**Common Language Runtime**

# Metadata Unmanaged API

This document specifies the API for emitting and importing metadata for the Common Language Runtime (CLR). This API is unmanaged and intended for use by compilers and loaders – low-level tools that require fast access to metadata with a minimum of assistance for traversing relationships (such as the class hierarchy) or for manipulating collections (such as members on a class)

Browsers and other tools, seeking a higher-level API, may instead use the managed Reflection interfaces

***This is preliminary documentation and subject to change***

Last revised: 2 August 2001

# 1 Overview of the Metadata API

This document defines a set of APIs for *emitting* and *importing* metadata. It explains what metadata is, and how it is used. It describes all of the data structures that are passed through this API: bitmasks, signatures, custom attributes and marshalling specifiers.

Metadata is used to describe, on the one hand, runtime types (classes, interfaces and valuetypes), fields and methods, and, on the other hand, internal implementation and layout information that is used by the runtime to JIT-compile MSIL, load classes, execute code, and interoperate with the COM classic or native world. This information is included with every CLR component, and is available to the runtime, tools, and services.

Compilers and tools emit metadata by calling the *emit* APIs during compilation and link or, with RAD tools, as a part of building components or applications. The APIs write-to and read-from in-memory data structures. At save time, these in-memory structures are compressed and persisted in binary format into the target compilation unit (.obj file), executable file, or stand-alone metadata binary file. When multiple compilation units are linked to form an .EXE or .DLL, the emit APIs provide a method used to merge the metadata sections from each compilation unit into a single integrated metadata binary.

The loader and other runtime tools and services *import* metadata to obtain information about components so that tasks such as loading and activation can be completed.

All manipulation of metadata is performed through the metadata APIs, insulating tools from the underlying data structures and enabling a pluggable persistence format architecture that allows runtime binary representations, COM classic type libraries, and other formats to be imported into or from memory transparently.

To learn more about the Runtime file format in general, of which the metadata binary is a part, see the "PE File Format Extensions" spec. For a description of the Runtime type model, refer to the "Virtual Object System" spec. To learn more about interoperability with COM, refer to the "COM integration" spec. To learn more about interoperability with native platform APIs, refer to the "Platform Invoke Metadata Guide". To learn more about Assemblies, and their metadata APIs, see "Assembly Metadata API" spec.

In order to emit and import metadata at the low-level described in this spec, you need to know two things:

- Each method, its arguments and return type – the API. That's what this document describes
- Any data structures you must supply as arguments. There are four: bitmasks, signatures, custom attributes and marshalling descriptors. This information is described later in this spec.

## 1.1 Metadata APIs

At any time you might have several distinct areas of in-memory metadata. For example, you may have one area that maps all of the metadata from an existing module, held in a file on-disk. At the same time, you may be emitting metadata into a distinct area of metadata, that you will afterwards save as a module into a new on-disk file. (We use the word "module" to mean a file that contains metadata; typically it will be a .OBJ, .EXE or .DLL file that also contains MSIL code; but it can also be a file containing only metadata)

We call each separate area of metadata a *scope*. Each scope corresponds to a *module*. Usually that module has been saved, or will be saved, to an on-disk file. But there's no need to do so: scripting tools frequently generate in-memory metadata that is never persisted into a file. We use the term *scope* because it represents the scope within which metadata tokens are defined. That's to say, a metadata token with value N completely identifies an in-memory structure (for example, holding details of a class definition) within a given scope. But that same value N may correspond to a completely different in-memory structure for a different scope.

To establish an in-memory metadata scope, use **CoCreateInstance** for *IMetadataDispenserEx* to create a new scope or to open an existing set of metadata data structures from a file or memory location. With each *Define* or *Open*, the caller specifies which API to receive: The *emit* API interface, used to write to a metadata scope, is *IMetadataEmit*. The *import* API, which allows tools to read from a metadata scope, is *IMetadataImport*.

The metadata APIs described in this specification allow a component's metadata to be accessed without the class being loaded by the runtime. The primary design goals for this API include maximizing performance and minimizing overhead – the metadata engine stops just short of providing direct access to the in-memory data structures. On the other hand, when a class is loaded at runtime, the loader imports the metadata into its own data structures, which can be browsed via the Runtime *Reflection* services. The Reflection services do much more work for the client than the metadata APIs do, such as automatically walking the inheritance hierarchy to obtain information about inherited methods and fields; the metadata APIs return only the direct member declarations for a given class and expect the API client to make additional calls to walk the hierarchy and enumerate inherited methods. The former approach exposes a higher-level view of metadata, where the latter approach puts the API client in complete control of walking the data structures.

Consistent with the primary design goals, the metadata APIs perform a minimum of semantic error checking. These methods assume that the tools and services that emit metadata are enforcing the object system rules outlined in the common type system and that any additional checking on the part of the metadata engine during development time is superfluous. Specific comments about what checks are being performed accompany the specification of each method in this document.

## 1.2 Metadata Abstractions

Metadata stores declarative information about runtime types (classes, value types, and interfaces), global-functions and global-variable. Each such abstraction in a given metadata scope carries an identity as an **mdToken** (metadata token), where an **mdToken** is used by the metadata engine to index into a specific metadata data table in that scope. The metadata APIs return a token from each *Define* method and it is this token that, when passed into the appropriate *Get* method, is used to obtain its associated attributes. Note that an **mdToken** is not an immutable metadata object identifier: when two scopes are merged, tokens from the import scope are remapped into tokens in the emit scope. When a metadata scope is saved, there are various format optimizations that can result in token remaps. Managing tokens is discussed further in the next section.

To be more concrete: a metadata token is a 4-byte value. The most-significant byte specifies what type of token this is. For example, a value of 1 means it's a TypeDef token, whilst a value of 4 means it's a FieldDef token. (For the full list, with their values, see the CorTokenType enumeration in CorHdr.h) The lower 3 bytes give the index of the row, within a MetaData table, that the token refers to. We call those lower 3 bytes the RID, or Record IDentifier. So, for example, the metadata token with value 0x01000007 is a 'shorthand' way to refer to row number 7 in the TypeDef table, in the current scope. Similarly, token 0x0400001A refers to row number 26 (decimal) in the FieldDef table in the current scope. We never store anything in row zero of a metadata table. So a metadata token, whose RID is zero, we call a "nil" token. The metadata API defines a host of such nil tokens – one for each token type (for example, mdTypeDefNil, with value 0x01000000).

[The above explanation of RIDs is conceptually correct – however, in reality, the physical layout of data is much more complicated. Moreover, string tokens mdString are slightly different: their lower 3 bytes are not a record identifier, but an offset to their start location in the metadata string pool]

The following abstractions and corresponding **mdToken types** will be encountered in the metadata APIs. More details on these abstractions are provided in the externalization section of the common type system and, to some extent, with the appropriate *Define* method in this API specification.

- Module (**mdModule**): The metadata in a given scope describes a compilation unit, executable, or other development-, deployment-, or run-time unit, referred to in this documentation generally as a *module*. It is possible, although not required, to declare a name, GUID identifier, custom attributes, etc on the module as a whole.
- Module references (**mdModuleRef**): Compile-time references to modules, recording the source for type and member imports.
- Type declarations (**mdTypeDef**): Declarations of runtime reference types -- classes and interfaces – and of value types.
- Type references (**mdTypeRef**): References to runtime reference types and value types, such as may occur when declaring variables as runtime reference or value types or in declaring inheritance or implementation hierarchies. In a very real sense, the collection of type references in a module is the collection of compile-time import dependencies.
- Method definitions (**mdMethodDef**): Definitions of methods as members of classes or interfaces or as global module-level methods.
- Parameter declarations (**mdParamDef**): The signature of a method (mdMethodDef) includes the number and types of each of the method parameters. Therefore, it is not necessary to emit a parameter declaration data structure for each parameter. However, when there is additional metadata to persist for the parameter, such as marshaling or type mapping information, an optional parameter data structure may be created, identified by an mdParamDef token.
- Field declarations (**mdFieldDef**): Declarations of data members as members of classes or interfaces or as global module-level data members.
- Property declarations (**mdProperty**): Declarations of properties as members of classes or interfaces.

- Event declarations (**mdEvent**): Declarations of named events as members of classes or interfaces.
- Member references (**mdMemberRef**): References to methods and fields. A member reference is generated in metadata for every method invocation or field access that is made by any implementation in this module and a token is persisted in the MSIL stream. (Note that there is no runtime support for property or event references)
- Interface implementations (**mdIfaceImpl**): Information about a specific class's implementation of a specific interface. This metadata abstraction allows information to be persisted about the intersection that is neither specific to the class nor to the interface.
- Method implementations (**mdMethodImpl**): Information about a specific class's implementation of a method inherited via interface inheritance. This metadata abstraction allows information to be persisted that is specific to the implementation rather than to the contract; method declaration information cannot be modified by the implementing class.
- Custom attributes (**mdCustomAttribute**): Arbitrary data structures associated with any metadata object that can be referenced with an **mdToken** (except that custom attributes themselves cannot have custom attributes).
- Permission set (**mdPermission**): A declarative security permission set associated with any one of: mdTypeDef, mdMethodDef and mdAssembly. For further information, see the specification called "Declarative Security Support"
- Type constructor (**mdTypeSpec**): An mdTypeSpec token is used to obtain a token for a type (e.g., a boxed value type) that can be used as input to any MSIL instruction that takes a type. Refer to the Signature specification for details.
- Signature (**mdSignature**): An mdSignature token is only needed when passing a full method signature to an MSIL instruction (e.g., calli) or to encode local variable signatures used in the PE file. These are referred to as "stand-alone signatures". Otherwise, the binary signature encoding associated with declarations of methods, fields, properties, or references to any of these, is supplied directly and the metadata manages the associated blob heap transparently.
- User string (**mdString**). Like mdSignature, an mdString token is only needed when passing a string to an MSIL instruction (e.g., ldstr). Otherwise, the metadata APIs handle all strings (and the associated blob heap) transparently.

Note that there are not two separate token types **mdFieldRef** and **mdMethodRef**, in the above list, as you might have expected. That's because field and method references are share the same table, and we have only the single, generic token type **mdMemberRef**. Nonetheless, for purposes of clarity, this spec will talk about mdFieldRef and mdMethodRef tokens as, invented, species of mdMemberRef tokens.

Runtime metadata is extensible. There are three scenarios where this is important:

- *The Common Language Subset* (CLS) is a specification for conventions that languages and tools agree to support in a uniform way for better language integration. The CLS may constrain parts of the common type system model, and the CLS may introduce higher-level abstractions that are layered over the common type system. It is important that the metadata be able to capture these sorts of development-time abstractions that are used by tools even though they are not recognized or supported explicitly by the runtime.

- It should be possible to represent language-specific abstractions in metadata that are neither common type system nor CLS language abstractions. For example, it should be possible, over time, to enable languages like VC to not require separate header files or IDL files in order to use types, methods, and data members exported by compiled modules.
- It should be possible to encode in member signatures types and type modifiers that are used in language-specific overloading.

This extensibility comes in the following forms:

- Every metadata object can carry custom attributes, and the metadata APIs provide a way to declare, enumerate, and retrieve custom attributes. Custom attributes may be identified by a type reference (mdTypeDef/Ref), where the structure of the attribute is self-describing (via data members declared on the type) and the value encoding may be browsed by any tool including the runtime Reflection services.
- In addition to common type system extensibility, it is possible to emit custom modifiers into member signatures. Runtime will honor these modifiers for purposes of method overloading and hiding, as well as for binding, but will not enforce any of the language-specific semantics.

## 1.3 Using the APIs and Metadata Tokens

The metadata APIs can be called from C++. The two header files that define the public APIs and all necessary enums and constants, are **CorHdr.h** and **Cor.h**. The way the metadata APIs are used will depend in part on the kind of client using them. We can think of clients as falling into one of two general categories:

- Compilers, like VC, that build interim .obj files and then, in a separate linker phase, merge the individual compilation units into a single target PE file
- RAD tools, that manage all code and data structures in the tool environment until build time, at which time they build and emit a PE file in a single step

### 1.3.1 The Complile/Link Style of Interaction

In the compile/link style of interaction, a compiler front end will use the IMetaDataDispenserEx API to establish an in-memory metadata scope and then use the IMetaDataEmit API to declare types and members, working with the metadata abstractions described in the previous section. However, the front end will not be able to supply method implementation information (e.g., whether the implementation is managed or unmanaged, MSIL or native code) or RVA information because it is not known at this time. Instead, the backend and/or linker will need to be able to supply this information *later*, as the actual code is compiled and emitted into the PE file.

The complexity here is that the tool needs to be able to obtain information about the target “save size” of the metadata binary in order to leave room for it in the PE file, but it is not ready to save it into the file until the method (and module-level static data member) RVAs are known and emitted into metadata. In order to calculate the target save size correctly, the metadata engine must first perform any pre-save optimizations, since these optimizations, ideally, make the target binary smaller. Such optimizations might include sorting data structures for faster searching, or optimizing away (early binding) mdTypeRefs and mdMemberRefs when the reference is to a type or member that is declared in the current scope. These sorts of optimizations may result in remapping metadata tokens that the tool is going to expect to be able to use again to emit the implementation and/or RVA information. This means that the tool and the metadata engine must work together to track token remaps.

The sequence of calls for persisting metadata during compilation, then, is:

> **IMetaDataEmit::SetHandler**, to supply an IUnknown interface that the metadata engine can use to query for IID_IMapToken to notify the client of token remaps. SetHandler may be called at any point after the metadata scope is created, but certainly before a call to GetSaveSize.
>
> **IMetaDataEmit::GetSaveSize**, to obtain the save size of the metadata binary. GetSaveSize uses the IMapToken interface supplied in SetHandler to notify the client of any token remaps. Note that if SetHandler was not used to supply an IMapToken interface, no optimizations are performed. This enables a compiler that is emitting an interim .obj file to skip unneeded optimizations that are likely to have to be redone after the link and Merge phase, anyway (see below).
>
> **IMetaDataEmit::Save**, to persist the metadata binary, after SetRVA and other IMetaDataEmit methods are used, as needed, to emit the final implementation metadata.

The next level of complication comes in the linker phase, when multiple compilation units are to be merged into a single integrated PE file. In this case, not only do the metadata scopes need to be merged, but the RVAs will change again as the new PE file is emitted. In the merge phase, the IMetaDataEmit::Merge method, working with a single import and a single emit scope with each call, remaps metadata tokens from the import scope into the emit scope. In addition, the merge may encounter continuable errors that it needs to be able to notify the client of. After the merge is complete, emitting the final PE file involves a call to IMetaDataEmit::GetSaveSize, and another round of token remapping.

The sequence of calls for emitting and persisting metadata by the linker is:

> **IMetaDataEmit::SetHandler**, to supply an IUnknown interface that the metadata engine can use to query for not only IID_IMapToken, as above, but also for IID_IMetaDataError. The latter interface is used to notify the client of any continuable errors that arise from Merge.
>
> **IMetaDataEmit::Merge**, to merge a specified metadata scope into the current emit scope. Merge uses the IMapToken interface to notify the client of token remaps and it uses IMetaDataError to notify the client of continuable errors.
>
> **IMetaDataEmit::GetSaveSize**, to obtain the target save size of the metadata binary. GetSaveSize uses the IMapToken interface supplied in SetHandler to notify the client of any token remaps. Observe that a tool must be prepared to handle token remaps in Merge and then **again** in GetSaveSize after various format optimizations are performed. The last notification for a token is the one that is the final mapping that the tool should rely on.
>
> **IMetaDataEmit::Save**, to persist the metadata binary, after SetRVA and other IMetaDataEmit methods are used, as needed, to emit the final implementation metadata.

### 1.3.2 The RAD Tool Style of Interaction

As in the compile/link style of interaction, a RAD tool will use the IMetaDataDispenserEx API to establish an in-memory metadata scope and then use the IMetaDataEmit API to declare types and members, working with the metadata abstractions described in the previous section. In contrast to the compile/link style, the RAD tool will typically emit the PE file in a single step. It will likely emit declaration and implementation information in a single pass. And, it will probably never need to call Merge. As such, the only reason it might have any need to handle the complexity of token remaps is if it wants to take advantage of the pre-save optimizations that are currently performed in GetSaveSize. Strictly speaking, though, a tool that understands how to emit the metadata in a fully-optimized fashion to start with doesn't need the metadata engine to emit a reasonably optimized file. Although it's a little dangerous, because future implementations of the metadata engine and file format might obsolete some optimizations and introduce others, there is a clear set of rules for how to emit optimized metadata (see Emitting Optimized Metadata Data Structures).

This means that, after emitting the metadata declarations and implementation information, the sequence of calls is simply:

> **IMetaDataEmit::Save**, to persist the metadata binary, after SetRVA and other IMetaDataEmit methods are used, as needed, to emit the final implementation metadata.

*In the general case, there are probably styles of interaction that lie between these two. Some tools may want the metadata engine to own optimizations but may not be interested in token remap information. Or, they may want remap information only for some token types and not others. In truth, a compiler may not even be interested in performing optimizations when emitting an .obj. In future milestones, we are looking at a degree of tuning that is client-specified that offers a range of balance between complexity and optimization.*

### 1.3.3 IMapToken

Any client that implements IMapToken must implement the following method(s):

```
Map (ULONG tkImp, ULONG tkEmit);
```

where *tkImp* is the original token (as known to the client) and *tkEmit* is the new token for that metadata object. When the token remap occurs during Merge, the original token is scoped in the import (source) metadata scope and the new token is scoped in the emit (target) metadata scope.

### 1.3.4 IMetaDataError

Any client that implements IMetaDataError must implement the following method(s):

```
OnError (HRESULT hr, mdToken token);
```

where *hr* is the recoverable error that occurred and *token* is the identity of the metadata token that was being merged in when the error occurred.

## 1.4 Related Specifications

The following related specifications are augmented, implemented, or enforced by several of the methods defined in this document:

- *Reflection* and *ReflectionEmit* interfaces, which are managed versions of these unmanaged interfaces
- Extensions to the PE File format, of which binary metadata is a part. These extensions are included in the "ECMA Partition II : Metadata" specification.
- The Common Type System, which defines the object model that underlies the comman language runtime (CLR), its externalization in metadata, and its implications for the runtime. This information is now included in the "ECMA Partition I : Architecture" specification.
- The "*Common Language Subset"*, which places a number of modeling restrictions on the metadata. The metadata design accommodates but does not explicitly enforce CLS rules. This information is now included in the "ECMA Partition I : Architecture" specification.
- "COM Integration" and "Platform Invoke", which describe requirements for metadata to control how Runtime method invocations and field accesses are mapped onto underlying legacy services. Further details can be found in the many documents listed under "Interop Specifications"

## 1.5 Coding Conventions

The following coding conventions are used by the metadata API.

### 1.5.1 Handling String Parameters

The metadata API exposes all strings as UNICODE (the on-disk format for symbol names is actually UTF8, but that is hidden from clients of the API).

Symbol Names

- String parameters that are symbol names are always assumed to be null-terminated, and no [in] length parameter is needed. Embedded nulls are not supported.
- If an [in] parameter string is too large to persist without truncation, an error will be returned.
- Every returned string is a triple of three parameters (actual param names vary): [in] ULONG cchString, [out]LPCWSTR wzString, [out] ULONG *pchString – where cchString is the count of characters allocated in the buffer including the terminating null, wzString is a pointer to the string buffer returned, and pchString returns the size of the persisted string (including the terminating null) in the event that the buffer did not allocate sufficient size to return the full string. If the returned string was truncated, an error indication will be returned and the client can reallocate the buffer and retry if desired.

User Strings

- User strings may have embedded nulls and should not have a null terminator.

- A length must be supplied with the [in] string parameter. The length supplied is exactly the length that will be stored. If the string ends in a null, it is interpreted to be part of the string value. If the string is null terminated, the length should not include the terminating null.

### 1.5.2 Optional Return Parameters

Many methods in the metadata API that return information, have optional *out* parameters – in the summary table for that method, in the "Required?" column, their entry says "no". This is common with returned strings, but occurs for other types of parameter too. If you want that information returned from the call, provide a non-null pointer value for that argument. If, on the other hand, you are not interested in that information, simply supply a null pointer, and the method will skip over.

