**Marshaling Overview**

The .NET Compact Framework version 2.0 provides expanded marshaling support through **IDispatch** and through platform invoke and vtable calls. This support includes the following:

* Using the [MarshalAsAttribute](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.marshalasattribute.aspx) attribute.
* Marshaling variant types that are supported on Windows Embedded CE.
* Marshaling types that call COM interfaces through a vtable.
* Marshaling structures with embedded arrays and strings.
* Specifying the layout for a structure.

You can marshal the following types either by value or by reference:

* **BStr**
* **IUnknown**
* **IDispatch**
* **SafeArray**
* **DateTime** (marshaled as an OLE **DATE**)
* **Variant**

Note that the .NET Compact Framework 2.0 supports the [AllocHGlobal](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.marshal.allochglobal.aspx) and [FreeHGlobal](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.marshal.freehglobal.aspx) methods.

http://i.msdn.microsoft.com/Global/Images/clear.gif Interop Logging

You can create log files of the function signatures to see how an interop call is marshaled, and also to isolate errors that may occur during the marshaling process. For information about how to create the files, see [How to: Create Log Files](http://msdn.microsoft.com/en-us/library/ms229650.aspx). For information about how to interpret the log files, see [Log File Information](http://msdn.microsoft.com/en-us/library/ms229667.aspx).

http://i.msdn.microsoft.com/Global/Images/clear.gif Marshaling Differences with the Full .NET Framework

The .NET Compact Framework does not support the following marshaling and interoperability features that are provided in the full .NET Framework:

* Custom marshaling.
* Obtaining a managed delegate from a native function pointer by using the [GetDelegateForFunctionPointer](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.marshal.getdelegateforfunctionpointer.aspx) method. You can, however, create a native function pointer from a managed delegate.
* Accessing .NET Compact Framework classes from native components.
* Passing structures (VT\_RECORD) through IDispatch.
* Passing Int64 and UInt64 types through IDispatch.

The .NET Compact Framework differs from the full .NET Framework in the following marshaling behaviors:

* The .NET Compact Framework provides limited support for marshaling fields in structures, elements in arrays, and return types compared to the .NET Framework. For information about overriding default marshaling behavior, see [Marshaling Overview](http://msdn.microsoft.com/en-us/library/ms172511.aspx).
* The .NET Compact Framework allows arrays of SCODE values to be marshaled; the full .NET Framework does not.
* The .NET Compact Framework marshals arrays of IUnknown and IDispatch pointers differently than the full .NET Framework.
* The .NET Compact Framework initializes all threads as multithreaded apartments and does not support other threading models or setting an apartment model. Consequently, the .NET Compact Framework does not support the [ApartmentState](http://msdn.microsoft.com/en-us/library/system.threading.thread.apartmentstate.aspx) property or the following methods:
  + [SetApartmentState](http://msdn.microsoft.com/en-us/library/system.threading.thread.setapartmentstate.aspx)
  + [GetApartmentState](http://msdn.microsoft.com/en-us/library/system.threading.thread.getapartmentstate.aspx)
  + [TrySetApartmentState](http://msdn.microsoft.com/en-us/library/system.threading.thread.trysetapartmentstate.aspx)

http://i.msdn.microsoft.com/Global/Images/clear.gif Marshaling with the Visual Basic Declare Statement

The Visual Basic **Declare** statement is an alternative to **declare** references to external procedures in a DLL. Note that **Ansi** keyword in the Declare statement is not supported.

Marshaling with the **Declare** statement is identical to marshaling with the [DllImportAttribute](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.dllimportattribute.aspx) class, except for **ByVal String** objects. In a **Declare** statement, a **ByVal String** parameter will be marshaled as an output parameter. Because strings are immutable, this forces the common language runtime to copy the string and return a new reference.

http://i.msdn.microsoft.com/Global/Images/clear.gif Differences Between IDispatch and Platform Invoke Marshalers

The following table lists the types that are marshaled differently by the two marshalers.

