pIPF->Release()` to tell the COM object that you're done using the interface.

Pay Close Attention - String Handling

I need to make a detour for a few moments, and discuss how to handle strings in COM code. If you are familiar with how Unicode and ANSI strings work, and know how to convert between the two, then you can skip this section. Otherwise, read on.

Whenever a COM method returns a string, that string will be in Unicode. (Well, all methods that are written to the COM spec, that is!) Unicode is a character encoding scheme, like ASCII, only all characters are 2 bytes long. If you want to get the string into a more manageable state, you should convert it to a `TCHAR` string.

`TCHAR` and the `_t` functions (for example, `_tcscopy()`) are designed to let you handle Unicode and ANSI strings with the same source code. In most cases, you'll be writing code that uses ANSI strings and the ANSI Windows APIs, so for the rest of this article, I will refer to `char` instead of `TCHARs`, just for simplicity. You should definitely read up on the `TCHAR` types, though, to be aware of them in case you ever come across them in code written by others.

When you get a Unicode string back from a COM method, you can convert it to a `char` string in one of several ways:

1. Call the `WideCharToMultiByte()` API.
2. Call the CRT function `wcstombs()`.
3. Use the `CString` constructor or assignment operator (MFC only).
4. Use an ATL string conversion macro.

WideCharToMultiByte()

You can convert a Unicode string to an ANSI string with the `WideCharToMultiByte()` API. This API's prototype is:

```
int WideCharToMultiByte (
    UINT      CodePage,
    DWORD     dwFlags,
    LPCWSTR  lpWideCharStr,
    int       cchWideChar,
    LPSTR     lpMultiByteStr,
    int       cbMultiByte,
    LPCSTR    lpDefaultChar,
    LPBOOL    lpUsedDefaultChar );
```

The parameters are:

CodePage

The code page to convert the Unicode characters into. You can pass `CP_ACP` to use the current ANSI code page. Code pages are sets of 256 characters. Characters 0-127 are always identical to the ASCII encoding. Characters 128-255 differ, and can contain graphics or letters with diacritics. Each language or region has its own code page, so it's important to use the right code page to get proper display of accented characters.

dwFlags

`dwFlags` determine how Windows deals with "composite" Unicode characters, which are a letter followed by a diacritic. An example of a composite character is `è`. If this character is in the code page specified in `CodePage`, then nothing special happens. However, if it is *not* in the code page, Windows has to convert it to something else.

Passing `WC_COMPOSITECHECK` makes the API check for non-mapping composite characters. Passing `WC_SEPCHARS` makes Windows break the character into two, the letter followed by the diacritic, for example `e``. Passing `WC_DISCARDNS` makes Windows discard the diacritics. Passing `WC_DEFAULTCHAR` makes Windows replace the composite characters with a "default" character, specified in the `lpDefaultChar` parameter. The default behavior is `WC_SEPCHARS`.

lpWideCharStr

The Unicode string to convert.

cchWideChar

The length of `lpWideCharStr` in Unicode characters. You will usually pass -1, which indicates that the string is zero-terminated.

lpMultiByteStr

A `char` buffer that will hold the converted string.

cbMultiByte

The size of `lpMultiByteStr`, in bytes.

lpDefaultChar

Optional - a one-character ANSI string that contains the "default" character to be inserted when `dwFlags` contains `WC_COMPOSITECHECK` | `WC_DEFAULTCHAR` and a Unicode character cannot be mapped to an equivalent ANSI character. You can pass NULL to have the API use a system default character (which as of this writing is a question mark).

lpUsedDefaultChar

Optional - a pointer to a `BOOL` that will be set to indicate if the default char was ever inserted into the ANSI string. You can pass NULL if you don't care about this information.

Whew, a lot of boring details! Like always, the docs make it seem much more complicated than it really is. Here's an example showing how to use the API:

```
// Assuming we already have a Unicode string wszSomeString...
char szANSIString [MAX_PATH];

WideCharToMultiByte ( CP_ACP, // ANSI code page
```

```

        WC_COMPOSITECHECK,    // Check for accented characters
        wszSomeString,       // Source Unicode string
        -1,                  // -1 means string is zero-terminated
        szANSIString,        // Destination char string
        sizeof(szANSIString), // Size of buffer
        NULL,                 // No default character
        NULL );              // Don't care about this flag

```

After this call, `szANSIString` will contain the ANSI version of the Unicode string.

wcstombs()

The CRT function `wcstombs()` is a bit simpler, but it just ends up calling `WideCharToMultiByte()`, so in the end the results are the same. The prototype for `wcstombs()` is:

```

size_t wcstombs (
    char*      mbstr,
    const wchar_t* wcstr,
    size_t     count );

```

The parameters are:

`mbstr`

A `char` buffer to hold the resulting ANSI string.