### 1.5.3 Storing Default Values

Constants can be stored into metadata as default values for Fields, Parameters and Properties. The constant is specified using 3 parameters called:

- dwDefType – specifies the type of the constant value (for example, ELEMENT_TYPE_UI2)
- pValue – a void* pointer to a blob giving the actual default value. (For example, a pointer to the 4-byte DWORD holding 0x0000002A will store a DWORD value of 42 decimal into the metadata)
- cchValue – count of the (Unicode) characters in the sequence pointed-to by pValue. This is only required if dwDefType = ELEMENT_TYPE_STRING – in all other cases, the length is inferred from the ELEMENT_TYPE_, obviously.

Note that such default values are **not** automatically inserted into initialization code, or into statically-initialized data areas – they are merely recorded into metadata.

The type provided as a default value, via the dwDefType, is limited to being a primitive, a string, or null. Specifically:

| | |
|---|---|
| ELEMENT_TYPE_BOOLEAN | ELEMENT_TYPE_WCHAR |
| ELEMENT_TYPE_I1 | ELEMENT_TYPE_U1 |
| ELEMENT_TYPE_I2 | ELEMENT_TYPE_U2 |
| ELEMENT_TYPE_I4 | ELEMENT_TYPE_U4 |
| ELEMENT_TYPE_I8 | ELEMENT_TYPE_U8 |
| ELEMENT_TYPE_R4 | ELEMENT_TYPE_R8 |
| ELEMENT_TYPE_STRING | ELEMENT_TYPE_CLASS |

(This list is a subset of the CorElementType enumeration in CorHdr.h)

In the particular case of ELEMENT_TYPE_CLASS, its value can only be null.

Indicate that you do not wish to specify a default value, by providing a value for dwDefType of all-bits-set (-1).

### 1.5.4 Null Pointers for Return Parameters

Since the metadata APIs do a minimum of error checking, it's useful to understand when they expect that you will provide a non-null pointer for return parameters:

- In **define** methods, a non-null pointer is always required for the return token for the thing that is being defined: we create one, you get back the token for it. Don't look at it if you don't want it.
- In **find** methods, we also always expect to return the token for the thing we successfully find.
- In **get** methods, you may pass null in for parameters you are not interested in getting back.
- In **set** methods, there's generally no return. You pass in the token for the thing to be updated, along with the values to update, and the metadata APIs perform the update.

### 1.5.5 "Ignore This Argument"

Several methods in the metadata API allow you to change the value an item that was defined earlier. For example:

```
HRESULT SetFieldProps(mdFieldDef fd, DWORD dwFieldFlags,
        DWORD dwDefType, void const *pValue, ULONG cchValue)
```

allows you to change *dwFieldFlags*, *dwDefType* and *pValue*, previously supplied in a call to DefineField. But what if you want to change *dwFieldFlags* but not *pValue* (or vice versa)? How do you specify this? We obey the following conventions for method parameters:

- Pointer – use a null pointer to indicate "ignore this argument"
- Value (typically a flags bitmask) – use a value of all bits set (–1) to indicate "ignore this argument"

### 1.5.6 Error Returns

Almost all methods in the IMetadataDispenserEx, IMetaDataEmit and IMetaDataImport interfaces return an HRESULT to indicate their result. This has the value S_OK if the operation was successful, or another value that describes the reason why the operation failed.

One general pattern across all the MetaData APIs is that if the caller provides a string buffer that is too small to hold the results, then we copy as many characters as will fit, but return the alternate success HRESULT of CLDB_S_TRUNCATION.

Recall that callers of the IMetadata* interfaces are compilers or tools – not end users. It is the responsibility of these callers to always check the return status from each call – since these reflect errors on the part of the direct caller (eg a compiler) than of the end user (eg a programmer).

# 2 IMetadataDispenserEx

The dispenser API is used to map existing metadata so that it can be inspected (and added to), or to create a fresh in-memory area to define new metadata. In this section, we also include methods to control how the metadata API operates.

## 2.1 DefineScope

```
HRESULT DefineScope(REFCLSID rclsid, DWORD dwCreateFlags,
        REFIID riid, IUnknown **ppIUnk)
```

Create a fresh area in memory, into which you can create new metadata using the MetaData Emit API. *DefineScope* creates a set of in-memory metadata tables of the specified class, generates a unique guid (module version identifier, or *mvid*) for the metadata, and creates an entry in the *Module* able for the compilation unit being emitted. If successful, the requested metadata interface is returned. Note that a developer may attach attributes to the metadata scope as a whole using *IMetadataEmit::SetModuleProps* or *IMetadataEmit::DefineCustomAttribute*, as appropriate.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | rclsid | The CLSID of the version of metadata structures to create | yes |
| in | dwCreateFlags | Used to tailor DefineScope behavior. Must be 0 | yes |
| in | riid | The IID of the interface required | yes |
| out | ppIUnk | The returned interface, on success. | |

*rclsic* should be specified as CLSID_ CorMetaDataRuntime in this release

*riid* must be one of *IID_IMetaDataEmit*, *IID_IMetaDataImport*, *IID_IMetaDataAssemblyEmit* or *IID_IMetaDataAssemblyImport*

## 2.2 OpenScope

```
HRESULT OpenScope(LPCWSTR wzScope, DWORD dwOpenFlags,
        REFIID riid, IUnknown **ppIUnk)
```

Open an existing file, and map its metadata into memory. That in-memory copy of the metadata can then be queried using methods from the *IMetaDataImport* or added-to using method from the *IMetaDataEmit* interfaces. Note that the target file must contain CLR metadata, else the method will fail.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | wzScope | target file | yes |
| in | dwOpenFlags | 0 = open for read, 1 = open for write | yes |
| in | riid | The IID of the interface required | yes |
| out | ppIUnk | The returned interface | |

*riid* must be one of *IID_IMetaDataEmit*, *IID_IMetaDataImport*, *IID_IMetaDataAssemblyEmit* or *IID_IMetaDataAssemblyImport*

**Example:**

```
HRESULT h;
IMetaDataImport* p;
h = OpenScope (L"file:c\\App.Exe", 0, IID_IMetaDataImport, (IUnknown**) &p);
```

## 2.3 OpenScopeOnMemory

```
HRESULT OpenScopeOnMemory(LPCVOID pData, ULONG cbData,
        DWORD dwOpenFlags, REFIID riid, IUnknown **ppIUnk);
```

Treat the area of memory specified by the pData and cbData arguments as CLR metaData. This metaData can then be queried using methods from the *IMetaDataImport* interface. This is similar to the *OpenScope* method, except that metaData of interest already exists in-memory, rather than in a file on-disk.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | pData | Pointer to start of memory | yes |
| in | cbData | Size of the memory area, in bytes | yes |
| in | dwOpenFlags | 0 = open for read, 1 = open for wrie | yes |
| in | riid | The IID of the interface required | yes |
| out | ppIUnk | The returned interface | |

*riid* must be one *of IID_IMetaDataEmit, IID_IMetaDataImport, IID_IMetaDataAssemblyEmit* or *IID_IMetaDataAssemblyImport*

## 2.4 SetOption

You can control how your calls to the metadata API are handled. These settings are transient; they are not persisted to disk.

The settings are gathered into the following categories:

**Duplicate checks** Each time you call a method on IMetaDataEmit that creates a new item, you can ask it to check whether the item already exists in the current scope. You can control which items are checked and which are not. For example, you can ask for checking on MethodDefs; in this case, when you call DefineMethod, it will check that the method does not already exist in the current scope. This check uses the *key* that uniquely identifies a given method: parent type, name and signature

**Ref-to-Def optimizations** By default, the metadata engine will convert Refs to Defs where it can (where the referenced item actually exists in the current scope). You can control which Refs are optimized in this way

**Notifications on token movement** Controls which token remaps (during metadata merge) call you back. (Use SetHandler to establish your IMapToken interface)

**ENC Modes** – allow control over behaviour of EditAndContinue

**EmitOutOfOrder** Allows you to control which out-of-order 'errors' call you back. (Use SetHandler to establish your IMetaDataError interface). Emitting metadata 'out-of-order' is not fatal – it's just that if you emit it in an order favoured by the metadata engine, the metadata is more compact and efficient to search)

**Import Options** Specify which sorts of deleted metadata tokens are returned in any enumeration. (See *DeleteToken* for more information)

**Generate TCE Adaptors** – yes or no

**NameSpace** Specifies a different namespace than the one provided by the type library being imported.

**ThreadSafetyOptions** Specifies whether you want the metadata engine to take out reader/writer locks to ensure thread safety (default assumes access is single-threaded by the caller, so no locks are taken)

```
HRESULT SetOption (REFGUID optionId, const VARIANT *pvalue)
```

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | optionId | Pointer to GUID that specifies required option | yes |
| in | pvalue | Value to set | yes |

*optionId* argument must point to one of the following GUIDs, defined in Cor.h:

- MetaDataCheckDuplicatesFor. *pvalue* must be a variant of type UI4, holding a bitmask of which duplicate checks you require. See the CorCheckDuplicatesFor enum in CorHdr.h
- MetaDataRefToDefCheck. *pvalue* must be a variant of type UI4, holding a bitmask of which checks you require. See the CorRefToDefCheck enum in CorHdr.h
- MetaDataNotificationForTokenMovement. *pvalue* must be a variant of type UI4, holding a bitmask of which notifications you require. See the CorNotificationForTokenMovement enum in CorHdr.h
- MetaDataSetUpdate. *pvalue* must be a variant of type UI4, holding a bitmask of which checks you require. See the CorSetUpdate enum in CorHdr.h
- MetaDataErrorIfEmitOutOfOrder. *pvalue* must be a variant of type UI4, holding a bitmask of which checks you require. See the CorErrorIfEmitOutOfOrder enum in CorHdr.h
- MetaDataImportOption. *pvalue* must be a variant of type UI4, holding a bitmask of which deleted items you want reported in an enumeration of the metadata. See the CorImportOptions enum in CorHdr.h
- MetaDataGenerateTCEAdapters. *pvalue* must be a variant of type BOOL. If set true, then when we import a type library, we will translate event source interfaces to add/remove methods.
- MetaDataTypeLibImportNamespace. *pvalue* must be a variant of type BSTR, EMPTY or NULL. If *pvalue* represents a nil value, then the current namespace is set to null; otherwise the current namespace is set to the string held in the variant's BSTR

- MetaDataThreadSafetyOptions. *pvalue* must be a variant of type UI4, holding a bitmask of which safety options you require. See the CorThreadSafetyOptions enum in CorHdr.h

## 2.5 GetOption

Returns the settings for the current metadata scope. See *SetOption* for details.

```
HRESULT GetOption(REFGUID optionId, const VARIANT *pvalue)
```

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | optionId | Pointer to GUID that specifies required option | yes |
| in | pvalue | Value to return | yes |

# 3 IMetaDataEmit

The emitter API is used by compilers to generate in-memory and on-disk metadata. This API is implemented directly over the low-level metadata engine APIs, generating records into the various data structures, which are converted at "save" time to the target on-disk format.

## 3.1 Defining, Saving, and Merging Metadata

### *3.1.1 SetModuleProps*

```
HRESULT SetModuleProps(LPCWSTR wzName)
```

Records a name for the current scope. This should be the name of the file in which this module is stored. Eg: "Foo.DLL" – no drive letter, no path

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | wzName | Module name in Unicode | no |

### *3.1.2 Save*

```
HRESULT Save(LPCWSTR wzFile, DWORD dwSaveFlags)
```

Saves all of the metadata in the current scope to the specified file. The method leaves all of the metadata intact

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | wzFile | Name of file to save to. If null, the in-memory copy will be saved to the last location that was used | no |
| in | dwSaveFlags | [reserved] | must be 0 |

### *3.1.3 SaveToStream*

```
HRESULT SaveToStream(IStream *pIStream, DWORD dwSaveFlags)
```

Saves all of the metadata in the current scope to the specified stream. The method leaves all of the metadata intact.

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | pIStream | Writeable stream to save to | yes |
| in | dwSaveFlags | [reserved] | must be 0 |

### *3.1.4 SaveToMemory*

```
HRESULT SaveToMemory(void *pbData, ULONG cbData)
```

Saves all of the metadata in the current scope to the specified area of memory. The method leaves all of the metadata intact.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | pbData | Start address at which to write metadata | yes |
| in | cbData | Size of allocated memory, in bytes | yes |

### 3.1.5 GetSaveSize

```
HRESULT GetSaveSize(CorSaveSize fSave, DWORD *pdwSaveSize)
```

Calculates the space required, in bytes, to save all of the metadata in the current scope. (Specifically, a call to the SaveToStream method would emit this number of bytes)

If the caller implements the IMapToken interface (via SetHandler or Merge), then GetSaveSize will perform two passes over the metadata in order to optimize and compress it. Otherwise, no optimizations are performed.

If optimization is performed, the first pass simply sorts the metadata structures so as to tune the performance of import-time searches. This step will likely result in moving records around, with the side-effect that tokens the tool has retained for future reference are invalidated. (Metadata does not inform its caller of these token changes until after the second pass, however). In the second pass, various optimizations are performed that are intended to reduce the overall size of the metadata, such as optimizing away (early binding) mdTypeRefs and mdMemberRefs when the reference is to a type or member that is declared in the current metadata scope. In this pass, another round of token mapping occurs. After this pass, the metadata engine notifies the caller, via its IMapToken interface, of any changed token values.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | fSave | Requests accurate, or approximate | no |
| out | pdwSaveSize | Size required to save file | |

*fSave* should be one of *cssAccurate* (the default), or *cssQuick* (see the *CorSaveSize* enum in CorHdr.h). *cssAccurate* will return the exact save size but takes longer to calculate. *cssQuick* will return a size, padded for safety, but takes less time to calculate. *fSave* can also have the *cssDiscardTransientCAs* bit set – this tells *GetSaveSize* that it can throw away discardable custom attributes

### 3.1.6 Merge

```
HRESULT Merge(IMetaDataImport *pImport, IMapToken *pIMap,
         IUnknown *pHandler)
```

Starts a merge of metadata from the scope defined by *pImport* into the current metadata scope. In so doing, tokens from the imported scope are remapped into the current scope. *Merge* uses the *IMapToken* interface supplied by the caller to notify the caller of each remap; it uses the *IMetaDataError* interface supplied by the caller to notify the caller of any errors.

This routine can be called for several import scopes. The actual merge operation, across all these import scopes is triggered by calling the routine *MergeEnd*

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | pImport | Identifies other metadata scope to be merged | yes |
| in | pIMap | Interface on which to notify token remaps | no |
| in | pHandleer | Interface on which to notify errors | no |

## 3.1.7 MergeEnd

```
HRESULT MergeEnd( )
```

This routine triggers the actual merge of metadata, of all import scopes specified by preceding calls to *Merge* into the current output scope.

During merge, various errors may be encountered, as follows:

The following special conditions apply to the merge:

- An MVID is never imported, since it is unique to that other metadata
- No existing module-wide properties are overwritten. So, if module properties were already set for the current scope, no module properties are imported. But, if module properties have not been set in the current scope, they will be imported once-only, when they are first encountered. If they are encountered again, they must be duplicates (eg, when merging .obj files during a VC link step); if they are not duplicates, based on comparing the values of all module properties (except MVID), we raise an error
- For TypeDefs, no duplicates will be merged into the current scope. The check for duplicates is based on fully-qualified name + guid + version number. If there is a match on name or on guid and any of the other two elements is different, we raise an error. Else, if there is a full match on 3 items, *Merge* does a cursory check to ensure the entries are indeed duplicates – we raise an error if they are not. This cursory check is based on:
    - Same member declarations, in same order. (However, members flagged as mdP*rivateScope* are not included in this check; they are merged specially; see later)
    - Same class layout

      Observe that this means that a TypeDef must always be fully and consistently defined in every metadata scope in which it is declared; if its member implementations (for a class) are spread across multiple compilation units (as in VC), the full definition is assumed to be present in every scope and not incremental to each scope. For example, if parameter names are relevant to the contract, they must be emitted the same way into every scope; if they are not relevant, they should not be emitted into metadata

      The exception is that a TypeDef may have incremental members flagged as *mdPrivateScope*. On encountering these, *Merge* will incrementally add them to the current scope without regard for duplicates (since only the compiler understands the private scope, the compiler must be responsible for enforcing rules)

- When merging members that have RVAs, we do not import/merge any of this information – the compiler is expected to re-emit it
- Custom attributes are merged only at the time we merge the item they are attached to. For example, custom attributes associated with a class will be merged when the class is first encountered. If custom attributes are associated with TypeDefs or MemberDefs that were specific to the compilation unit (e.g., time stamp of member compile), these will not be handled specially and it is up to the compiler to remove or update such metadata.

### *3.1.8 SetHandler*

```
HRESULT SetHandler(IUnknown *pUnk)
```

Registers a handler interface through which the caller may receive notification of errors (IMetaDataError) and of token remaps (IMapToken).

The metadata engine sends notification on the map token interface provided by SetHandler() for compilers who do not generate records in an optimized way and would like to save optimized. If IMapToken is not provided via SetHandler, no optimization will be performed on save *except* where several import scopes have been merged using the provided IMapToken on merge for each scope.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | pUnk | Handler to register | yes |

## 3.2 Custom Attributes

Custom attributes are just what they say – attributes you can attach to a programming element, such as a method or field. But these attributes are defined by the customer – the programmer and/or language – rather than pre-defined by the runtime itself.

Think of a custom **attribute** as a triple of (tokenParent, tokenMethod, blob) stored into metadata. The blob holds the arguments to the class constructor method specified by tokenMethod. The runtime has a full understanding of the contents of this blob; on request, it will instantiate the attribute-object that the blob represents, attaching it to the item whose token is tokenParent.

### *3.2.1 Using Custom Attributes*

The model for using custom attributes has two steps. First, the programmer defines a custom attribute-class, and the language emits that definition into the metadata, just as it would for any regular class. Here is an example of defining an attribute-class, called Location, in some invented programming language:

```
[attribute] class Location {
    string name;
    Location (string n) {name = n;}
}
```

Second, the programmer defines an instance of that attribute class (let’s call it an attribute-object) and attaches it to some programming element. Here is an example of defining two Location attribute-objects and attaching them to two classes, Television and Refrigerator. Note that we define the attribute-object by providing a literal string argument to its Location constructor method:

```
[Location (“Aisle 3”)]  class Television { . . . }
[Location (“Aisle 42”)] class Refrigerator { . . . }
```

As a result, the Television class at runtime will always have an attribute-object attached (whose name field holds the string “Aisle 3”) whilst the Refrigerator class at runtime will have an attribute-object attached (whose name field holds the string “Aisle 42”)

Note that attribute-classes are not distinguished in any way whatsoever by the runtime – their definition within metadata looks just like any regular type definition. Our use therefore of “attribute-class” in this spec is simply to help understanding.

Custom attribute-objects can be attached to any metadata item that has a metadata token: mdTypeDef, mdTypeRef, mdMethod, mdField, mdParameter, etc. Duplicates are supported, such that a given programming element may well have multiple attribute-objects of the same attribute-class attached to it. [so, in the example above, class Television might have two Location attribute-objects – with name fields of “Aisle 42” and “Back Store”]

It is legal to attach a custom attribute-object to a custom attribute-class. (but you cannot attach a custom-attribute object to any individual runtime *object*)

Custom attributes have the following characteristics:

- Require up-front design before attributes can be emitted
- Capitalize on the runtime infrastructure for class identity, structure, and versioning
- Allow tools, services, and third parties (the primary customers for this mechanism) to extend the types of information that may be carried in metadata without having to depend on the runtime to maintain and version that information
- Although each language or tool will provide a language-specific syntax and conventions for using custom attributes, the self-describing nature of these attributes will enable tools to provide drop-down lists and other developer aids
- Runtime Reflection services will support browsing over these custom attributes, since they are self-describing.

### 3.2.2 DefineCustomAttribute

```
HRESULT DefineCustomAttribute(mdToken tkOwner, mdToken tkAttrib,
        void const *pBlob, ULONG cbBlob, mdCustomAttribute *pca)
```

Defines a custom attribute-object, attached to the specified parent (tkOwner)

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | tkOwner | Token for the owner item | yes |
| in | tkCtor | Token that identifies the custom attribute | yes |
| in | pBlob | Pointer to blob | no |
| in | cbBlob | Count of bytes in pBlob | no |
| out | pca | CustomAttribute token assigned | |

*tkOwner* may be any valid metadata token, except an mdCustomAttribute.