|  |  |  |
| --- | --- | --- |
| **Type** | **IDispatch** | **Platform invoke and vtable** |
| [String](http://msdn.microsoft.com/en-us/library/system.string.aspx) | **BStr** | wchar\* |
| [Object](http://msdn.microsoft.com/en-us/library/system.object.aspx) | **Variant** | NULL |
| [Boolean](http://msdn.microsoft.com/en-us/library/system.boolean.aspx) | VARIANT\_BOOL | byte |
| [Array](http://msdn.microsoft.com/en-us/library/system.array.aspx) | **SafeArray** | C-style array |

The .NET Compact Framework marshals a class through platform invoke without the [StructLayoutAttribute](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.structlayoutattribute.aspx) as an auto-layout structure; the full .NET Framework marshals it as a COM callable wrapper (CCW).

Note that the .NET Compact Framework marks a **SafeArray** with FADF\_FIXEDSIZE and throws an exception if you resize it in native code.

In situations where [Boolean](http://msdn.microsoft.com/en-us/library/system.boolean.aspx) translates to a native byte type, you cannot marshal [Boolean](http://msdn.microsoft.com/en-us/library/system.boolean.aspx) as a return type; you can marshal it only as an argument.

http://i.msdn.microsoft.com/Global/Images/clear.gif Marshaling Delegates

By default, delegates are marshaled as function pointers. You can also explicitly use the [FunctionPtr](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.unmanagedtype.functionptr.aspx) value from the [UnmanagedType](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.unmanagedtype.aspx) enumeration for creating an instance of the [MarshalAsAttribute](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.marshalasattribute.aspx). See [Marshaling Delegates as Function Pointers](http://msdn.microsoft.com/en-us/library/ms172513.aspx) for examples.

http://i.msdn.microsoft.com/Global/Images/clear.gif Specifying a Character Set

You can use the [CharSet](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.dllimportattribute.charset.aspx) field of the [DllImportAttribute](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.dllimportattribute.aspx) to specify a character set when marshaling strings through platform invoke.

The .NET Compact Framework supports the following two values:

* [Auto](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.charset.auto.aspx). Strings are marshaled by using the appropriate character set for the operating system, which is the Unicode character set. This is the default value.
* [Unicode](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.charset.unicode.aspx). Strings are marshaled by using the Unicode character set.

The [Ansi](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.charset.ansi.aspx) value is not supported because Windows Embedded CE is Unicode only. [None](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.charset.none.aspx) is equivalent to [Ansi](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.charset.ansi.aspx) and is not supported.

Because the .NET Compact Framework does not support the [ExactSpelling](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.dllimportattribute.exactspelling.aspx) field, the common language runtime automatically searches for an entry point according to the values specified by [CharSet](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.dllimportattribute.charset.aspx).

http://i.msdn.microsoft.com/Global/Images/clear.gif Object Pinning

When the .NET Compact Framework common language runtime marshals an object, the object is pinned for the duration of the platform invoke call to ensure that the garbage collector does not free or move the object.

http://i.msdn.microsoft.com/Global/Images/clear.gif Memory Usage

Use the following guidelines for handling memory with unmanaged code in the .NET Compact Framework:

* Always allocate memory in managed code and pass it to unmanaged code.
* If unmanaged code holds a pointer to a managed component, you must manually pin the object by using the [GCHandle](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.gchandle.aspx) structure.

The .NET Compact Framework common language runtime coinitializes threads at startup and couninitializes them on shutdown. Threads are marked as "free threading".

**Marshaling**

The COM technique of marshaling allows interfaces exposed by an object in one process to be used in another process. In marshaling, COM provides code (or uses code provided by the interface implementor) both to pack a method's parameters into a format that can be moved across processes (as well as, across the wire to processes running on other machines) and to unpack those parameters at the other end. Likewise, COM must perform these same steps on the return from the call.

|  |
| --- |
| **9a0b162x.alert_note(en-us,VS.90).gifNote:** |
| Marshaling is typically not necessary when an interface provided by an object is being used in the same process as the object. However, marshaling may be needed between threads. |

**Introduction to COM**

COM is the fundamental "object model" on which ActiveX Controls and OLE are built. COM allows an object to expose its functionality to other components and to host applications. It defines both how the object exposes itself and how this exposure works across processes and across networks. COM also defines the object's life cycle.