`wcstr`

The Unicode string to convert.

`count`

The size of the `mbstr` buffer, in bytes.

`wcstombs()` uses the `WC_COMPOSITECHECK` | `WC_SEPCHARS` flags in its call to `WideCharToMultiByte()`. To reuse the earlier example, you can convert a Unicode string with code like this:

```

wcstombs ( szANSIString, wszSomeString, sizeof(szANSIString) );

```

CString

The MFC `CString` class contains constructors and assignment operators that accept Unicode strings, so you can let `CString` do the conversion work for you. For example:

```

// Assuming we already have wszSomeString...

CString str1 ( wszSomeString );    // Convert with a constructor.
CString str2;

str2 = wszSomeString;             // Convert with an assignment operator.

```

ATL macros

ATL has a handy set of macros for converting strings. To convert a Unicode string to ANSI, use the `W2A()` macro (a mnemonic for "wide to ANSI"). Actually, to be more accurate, you should use `OLE2A()`, where the "OLE" indicates the string came from a COM or OLE source. Anyway, here's an example of how to use these macros.

```

#include <atlconv.h>

// Again assuming we have wszSomeString...

{
    char szANSIString [MAX_PATH];
    USES_CONVERSION; // Declare local variable used by the macros.

    lstrcpy ( szANSIString, OLE2A(wszSomeString) );
}

```

The `OLE2A()` macro "returns" a pointer to the converted string, but the converted string is stored in a temporary stack variable, so we need to make our own copy of it with `lstrcpy()`. Other macros you should look into are `W2T()` (Unicode to `TCHAR`), and `W2CT()` (Unicode string to `const TCHAR` string).

There is an `OLE2CA()` macro (Unicode string to a `const char` string) which we could've used in the code snippet above. `OLE2CA()` is actually the correct macro for that situation, since the second parameter to `lstrcpy()` is a `const char*`, but I didn't want to throw too much at you at once.

Sticking with Unicode

On the other hand, you can just keep the string in Unicode if you won't be doing anything complicated with the string. If you're writing a console app, you can print Unicode strings with the `std::wcout` global variable, for example:

```
wcout << wszSomeString;
```

But keep in mind that `wcout` expects all strings to be in Unicode, so if you have any "normal" strings, you'll still need to output them with `std::cout`. If you have string literals, prefix them with `L` to make them Unicode, for example:

```
wcout << L"The Oracle says..." << endl << wszOracleResponse;
```

If you keep a string in Unicode, there are a couple of restrictions:

- You must use the `wcsXXX()` string functions, such as `wcslen()`, on Unicode strings.
- With very few exceptions, you cannot pass a Unicode string to a Windows API on Windows 9x. To write code that will run on 9x and NT unchanged, you'll need to use the `TCHAR` types, as described in MSDN.

Bringing it All Together - Sample Code

Following are two examples that illustrate the COM concepts covered in the article. The code is also contained in the article's sample project.

Using a COM object with a single interface

The first example shows how to use a COM object that exposes a single interface. This is the simplest case you'll ever encounter. The code uses the Active Desktop coclass contained in the shell to retrieve the filename of the current wallpaper. You will need to have the Active Desktop installed for this code to work.

The steps involved are:

1. Initialize the COM library.
2. Create a COM object used to interact with the Active Desktop, and get an `IActiveDesktop` interface.
3. Call the `GetWallpaper()` method of the COM object.
4. If `GetWallpaper()` succeeds, print the filename of the wallpaper.
5. Release the interface.
6. Uninitialize the COM library.

```
WCHAR    wszWallpaper [MAX_PATH];
CString  strPath;
HRESULT  hr;
IActiveDesktop* pIAD;
```

```
// 1. Initialize the COM library (make Windows load the DLLs). Normally you would
// call this in your InitInstance() or other startup code. In MFC apps, use
// AfxOleInit() instead.
CoInitialize ( NULL );
```

```
// 2. Create a COM object, using the Active Desktop coclass provided by the shell.
// The 4th parameter tells COM what interface we want (IActiveDesktop).
hr = CoCreateInstance ( CLSID_ActiveDesktop,
                        NULL,
                        CLSCTX_INPROC_SERVER,
                        IID_IActiveDesktop,
```

```

        (void**) &pIAD );

if ( SUCCEEDED(hr) )
{
    // 3. If the COM object was created, call its GetWallpaper() method.
    hr = pIAD->GetWallpaper ( wszWallpaper, MAX_PATH, 0 );

    if ( SUCCEEDED(hr) )
    {
        // 4. If GetWallpaper() succeeded, print the filename it returned.
        // Note that I'm using wcout to display the Unicode string wszWallpaper.
        // wcout is the Unicode equivalent of cout.
        wcout << L"Wallpaper path is:\n    " << wszWallpaper << endl << endl;
    }
    else
    {
        cout << _T("GetWallpaper() failed.") << endl << endl;
    }

    // 5. Release the interface.
    pIAD->Release();
}
else
{
    cout << _T("CoCreateInstance() failed.") << endl << endl;
}

// 6. Uninit the COM library. In MFC apps, this is not necessary since MFC does
// it for us.
CoUninitialize();

```

In this sample, I used `std::wcout` to display the Unicode string `wszWallpaper`.

Using a COM object with a multiple interfaces

The second example shows how to use `QueryInterface()` with a COM object that exposes a single interface. The code uses the Shell Link coclass contained in the shell to create a shortcut to the wallpaper file that we retrieved in the last example.

The steps involved are:

1. Initialize the COM library.
2. Create a COM object used to create shortcuts, and get an `IShellLink` interface.
3. Call the `SetPath()` method of the `IShellLink` interface.
4. Call `QueryInterface()` on the COM object and get an `IPersistFile` interface.
5. Call the `Save()` method of the `IPersistFile` interface.
6. Release the interfaces.
7. Uninitialize the COM library.

```

CString      sWallpaper = wszWallpaper; // Convert the wallpaper path to ANSI
IShellLink*  pISL;
IPersistFile* pIPF;

// 1. Initialize the COM library (make Windows load the DLLs). Normally you would
// call this in your InitInstance() or other startup code. In MFC apps, use
// AfxOleInit() instead.
CoInitialize ( NULL );

// 2. Create a COM object, using the Shell Link coclass provided by the shell.
// The 4th parameter tells COM what interface we want (IShellLink).
hr = CoCreateInstance ( CLSID_ShellLink,
                       NULL,
                       CLSCTX_INPROC_SERVER,
                       IID_IShellLink,
                       (void**) &pISL );

if ( SUCCEEDED(hr) )
{
    // 3. Set the path of the shortcut's target (the wallpaper file).
    hr = pISL->SetPath ( sWallpaper );

    if ( SUCCEEDED(hr) )
    {

```

```

// 4. Get a second interface (IPersistFile) from the COM object.
hr = pISL->QueryInterface ( IID_IPersistFile, (void**) &pIPF );

if ( SUCCEEDED(hr) )
{
// 5. Call the Save() method to save the shortcut to a file. The
// first parameter is a Unicode string.
hr = pIPF->Save ( L"C:\\wallpaper.lnk", FALSE );

// 6a. Release the IPersistFile interface.
pIPF->Release();
}

// 6b. Release the IShellLink interface.
pISL->Release();
}

// Printing of error messages omitted here.

// 7. Uninit the COM library. In MFC apps, this is not necessary since MFC
// does it for us.
CoUninitialize();

```

Handling HRESULTs

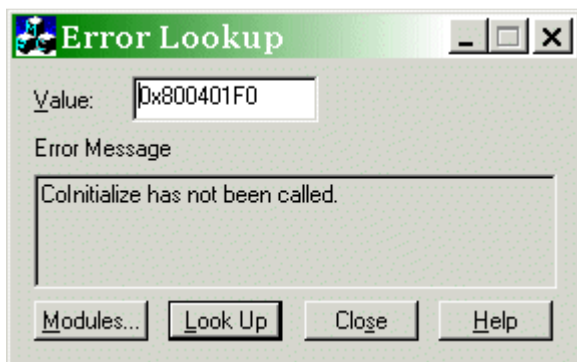
I've already shown some simple error handling, using the `SUCCEEDED` and `FAILED` macros. Now I'll give some more details on what to do with the `HRESULTs` returned from COM methods.