*tkCtor* is the token that identifies the constructor method to execute to create the custom attribute-object.

The format of *pBlob* for defining a custom attribute is defined in later in this spec. (broadly speaking, the blob records the argument values to the class constructor, together with zero or more values for named fields/properites – in other words, the information needed to instantiate the object specified at the time the metadata was emitted). If the constructor requires no arguments, then there is no need to provide a blob argument.

### 3.2.3 SetCustomAttributeValue

```
HRESULT SetCustomAttributeValue(mdCustomAttribute pca,
        void const *pBlob, ULONG cbBlob)
```

Sets the value of an existing custom attribute to have a new value. The value that was previously defined is replaced with this new value.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | pca | Token of target custom attribute | yes |
| in | pBlob | Pointer to blob | yes |
| in | cbBlob | Count of bytes in pBlob | yes |

## 3.3 Building Type Definitions

### 3.3.1 DefineTypeDef

```
HRESULT DefineTypeDef(LPCWSTR wzName, DWORD dwTypeDefFlags,
        mdToken tkExtends, mdToken rtkImplements[], mdTypeDef *ptd)
```

Defines a type. A flag in *dwTypeDefFlags* specifies whether the type being created is a common type system reference type (class or interface) or a common type system value type.

Duplicates are disallowed. So, within any scope, *wzName* must be unique.

Depending on the parameters supplied, this method, as a side effect, may also create an *InterfaceImpl* record for each interface inherited or implemented by this type. None of these *InterfaceImpl* tokens are returned by this method – if a client wants to later add/modify these *InterfaceImpls*, it must use *IMetaDataImport* to enumerate them. If COM semantics of 'default interface' are desired, then it's important to supply the default interface as the first in *rtkImplements[]*; a custom attribute set on the class will indicate that it does have a default interface (which is always assumed to be the first *InterfaceImpl* declared for the class). Refer to the COM Integration spec for more details.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | wzName | Name of type in Unicode | yes |
| in | dwTypeDefFlags | Typedef attributes | yes |
| in | tkExtends | Token of the superclass | no |
| in | rtkImplements[] | Array of tokens specifying the interfaces that this class or interface implements | no |
| out | ptd | TypeDef token assigned | |

*dwTypeDefFlags* is a bitmask from the CorTypeAttr enum in CorHdr.h.

*tkExtends* must be an mdTypeDef or an mdTypeRef.

Each element of the rtkImplements[] array holds an mdTypeDef or an mdTypeRef. The last element in the array must be mdTokenNil.

## 3.3.2 SetTypeDefProps

```
HRESULT SetTypeDefProps(mdTypeDef td, DWORD dwTypeDefFlags,
        mdToken tkExtends, DWORD mdToken rtkImplements[])
```

Sets the attributes of an existing type, previously defined using the *DefineTypeDef* method. This is useful when the original definition supplied only minimal information, perhaps corresponding to a forward reference in the compiler's source language. Note that you cannot use this method to change the type's name. In all other respects however, *SetTypeDefProps* has essentially the same behavior as *DefineTypeDef* and, depending on the parameters supplied, it may also create one or more *InterfaceImpl* data structures.

If you supply a value for any argument, it will supersede the value you supplied in the earlier call to *DefineTypeDef*. If you want to leave the original value unchanged, mark that argument as "to be ignored" – see section 1.5.5 for details.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | td | TypeDef token obtained from original call to DefineTypeDef | yes |
| in | dwTypeDefFlags | Typedef attributes | no |
| in | tkExtends | Token of the superclass. Obtained from a previous call to DefineImportType, or null. | no |
| in | rtkImplements[] | Array of tokens for the interfaces that this type implements. These TypeRef tokens are obtained via DefineImportType | no |

*dwTypeDefFlags* is a bitmask from the *CorTypeAttr* enumeration in CorHdr.h.

*tkExtends* must be an *mdTypeDef* or an *mdTypeRef* or nil

Each element of *rtkImplements[]* is an *mdTypeDef* or an *mdTypeRef*. (Typically, you obtain required *TypeRef* tokens by a call to *DefineImportType*) The last element in the array must be *mdTokenNil*.

## 3.4 Declaring and Defining Members

### 3.4.1 DefineMethod

```
HRESULT DefineMethod(mdTypeDef td, LPCWSTR wzName,
        DWORD dwMethodFlags, PCCOR_SIGNATURE pvSig, ULONG cbSig,
        ULONG ulCodeRVA, DWORD dwImplFlags, mdMethodDef *pmd)
```

Defines a method (of a class or interface), or a global-function. If a method, then use *td* to specify the TypeDef token for its enclosing class or interface. If a global-function, then set *td* to *mdTokenNil*.

The metadata API guarantees to persist methods in the same order as the caller emits them for a given enclosing class or interface (its *td* argument).

See later in this spec for details on how to set the method declaration flags (dwMethodFlags) and method implementation flags (dwImplFlags).

The runtime uses MethodDefs to set up vtable slots. In the case where one or more slots need to be skipped (e.g., to preserve parity with a COM interface layout), a dummy method would be defined in order to take up the slot(s) in the vtable. The method would be defined using the "special name" flag (mdRTSpecialName), with the name encoded as:

_VtblGap<SequenceNumber><_CountOfSlots>

where *SequenceNumber* is the sequence number of the method and *CountOfSlots* is the number of slots to skip in the vtable.

If *CountOfSlots* is omitted, 1 is assumed. These dummy methods are not callable from either managed or unmanaged code. Any attempt to call these methods, either from managed or unmanaged code will generate an exception. Their only purpose is to take up space in the vtable that the runtime generates for COM integration. They have no impact on managed clients that may be using the interface.

The format of the signature blob is specified later in this spec. (briefly, the blob captures the calling convention, the type of each parameter, and the return type). The caller builds the signature blob. This API assumes it is a valid method signature in the emit scope. No checks are performed; the signature is persisted as supplied. If you need to specify additional information for any parameters, use the SetParamProps method.

You should not define duplicate methods. That's to say, the triple (td, wzName, pvSig) should be unique. There is one exception to this rule: you can define a duplicate triple so long as one of those definitions sets the mdPrivateScope bit in the dwMethodFlags argument. (The mdPrivateScope bit means the compiler will not emit a reference to this methodDef). A typical use is when defining a function that is private to a compiland (the runtime does not recognize or support compiland scope). Note that any mdPrivateScope methods do not affect the metadata ordering guarantee. Ideally, tools and compilers would emit scoped statics after all the other methods, but it should be sufficient to say that even if mdPrivateScope members are interleaved in method sequences they are simply ignored when it comes to layout.

Method implementation information is often not known at the time the method is declared, e.g. in languages where the front-end calls DefineMethod but it is the backend that supplies implementation information and the linker that supplies code address information. As such, ulCodeRVA and dwImplFlags are not required to be supplied with DefineMethod. They may be supplied later via SetMethodImplFlags or SetRVA, as appropriate.

In some situations, such as PInvoke or COMinterop scenarios, the method body will not be supplied, and ulCodeRVA will remain 0. In these situations, the method should not be tagged as *abstract*, since the runtime will locate the implementation. (See interop specs for more detail).

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | td | Typedef token of parent | no |
| in | wzName | Member name in Unicode | yes |
| in | dwMethodFlags | Member attributes | yes |
| in | pvSig | Method signature | yes |
| in | cbSig | Count of bytes in pvSig | yes |
| in | ulCodeRVA | Address of code | no, may be 0 |
| in | dwImplFlags | Implementation flags for method | no, may be 0 or all 1s |
| out | pmd | Member token | |

dwMethodFlags is a bitmask from the CorMethodAttr enum in CorHdr.h.

dwImplFlags is a bitmask from the CorMethodImpl enum in CorHdr.h.

### 3.4.2 SetMethodProps

```
HRESULT SetMethodProps(mdMethodDef md, DWORD dwMethodFlags,
        ULONG ulCodeRVA, DWORD dwImplFlags)
```

Changes the settings for a previously-defined method.

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | md | Token for method to be changed | yes |
| in | dwMethodFlags | Member attributes | no |
| in | ulCodeRVA | Address of code | no |
| in | dwImplFlags | Implementation flags for method | no |

dwMethodFlags is a bitmask from the CorMethodAttr enumeration in CorHdr.h.

ulCodeRVA is the address at which the method’s code starts.

dwImplFlags is a bitmask from the CorMethodImpl enumeration in CorHdr.h.

If you supply a value for any optional argument, that value will supersede the previous, supplied to DefineMethod. If you want to leave the original value unchanged, mark the argument as “to be ignored” – see section 1.5.5 for details.

### 3.4.3 DefineField

```
HRESULT DefineField(mdTypeDef td, LPCWSTR wzName,
        DWORD dwFieldFlags, PCCOR_SIGNATURE pvSig, ULONG cbSig,
        DWORD dwDefType, void const *pValue, ULONG cchValue,
        mdFieldDef *pmd)
```

Defines a field. The field may be specified as global (if td = mdTokenNil) or as a member of an existing class or interface (td = the TypeDef token for that parent class or interface).

The metadata API guarantees to persist the fields in the same order as the caller emits them for a given parent (the td argument).

See section 10 for the format of the signature blob. It is built by the client and is assumed to be a valid type signature in the current scope. No checks are performed: the signature is persisted as supplied.

You should not define duplicate fields. That’s to say, the triple (td, wzName and pvSig) should be unique. However, there is one exception to this rule: you can define a duplicate triple so long as one of those definitions sets the fdPrivateScope bit in the dwFieldFlags argument. (The fdPrivateScope bit means this field was emitted solely for use by the compiler – for example, to obtain a metadata token to pass to MSIL. The compiler takes on responsiblity to never create a FieldRef in any other module, to this field. A typical use is when defining a static local variable in a method – static in the sense that its visibility is limited to the current compiland).

You can use this method to save a default value for the property, via the dwDefType, pValue and cchValue parameters – see 1.5.3 for details.

*Global data may need initialization upon module load. The design approach is for the compiler to emit one or more function definitions that correspond to the initializers. Rather than providing any runtime support for calling the initializers, the compiler will call them explicitly, in the appropriate sequence, from the body of the module entry point. As such, there is neither special-purpose metadata nor runtime support needed to initialize the module’s static data members.*

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | td | Typedef token for the enclosing class or interface | yes |
| in | wzName | Field name in Unicode | yes |
| in | dwFieldFlags | Field attributes | yes |
| in | pvSig | Field signature as a blob | yes |
| in | cbSig | Count of bytes in pvSig | yes |
| in | dwDefType | ELEMENT_TYPE_* for the constant value | no |
| in | pValue | Constant value for field | no |
| in | cchValue | Size in (Unicode) characters of pValue | no |
| out | pmd | FieldDef token assigned | |

dwFieldFlags is a bitmask from the CorFieldAttr enumeration in CorHdr.h.

dwDefType is a value from the CorElementType enumeration in CorHdr.h. If you do not want to define any constant value for this field, supply a value of ELEMENT_TYPE_END for dwDefType.

### 3.4.4 SetFieldProps

```
HRESULT SetFieldProps(mdFieldDef fd, DWORD dwFieldFlags,
        DWORD dwDefType, void const *pValue, ULONG cchValue)
```

Sets the properties of an existing field. See the description of *DefineField* for more information.

If you supply a value for any optional argument, that value will supersede the previous, supplied to *DefineField*. If you want to leave the original value unchanged, mark the argument as “to be ignored” – see section 1.5.5 for details.

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | fd | Token for the target field | yes |
| in | dwFieldFlags | Field attributes | no |
| in | dwDefType | ELEMENT_TYPE_* for the constant value | no |
| in | pValue | Constant value for field | no |
| in | cchValue | Size in (Unicode) characters of pValue | no |

dwFieldFlags is a bitmask from the CorFieldAttr enum in CorHdr.h.

dwDefType is a value from the CorElementType enum in CorHdr.h. If you do not want to define any constant value for this field, supply a value of ELEMENT_TYPE_END.

### 3.4.5 DefineNestedType

```
HRESULT DefineNestedType(LPCWSTR wzName, DWORD dwTypeDefFlags,
        mdToken tkExtends, mdToken rtkImplements[],
        mdTypeDef tdEncloser, mdTypeDef *ptd)
```

Defines a type that is lexically nested within an enclosing type. This call is analogous to *DefineTypeDef* – but has an extra argument, *tdEncloser,* to denote the type that encloses this type. (see *DefineTypeDef* – section 3.3.1 for more detail)

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | wzName | Name of type in Unicode | yes |
| in | dwTypeDefFlags | Typedef attributes | yes |
| in | tkExtends | Token of the superclass | yes |
| in | rtkImplements[] | Array of tokens specifying the interfaces that this class or interface implements | no |
| in | tdEncloser | Token of the enclosing type | yes |
| out | ptd | TypeDef token assigned | |

Supply the simple, unmangled name of the type in *wzName*

*dwFlags* is a bitmask from the CorTypeAttr enum in CorHdr.h. You must set one of the tdNestedXXX bits – that's to say, one of tdNestedPublic, tdNestedPrivate, tdNestedFamily, tdNestedAssembly, tdNestedFamANDAssem or tdNestedFamORAssem.

*tkExtends* must be a TypeDef or a TypeRef

*tdEncloser* must be a TypeDef (in other words, the enclosing class is defined within this same module). It cannot be a TypeRef.

Each element of the *rtkImplements[]* array holds an *mdTypeDef* or an *mdTypeRef*. The last element in the array must be *mdTokenNil*.

### 3.4.6 DefineParam

```
HRESULT DefineParam(mdMethodDef md, ULONG ulParamSeq,
        LPCWSTR wzName, DWORD dwParamFlags, DWORD dwDefType,
        void const *pValue, ULONG cchValue, mdParamDef *ppd)
```

Defines extra information for a method parameter (beyond what could have been supplied in the definition of its corresponding method signature)

You can use this method to save a default value for the property, via the dwDefType, pValue and cchValue parameters – see 1.5.3 for details.

Note that even if you specify that all optional parameters to this call are to be ignored (see 1.5.5), metadata will still create a ParamDef record and return its assigned token.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | md | Token for the method whose parameter is being defined | yes |
| in | ulParamSeq | Parameter sequence number | yes |
| in | wzName | Name of parameter in Unicode | no |
| in | dwParamFlags | Flags for parameter | no |
| in | dwDefType | ELEMENT_TYPE_* for the constant value | no |
| in | pValue | Constant value for parameter | no |
| in | cchValue | Size in (Unicode) characters of pValue | no |
| out | ppd | ParamDef token assigned | |

ulParamSeq specifies the parameter sequence number, starting at 1. Use a value of 0 to mean the method return value.

wzName is the name to give the parameter. If you specify null, this argument is ignored. If you wish to remove any previous-supplied name, supply an empty strring for wzName.

dwParamFlags is a bitmask from the CorParamAttr enumeration in CorHdr.h. If you specify all-bits-set (-1), then this argument will be ignored (see 1.5.5)

### 3.4.7 SetParamProps

```
HRESULT SetParamProps(mdParamDef pd, LPCWSTR wzName,
        DWORD dwParamFlags, DWORD dwDefType,
        void const *pValue, ULONG cchValue)
```

Sets the attributes for a specified method parameter. See the description of *DefineParam* for details.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | pd | Token for target parameter | yes |
| in | wzName | Name of parameter in Unicode | no |
| in | dwParamFlags | Flags for parameter | no |
| in | dwDefType | ELEMENT_TYPE_* for the constant value | no |
| in | pValue | Constant value for parameter | no |
| in | cchValue | Size in (Unicode) characters of pValue | no |

If you supply a value for any optional argument, that value will supersede the previous, supplied to DefineParam. If you want to leave the original value unchanged, mark the argument as “to be ignored” – see section 1.5.5 for details.

### 3.4.8 DefineMethodImpl

```
HRESULT DefineMethodImpl(mdTypeDef td, mdToken tkBody,
        mdToken tkDecl)
```

Defines how a class implements a method that it inherits from an interface. *td* specifies the class that is implementing the method. *tkBody* specifies the code that is to be used to implement the method. *tdDecl* specifies the method in the interface for which we are providing a code body

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | td | Typedef token of the implementing class | yes |
| in | tkBody | MethodDef or MethodRef token of the code body | yes |
| in | tkDecl | MethodDef or MethodRef token of the interface method being implemented | yes |

## 3.4.9 SetRVA

```
HRESULT SetRVA(mdMethodDef md, ULONG ulRVA)
```

Sets or replaces the RVA for an existing MethodDef

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | tk | Token for target method or method implementation | yes |
| in | ulRVA | Address of code or data area | yes |

## 3.4.10SetFieldRVA

```
HRESULT SetFieldRVA(mdFieldDef fd, ULONG ulRVA)
```

Sets or replaces the RVA for an existing global-variable. In general, global-variables don’t need to be declared at all in metadata: they are static data laid out by the compiler and allocated in the PE file in which they are declared and used; access to them is entirely an internal implementation issue. However, when a global variable is to be exported to managed code from the module, a metadata declaration is needed.

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | fd | Token for target field | yes |
| in | ulRVA | Address of code or data area | yes |

## 3.4.11DefinePinvokeMap

```
HRESULT DefinePinvokeMap(mdToken tk, DWORD dwMappingFlags,
        LPCWSTR wzImportName, mdModuleRef mrImportDLL)
```

Defines information for a method that will be used by PInvoke (Runtime service that supports inter-operation with unmanaged code)

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | tk | Token for target method | yes |
| in | dwMappingFlags | Flags used by Pinvoke to do the mapping | no |
| in | wzImportName | Name of target export method in unmanaged DLL | no |
| in | mrImportDLL | Token for target native DLL | yes |

*tk* is an mdMethodDef token

*dwMappingFlags* is a bitmask from the CorPinvokeMap enum in CorHdr.h

*wzImportName* may be the simple name of the imported function (eg "MessageBox") or its ordinal, encoded as a decimal integer preceded by a # character (eg "#123")

## 3.4.12SetPinvokeMap

```
HRESULT SetPinvokeMap(mdToken tk, DWORD dwMappingFlags,
        LPCWSTR wzImportName, mdModuleRef mrImportDLL)
```

Sets information for a method that will be used by PInvoke (runtime service that supports inter-operation with unmanaged code)

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | tk | Token to which mapping info applies | yes |
| in | dwMappingFlags | Flags used by pinvoke to do the mapping | no |
| in | wzImportName | Name of target export in native DLL | no |
| in | mdImportDLL | mdModuleRef token for target unmanaged DLL | no |

*tk* is an mdMethodDef token

*dwMappingFlags* is a bitmask from the CorPinvokeMap enum in CorHdr.h.

*wzImportName* may be the simple name of the imported function (eg "MessageBox") or its ordinal, encoded as a decimal integer preceded by a # character (eg "#123")

## 3.4.13SetFieldMarshal

```
HRESULT SetFieldMarshal(mdToken tk, PCCOR_SIGNATURE pvUnmgdType,
        ULONG cbUnmgdType)
```

Sets marshaling information for a field, method return, or method parameter. Specifically, you specify the unmanaged type that this data item should be marshalled to and from. See the "COM Integration" and "Platform Invoke" specs for details on when/where unmananaged type information is used and for the format of the unmanaged type signature blob

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | tk | Token for target data item | yes |
| in | pvUnmgdType | Signature for unmanaged type | yes |
| in | cbUnmgdType | Count of bytes in pvUnmgdType | yes |

*tk* is an mdFieldDef or mdParamDef that specifies the target field or parameter

## 3.5 Building Type and Member References

### 3.5.1 *DefineTypeRefByName*

```
HRESULT DefineTypeRefByName(mdToken tkResScope,
        LPCWSTR wzName, mdTypeRef *ptr)
```

Defines a reference to a type that exists in another module. This method does not look into that other module. Therefore, attempting to resolve the type reference might fail at runtime

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | tkResScope | Token for the resolution scope: ModuleRef if defined in same assembly as caller; AssemblyRef if defined in a different assembly than caller; TypeRef if this is a nested type; Module if defined in same module; or nil | yes |
| in | wzName | Name of target type in Unicode | yes |
| out | ptr | TypeRef token assigned | |

tkResScope must be an mdModuleRef, mdAssemblyRef, mdTypeRef, mdModule or nil. These are used as follows:

- If the target Type is defined in a different module, but one which lies in the same assembly as the current module, then you should supply an mdModuleRef to that other module (eg to "Foo.DLL")
- If the target Type is defined in a module which lies in a different assembly from the current module, then you should supply an mdAssemblyRef to that other assembly (eg to "MyAssem" – no file extension)
- If the target Type is a nested Type, then supply an mdTypeRef to its enclosing Type
- If the target Type exists in this same module, then supply an mdModule for the current module – the one you obtain by calling *GetModuleFromScope*) Note that this is a legal, but rare, case – you can almost use a TypeDef instead!