Fundamental to COM are these concepts:

* [Interfaces](http://msdn.microsoft.com/en-us/library/70h1cyaz.aspx) — the mechanism through which an object exposes its functionality.
* [IUnknown](http://msdn.microsoft.com/en-us/library/75ky7wat.aspx) — the basic interface on which all others are based. It implements the reference counting and interface querying mechanisms running through COM.
* [Reference counting](http://msdn.microsoft.com/en-us/library/4947zb56.aspx) — the technique by which an object (or, strictly, an interface) decides when it is no longer being used and is therefore free to remove itself.
* [QueryInterface](http://msdn.microsoft.com/en-us/library/78xw8kzk.aspx) — the method used to query an object for a given interface.
* [Marshaling](http://msdn.microsoft.com/en-us/library/9a0b162x.aspx) — the mechanism that enables objects to be used across thread, process, and network boundaries, allowing for location independence.
* [Aggregation](http://msdn.microsoft.com/en-us/library/ht2b2bfy.aspx) — a way in which one object can make use of another.

**The Component Object Model**

The Microsoft Component Object Model (COM) is a platform-independent, distributed, object-oriented system for creating binary software components that can interact. COM is the foundation technology for Microsoft's OLE (compound documents), ActiveX® (Internet-enabled components), as well as others.

To understand COM (and therefore all COM-based technologies), it is crucial to understand that it is not an object-oriented language but a standard. Nor does COM specify how an application should be structured; language, structure, and implementation details are left to the application programmer. Rather, COM specifies an object model and programming requirements that enable COM objects (also called COM components, or sometimes simply *objects*) to interact with other objects. These objects can be within a single process, in other processes, and can even be on remote machines. They can have been written in other languages, and they may be structurally quite dissimilar, which is why COM is referred to as a *binary standard*—a standard that applies after a program has been translated to binary machine code.

The only language requirement for COM is that code is generated in a language that can create structures of pointers and, either explicitly or implicitly, call functions through pointers. Object-oriented languages such as Microsoft® Visual C++® and Smalltalk provide programming mechanisms that simplify the implementation of COM objects, but languages such as C, Pascal, Ada, Java, and even BASIC programming environments can create and use COM objects.

COM defines the essential nature of a COM object. In general, a software object is made up of a set of data and the functions that manipulate the data. A COM object is one in which access to an object's data is achieved exclusively through one or more sets of related functions. These function sets are called *interfaces,* and the functions of an interface are called *methods.* Further, COM requires that the only way to gain access to the methods of an interface is through a pointer to the interface.

Besides specifying the basic binary object standard, COM defines certain basic interfaces that provide functions common to all COM-based technologies, and it provides a small number of API functions that all components require. COM also defines how objects work together over a distributed environment and has added security features to help provide system and component integrity.

The following topics in this section describe basic COM issues related to designing COM objects:

* [COM Objects and Interfaces](http://msdn.microsoft.com/en-us/library/ms690343(VS.85).aspx)
* [Using and Implementing IUnknown](http://msdn.microsoft.com/en-us/library/ms693423(VS.85).aspx)
* [Reusing Objects](http://msdn.microsoft.com/en-us/library/ms678443(VS.85).aspx)
* [The COM Library](http://msdn.microsoft.com/en-us/library/ms682442(VS.85).aspx)
* [Managing Memory Allocation](http://msdn.microsoft.com/en-us/library/ms688453(VS.85).aspx)

**Introduction to ATL**

ATL is the Active Template Library, a set of template-based C++ classes with which you can easily create small, fast Component Object Model (COM) objects. It has special support for key COM features including: stock implementations of [IUnknown](http://msdn.microsoft.com/en-us/library/ms680509.aspx), [IClassFactory](http://msdn.microsoft.com/en-us/library/ms694364.aspx), [IClassFactory2](http://msdn.microsoft.com/en-us/library/ms692720.aspx), and **IDispatch**; dual interfaces; standard COM enumerator interfaces; connection points; tear-off interfaces; and ActiveX controls. For more information, see [Implementing the IDispatch Interface](http://msdn.microsoft.com/en-us/library/ms221037.aspx).