An `HRESULT` is a 32-bit signed integer, with nonnegative values indicating success, and negative values indicating failure. An `HRESULT` has three fields: the severity bit (to indicate success or failure), the facility code, and the status code. The "facility" indicates what component or program the `HRESULT` is coming from. Microsoft assigns facility codes to the various components, for example COM has one, the Task Scheduler has one, and so on. The "code" is a 16-bit field that has no intrinsic meaning; the codes are just an arbitrary association between a number and a meaning, just like the values returned by `GetLastError()`.

If you look up error codes in the `winerror.h` file, you'll see a lot of `HRESULTs` listed, with the naming convention `[facility]_[severity]_[description]`. Generic `HRESULTs` that can be returned by any component (like `E_OUTOFMEMORY`) have no facility in their name. Examples:

- `REGDB_E_READREGDB`: Facility = REGDB, for "registry database"; E = error; READREGDB is a description of the error (couldn't read the database).
- `S_OK`: Facility = generic; S = success; OK is a description of the status (everything's OK).

Fortunately, there are easier ways to determine the meaning of an `HRESULT` than looking through `winerror.h`. `HRESULTs` for built-in facilities can be looked up with the Error Lookup tool. For example, say you forgot to call `CoInitialize()` before `CoCreateInstance()`. `CoCreateInstance()` will return a value of `0x800401F0`. You can enter that value into Error Lookup and you'll see the description: "CoInitialize has not been called."



You can also look up `HRESULT` descriptions in the debugger. If you have an `HRESULT` variable called `hres`, you can view the description in the Watch window by entering "`hres,hr`" as the value to watch. The "`hr`" tells VC to display the value as an `HRESULT` description.

| Name | Value |
|----------|--|
| @ERR, hr | S_OK |
| hres, hr | 0x800401f0 CoInitialize has not been called. |
| | |

References

Essential COM by Don Box, ISBN 0-201-63446-5. Everything you'd ever want to know about the COM spec and IDL (interface definition language). The first two chapters go into great detail about the COM spec and the problems it was designed to solve.

MFC Internals by George Shepherd and Scot Wingo, ISBN 0-201-40721-3. Contains an in-depth look at MFC's COM support.

Beginning ATL 3 COM Programming by Richard Grimes, et al, ISBN 1-861001-20-7. This book goes into depth about about writing your own COM components using ATL.

License

This article has no explicit license attached to it but may contain usage terms in the article text or the download files themselves. If in doubt please contact the author via the discussion board below.

A list of licenses authors might use can be found [here](#)

About the Author

Michael Dunn



Member

Michael lives in sunny Mountain View, California. He started programming with an Apple //e in 4th grade, graduated from [UCLA](#) with a math degree in 1994, and immediately landed a job as a QA engineer at Symantec, working on the Norton AntiVirus team. He pretty much taught himself Windows and MFC programming, and in 1999 he designed and coded a new interface for Norton AntiVirus 2000. Mike has been a a developer at [Napster](#) and at his own lil' startup, [Zabersoft](#), a development company he co-founded with offices in Los Angeles and Odense, Denmark. Mike is now a senior engineer at [VMware](#).

He also enjoys his hobbies of playing pinball, bike riding, photography, and the occasional 360 or MAME game (current favorite: *Space Invaders Extreme*). He would get his own snooker table too if they weren't so darn big! He is also sad that he's forgotten the languages he's studied: French, Mandarin Chinese, and Japanese.

Mike was a [VC MVP](#) from 2005 to 2009.

Occupation: Software Developer (Senior)

Company: VMware

Location: United States

Discussions and Feedback

247 messages have been posted for this article. Visit <http://www.codeproject.com/KB/COM/comintro.aspx> to post and view comments on this article, or click [here](#) to get a print view with messages.

[PermaLink](#) | [Privacy](#) | [Terms of Use](#)
 Last Updated: 27 Jul 2000
 Editor: [Rob Manderson](#)

Copyright 2000 by Michael Dunn
 Everything else Copyright © [CodeProject](#), 1999-2010
 Web22 | [Advertise on the Code Project](#)