- If you don't know the final module in which the reference will resolve, you may supply a nil token. However, this is only valid as a temporary state. The token must be fixed up by the time the Runtime loader 'sees' this TypeRef. One example where this is used is when VC compiles separate .cpp files into separate .obj files. The Linker 'joins' them together into one image (.dll or .exe file) – as part of that process, it calls metadata Merge code with optimizes these nil-scoped TypeRefs to be replaced by the corresponding TypeDef. This 'trick' does not work if the TypeRef would have to resolve outside the merged image

### 3.5.2 DefineImportType

```
HRESULT DefineImportType(IMetaDataAssemblyImport *pAssemImport,
        const void *pbHashValue, ULONG cbHashValue,
        IMetaDataImport *pImport,
        mdTypeDef tdImport, IMetaDataAssemblyEmit *pAssemEmit,
        mdTypeRef *ptr)
```

Defines a reference to a type that exists in another module or assembly. The method looks up the *tdImport* token in that other module, specified via a combination of *pAssemImport*, *pbHashValue*, *cbHashValue* and *pImport*, and retrieves its properties. It uses this information to define a TypeRef in the current scope.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | pAssemImport | Assembly scope containing the tdImport TypeDef | yes |
| in | pbHashValue | Blob holding hash for assembly pAssemImport | yes |
| in | cbHashValue | Count of bytes in pbHashValue | yes |
| in | pImport | Metadata scope (module) holding target Type | yes |
| in | tdImport | TypeRef token for target Type within pImport scope | yes |
| in<br>out | pAssemEmit<br>ptr | Assembly scope for output<br>TypeRef token assigned | yes |

### 3.5.3 DefineMemberRef

```
HRESULT DefineMemberRef(mdToken tkImport, LPCWSTR wzName,
        PCCOR_SIGNATURE pvSig, ULONG cbSig, mdMemberRef *pmr)
```

Defines a reference to a member (field, method, global-variable, global-function) that exists in another module. This method does not look up that other module; so the compiler takes on responsibility to ensure the MemberRef will bind successfully at runtime.

You specify the member you are interested in by giving its name (*wzName*), its signature (*pvSig, cbSig*), and the a reference to the class or interface in that other module , for its class or interface (*tkImport*). If the target member is a global-variable or global-function, then *tkImport* must be the *mdModuleRef* token for that module.

You obtain the *tkImport* token from a previous call to *DefineTypeRefByName*, *DefineImportType*, or *DefineModuleRef*.

You can specify *tkImport* as *mdTokenNil*. This indicates that the imported member's parent will be resolved later by the compiler or linker (the typical scenario is when a global function or data member is being imported from a .obj file that will ultimately be linked into the current module and the metadata merged). Ultimately, all *MemberRefs* must be fully-resolved to have a consistent, loadable module.

Note: every member reference must have a reference scope that is one of:

- TypeRef token, if member is referenced on an imported type
- ModuleRef token, if member is a global-variable or global-function
- MethodDef token, if member is a call site signature for a vararg method defined in the same module
- TypeSpec token, if member is a member of a constructed type (eg an array)

Note too: as an optimization (see Metadata Optimizations), tkImport may be an mdMethodDef, if the reference is not really an import but is simply a callsite reference that could not be optimized away. This can occur when a call is made to a vararg function where additional arguments are passed on the call. In this case, we can't just optimize the MemberRef away if we otherwise could (see Metadata Optimizations for details), but at the same time there is no need to incur the extra runtime overhead to do a full resolution when the resolution may be early bound. So, we persist the "parent" of the MemberRef as the MethodDef token of the method declaration and the MemberRef is called "fully resolved."

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | tkImport | Token for the target member's class or interface. Or, if the member is global, the ModuleRef for that other file | yes |
| in | wzName | Name of the target member | yes |
| in | pvSig | Signature of the target member | yes |
| in | cbSig | Count of bytes in pvSig | yes |
| out | pmr | MemberRef token assigned | |

*tkImport* must be one of *mdTypeRef*, *mdModuleRef*, *mdMethodDef* or *mdTypeSpec*, or nil. In the latter case, we look up a the function declared global in the current scope.

## 3.5.4 DefineImportMember

```
HRESULT DefineImportMember(IMetaDataAssemblyImport *pAssemImport,
        const void *pbHashValue, ULONG cbHashValue,
        IMetaDataImport *pImport,
        mdToken mbMember, IMetaDataAssemblyEmit *pAssemEmit,
        mdToken tkParent, mdMemberRef *pmr)
```

Defines a reference to a member (field, method), global-variable or global-function, that exists in another module.

Generally, before you create a MemberRef to any member in that other module, you need to create a TypeRef for its enclosing class or module, that *parallels* its enclosing class or module in the other module. It is this enclosing TypeRef of MemberRef that you supply as the *tkParent* argument. So:

- If the target member is a field or method, then you must create a TypeRef, in the current scope, for its enclosing class; do this with a call to *DefineTypeRefByName* or *DefineImportType*
- If the target member is a global-variable or global-function (ie not a member of any class or interface), then you must create a ModuleRef, in the current scope, for that other module; do this with a call to *DefineModuleRef*.

There is one exception to having to supply a valid TypeRef or ModuleRef for the *tkParent* argument: if the enclosing class, interface or module will be resolved later by the compiler or linker, then supply it as mdTokenNil. (The only scenario is when a global-function or global-variable is being imported from a .obj file that will ultimately be linked into the current module and the metadata merged).

The method looks up the *mbMember* token in that other module, specified by *PImport,* and retrieves its properties. It uses this information to call the *DefineMemberRef* method, in the current scope.

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | pAssemImport | Assembly scope containing the tdImport TypeDef | no |
| in | pbHashValue | Blob holding hash for assembly pAssemImport | no |
| in | cbHashValue | Count of bytes in pbHashValue | no |
| in | pImport | Metadata scope (module) holding target Type | yes |
| in | mbMember | MethodDef or FieldDef token for target member within pImport scope | yes |
| in | pAssemEmit | Assembly scope for output | no |
| in | tkParent | TypeRef or ModuleRef token for the class that owns the target member member | yes |
| out | ptr | TypeRef token assigned | |

*mdMember* is an mdFieldDef, mdMethodDef or mdProperty

### 3.5.5 DefineModuleRef

```
HRESULT DefineModuleRef(LPCWSTR wzName, mdModuleRef *pmur)
```

Defines a reference to another module. Note that the method does not check whether the specified external module actually exists.

*wzName* should be a file name and extension – but no drive letter or file path. For example, “c:\MyApp\Widgets.dll” is wrong – use “Widgets.dll”

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|

| | | | |
|---|---|---|---|
| in | wzName | Name of the other metadata file. Typically, a DLL | yes |
| out | pmur | ModuleRef token assigned | |

### 3.5.6 SetParent

```
HRESULT SetParent(mdMemberRef mr, mdToken tk)
```

Sets the parent of a MemberRef to a new value. This method is typically used by a compiler or tool (like VC) that emits individual .obj files, each with its own metadata; these .obj files are later merged into a single image. This method is used to fix up module import scopes.

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | mr | The MemberRef token to be re-parented | yes |
| in | tk | Token for the new parent | yes |

The parent token (*tk*) may be any of *mdTypeRef*, *mdModuleRef*, *mdMethodDef*, *mdTypeDef* or *mdTokenNil*

## 3.6 Declaring Events and Properties

### 3.6.1 DefineProperty

```
HRESULT DefineProperty(mdTypeDef td, LPCWSTR wzProperty,
        DWORD dwPropFlags, PCCOR_SIGNATURE pvSig, ULONG cbSig,
        DWORD dwDefType, void const *pValue, ULONG cchValue,
        mdMethodDef mdSetter, mdMethodDef mdGetter,
        mdMethodDef rmdOtherMethods[], mdFieldDef fdBackingField,
        mdProperty *pmdProp)
```

A *property* is like a field within a class. But instead of accessing the value stored in that field location, a property can execute set/get code. You might use this, for example, to range-check a value before setting the property; but the code can also be as complex as the developer chooses. A language may choose to have users write syntax that looks like regular field access (x = foo.prop) but execute property accessor code, 'behind the scenes'.

Examples of using properties include:

- enhanced UI semantics, by presenting the object's state as the values of its properties and allowing the user to manipulate state by changing the values through the UI
- enhanced language support, by abstracting a notion of a property name/identifier that can be used in lieu of explicit method invocation in assignment statements and expressions
- rich infrastructure services such as transparent persistence for properties that are tagged as being part of the persistent state of the object

A property is defined, using *DefineProperty*, in a similar way to how you would define a method of a class. As for a method, you specify the property by giving its owner, name, type, and formal parameter list. For indexed properties, the property can be said to have a signature that is its return type plus the types of its parameters.

You can define more than just setter and getter methods for a property. Simply provide their tokens in the rmdOtherMethods[] array.

In this version of the runtime, there is no built-in support for properties *at runtime*. That's to say, compilers that provide properties must resolve any compile-time reference to a property into its corresponding method invocation; the metadata provides the information necessary for the compiler to do that resolution. In support of dynamic invocation, the Reflection APIs provide this same feature, to resolve property-to-method.

You can use this method to save a default value for the property, via the dwDefType, pValue and cchValue parameters – see 1.5.3 for details.

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | td | Token for class or interface on which property is being defined | yes |
| in | wzProperty | Name of property | yes |
| in | dwPropFlags | Property flags | yes |
| in | pvSig | Property signature | yes |
| in | cbSig | Count of bytes in pvSig | yes |
| in | dwDefType | Type of property's default value | no |
| in | pValue | Default value for property | no |
| in | cchValue | Count of (Unicode) characters in pValue | no |
| in | mdSetter | Method that sets the property value | no |
| in | mdGetter | Method that gets the property value | no |
| in | rmdOtherMethods[] | Array of other methods associated with the property. Terminate array with an mdTokenNil. | no |
| in | fdBackingField | Field on the same enclosing class or interface that backs the property | no |
| out | pmdProp | Property token assigned | |

dwPropFlags is drawn from the CorPropertyAttr enum in CorHdr.h.

### 3.6.2 SetPropertyProps

```
HRESULT SetPropertyProps(mdProperty pr, DWORD dwPropFlags,
        DWORD dwDefType, void const *pValue, ULONG cchValue,
        mdMethodDef mdSetter, mdMethodDef mdGetter,
        mdMethodDef rmdOtherMethods[], mdFieldDef fdBackingField)
```

Sets the information stored in metadata for a property, previously defined with a call to *DefineProperty*.

You can use this method to save a default value for the property, via the dwDefType, pValue and cchValue parameters – see 1.5.3 for details.

If you supply a value for any optional argument, that value will supersede the previous, supplied to *DefineProperty*. If you want to leave the original value unchanged, mark the argument as "to be ignored" – see section 1.5.5 for details.

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | pr | Token for property to be changed | yes |
| in | dwPropFlags | Property flags | yes |
| in | dwDefType | Type of property's default value | no |
| in | pValue | Default value for property | no |
| in | cchValue | Count of (Unicode) characters in pValue | no |
| in | mdSetter | Method that sets the property value | no |
| in | mdGetter | Method that gets the property value | no |
| in | rmdOtherMethods[] | Array of other methods associated with the property. Terminate array with an mdTokenNil. | no |
| in | fdBackingField | Field on the same enclosing class or interface that backs the property | no |

dwPropFlags is drawn from the CorPropertyAttr enum in CorHdr.h.

### 3.6.3 DefineEvent

An event is treated in metadata in a similar manner to a property – as a collection of methods defined upon a class or interface. But runtime provides no support for events: the compiler must translate all references to events into calls to the appropriate method.

```
HRESULT DefineEvent(mdTypeDef td, LPCWSTR wzEvent,
        DWORD dwEventFlags, mdToken tkEventType, mdMethodDef mdAddOn,
        mdMethodDef mdRemoveOn, mdMethodDef mdFire,
        mdMethodDef rmdOtherMethod[], mdEvent *pmdEvent)
```

Defines an event source for a class or interface.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | td | Token of target class or interface | yes |
| in | wzEvent | Name of event | yes |
| in | dwEventFlags | Event flags | no |
| in | tkEventType | Token for the Event class | yes |
| in | mdAddOn | Method used to subscribe to the event, or nil | yes |
| in | mdRemoveOn | Method used to unsubscribe to the event, or nil | yes |
| in | mdFire | Method used (by a subclass) to fire the event | yes |
| in | rmdOtherMethods[] | Array of tokens for other methods associated with the event | no |
| out | pmdEvent | Event token assigned | |

*td* must be an mdTypeDef or mdTypeDefNil

*wzEvent* specifies the name of the event.  This must be unique across all event names that the class or interface exposes

*dwEventFlags* is drawn from the CorEventAttr enum in CorHdr.h

*tkEventType* must be an mdTypeDef, mdTypeRef or nil

*mdAddOn, mdRemoveOn* and *mdFire* must each be an mdMethodDef, mdMethodRef or nil

*rmdOtherMethods []* must each be an mdMethodDef, mdMethodRef.  Terminate the array with an mdMethodDefNil token.

### 3.6.4 SetEventProps

```
HRESULT DefineEvent(mdEvent ev, DWORD dwEventFlags,
        mdToken tkEventType, mdMethodDef mdAddOn,
        mdMethodDef mdRemoveOn, mdMethodDef mdFire,
        mdMethodDef rmdOtherMethod[])
```

Changes the properties of an existing event.  See *DefineEvent* for more information.

If you supply a value for any argument, it will supersede the value you supplied in the earlier call to *DefineEvent*.  If you want to leave the original value unchanged, mark that argument as "to be ignored" – see section 1.5.5 for details.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | ev | Event token | yes |
| in | dwEventFlags | Event flags | no |
| in | tkEventType | Token for the Event class | no |
| in | mdAddOn | Method used to subscribe to the event, or nil | no |
| in | mdRemoveOn | Method used to unsubscribe to the event, or nil | no |
| in | mdFire | Method used (by a subclass) to fire the event | no |
| in | rmdOtherMethods[] | Array of tokens for other methods associated with the event | no |

## 3.7 Specifying Layout Information for a Class

### 3.7.1 SetClassLayout

```
HRESULT SetClassLayout (mdTypeDef td, DWORD dwPackSize,
        COR_FIELD_OFFSET rFieldOffsets[], ULONG ulClassSize)
```

Sets the layout of fields for an existing class.

The original definition of the class, made by a call to *DefineTypeDef,* marked it as having one of three layouts: tdAutoLayout, tdLayoutSequential or tdExplicitLayout. Normally, you would specify tdAutoLayout, and let the runtime choose how best to lay out the fields for objects of that class; for example, changing their order might result in faster garbage collection.

However you may want objects of a class laid out in-memory to match how unmanaged code would have done that; in this case, choose tdLayoutSequential or tdExplicit layout; and call SetClassLayout to complete the layout information, as follows:

- tdLayoutSequential – specify the packing size between adjacent fields. Must be 1, 2, 4, 8 or 16 bytes. (A field will be aligned to its natural size, or to the packing size, whichever results in the *smaller* offset)
- tdExplicitLayout – specifiy the offsets, at which each field starts. Or specify the overall size (and optionally, the packing size)

Note that you can use this method to define unions (where multiple fields have the same offset within the class)

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | td | Token for the class being laid out | yes |
| in | dwPackSize | Packing size: 1, 2, 4, 8 or 16 bytes | no |
| in | rFieldOffsets | Array of mdFieldDef / ululByteOffset values for each field on the class for which sequence or offset information is specified. Terminate array with mdTokenNil. | no |
| in | ulClassSize | Overall size of these class objects, in bytes | no |

The COR_FIELD_OFFSET is a simple struct with two fields: an mdFieldDef to define the field, and a ULONG to specify the byte offset from the start of the object, at which this field should start (offsets start at zero).

## 3.8 Miscellaneous

### 3.8.1 GetTokenFromSig

```
HRESULT GetTokenFromSig(PCCOR_SIGNATURE pvSig, ULONG cbSig,
        mdSignature *pmsig)
```

Stores a signature into the Blob heap, returning a metadata token that can be used to reference it later. That token represents an index into the *StandAloneSig* table. This method creates new entries in metadata, so its name is perhaps misleading – you might think of it instead as being "DefineStandAloneSig"

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | pvSig | Signature to be persisted stored | yes |
| in | cbSig | Count of bytes in pvSig | yes |
| out | pmsig | Signature token assigned | |

### 3.8.2 GetTokenFromTypeSpec

```
HRESULT GetTokenFromTypeSpec(PCCOR_SIGNATURE pvSig, ULONG cbSig,
        mdTypeSpec *ptypespec)
```

Stores a type specification into the Blob heap, returning a metadata token that can be used to reference it later. That token represents an index into the *TypeSpec* table. This method creates new entries in metadata, so its name is perhaps misleading – you might think of it instead as being "DefineTypeSpec"

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | pvSig | Signature being defined | yes |
| in | cbSig | Count of bytes in pvSig | yes |
| out | ptypespec | TypeSpec token assigned | |

### 3.8.3 DefineUserString

```
HRESULT DefineUserString(LPCWSTR wzString, ULONG cchString,
        mdString *pstk)
```

Stores a user string into the UserString heap in metadata, returning a token that can be used to retrieve it later. This token is unlike any other in metadata – it is not an index for a row in a metadata table – its lower 3 bytes are the actual byte offset within the UserString heap at which the string is stored.

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | wzString | User string to store | yes |
| in | cchString | Count of (wide) characters in wzString | yes |
| out | pstk | String token assigned | |

## 3.8.4 DeleteToken

```
HRESULT DeleteToken(mdToken tk)
```

Deletes the specified token from the current metadata scope. The only sorts of token you can delete are: TypeDef, MethodDef, FieldDef, Event, Property, ExportedType and CustomAttribute.

This support is for EditAndContinue and incremental-compilation scenarios – where a compiler wants to make a small change to the metadata, without re-emitting all of it again. The information identified by the token is not physically erased (an expensive operation that would require a token remap). Instead, they are marked 'deleted' – we set the xxRTSpecialName bit in their attributes flag, and append "_Delete" to their name. For CustomAttribute, their parent is set to nil.

Compilers who use this method take on responsibility for dealing with any inconsistencies it makes in the metadata (eg live references to these deleted tokens)

In order to use this method, you must first call SetOption, specifying the MetaDataSetUpdate guid, and setting the MDUpdateIncremental flag. You must then open reopen the scope in read-write mode.