ATL code can be used to create single-threaded objects, apartment-model objects, free-threaded model objects, or both free-threaded and apartment-model objects.

**Marshaling Details**

If you use standard marshaling, COM handles all of the details described in this section for you. This section is provided for those few programmers who need these details and for those interested in the underlying information. Marshaling is the process of packaging and unpackaging parameters so a remote procedure call can take place.

Different parameter types are marshaled in different ways. For example, marshaling an integer parameter involves simply copying the value into the message buffer. (Although even in this simple case, there are issues such as byte ordering to deal with in cross-machine calls.) Marshaling an array, however, is a more complex process. Array members are copied in a specific order so that the other side can reconstruct the array exactly. When a pointer is marshaled, the data that the pointer is pointing to is copied following rules and conventions for dealing with nested pointers in structures. Unique functions exist to handle the marshaling of each parameter type.

With standard marshaling, the proxies and stubs are systemwide resources for the interface and they interact with the channel through a standard protocol. Standard marshaling can be used both by standard COM-defined interfaces and by custom interfaces, as follows:

* In the case of most COM interfaces, the proxies and stubs for standard marshaling are in-process component objects which are loaded from a systemwide DLL provided by COM in ole32.dll.
* In the case of custom interfaces, the proxies and stubs for standard marshaling are generated by the interface designer, typically with MIDL. These proxies and stubs are statically configured in the registry, so any potential client can use the custom interface across process boundaries. These proxies and stubs are loaded from a DLL that is located via the system registry, using the interface ID (IID) for the custom interface they marshal.
* An alternative to using MIDL to generate proxies and stubs for custom interfaces, a type library can be generated instead and the system provided, type-library–driven marshaling engine will marshal the interface.

As an alternative to standard marshaling, an interface (standard or custom) can use custom marshaling. With custom marshaling, an object dynamically implements the proxies at run time for each interface that it supports. For any given interface, the object can select COM-provided standard marshaling or custom marshaling. This choice is made by the object on an interface-by-interface basis. Once the choice is made for a given interface, it remains in effect during the object's lifetime. However, one interface on an object can use custom marshaling while another uses standard marshaling.

Custom marshaling is inherently unique to the object that implements it. It uses proxies implemented by the object and provided to the system on request at run time. Objects that implement custom marshaling must implement the [IMarshal](http://msdn.microsoft.com/en-us/library/ms688712(VS.85).aspx) interface, whereas objects that support standard marshaling do not.

If you decide to write a custom interface, you must provide marshaling support for it. Typically, you will provide a standard marshaling DLL for the interface you design. You can use the tools contained in the Platform SDK CD to create the proxy/stub code and the proxy/stub DLL. Alternatively, you can use these tools to create a type library which COM will use to do data-driven marshaling (using the data in the type library).

For a client to make a call to an interface method in an object in another process involves the cooperation of several components. The standard proxy is a piece of interface-specific code that resides in the client's process space and prepares the interface parameters for transmittal. It packages, or marshals, them in such a way that they can be re-created and understood in the receiving process. The standard stub, also a piece of interface-specific code, resides in the server's process space and reverses the work of the proxy. The stub unpackages, or unmarshals, the sent parameters and forwards them to the object application. It also packages reply information to send back to the client.

|  |
| --- |
| **http://i.msdn.microsoft.com/ms692621.note(en-us,VS.85).gifNote:** |
| Readers more familiar with RPC than COM may be used to seeing the terms client stub and server stub. These terms are analogous to proxy and stub. |

**Components of Interprocess communications**

The following diagram shows the flow of communication between the components involved. On the client side of the process boundary, the client's method call goes through the proxy and then onto the channel, which is part of the COM library. The channel sends the buffer containing the marshaled parameters to the RPC run-time library, which transmits it across the process boundary. The RPC run time and the COM libraries exist on both sides of the process. The distinction between the channel and the RPC run time is a characteristic of this implementation and is not part of the programming model or the conceptual model for COM client/server objects. COM servers see only the proxy or stub and, indirectly, the channel. Future implementations may use different layers below the channel or no layers.