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | tk | Token to delete | yes |

## 3.8.5 DefineSecurityAttributeSet

```
HRESULT DefineSecurityAttributeSet(mdToken tkObj,
        COR_SECATTR rSecAttrs[],
        ULONG cSecAttrs,
        ULONG *pulErrorAttr)
```

Defines a collection of security permissions, attached to the item whose metadata token is *tkObj*.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | tkObj | Token to which the security info is attached | yes |
| in | rSecAttrs[] | Array of COR_SECATTR structs | yes |
| in | cSecAttrs | Number of elements in rSecAttrs[] | yes |
| out | pulErrorAttr | If method fails, specifies the index in rSecAttrs[] of element that caused the problem | yes |

*tkObj* must be an mdtTypeDef, a mdtMethodDef or an mdtAssembly

*rSecAttrs[]* is an array or COR_SECATTR structs.  These, in turn, are defined in CorHdr.h, as follows:

```
typedef struct COR_SECATTR {
  mdMemberRef tkCtor;             // Ref to constructor of security attribute
  const void  *pCustomAttribute;  // Blob describing ctor args and field/property values
  ULONG       cbCustomAttribute;  // Length of the above blob
} COR_SECATTR;
```

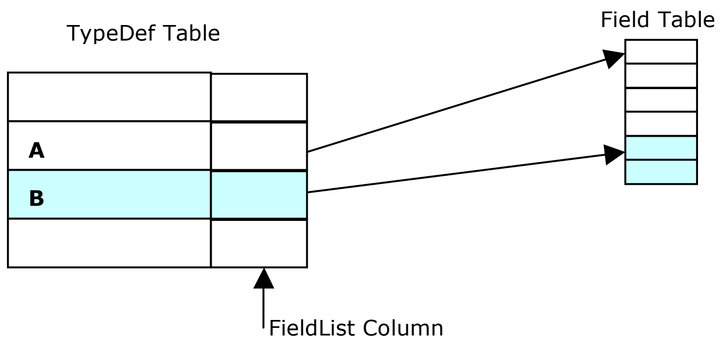
Each element of the array defines a custom attribute blob that describes one corresponding Security Attribute (see section 11)

## 3.9 Order of Emission

If you call the methods in the IMetaDataEmit interface in a certain order, then the metadata engine can store the resulting data in a compact form.  If you break these ordering constraints, then everything still works, but the metadata engine has to introduce intermediate 'map' tables – these take up more space in the stored PE file, and are slower to query at runtime.  If you emit definitions in the following order, you avoid these intermediate 'map' tables --

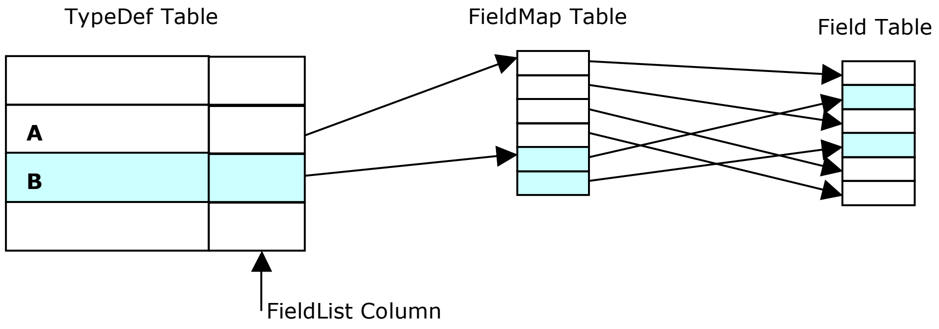
- Emit global functions and fields first
- If you emit TypeDef-A before TypeDef-B, then emit MethodDefs, FieldDefs, Properties, and Events of TypeDef-A before those of TypeDef-B
- If you emit MethodDef-A before MethodDef-B, then emit any Parameters for MethodDef-A before those of MethodDef-B

The reason for these rules is illustrated by the picture below –



Each time you call DefineTypeDef, the metadata engine stores the information you supply into the next row of the TypeDef table. The picture shows a row for Type-A, and one for Type-B. Similarly, each time you call DefineField, the metadata engine stores the information you supply into the next row of the Field table. The TypeDef table includes a column called FieldList that points to the first field for that type. This picture shows what happens if you define in the order – Type-A, Type-B, Field-A-1 thru Field-A-4, Field-B-1, Field-B-2. With this ordering, the fields owned by each Type lie in a contiguous run in the Field table.

The next picture shows what happens if you interleave the definitions –



Here, the order of definition was: Type-A, Type-B, Field-A-1, Field-B-1, Field-A-2, Field-B-2, Field-A-3, Field-A-4. The metadata engine creates an intermediate FieldMap table – as far as the TypeDef table is concerned, it looks like all the Fields for Type-A lie in a contiguous run – but their order in the Field table is not contiguous.

Global functions and fields are parented by an artificial Type created by the metadata engine (it's called <Module> and is always the first row in the TypeDef table) – in all other respects, for ordering rules, they behave like members of a genuine Type.

Hopefully, this simple picture makes the ordering constraints easy to understand. Just as you should emit a Type's Fields so they lie in a contiguous run, the same holds true for each Type's Methods, Events and Properties. Similarly, for each Method, emit its Parameters so they also lie in a contiguous run.

Note that there's no other ordering constraint omitted by the above rules. In particular, for our example, there's no ordering constraint between definition of Type-A's fields, and definition of Type-B. To be absolutely clear, the following orders are all good (we abbreviate Type-A to tA, and Field-A-1 to fA1, etc) –

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| tA | tB | fA1 | fA2 | fA3 | fA4 | fB1 | fB2 |
| tA | fA1 | tB | fA2 | fA3 | fA4 | fB1 | fB2 |
| tA | fA1 | fA2 | tB | fA3 | fA4 | fB1 | fB2 |
| tA | fA1 | fA2 | fA3 | tB | fA4 | fB1 | fB2 |
| tA | fA1 | fA2 | fA3 | fA4 | tB | fB1 | fB2 |

Note that, you can choose to emit definitions interleaved, but then have the metadata engine remove these intermediate 'map' tables before saving the metadata to disk. This involves moving table rows to make them contiguous – but since these row numbers, or RIDs, make up the corresponding metadata tokens, this results in a remapping of tokens already assigned. If you want to do this, you must call SetHandle (as explained earlier) to register for these token remap ‘events’ – you must then fix up any affected tokens that you already generated into your MSIL code stream. (most compilers jump thru any hoops they can to avoid doing this!)

# 4 MetaDataImport

The import interface is used to consume an existing metadata section from a PE file or other source (eg stand-alone runtime metadata binary or type library). The design of these interfaces is intended primarily for tools/services that will be importing type information (eg development tools) or managing deployed components (eg resolution/activation services). The following groups of methods are defined:

- Enumerating collections of items in the metadata scope
- Finding a specific item with a specific set of characteristics
- Getting properties of a specified item
- Resolving import references

## 4.1 Enumerating Collections

To use the EnumXXX methods, you allocate an array to hold the results, then call the required EnumXXX method. Of course, there might be more entries in the table than your array can hold. Just keep calling EnumXXX until eventually the count argument returns zero. Then tidy off by calling the CloseEnum method.

You can determine how many items are in the collection ahead of time by calling the CountEnum method.

Example:

```
const int siz = 5;     // size of array
HCORENUM  enr = 0;     // enumerator
mdTypeDef toks[siz];   // array to hold returned tokens
ULONG     count;       // count of tokens returned
HRESULT   h;
h = pImp->EnumTypeDefs(&enr, toks, siz, &count);
while(count > 0) {
    for(int i = 0; i < count; i++) cout << toks[i] << " ";
    h = pImp->EnumTypeDefs(&enr, toks, siz, &count);
}
pImp->CloseEnum(enr);
```

In this example, pImp is the IMetaDataImport pointer returned from a previous call to OpenScope. (We have omitted error handling to keep the example simple)

Note: When enumerating collections of members for a class, EnumMembers returns only members defined directly on the class: it does not return any members that the class inherits, even if it provides an implementation for those inherited members. To enumerate those inherited members, the caller must explicitly walk the inheritance chain (the rules for which may vary depending upon the language/compiler that emitted the original metadata).

### 4.1.1 CloseEnum Method

```
void CloseEnum(HCORENUM hEnum)
```

Frees the memory previously allocated for the enumeration.   Note that the hEnum argument is that obtained from a previous EnumXXX call (for example, EnumTypeDefs)

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | hEnum | Handle for the enumeration you wish to close | yes |

### 4.1.2 CountEnum Method

```
HRESULT CountEnum(HCORENUM hEnum, ULONG *pulCount);
```

Returns the number of items in the enumeration.  Note that the hEnum argument is that obtained from a previous EnumXXX call (for example, EnumTypeDefs).

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | hEnum | Handle for the enumeration of interest | yes |
| out | pulCount | Count of items in the enumeration | |

### 4.1.3 ResetEnum

```
HRESULT ResetEnum(HCORENUM hEnum, ULONG ulPos);
```

Reset the enumeration to the position specified by pulCount.  So, if you reset the enumeration to the value 5, say, then a subsequent call to the corresponding EnumXXX method will return items, starting at the 5th (where counting starts at item number zero).  Note that the hEnum argument is that obtained from a previous EnumXXX call (for example, EnumTypeDefs)

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| out | hEnum | Handle for the enumeration of interest | yes |
| in | ulPos | Item number to reset to | yes |

### 4.1.4 IsValidToken

```
BOOL IsValidToken(mdToken tk)
```

Returns true if tk is a valid metadata token in the current scope.  [The method checks the token type is one of those in the CorTokenType enumeration in CorHdr.h, and then that its RID is less than or equal to the current count of those token types]

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | tk | Metadata token | yes |

## 4.1.5 EnumTypeDefs

```
HRESULT EnumTypeDefs(HCORENUM *phEnum, mdTypeDef rTokens[],
        ULONG cTokens, ULONG *pcTokens)
```

Enumerates all TypedDefs within the current scope.  Note: the collection will contain Classes, Interfaces, etc, as well as any TypeDefs added via an extensibility mechanism.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| inout | phEnum | Enumeration handle.  Must be 0 on first call | yes |
| out | rTokens [] | Array to hold the returned tokens | |
| in | cTokens | Size of rTokens [] array | yes |
| out | pcTokens | Number of  tokens actually returned | |

## 4.1.6 EnumInterfaceImpls

```
HRESULT EnumInterfaceImpls(HCORENUM *phEnum, mdTypeDef td,
        mdInterfaceImpl rTokens[], ULONG cTokens, ULONG *pcTokens)
```

Enumerates all interfaces implemented by the specified TypeDef.  Tokens will be returned in the order the interfaces were specified (through *DefineTypeDef* or *SetTypeDefProps*).

[See *GetInterfaceImplProps* for more detail of how this method works]

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| inout | phEnum | Enumeration handle.  Must be 0 on first call | yes |
| in | td | Token specifying the TypeDef whose InterfaceImpls are required | yes |
| out | rTokens [] | Array to hold returned tokens | |
| in | cTokens | Size of rTokens [] array | yes |
| out | pcTokens | Number of tokens actually returned | |

## 4.1.7 EnumMembers

```
HRESULT EnumMembers(HCORENUM *phEnum, mdTypeDef cl,
        mdToken rTokens[], ULONG cTokens, ULONG *pcTokens)
```

Enumerates all members (fields and methods, but **not** properties or events) defined by the class specified by *cl*.  This does not include any members inherited by that class; even in the case where this TypeDef actually implements an inherited method.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| inout | phEnum | Enumeration handle. Must be 0 on first call | yes |
| in | cl | TypeDef for the class whose members are required | yes |
| out | rTokens[] | Array to hold returned tokens | |
| in | cTokens | Size of rTokens [] array | yes |
| out | pcTokens | Number of tokens actually returned | |

The tokens returned in the *rTokens[]* array will be of mdMethodDefs or mdFieldDefs

### 4.1.8 EnumMembersWithName

```
HRESULT EnumMembersWithName(HCORENUM *phEnum, mdTypeDef cl,
        LPCWSTR wzName, mdToken rTokens[], ULONG cTokens,
        ULONG *pcTokens)
```

Enumerates all members (fields and methods, but **not** properties or events) defined by the specified TypeDef, and that also have the specified name. This does not include any members inherited by the TypeDef; even in the case where this TypeDef actually implements an inherited method. This method is like calling EnumMembers, but discarding all tokens except those corresponding to the specified name.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| inout | phEnum | Enumeration handle. Must be 0 on first call | yes |
| in | cl | TypeDef for the class whose members are required | yes |
| in | wzName | Name of members required. | no |
| out | rTokens [] | Array to hold returned tokens | |
| in | cTokens | Size of rTokens [] array | yes |
| out | pcTokens | Number of tokens actually returned | |

The tokens returned in the *rTokens[]* array will be mdMethodDefs or mdFieldDefs

### 4.1.9 EnumMethods

```
HRESULT EnumMethods(HCORENUM *phEnum, mdTypeDef cl,
        mdMethodDef rTokens[], ULONG cTokens, ULONG *pcTokens)
```

Enumerates all methods defined by the specified TypeDef. Tokens are returned in the same order they were emitted. If you supply a nil token for the *cl* argument the method will enumerate the global functions defined for the module as a whole.

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| inout | phEnum | Enumeration handle. Must be 0 on first call | yes |
| in | cl | Token specifying the TypeDef whose methods are required | no |
| out | rTokens [] | Array to hold returned tokens | |
| in | cTokens | Size of rTokens [] array | yes |
| out | pcTokens | Number of tokens actually returned | |

## 4.1.10EnumMethodsWithName

```
HRESULT EnumMethodsWithName(HCORENUM *phEnum, mdTypeDef cl,
        LPCWSTR wzName, mdMethodDef rTokens[], ULONG cTokens,
        ULONG *pcTokens)
```

Enumerates all methods defined by the specified TypeDef (*cl*), and that also have the specified name (*wzName*). This method is like calling *EnumMethods*, but discarding all tokens except those corresponding to the specified name.

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| inout | phEnum | Enumeration handle. Must be 0 on first call | yes |
| in | cl | TypeDef for the class whose methods are required | yes |
| in | wzName | Name of methods required | no |
| out | rTokens [] | Array to hold returned tokens | |
| in | cTokens | Size of rTokens [] array | yes |
| out | pcTokens | Number of tokens actually returned | |

Note that supplying a nil token for the *cl* parameter will enumerate only the global functions with that name defined for the module as a whole.

## 4.1.11EnumUnresolvedMethods

```
HRESULT EnumUnresolvedMethods(HCORENUM *phEnum,
        mdMethodDef rTokens[], ULONG cTokens, ULONG *pcTokens)
```

Enumerates all methods in the current scope that have been declared but are not implemented.

The enumeration excludes all methods defined at modules scope (globals), or those defined on Interfaces or Abstract classes. Beyond those, for each method marked miForwardRef, it is included into the "unresolved" enumeration if either:

- mdPinvokeImpl = 0
- miRuntime = 0

Put another way, "unresolved" methods are class methods marked miForwardRef but which are not implemented in unmanaged code (reached via PInvoke) nor implemented internally by the Runtime itelf

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| inout | phEnum | Enumeration handle.  Must be 0 on first call | yes |
| out | rTokens [] | Array to hold returned tokens | |
| in | cTokens | Size of rTokens [] array | yes |
| out | pcTokens | Number of tokens actually returned | |

## 4.1.12EnumMethodSemantics

```
HRESULT EnumMethodSemantics(HCORENUM *phEnum, mdMethodDef mb,
        mdToken rTokens[], ULONG cTokens, ULONG *pcTokens)
```

Enumerates all semantics for a given method.  (See GetMethodSemantics for how a method's semantics are derived)

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| inout | phEnum | Enumeration handle.  Must be 0 on first call | yes |
| in | md | Token for required method | yes |
| out | rTokens [] | Array to hold returned tokens | |
| in | cTokens | Size of rTokens [] array | yes |
| out | pcTokens | Number of tokens actually returned | |

## 4.1.13EnumFields

```
HRESULT EnumFields(HCORENUM *phEnum, mdTypeDef cl,
        mdFieldDef rTokens[], ULONG cTokens, ULONG *pcTokens)
```

Enumerates all fields defined on a specified TypeDef.  The tokens are returned in the same order as originally emitted into metadata.  If you specify *cl* as nil, the method will enumerate all the global static data members defined in the current scope.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| inout | phEnum | Enumeration handle.  Must be 0 on first call | yes |
| in | cl | Token specifying the TypeDef whose methods are required | yes |
| out | rTokens [] | Array to hold returned tokens | |
| in | cTokens | Size of rTokens [] array | yes |
| out | pcTokens | Number of tokens actually returned | |

## 4.1.14EnumFieldsWithName

```
HRESULT EnumFieldsWithName(HCORENUM *phEnum, mdTypeDef cl,
        LPCWSTR wzName, mdFieldDef rFields[], ULONG cMax,
        ULONG *pcTokens)
```

Enumerates all fields defined by the specified TypeDef (*cl*), and that also have the specified name (wzName).

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| inout | phEnum | Enumeration handle.  Must be 0 on first call | yes |
| in | cl | TypeDef for the class whose fields are required | yes |
| in | wzName | Name of field required | no |
| out | rTokens [] | Array to hold returned tokens | |
| in | cTokens | Size of rTokens [] array | yes |
| out | pcTokens | Number of tokens actually returned | |

Note that supplying a nil token for the *cl* parameter will enumerate any module-global functions with the specified name.

### 4.1.15EnumParams

```
HRESULT EnumParams(HCORENUM *phEnum, mdMethodDef md,
        mdParamDef rTokens[], ULONG cTokens, ULONG *pcTokens)
```

Enumerates all *attributed* parameters for the method specified by *md*.  By *attributed* parameters, we mean those parameters of a method which have been explicitly defined via a call to *DefineParam*

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| inout | phEnum | Enumeration handle.  Must be 0 on first call | yes |
| in | md | MethodDef for the method whose parameters are required | yes |
| out | rTokens [] | Array to hold returned tokens | |
| in | cTokens | Size of rTokens [] array | yes |
| out | pcTokens | Number of tokens actually returned | |

Note that you can find the number of parameters and their types from the signature returned in GetMethodProps

### 4.1.16EnumMethodImpls

```
HRESULT EnumMethodImpls(HCORENUM *phEnum, mdTypeDef td,
        mdToken rBody[], mdToken rDecl[], ULONG cTokens,
        ULONG *pcTokens)
```

Enumerates all MethodImpls in the current scope for the TypeDef specified by *td*

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| inout | phEnum | Enumeration handle. Must be 0 on first call | yes |
| in | td | TypeDef for which MethodImpls are requested | yes |
| out | rBody [] | Array to hold returned tokens for method bodies | |
| out | rDecl [] | Array to hold returned tokens for method declarations | |
| in | cTokens | Size of rBody and rDecl arrays | yes |
| out | pcTokens | Number of tokens actually returned | |

### 4.1.17EnumProperties

```
HRESULT EnumProperties (HCORENUM *phEnum, mdTypeDef td,
        mdProperty rTokens[], ULONG cTokens, ULONG *pcTokens)
```

Enumerates all Property tokens for a specified class, interface or valuetype.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| inout | phEnum | Enumeration handle. Must be 0 on first call | yes |
| in | td | Token for the type whose properties you want | yes |
| out | rTokens [] | Array to hold returned tokens | |
| in | cTokens | Size of rTokens [] array | yes |
| out | pcTokens | Number of tokens actually returned | |

### 4.1.18EnumEvents

```
HRESULT EnumEvents (HCORENUM *phEnum, mdTypeDef td, mdEvent rTokens[],
        ULONG cTokens, ULONG *pcTokens)
```

Enumerates all Event tokens for a specified type

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| inout | phEnum | Enumeration handle. Must be 0 on first call | yes |
| in | td | Token for the type on which the events are defined | yes |
| out | rTokens [] | Array to hold returned tokens | |
| in | cTokens | Size of rTokens [] array | yes |
| out | pcTokens | Number of tokens actually returned | |

### 4.1.19EnumTypeRefs

```
HRESULT EnumTypeRefs(HCORENUM *phEnum, mdTypeRef rTokens[],
        ULONG cTokens, ULONG *pcTokens)
```

Enumerates all TypeRef tokens that are defined in the current scope.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| inout | phEnum | Enumeration handle.  Must be 0 on first call | yes |
| out | rTokens [] | Array to hold returned tokens | |
| in | cTokens | Size of rTokens [] array | yes |
| out | pcTokens | Number of tokens actually returned | |

### 4.1.20EnumMemberRefs

```
HRESULT EnumMemberRefs(HCORENUM *phEnum, mdToken tkParent,
        mdMemberRef rTokens[], ULONG cTokens, ULONG *pcTokens)
```

Enumerates all MemberRef tokens in the current scope for the specified parent. *tkParent* may be a TypeRef, MethodDef, TypeDef, ModuleRef or nil; in the latter case, we return tokens that reference global-fields or global-functions.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| inout | phEnum | Enumeration handle.  Must be 0 on first call | yes |
| in | tkParent | Token of parent | yes |
| out | rTokens [] | Array to hold returned tokens | |
| in | cTokens | Size of rTokens [] array | yes |
| out | pcTokens | Number of tokens actually returned | |

### 4.1.21EnumModuleRefs

```
HRESULT EnumModuleRefs (HCORENUM *phEnum, mdModuleRef rTokens[],
        ULONG cTokens, ULONG *pcTokens)
```

Enumerates all module references in the current scope.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| inout | phEnum | Enumeration handle.  Must be 0 on first call | yes |
| out | rTokens [] | Array to hold the returned tokens | |
| in | cTokens | Size of rTokens [] array | yes |
| out | pcTokens | Number of tokens actually returned | |

### 4.1.22EnumCustomAttributes

```
HRESULT EnumCustomAttributes (HCORENUM *phEnum, mdToken tk,
        mdToken tkType, mdCustomAttribute rTokens[],
        ULONG cTokens, ULONG *pcTokens)
```

Enumerates all custom attributes for a specified owner.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| inout | phEnum | Enumeration handle.  Must be 0 on first call | yes |
| in | tkOwner | Token for owner. | no |
| in | tkType | Token for constructor method, or mdTypeRef, or nil | no |
| out | rTokens [] | Array to hold returned tokens | |
| in | cTokens | Size of rTokens [] array | yes |
| out | pcTokens | Number of tokens actually returned | |

*tkOwner* is the token for the owner – that's to say, the metadata item this custom attribute is attached to.  If you specify *tkOwner* as nil, we enumerate all custom attributes in the scope

If you want to enumerate custom attributes, then supply *tkType* as the mdMethodDef or mdMemberRef token for its constructor method.  *tkType* is used to filter the answer: if specified as null, no filtering is done

### 4.1.23EnumSignatures

```
HRESULT EnumSignatures(HCORENUM *phEnum, mdSignature rTokens[],
        ULONG cTokens, ULONG *pcTokens)
```

Enumerates all stand-alone signatures defined within the current scope, by looking at each row of the *StandAloneSig* table.  These signatures were defined by previous calls to the *GetTokenFromSig* method

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| inout | phEnum | Enumeration handle.  Must be 0 on first call | yes |
| out | rTokens [] | Array to hold the returned tokens | |
| in | cTokens | Size of rTokens [] array | yes |
| out | pcTokens | Number of tokens actually returned | |

### 4.1.24EnumTypeSpecs

```
HRESULT EnumTypeSpecs(HCORENUM *phEnum, mdTypeSpec rTokens[],
        ULONG cTokens, ULONG *pcTokens)
```

Enumerates all TypeSpecs defined within the current scope, by looking at each row of the *TypeSpec* table. These TypeSpecs were previously defined by previous calls to the *GetTokenFromTypeSpec* method

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| inout | phEnum | Enumeration handle. Must be 0 on first call | yes |
| out | rTokens [] | Array to hold the returned tokens | |
| in | cTokens | Size of rTokens [] array | yes |
| out | pcTokens | Number of tokens actually returned | |

### 4.1.25EnumUserStrings

```
HRESULT EnumUserStrings(HCORENUM *phEnum, mdString rTokens[],
        ULONG cTokens, ULONG *pcTokens)
```

Enumerates all user strings stored within the current scope, by scanning the entire UserString heap. These are the strings stored by previous calls to the *DefineUserString* method

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| inout | phEnum | Enumeration handle. Must be 0 on first call | yes |
| out | rTokens [] | Array to hold the returned tokens | |
| in | cTokens | Size of rTokens [] array | yes |
| out | pcTokens | Number of tokens actually returned | |

WARNING: The only scenario in which this method is expected to be used is for a metadata browser, rather than by a compiler

## 4.2 Finding a Specific Item in Metadata

### 4.2.1 FindTypeDefByName

```
HRESULT FindTypeDefByName(LPCWSTR wzName, mdToken tkEncloser,
        mdTypeDef *ptd)
```

Finds the type definition (class, interface, value-type) with the given name. If this is a nested type, supply *tkEncloser* as the TypeDef or TypeRef token for its immediately-enclosing type. If this is not a nested type, supply *tkEncloser* as nil

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | wzName | Name of required type | yes |
| in | tkEncloser | Token for enclosing type, or nil | no |
| out | ptd | Token for type definition | |

## 4.2.2 FindMember

```
HRESULT FindMember(mdTypeDef td, LPCWSTR wzName,
        PCCOR_SIGNATURE pvSig, ULONG cbSig, mdToken *pmd)
```

Finds a specified member (field or method) in the current metadata scope. The member you want is specified by *td* (its enclosing class or interface), *wzName* (its name) and, optionally, its signature (*pvSig*, *cbSig*). If *td* is specified as mdTokenNil, then the lookup is done for a global-variable or global-function. Recall that you may have multiple members with the same name on a class or interface; supply its signature to find the unique match.

*FindMember* only finds members that were defined directly on the class or interface; it does not find inherited members. (*FindMember* is simply a helper method – it first calls FindMethod; if that doesn't find a match, it then calls *FindField*)

The signature passed in to FindMember must have been generated in the current scope. That's because signatures are bound to a particular scope. As discussed later in this spec, a signature can embed a token that identifies the enclosing Class or ValueType (the token is an index into the local TypeRef table). In other words, you cannot build a runtime signature outside the context of the current scope that can be used as input to FindMember.

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | td | Token of enclosing type | yes |
| in | wzName | Name of the required member | yes |
| in | pvSig | Signature of the required member | no |
| in | cbSig | Count of bytes in pvSig | no |
| out | pmd | Token for matching method or field | |

*pmd* can be an mdMethodDef or an mdFieldDef

## 4.2.3 FindMethod

```
HRESULT FindMethod(mdTypeDef td, LPCWSTR wzName,
        PCCOR_SIGNATURE pvSig, ULONG cbSig, mdMethodDef *pmd)
```

Finds a specified method in the current metadata scope. The field you want is specified by *td* (its enclosing class or interface), *wzName* (its name) and optionally, its signature (*pvSig, cbSig*). If *td* is specified as mdTokenNil, then the lookup is done for a global-function.

See *FindMember* description for more details.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | td | Token of enclosing type | yes |
| in | wzName | Name of required method | yes |
| in | pvSig | Signature of the required method | no |
| in | cbSig | Count of bytes in pvSig | no |
| out | pmd | Token for matching method | |

### 4.2.4 FindField

```
HRESULT FindField(mdTypeDef td, LPCWSTR wzName,
        PCCOR_SIGNATURE pvSig, ULONG cbSig, mdFieldDef *pmd)
```

Finds a specified field in the current metadata scope.  The field you want is specified by *td* (its enclosing class or interface), *wzName* (its name) and optionally, its signature (*pvSig, cbSig*).  If *td* is specified as mdTokenNil, then the lookup is done for a global-variable.

See *FindMember* description for more details.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | td | Token of enclosing type | yes |
| in | wzName | Name of required field | yes |
| in | pvSig | Signature of the required field | no |
| in | cbSig | Count of bytes in pvSig | no |
| out | pfd | Token for matching field | |

### 4.2.5 FindMemberRef

```
HRESULT FindMemberRef(mdTypeRef td, LPCWSTR wzName,
        PCCOR_SIGNATURE pvSig, ULONG cbSig, mdMemberRef *pmr)
```

Finds a member reference in the current metadata scope.  The reference you want is specified by *td* (its owner class or interface), *wzName* (its name) and optionally, its signature (*pvSig, cbSig*).  If *td* is specified as mdTokenNil, then the lookup is done for a global-variable or global-function.

The signature passed in to FindMember must have been generated in the current scope.  See *FindMember* description for details.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | td | Token of enclosing type | yes |
| in | wzName | Name of required member reference | yes |
| in | pvSig | Signature of the required member | no |
| in | cbSig | Count of bytes in pvSig | no |
| out | pmr | Token for matching member reference | |

tr must be one of mdTypdDef, mdTypeRef, mdMethodDef, mdModuleRef or mdTypeSpec or mdTokenNil.

### 4.2.6 FindTypeRef

```
HRESULT FindTypeRef(mdToken tkResScope, LPCWSTR wzName, mdTypeRef *ptr)
```

Returns information about an existing type reference

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | tkResScope | Token for scope in which type is defined | no |
| in | wzName | Name of required type | yes |
| out | ptr | TypeRef token returned | |

*tkResScope* may be an mdModuleRef, mdAssemblyRef, mdTypeRef token (this is required to disambiguate, for example, a reference to Type X in Assembly A, from a reference to Type X in Assembly B). If the target type is nested, specify *tkResScope* as the mdTypeRef token for its immediately-enclosing type

## 4.3 Obtaining Properties of a Specified Object

These methods are specifically designed to return single-valued properties of metadata items. When the property is a reference to another item, a token for that item is returned for the property. Any pointer input type can be null to indicate that the particular value is not being requested. To obtain properties that are essentially collection objects (e.g., the collection of interfaces that a class implements), see the earlier section on enumerations.

### 4.3.1 GetScopeProps

```
HRESULT GetScopeProps (LPWSTR wzName, ULONG cchName, ULONG *pchName,
        GUID *pmvid)
```

Gets the properties for the current metadata scope that were set with a previous call to *SetModuleProps*.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| out | wzName | Buffer to hold name of current module. | no |
| in | cchName | Count of characters allocated in wzName buffer | no |
| out | pchName | Actual count of characters returned | |
| out | pmvid | Returned module VID | |

### 4.3.2 GetModuleFromScope

```
HRESULT GetModuleFromScope (mdModule *pModule)
```

Gets the token for the module definition for the current scope.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| out | pModule | Module token returned | |

### 4.3.3 GetTypeDefProps

```
HRESULT GetTypeDefProps (mdTypeDef td, LPWSTR wzTypeDef,
        ULONG cchTypeDef, ULONG *pchTypeDef,
        DWORD *pdwTypeDefFlags, mdToken *ptkExtends)
```

Gets the information stored in metadata for a specified type definition.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | td | Token for required type definition | yes |
| out | wzTypeDef | Name of type. | no |
| in | cchTypedef | Count of characters allocated in wzTypeDef buffer | no |
| out | pchTypedef | Actual count of characters returned | no |
| out | pdwTypeDefFlags | Flags set on type definition. | no |
| out | ptdExtends | Token for superclass. | no |

*pdwTypeDefFlags* is a bitmask from the CorTypeAttr enum in CorHdr.h.
*ptdExtends* is an mdTypeDef or mdTypeRef

### 4.3.4 GetNestedClassProps

```
HRESULT GetNestedClassProps (mdTypeDef tdNested,
        mdTypeDef *ptdEncloser)
```

Gets the enclosing class for a specified nested class.

| in/out | Parameter | Description | Required? |
|---|---|---|---|

| | | | |
|---|---|---|---|
| in | tdNested | Token for required nested class | yes |
| out | ptdEncloser | Token for the enclosing class | yes |

### 4.3.5 GetInterfaceImplProps

```
HRESULT GetInterfaceImplProps (mdInterfaceImpl iImpl,
        mdTypeDef *pClass, mdToken *ptkIface)
```

Gets the information stored in metadata for a specified interface implementation.

Each time you call *DefineTypeDef* or *SetTypeDefProps* to define a type, you can specify which interfaces that class implements, if any. For example, suppose a class has an mdTypeDef token value of 0x02000007. And suppose it implements three interfaces whose types have tokens 0x02000003 (TypeDef), 0x0100000A (TypeRef) and 0x0200001C (TypeDef). Conceptually, this information is stored into an interface implementation table like this:

| Row Number | Class Token | Interface Token |
|---|---|---|
| 4 | | |
| 5 | 02000007 | 02000003 |
| 6 | 02000007 | 0100000A |
| 7 | | |
| 8 | 02000007 | 0200001C |

*GetInterfaceImplProps* will return the information held in the row whose token you provide in the *iImpl* argument. (Recall, the token is a 4-byte value; the lower 3 bytes hold the row number, or RID; the upper byte holds the token type – 0x09 for mdtInterfaceImpl).

[You obtain the value for iImpl by calling the EnumInterfaceImpls method]

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | iImpl | Token for the required interface implementation | yes |
| out | pClass | Token for the class. | no |
| out | ptkIface | Token for the interface that *pClass* implements. | no |

ptkIface will be an mdTypeDef or an mdTypeRef

### 4.3.6 GetCustomAttributeProps

```
HRESULT GetCustomAttributeProps (mdCustomAttribute ca,
        mdToken *ptkOwner, mdToken *ptkType,
        void const **ppBlob, ULONG *pcbBlob)
```

Returns information about a custom attribute.

A custom attribute is stored as a blob whose format is understood by the metadata engine, and by Reflection; essentially a list of argument values to a constructor method which will create an instance of the custom attribute.

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | ca | Token for required custom attribute | yes |
| out | ptkOwner | Token for owner | |
| out | ptkType | Token for custom attribute | no |
| out | ppBlob | Pointer to blob for custom attribute | no |
| out | pcbBlob | Count of bytes in *ppBlob* | no |

*ptk* is the token for the owner – that's to say, the metadata item this custom attribute is attached to.  A custom attribute can be attached to any sort of owner, with the sole exception of an mdCustomAttribute.

If *ca* is a custom attribute, then *ptkType* is the mdMethodDef or mdMemberRef token for its constructor method.

### 4.3.7 GetCustomAttributeByName

```
HRESULT GetCustomAttributeByName (mdToken tdOwner, LPCWSTR wzName,
        const void **ppBlob, ULONG *pcbBlob)
```

Returns the information stored for a custom attribute, where you specify the target by its owner and name.  See GetCustomAttributeProps for more detail.

The name is the name of the attribute class (see Section 3.2.2)

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | tdOwner | Token for owner of custom attribute or custom value | yes |
| in | wzName | Name of custom attribute or custom value | yes |
| out | ppBlob | Pointer to blob for custom attribute or value | yes |
| out | pcbBlob | Count of bytes in *ppBlob* | yes |

Note that it is quite legal to define multiple custom attributes for the same owner; they may even have the same name.  GetCustomAttributeByName returns only one of those multiple instances (in fact, the first it encounters, but that behaviour is not guaranteed).  Use the EnumCustomAttributes if you want to find them all.

### 4.3.8 GetMemberProps

```
HRESULT GetMemberProps(mdToken md, mdTypeDef *pClass, LPWSTR wzName,
   ULONG cchName, ULONG *pchName, DWORD *pdwAttr,
   PCCOR_SIGNATURE *ppSig, ULONG *pcbSig, ULONG *pulCodeRVA,
   DWORD *pdwImplFlags, DWORD *pdwDefType, void const **ppValue, ULONG
   *pcbValue)
```

Gets the information stored in metadata for a specified member definition. This is a simple helper method: if *md* is a MethodDef, then we call *GetMethodProps*; if *md* is a FieldDef, then we call *GetFieldProps*. See these other methods for details.

### 4.3.9 GetMethodProps

```
HRESULT GetMethodProps(mdMethodDef md, mdTypeDef *pClass,
        LPWSTR wzName, ULONG cchName, ULONG *pchName,
        DWORD *pdwAttr, PCCOR_SIGNATURE *ppvSig, ULONG *pcbSig,
        ULONG *pulCodeRVA, DWORD *pdwImplFlags)
```

Retrieves a method definition in the current metadata scope. The method you want is specified by *md* (its MethodDef token).

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | md | Token of required method | yes |
| out | pClass | Token for class in which this method is defined. | no |
| out | wzName | Buffer to hold name of method. | no |
| in | cchName | Count of wide characters in wzName | no |
| out | pchName | Actual count of wide characters copied to wzName | no |
| out | ppvSig | Pointer to signature of the required method. | no |
| out | pcbSig | Count of bytes in ppvSig | no |
| out | pulCodeRVA | RVA for code. | no |
| out | pdwImpleFlags | Implementation flags. | no |

*pdwAttr* is from the CorMethodAttr enum in CorHdr.h.

### 4.3.10GetFieldProps

```
HRESULT GetFieldProps(mdFieldDef fd, mdTypeDef *pClass, LPWSTR wzName,
   ULONG cchMember, ULONG *pchMember, DWORD *pdwAttr, PCCOR_SIGNATURE
   *ppvSig, ULONG *pcbSig, DWORD *pdwDefType, void const **ppValue,
   ULONG *pcbValue)
```

Retrieves the information stored in metadata for a specified field.

| in/out | Parameter | Description | Required? |
|---|---|---|---|

| | | | |
|---|---|---|---|
| in | Fd | Token of required field | yes |
| out | pClass | Token for class on on which the field is defined. | no |
| in | wzName | Buffer to hold name of property. | no |
| in | cchName | Count of wide characters in wzName | no |
| out | pchName | Actual count of wide characters copied to wzName | no |
| out | pdwAttr | Attribute flags. | no |
| out | pdwDefType | ELEMENT_TYPE_* for the constant value. | no |
| out | ppValue | Pointer to the parameter default value. | no |
| out | pcbValue | Count of bytes in *ppValue* | no |

pdwAttr is drawn from the CorFieldAttr enum in CorHdr.h

### 4.3.11GetParamProps

```
HRESULT GetParamProps (mdParamDef pd, mdMethodDef pmd,
        ULONG *pulSequence, LPWSTR wzName, ULONG cchName,
        ULONG *pchName, DWORD *pdwAttr, DWORD *pdwDefType,
        void const **ppValue, ULONG *pcbValue)
```

Retrieves the information stored in metadata for a specified parameter on a method, or global-function.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | pd | Token of required parameter | yes |
| in | pmd | Token for method on which the parameter is defined | no |
| out | pulSequence | Ordinal value of parameter in method signature; 0 indicates the return value | no |
| out | wzName | Buffer to hold name of parameter | no |
| in | cchName | Count of wide characters in wzName | no |
| out | pchName | Actual count of wide characters copied to wzName | no |
| out | pdwAttr | Attribute flags | no |
| out | pdwDefType | ELEMENT_TYPE_* for the constant value | no |
| out | ppValue | Pointer to the parameter default value | no |
| out | pcbValue | Count of bytes in *ppValue* | no |

*pdwAttr* is drawn from the CorParamAttr enum in CorHdr.h

### 4.3.12GetParamForMethodIndex

```
HRESULT GetParamForMethodIndex(mdMethodDef md, ULONG ulParamSeq,
        mdParamDef *ppd)
```

Returns the definition of parameter number *ulParamSeq* for the method (or global-function) whose token is *md.*  A value of 0 for *ulParamSeq* denotes the return value; parameters are numbered starting at 1.

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | md | Token of target method | yes |
| in | ulParamSeq | Ordinal value of parameter in method signature; 0 indicates the return value | no |
| out | ppd | Pointer to the parameter defintion | |

### 4.3.13GetPinvokeMap

```
HRESULT GetPinvokeMap(mdToken tk, DWORD *pdwMappingFlags,
        LPCWSTR wzName, ULONG cchName,
        ULONG *pchName, mdModuleRef *pmrImportDLL)
```

Returns the PInvoke information stored for a given method.  (PInvoke is a Runtime service that supports inter-operation with unmanaged code)

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | tk | Token for the method required | yes |
| out | pdwMappingFlags | Flags stored to describe mapping | yes |
| out | wzName | Buffer to hold name of method in unmanaged DLL | no |
| in | cchName | Count of characters allocated in wzName buffer | no |
| out | pchName | Actual count of characters returned | no |
| out | pmdImportDLL | mdModuleRef token for target unmanaged DLL | no |

*tk* must be a MethodDef token

*dwMappingFlags* is a bitmask from the CorPinvokeMap enum in CorHdr.h

### 4.3.14GetFieldMarshal

```
HRESULT GetFieldMarshal(mdToken tk, PCCOR_SIGNATURE *ppNativeType,
        ULONG *pcbNativeType)
```

Returns the marshaling information for a field, method return, or method parameter (See *SetFieldMarshal* for details)

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | tk | Token for target data item | yes |
| out | ppNativeType | Pointer to the native type signature | |
| out | pcbNativeType | Actual count of bytes in ppNativeType | |

*tk* is an mdFieldDef or mdParamDef

### 4.3.15GetRVA

```
HRESULT GetRVA(mdToken tk, ULONG *pulCodeRVA, DWORD *pdwImplFlags)
```

Returns the code RVA and implementation flags for a given member.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | tk | Token for the required member | yes |
| out | pulRVA | RVA for required member | no |
| out | pdwImplFlags | Implementation flags for required member | no |

*tk* must be one of mdMethodDef mdFieldDef.  In the latter case, the field must be a global-variable

*dwImplFlags* is a bitmask from the CorMethodImpl enum in CorHdr.h (not relevant if *tk* is an mdFieldDef)

### 4.3.16GetTypeRefProps

```
HRESULT GetTypeRefProps(mdTypeRef tr, mdToken *ptkResScope,
        LPWSTR wzName, ULONG cchName, ULONG *pchName)
```

Retrieve information for a type reference in the current metadata scope

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | tr | Token of required method reference | yes |
| out | ptkResScope | Token for resolution scope – a ModuleRef or AssemblyRef | no |
| in | wzName | Buffer to hold name of type | no |
| in | cchName | Count of wide characters in wzName | no |
| out | pchName | Actual count of wide characters copied to wzName | no |

### 4.3.17GetMemberRefProps

```
HRESULT GetMemberRefProps(mdMemberRef mr, mdToken *ptk,
        LPWSTR wzMember, ULONG cchMember, ULONG *pchMember,
        PCCOR_SIGNATURE  *ppSig, ULONG *pcbSig)
```

Returns the information stored in metadata for a specified member reference.

| in/out | Parameter | Description | Required? |
|---|---|---|---|

| | | | |
|---|---|---|---|
| in | mr | Token for required member reference | yes |
| out | ptk | Token for class or interface on which member is defined. | no |
| out | wzName | Buffer to hold name of member. | no |
| in | cchName | Count of wide characters in wzName | no |
| out | pchName | Actual count of wide characters copied into wzName | no |
| out | ppSig | Pointer to signature blob | no |
| out | pcbSig | Count of bytes in ppSig | no |

### 4.3.18GetModuleRefProps

```
HRESULT GetModuleRefProps(mdModuleRef mr, LPWSTR wzName,
        ULONG cchName, ULONG *pchName)
```

Returns the information stored in metadata for a specified module reference.

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | mr | Token for required module reference | yes |
| out | wzName | Buffer to hold name | no |
| in | cchName | Count of characters allocated in wzName buffer | no |
| out | pchName | Actual count of characters returned | no |

### 4.3.19GetPropertyProps

```
HRESULT GetPropertyProps(mdProperty prop, mdTypeDef *pClass,
        LPWSTR wzName, ULONG cchName, ULONG *pchName,
        DWORD *pdwFlags, PCCOR_SIGNATURE *ppSig, ULONG *pbSig,
        DWORD *pdwDefType, const void **ppValue,
        ULONG *pcbValue, mdMethodDef *pmdSetter,
        mdMethodDef *pmdGetter, mdMethodDef rmdOtherMethods[],
        ULONG cMax, ULONG *pcOtherMethods, mdFieldDef *pmdBackingField)
```

Returns information stored in metadata for a specified property.

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | prop | Token of required property | yes |
| out | pClass | Token for type on which the property is defined | no |
| out | wzName | Buffer to hold name of property | no |
| in | cchName | Count of wide characters in wzName | no |
| out | pchName | Actual count of wide characters copied to wzName | no |
| out | pdwFlags | Property flags | no |
| out | ppSig | Pointer to property signature | no |
| out | pbSig | Count of bytes in *ppSig* | no |
| out | pdwDefType | ELEMENT_TYPE_* for the constant value | no |
| out | ppValue | Pointer to the property default value | no |
| out | pcbValue | Count of bytes in *ppValue* | no |
| out | pmdSetter | Token for setter method | no |
| out | pmdGetter | Token for getter method | no |
| out | rmdOtherMethods[] | Array to hold tokens for other property methods | no |
| in | cMax | Count of elements in the *rmdOtherMethods* array | no |
| out | pcOtherMethods | Count of elements filled in *mdOtherMethods* array | no |
| out | pmdBackingFiled | Token for property's backing field | no |

pdwFlags is drawn from the CorPropertyAttr enum in CorHdr.h.

Note that only *cMax* other methods can be returned by this method.  If the property has more methods defined than you provide array space to hold, they are skipped without warning.

### 4.3.20GetEventProps

```
HRESULT GetEventProps(mdEvent ev, mdTypeDef *pClass, LPCWSTR wzEvent,
        ULONG cchEvent, ULONG *pchEvent, DWORD *pdwEventFlags,
        mdToken *ptkEventType, mdMethodDef *pmdAddOn,
        mdMethodDef *pmdRemoveOn, mdMethodDef *pmdFire,
        mdMethodDef rOtherMethods[], ULONG cOtherMethods,
        ULONG *pcOtherMethods)
```

Returns the information previously defined for a given event.  See *DefineEvent* for more information

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | ev | Token of required event | yes |
| out | pClass | Class or interface on which event is defined. | no |
| out | wzEvent | Buffer to hold name of event. | no |
| in | cchEvent | Length of wzEvent in (wide) characters | no |
| out | pchEvent | Number of characters returned into wzEvent. | no |
| out | pdwEventFlags | Event flags. | no |
| out | ptkEventType | Token for the event class | no |
| out | pmdAddOn | Method used to subscribe to the event | no |
| out | pmdRemoveOn | Method used to unsubscribe to the event | no |
| out | pmdFire | Method used (by a subclass) to fire the event | no |
| out | rOtherMethods[] | Array to hold tokens for the event's other methods | no |
| in | cOtherMethods | Size of rOtherMethods array | no |
| out | pcOtherMethods | Number of other methods actually returned | no |

### 4.3.21GetMethodSemantics

```
HRESULT GetMethodSemantics(mdMethodDef md, mdToken tkProp,
        DWORD *pdwSemantics)
```

Returns the semantic flags for a given property.  Note that there is no *Define* method that creates those flags – they are derived from the *DefineProperty* method – for example, if a method was specified as a Getter, then that method's semantic flags would have the msGetter bit set.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | md | Token for required method | yes |
| in | tkProp | Token for required property | yes |
| out | pdwSemantics | Array to hold the method semantics DWORD | |

*pdwSemantics* is drawn from the CorMethodSemanticsAttr in CorHdr.h

### 4.3.22GetClassLayout

```
HRESULT GetClassLayout(mdTypeDef td, DWORD *pdwPackSize,
        COR_FIELD_OFFSET rFieldOffsets[], ULONG cMax,
        ULONG *pcFieldOffsets, ULONG *pulClassSize)
```

Returns the layout of fields for a class, defined by an earlier call to *SetClassLayout*.

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | td | Token for required class | yes |
| out | pdwPackSize | Packing size: 1, 2, 4, 8 or 16 bytes | no |
| out | rOffsets [] | Array to hold the offsets of class fields | no |
| out | cOffsets | Size of rOffsets [] array | no |
| out | pcOffsets | Number of offsets actually returned | no |
| out | pulClassSize | Overall size of the class object, in bytes | no |

See SetClassLayout for more information.

### 4.3.23GetSigFromToken

```
HRESULT GetSigFromToken(mdSignature tkSig, PCCOR_SIGNATURE *ppSig,
        ULONG *pcbSig)
```

Returns the signature for a given standalone-signature token (the *tkSig* parameter effectively indexes a row in the *StandAloneSig* table)

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | tkSig | Token for the required signature | yes |
| out | ppSig | Pointer to required signature blob | |
| out | pchSig | Count of bytes in the signature blob pointed to by *ppSig* | |

### 4.3.24GetTypeSpecFromToken

```
HRESULT GetTypeSpecFromToken(mdTypeSpec typespec,
        PCCOR_SIGNATURE *ppSig, ULONG *pcbSig)
```

Returns the TypeSpec whose token is *typespec* (the *typespec* parameter effectively indexes a row in the *TypeSpec* table)

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | typespec | Token for the required TypeSpec | yes |
| out | ppSig | Pointer to required TypeSpec | |
| out | pchSig | Count of bytes in the TypeSpec pointed to by *ppSig* | |

### 4.3.25GetUserString

```
HRESULT GetUserString(mdString stk, LPWSTR wzString,
        ULONG cchString, ULONG *pchString)
```

Returns the user string, previously stored into metadata (by the DefineUserString method), whose token is *stk*.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | stk | Token for the required string | yes |
| out | wzString | Buffer to hold the retrieved string | no |
| in | cchString | Length of wzString buffer in (wide) characters | no |
| out | pchString | Number of characters returned into wzString | no |

### 4.3.26GetNameFromToken

```
HRESULT GetNameFromToken(mdToken tk, MDUTF8CSTR *pwzName)
```

Returns a pointer, within metadata structures, to the name string for token *tk*. *tk* must be one of mdModule, mdTypeRef, mdTypeDef, mdFieldDef, mdMethodDef, mdParamDef, mdMemberRef, mdEvent, mdProperty, mdModuleRef, else the method will return failure

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | tk | Token to be inspected | yes |
| out | pwzName | Pointer to the token’s name, in UTF8 format | |

### 4.3.27ResolveTypeRef

```
HRESULT ResolveTypeRef(mdTypeRef tr, REFIID riid, IUnknown **ppIScope,
        mdTypeDef *ptd)
```

Resolves a given TypeRef token, by looking for its definition in other modules. If found, it returns an interface to that module scope in *ppIScope*, as well as the type definition token, in that module, for the requested type.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | tr | Token for the type reference of interest | yes |
| in | riid | Interface to return on the target scope | yes |
| out | ppIScope | Returned interface on target scope | |
| out | ptd | Token for a TypeDef | |

*riid* specifies the interface you would like returned for the module that holds the definition of the referenced type. Typically this would be IID_IMetaDataImport; see *OpenScope* for more information

Note: if the TypeRef token to be resolved has a resolution scope of AssemblyRef, then the *ResolveTypeRef* method looks for a match only in those metadata scopes that have already been opened (via calls to *IMetadataDispenserEx::OpenScope*). This is because it has no way to determine, from only the AssemblyRef, exactly where on disk, or in the Global Assembly Cache, that assembly is stored.

# 5 IMetaDataTables

There is a further interface used to query metadata – it is called IMetaDataTables, and is defined in the Cor.h header file. It provides very low-level read-access to metadata information – at the level of the physical tables. The layout of these tables is not guaranteed stable, and may change.

In order to see an example of how to use this API, see the sample code for the "MetaInfo" tool which ships with the .NET SDK. In particular, those code paths corresponding to the "-raw" and "-heaps" command line switches to that tool.

# 6 MethodImpls

## 6.1 Intro

A MethodImpl is a record in MetaData that allows a class to implement two or more inherited methods, whose names and signatures match. For example, class *C* implements interfaces *I* and *J* – both interfaces include a method *int Foo (int)*. How does *C* provide two implementations, one for *I::Foo* and one for *J::Foo*? [The only solution today is for the programmer to avoid the name collision by changing one of *I::Foo* or *J::Foo*]

## 6.2 Details

MethodImpls record a 3-way association among tokens. The three items associated together are:

- class being defined
- method whose body we want to use for the implementation
- method whose MethodTable slot we want to use

For example (all instances of the F function are assumed virtual) --

```
Interface I                  // DefineTypeDef   returns tdI
   int F (int)               // DefineMethod    returns mdFinI
Interface J                  // DefineTypeDef   returns tdJ
   int F (int)               // DefineMethod    returns mdFinJ
class C implements I, J      //  DefineTypeDef  returns tdC
   int F(int) rename I.F select I {...}  //  DefineMethod   returns mdI.F
                                         //  DefineMethodImpl (tdC, mdI.F, mrFinI)
   int F(int) rename J.F select J {...}      //  DefineMethod   returns mdJ.F
                                         //  DefineMethodImpl (tdC, mdJ.F, mrFinJ)
```

MethodImpls are stored in a 3-column table – TypeDef, MethodDef/Ref of body, MethodDef/Ref of the 'owner' of the MethodTable slot. In this example, that second column is a MethodDef that refers to the just-defined method body. So at this stage, it seems we're only *really* using two columns of the MethodImpl table . . .

However, instead of a class providing its own code, it may choose to reuse the code body already supplied, for this method, by a super class. So, let's add a base class B, and change C like this:

```
class B                               // DefineTypeDef   returns tdB
    int F (int) {...}                 // DefineMethod    returns mdFinB

class C extends B implements I, J {       //  DefineTypeDef  returns tdC
    int F(int) rename I.F select I {...}  //  DefineMethod   returns mdI.F
                                          //  DefineMethodImpl (tdC, mdI.F, mrFinI)
    int F(int) rename J.F select J uses B //
                                          //  DefineMethodImpl (tdC, mrFinB, mrFinJ)
```

We don't require a MethodDef for the last method, since we are providing no code. Instead, we *hijack* the code for method F provided by class B.

The prefix *mr* represents a MethodRef token. In the case where I, J, B and C all lie in the same module, then MethodRefs such as mrFinB would be changed to the corresponding MethodDef mdFinB through Ref-to-Def folding (or the compiler might do so itself)

Note that the body referenced in a MethodImpl has two constraints:

- It must be a virtual function. It cannot be for a non-virtual
- It must be implemented by a parent of the current class. You cannot *hijack* arbitrary virtual functions from classes that are unrelated to the current class – even when their name and signature matches what's required.

## 6.3 ReNaming Recommendations

The *I.F, J.F,* etc names in the above examples were invented for illustration. These mangled names might be provided by the user (if the compiler allows), or created automatically by the compiler. Mangling is required *only* to avoid name collisions within the class. With this proposal, the Runtime does not depend in any way upon unmangling to work out what to do – that information is all captured unambiguously in the MetaData tables.

All that said, we recommend that all compilers that target the Runtime adopt the same name mangling scheme. This will make life easier for tools such as browsers, debuggers, profilers, etc.

The suggested scheme is this: that the method *Foo* within class *C* which is going to use the MethodTable slot provided by method *Foo* in interface *IFace* be called *IFace.Foo.* Similarly, if the slot were provided by method *Foo* in base class *BClass*, that the method be called *BClass.Foo*

The prefix should be the fully-qualified Interface or Class name.

We also recommend that compilers mark each MethodDef that has a MethodImpl with mdSpecialName. Doing so alerts browsers that the method has been renamed, or mangled, away from the method it implements.

## 6.4 Notes

- The third column in a MethodImpl **must** be supplied as non-nil. (otherwise, there's a possible loophole that allows a compiler to use MethodImpls to rename an inherited method within a sub-class)

- MethodImpl tokens are not required
- We recommend that compilers use MethodImpls only in the case where there is ambiguity.  So, for example, if interface *J* had no method *int F(int)*, then there is no need to emit a MethodImpl.  Put another way, there is no requirement to emit a MethodImpl for every interface method that a class implements – although this will work, it is discouraged to avoid the consequent bloat in MetaData.
- A given method may have zero, one or more associated MethodImpls.
- A given class may have have multiple methods, all with the same name and signature, for which it can provide code – via the class derivation tree (single), via the interface tree (multiple), or defined within this class itself (single).  As before, wherever there is ambiguity over which vtable slot is to be matched with a given implemenation, compilers should emit a MethodImpl.  This can even be required for the slot inherited from a base class, *B* say, if class *C* itself defines a *virtual int F(int)* asking for a "new slot"
- With this design, there is no benefit in allocating a bit in the MethodImpl flags field of each MethodDef as a hint bit – if set, then the method has an associated MethodImpl?

# 7 NestedTypes

## 7.1 Introduction

This appendix summarizes support for nested types. It explains both how they are stored and retrieved from metadata, and the semantics given them by the runtime. We include examples, and explanation/exploration of what is provided and what not.

## 7.2 Definition

A Type is any of: Class, ValueType, Interface or Delegate. A NestedType is a Type whose definition, in the source language, is lexically enclosed within the definition of anotherTtype. We refer to these two by the names, "nested" Type and "encloser" Type; sometimes as "nestee" and "nester", respectively.

The support provided by the Runtime for nested Types is quite minimal. In essence, metadata provides an extra association for a NestedType – the TypeDef token of its encloser. And, at runtime, the Common Language Runtime (CLR) provides access from nestee methods to members defined within its encloser.

## 7.3 Supported Features

Here is the support provided for NestedTypes, both in metadata, and at runtime, by the CLR:

1. We support NestedTypes, not *inner* types. The distinction is that NestedTypes are only lexically nested; there is no access, by a sort of *this* or *super* pointer, to the enclosing type
2. The layout of an enclosing type is based only on its fields – it is totally unaffected by any Types that it nests. If the language wants the enclosing Type to be, for example, a struct containing a nested struct, then the compiler must emit a field definition into the enclosing type to hold that reference. Note that, whilst metadata preserves the order in which fields are defined, it does not preserve the order for multiple NestedTypes within an encloser
3. The relationship between an enclosing type and a NestedType, with respect to visibility and member access, is the same as that between a Type and its method/field members:
    - A NestedType does not have visibility independent of its enclosing Type. That is:

```
[non-exported] class EnclosingClass   // not visible outside of the assembly
{
    public void Foo() {...}       // visible only to anyone that can see
                                  // EnclosingClass ... that is, only within
                                  // the assembly
    public class NestedClass    {...}  // ditto
}
```

    - An enclosing type may control access to its nested type, by marking a nested type with any of the member access rules: private, family, assembly, assemblyORfamily, assemblyANDfamily, public. The runtime enforces these member access rules

4. A NestedType has access to all members of its enclosing type, without restriction. In this regard, it behaves just like a part of the implementation of the enclosing type. That is:

```
[exported] class EnclosingClass {
    family static int i;
    private static int j;
    public class NestedClass {
        void bar () {
            j = 1;                // OK
            EnclosingClass.j = 1; // OK
            i = 1;                // OK
            EnclosingClass.i = 1; // OK
        }
    }
}
```

5. NestedTypes may be nested arbitrarily deep
6. A NestedType may be subtyped, and may subtype another Type, entirely independently of its nesting hierarchy. For example:

```
class A {
  family static int i;
  class X {
      X() {
          i = 1;     // OK - sets A.i
          A.i = 1;   // OK - same effect as previous line
      }
  }
}
class Y : A.X {
  void foo() {
      i = 1;     // won't compiler - unknown identifier
      A.i = 1;   // won't compile - i not accessible - Y not in the scope of A,
                 // even though Y inherits from A.X
  }
}
class B : A {
  class Z : X {
      void bar () {
          i = 1;     // OK - sets the i field (in base class A)
          A.i = 1;   // OK - allowed since Z, via inheritance from X, is within
                     //      the scope of A
          B.i = 1;   // OK - since B derives from A
      }
  }
}
```

Note that in the *bar* method, all 3 assignments update the same field (*ie* the same cell in memory). As another example, a NestedType could inherit from its enclosing type:

```
public class EnclosingClass {
  public  static int i;
  private static int j;
  private virtual void Foo() { }
  public class NestedClass : EnclosingClass {
      private override void Foo() { }
      void bar () {
          j = 1;           // OK
          this.j = 1;      // OK
          i = 1;           // OK
          this.i = 1;      // OK
      }
  }
}
```

As the above examples illustrate, resolving references through the inheritance chain and through the nesting hierarchy can get complex. The CLR will give precedence to the inheritance chain; if there is no resolution within that chain, then it will traverse the nesting hierarchy.

7. When emitting metadata:
    - A NestedType will be emitted using DefineNestedType:
        - Mark its visibility *nested* (see revised type visibility rules, below).
        - Like any member within a Type, its name must be unique within that Type. Because the Type is a NestedType, it does *not* conflict with any module-level Type of the same name. There is no need for compilers to mangle the name of the nestee in order to make it unique at module-level. The runtime loader will take account of the Type's *nested* status when it comes to find the right Type to load.
        - The definition of a NestedType **must** occur in the same module as that of its encloser
        - In the current implementation, metadata actually preserves the order of emitted TypeDefs; however, tools should not rely that metadata enumerations will return NestedTypes before, or after, their enclosers
    - The TypeDef for a NestedType has one extra item of metadata, compared with a regular TypeDef. This additional item is the token for its enclosing type. This is persisted internally in a two-column look-aside table, holding the Typedefs of nestee and encloser. The Runtime uses this to determine whether the enclosing Type is, or is not, visible outside of the assembly. Note that because we losslessly capture the nesting hierarchy in metadata, there is no parsing of mangled type names required to "guess" the nesting structure
    - References to a NestedType will be emitted as TypeRefs. Upon resolution, the Runtime will observe that the visibility is *nested* and thus will apply member access rules
8. While importing a metadata file, suppose a language or tool, that is blind to NestedTypes, stumbles upon the definition of a NestedType. Firstly, it must recognize the Type as nested because it has one of the possible tdNestedXXX bits set in its TypeDef flags. [Every language or tool must recognize all bits in the CorTypeAttr enum – including tdNestedXXX. It need not implement the semantics those bits demand; it can simply stay away; but it must know enough to make that choice]   Having found a NestedType, the compiler/tool has two choices:
    - Don't expose that nested type
    - Expose that nested type as a module-level type, iff its encloser were visible and the member access rule on the nested type is NestedPublic
9. You can freely nest all Types – Classes, ValueTypes, Interfaces and Delegates. [The common case will be a Class which nests an Interface, but Runtime supports any permutations]

## 7.4 Visibility, Subclassing, and Member Access

Types carry visibility rules, one of:

- public -- meaning it is visible to any type in the same assembly, and may be exported outside of the assembly

  - non-public -- meaning it is visible to any type in the same assembly and may NOT be exported outside of the assembly
  - nested -- meaning it does not have visibility independent of its enclosing type; note that languages that do not support NestedTypes will either need to simply not expose these Types during import, or will need to test to make sure that member access rule on such a type is 'public' and the enclosing Type is visible before exposing the Type

Types carry subclassing rules, one of:

- sealed -- meaning that it may not be subclassed
- non-sealed -- meaning that any class that has visibility to the Type (see above) may subclass it

If a Type author wants to restrict subclassing to "this assembly" and yet make the Type available more widely, he could declare a non-sealed class with non-public visibility and a derived sealed Type that's visible to the world (public).

*Note: A Class may also be abstract, in which case it cannot be directly instantiated and must be subclassed with full implementations provided for all of its members. As such, an abstract Class cannot be sealed*

Types may specify access rules for their members, one of:

- private -- meaning that only this declaring Type may access the member
- family -- meaning that only a subtype of this Type may access the member
- assembly -- meaning that any Type in the same assembly as this Type may access the member
- familyANDassembly -- meaning that only subtypes in the same assembly as this Type may access the member
- familyORassembly -- meaning that any Type in the same assembly as this Type may access the member, as well as any subtype of the Type
- public -- meaning that any Tlass that has visibility to this Type may access the member

Any subtype that has access to a member may override the implementation for the member, if virtual, or may hide the member, if non-virtual.

## 7.5 Naming

Within a module, all Types must of course have a unique name. And within a Type, all NestedTypes must of course have a unique name. Note that metadata does *not* require that the names of NestedTypes be unique within the module.

Let's recap on an earlier example to clarify the problem:

```
class A {         =>  DefineTypeDef ("A") returns tokA
  class B {       =>  DefineNestedType ("B", tdNestedPublic, tokA) returns tokB
      class C {   =>  DefineNestedType ("C", tdNestedFamily, tokB) returns tokC
      }
  }
}
```

The problem comes in creating a TypeRef. For example, what TypeRef's do you emit in response to a source statement like:

    A.B.C.foo (42)

The answer is, that the compiler should emit one MemberRef, for foo, and 3 TypeRefs, for C, B and A. The resolution scope for each TypeRef is the token for its encloser. The resolution scope for A is the assembly or module where it is defined. Here's the DefineTypeRefByName prototype, as a reminder:

```
DefineTypeRefByName (mdToken  tkResolutionScope,  LPCWSTR  szNamespace,
                      LPCWSTR   szType,   mdTypeRef   *ptr)
```

In order to resolve this reference, walk 'up the tree' towards the root. Since this is driven by token, rather than by name, we find a unique path to the root. (If done by name, we could start with 10 "foo"s. We would end up with one unique path to root, but the interim tracking cost would be high)

This scheme works because NestedTypes must be defined within the same module as their enclosing type – never by a TypeRef.

[This proposed scheme replaces the current *de facto* practice where the compile invents a mangled name of A$B$C or similar. With this new proposal, we don't mangle names. And we circumvent the view that a TypeRef for A$B$C refers to a global class with a name of A$B$C]

## 7.6 *Naked* Instances

A language may choose, at its discretion, to allow a user to create an instance of a NestedType, without any encloser instance – in our running example, an instance of B, without any instance of A. So:

```
public static void main(String[] args) {
    A.B b = new A.B();                      // create a 'naked' B object
    b.bi = 9;                               // works fine
}
```

There is no instance of *A* to enclose *B*. Again, the Runtime has no qualms about any code that creates naked instances of a NestedType. So long as the language allows the user to name them, and therefore identify their TypeDef token, Runtime is content.

## 7.7 C++ "Member Classes"

The methods within a C++ member (*ie* nested) class have no special access to the members of an enclosing class. So, in the following example declaration,

```
class A {
    private: static int i;
    class B {
        void m();
    }
}
```

method *m* cannot reference the *private* field *i* of its enclosing class. However, in the definition of Runtime NestedTypes, it is clear that the Runtime would allow such an access to proceed – it would pass verification and run. Does this represent a problem?

The answer is no. The C++ compiler can use the support provided by Runtime for NestedTypes; it can deny any attempted access to field *i* by method *m* at compile time, and so preserve the semantics of the language. If another language imports that module's metadata there is again no problem, since it cannot emit code that runs within the nested lexical scope of class A. [It can of course emit code that attempts to access field *i* from *outside* of class A, but the Runtime would correctly fail that access since the field is marked *private*]

## 7.8 C++ "Friends"

**Friends will not be supported in first release of the Runtime.** We had earlier proposed to introduce a 'friends' mechanism, whereby a TypeDef could carry an explicit set of TypeDef/Ref tokens that are its Friends. Those friends would be allowed to have access to all of the members of the declaring type. This had been introduced in lieu of NestedTypes, in order to support some of the nested type semantics; however, with the above proposal to support NestedTypes, we will not provide direct support for "friends". But languages that have a notion of 'friend' may still carry such information as CustomAttributes in metadata.

## 7.9 Example - Simple

Here is an example, of a nested class definition, using SMC-like syntax:

```
public class A {               // DefineTypeDef("A", tdPublic) => tdA
  public static int asi;       // DefineField("asi", fdPublic|fdStatic, sig) => fdasi
  public        int aii;       // DefineField("aii", fdPublic, sig) => fdaii
  public class B {             // DefineNestedType("B", tdNestedPublic, tdA) => tdB
      public static int bsi; // DefineField("bsi", fdPublic|fdStatic, sig) => fdbsi
      public        int bii; // DefineField("bi, fdPublic, sig) => fdbi
  }
}
```

In compiling this fragment, the compiler will tell metadata about two classes. One class is called *A* and has visibility *public* -- that's to say, it can be seen outside of its assembly. To convey this info, the compiler calls DefineTypeDef, passing the name *A*, and a flags value of tdPublic (it can pass other goop too, but I'm only talking about those arguments that affect the real picture). This call returns a TypeDef token for A; let's call it *tdA*. Class *A* has one static field, *asi,* of type int. The compiler calls DefineField, passing the name *asi*, a flags value of fdPublic|fdStatic, and a signature of ELEMENT_TYPE_I4. This call returns a FieldDef token, which we'll call *fdasi*. Class *A* also has one instance field, *aii,* of type int. The compiler calls DefineField, passing the name *aii,* a flags value of fdPublic, and a signature of ELEMENT_TYPE_I4. This call returns a FieldDef token, which we'll call *fdaii.*

The other class is called *B* and has visibility *nestedPublic.* The compiler calls DefineNestedType, passing the name *B*, a flags value of tdNestedPublic, and an encloser of *tdA*. This call returns a TypeDef token for *B*; let's call it *tdB*. Class *B* has one static field, *bsi,* of type int. The compiler calls DefineField, passing the name *bsi*, a flags value of fdPublic|fdStatic, and a signature of ELEMENT_TYPE_I4. This call returns a FieldDef token, which we'll call *fdbsi*. Class *B* also has one instance field, *bii,* of type int. The compiler calls DefineField, passing the name *bii,* a flags value of fdPublic, and a signature of ELEMENT_TYPE_I4. This call returns a FieldDef token, which we'll call *fdbii.*

From this point forwards in time, metadata recognizes class *B* as nested solely because its flags value is one of the tdNestedXXX bunch – that's to say, one of tdNestedPublic, tdNestedPrivate, tdNestedFamily, tdNestedAssembly, tdNestedFamANDAssem or tdNestedFamORAssem.

Note that, as far as metadata is concerned, the only thing different about class *B* compared with class *A*, is that it is marked as "nested", and therefore has an associated encloser class (defined via *tdA*). Conversely, class *A* is not nested – its flags value is simply tdPublic – and it therefore has no associaated encloser.

Note too that, as explained above, an instance of class *A* (*ie* an *A* object) has only one field, which we called *aii.* In particular, there is no field containing a reference to a *B* object; nor yet space allocated within the *body* of *A* to hold a *B* object. So, if we attempt to compile the following code fragment, it will fail, as noted in the comments:

```
public static void main(String[] args) {
  A a = new A();                            // create an A object
  a.aii = 42;                               // works fine
  a.bii = 9;                                // doesn't work - no such field
  a.B.bii = 9;                              // doesn't work - no such field
}
```

Let’s go on and now create an instance of the nestee and see how nester and nestee relate:

```
public static void main(String[] args) {
    A a     = new A();                      // create an A object
    A.B b   = new A.B();                    // create a B object
    A.asi   = 1;                            //
    a.aii   = 2;                            //
    A.B.bsi = 3;                            //
    b.bsi   = 4;                            // reach static field via object
    b.bii   = 4;                            //
}
```

So far there is nothing at all surprising – the user of course has to specify he wants to create that nested class B – saying *B* on its own doesn’t work since there is no TypeDef for anything called *B* (just a NestedTypeDef). Each language may invent its own syntax for how to ‘reach’ *B* – the example has chosen the ‘obvious’ one of A.B. But apart from this naming wrinkle, everything would work the same if, instead of the nested *B*, we had been creating and operating upon a class *C,* defined at top level.

Where the behaviour of nested types *does* differ is that they lie within the lexical scope of their encloser, and so have unbridled access to all fields, properties and methods of that encloser – even if those fields, properties and methods are marked private. In this respect, the nested type is on a par with other methods and properties defined within the encloser. Here is an example that illustrates this:

```
public class Foo {
   public class A {
      private static int asi = 1;
      private        int aii = 2;
      public static  void ShowAsi() {Console.WriteLine("A.asi = " + A.asi);}
      public         void ShowAii() {Console.WriteLine("  aii = " +   aii);}

      public class B {
         private static int bsi = 3;
         private        int bii = 4;
         public static  void ShowBsi() {Console.WriteLine("B.bsi = " + B.bsi);}
         public         void ShowBii() {Console.WriteLine("  bii = " +   bii);}
         public static  void bsm() {
            asi = 10;       // same as: A.asi = 10;
            bsi = 11;       // same as: B.bsi = 11;
         }
         public void bim(A x) {
            asi = 13;       // same as: A.asi = 13;
            bsi = 14;       // same as: B.bsi = 14;
            bii   = 15;
            x.asi = 16;
            x.aii = 17;
         }
      }
   }

   public static void main (String[] args) {
      A   a = new A();
      A.B b = new A.B();
      A.ShowAsi(); a.ShowAii(); A.B.ShowBsi(); b.ShowBii();
      A.B.bsm();   Console.WriteLine(">>>>>>>call A.B.bsm");
      A.ShowAsi(); a.ShowAii(); A.B.ShowBsi(); b.ShowBii();
      b.bim(a);    Console.WriteLine(">>>>>>>call b.bim");
      A.ShowAsi(); a.ShowAii(); A.B.ShowBsi(); b.ShowBii();
   }
}
```

This program shows how, from within the static method *A.B.bsm*, we can update the private static field of our encloser class, *A.asi*. Similarly, from within the instance method *b.bim*, we can update the private instance field *aii* of any *A* object that we are passed as an argument.

Note that the visibility of a nested class affects whether it can be exported outside of the assembly in which it is defined. However, that visibility is qualified by that of its encloser. So, if the nested class has public visibility, but its encloser has non-public visibility, the nested class *cannot* be exported. This is a simple consequence of the fact their is no way to actually actually *name* the nested class from outside the assembly.

## 7.10 Example – Less Simple

Our simple example pointed out that the Runtime does not include any field in an encloser object, *A*, that references an object of its nested class, *B*. However, the user may explicitly add a field within *A* that holds a reference to a *B*. He may even add a 'backpointer' to an instance of his encloser, like this:

```
public class AA {
  public int aii;
  public BB  pbb;        // DefineField("pbb", fdPublic, sig) => fdpbb
  public class BB {
      public int bii;
      public AA paa;     // DefineField("paa", fdPublic, sig) => fdpaa
  }
}
```

With this definition of *AA*, we can write programs like the following, and things work fine:

```
public static void main(String[] args) {
  AA aa = new AA();          // create an AA object
  AA.BB bb = new AA.BB();    // create a BB object
  aa.pbb = bb;               // hook 'em one way
  bb.paa = aa;               // hook 'em the other
  aa.aii = 1;                // works fine
  aa.pbb.bii = 2;            // works fine
  bb.paa.aii = 3;            // works fine, even if aii were private
}
```

A language may choose to hide all of this plumbing detail from their users, and present a model that automatically provides an object reference field with the encloser, and nestee, etc.  The point is, metadata will *not* generate such plumbing. If a language wants it, the compiler must emit, via DefineField calls, as shown in the code comment above.

Such additions, by the compiler, can clearly be used as a route to implement "inner" classes

# 8 *Distinguished* Custom Attributes

The metadata engine implements two sorts of Custom Attribute, called (genuine) Custom Attributes, and pseudo Custom Attributes. In the remainder of this appendix, we'll abbreviate these terms to CA and PCA. Both CAs and PCAs are 'handed over' to metadata via the DefineCustomAttribute method. But they are treated differently, as follows:

- a CA is stored directly into the metadata. The 'blob' which holds its defining data is not checked or parsed. That 'blob' can be retrieved later
- a PCA is recognized because its name is one of a handful on metadata's hard-wired list of PCAs. The engine parses its 'blob' and uses this information to set bits and/or fields within the metadata tables. The engine then totally discards the 'blob'. So you cannot retrieve that 'blob' later – it doesn't exist

PCAs therefore serve to capture user 'directives', using the same familiar syntax the compiler provides for regular CAs – but these 'directives' are then stored into the more space-efficient form of metadata tables. Tables are also faster to check at runtime than full-bloodied (genuine) CAs. An example of a PCA is the SerializableAttribute – if the compiler calls DefineCustomAttribute with this PCA as an argument, the metadata engine simply sets the tdSerializable bit on the target class definition.

Many CAs are invented by higher layers of software. Metadata stores them, and returns them, without knowing, or caring, what they 'mean'. But all PCAs, plus a handful of regular CAs are of special interest to compilers and to the Runtime. An example of such 'distinguished' CAs is System.Reflection.DefaultMemberAttribute. This is stored in metadata as a regular CA 'blob', but Reflection uses this CA when called to Invoke the default member (property) for a Class.

This appendix lists all of the PCAs and 'distinguised' CAs – where 'distinguished' means that the Runtime and/or Compilers pay direct attention to them.

Note that it is a .net framework design guideline that all CAs should be named to end in "Attribute" (Neither metadata or runtime check, or care, about this convention)

## 8.1 Pseudo Custom Attributes (PCAs)

The metadata engine checks for the following CAs, as part of the processing for the DefineCustomAttribute method. The check is solely on their name – for example "DllImportAttribute" – their namespace is ignored. If a name match is found, the metadata engine parses the 'blob' argument and sets bits and/or fields within the metadata tables. It then throws the 'blob' on the floor (this is the definition of a PCA – see above):

System.InteropServices.**DllImportAttribute**
System.InteropServices.**GuidAttribute**
System.InteropServices.**ComImportAttribute**
System.InteropServices.**MethodImplAttribute**
System.InteropServices.**MethodImpl2Attribute**
System.InteropServices.**MarshalAsAttribute**
System.InteropServices.**PreserveSigAttribute**
System.InteropServices.**InAttribute**
System.InteropServices.**OutAttribute**

System.InteropServices.**InterfaceTypeAttribute**
System.InteropServices.**ClassInterfaceAttribute**
System.InteropServices.**OptionalAttribute**
System.InteropServices.**StructLayoutAttribute**
System.InteropServices.**FieldOffsetAttribute**
System.InteropServices.**DebuggableAttribute**

System.**SerializableAttribute**
System.**NonSerializedAttribute**

For a definition of these PCAs, see the online doc for .NET Framwork class libraries, or the “Data Interop” spec.

## 8.2 CAs that affect Runtime

The Runtime ‘pays attention’ to the CAs listed below. So, if a compiler attaches any of these CAs to a programming element (Class, Field, Assembly, etc, etc), then it will affect how that element is treated at runtime. For further details on this long list of CAs, consult the online doc for the .NET Framework class library, or appropriate specs in the area that each covers.

CAs that control runtime behavior of the JIT-compiler and the debugger:

System.Diagnostics.**DebuggerHiddenAttribute**
System.Diagnostics.**DebuggerStepThroughAttribute**

CA that is used by Reflection’s Invoke call – it invokes the property for the Type defined in this CA:

System.Reflection.**DefaultMemberAttribute**

CAs that control behavior of Interop services (inter-operation with ‘classic’ COM objects, and PInvoke dispatch to unmanaged code):

System.Runtime.InteropServices.**ComConversionLossAttribute**
System.Runtime.InteropServices.**ComEmulateAttribute**
System.Runtime.InteropServices.**ComImportAttribute**
System.Runtime.InteropServices.**ComRegisterFunctionAttribute**
System.Runtime.InteropServices.**ComSourceInterfacesAttribute**
System.Runtime.InteropServices.**ComUnregisterFunctionAttribute**
System.Runtime.InteropServices.**DispIdAttribute**
System.Runtime.InteropServices.**ExposeHResultAttribute**
System.Runtime.InteropServices.**FieldOffsetAttribute**
System.Runtime.InteropServices.**GlobalObjectAttribute**
System.Runtime.InteropServices.**HasDefaultInterfaceAttribute**
System.Runtime.InteropServices.**IDispatchImplAttribute**
System.Runtime.InteropServices.**ImportedFromTypeLibAttribute**
System.Runtime.InteropServices.**InterfaceTypeAttribute**
System.Runtime.InteropServices.**NoComRegistrationAttribute**
System.Runtime.InteropServices.**NoIDispatchAttribute**
System.Runtime.InteropServices.**PredeclaredAttribute**
System.Runtime.InteropServices.**StructLayoutAttribute**
System.Runtime.InteropServices.**TypeLibFuncAttribute**
System.Runtime.InteropServices.**TypeLibTypeAttribute**
System.Runtime.InteropServices.**TypeLibVarAttribute**

CAs that affect behavior of remoting:

System.Runtime.Remoting.**ContextAttribute**
System.Runtime.Remoting.**Synchronization**
System.Runtime.Remoting.**ThreadAffinity**
System.Runtime.Remoting.**OneWayAttribute**

CAs that affect the security checks performed upon method invocations at runtime:

System.Security.**DynamicSecurityMethodAttribute**
System.Security.Permissions.**SecurityAttribute**
System.Security.Permissions.**CodeAccessSecurityAttribute**
System.Security.Permissions.**EnvironmentPermissionAttribute**
System.Security.Permissions.**FileDialogPermissionAttribute**
System.Security.Permissions.**FileIOPermissionAttribute**
System.Security.Permissions.**IsolatedStoragePermissionAttribute**
System.Security.Permissions.**IsolatedStorageFilePermissionAttribute**
System.Security.Permissions.**PermissionSetAttribute**
System.Security.Permissions.**PublisherIdentityPermissionAttribute**
System.Security.Permissions.**ReflectionPermissionAttribute**
System.Security.Permissions.**RegistryPermissionAttribute**
System.Security.Permissions.**SecurityPermissionAttribute**
System.Security.Permissions.**SiteIdentityPermissionAttribute**
System.Security.Permissions.**StrongNameIdentityPermissionAttribute**
System.Security.Permissions.**UIPermissionAttribute**
System.Security.Permissions.**ZoneIdentityPermissionAttribute**
System.Security.Permissions.**PrincipalPermissionAttribute**
System.Security.**SuppressUnmanagedCodeSecurityAttribute**
System.Security.**UnverifiableCodeAttribute**

CA that denotes a TLS (thread-local storage) field:

System.**ThreadStatic**

The following CAs are used by the ALink tool to transfer information between Modules and Assemblies (they are temporarily 'hung off' a TypeRef to a class called AssemblyAttributesGoHere) then merged by ALink and 'hung off' the assembly:

System.Runtime.CompilerServices.**AssemblyOperatingSystemAttribute**
System.Runtime.CompilerServices.**AssemblyProcessorAttribute**
System.Runtime.CompilerServices.**AssemblyCultureAttribute**
System.Runtime.CompilerServices.**AssemblyVersionAttribute**
System.Runtime.CompilerServices.**AssemblyKeyFileAttribute**
System.Runtime.CompilerServices.**AssemblyKeyNameAttribute**
System.Runtime.CompilerServices.**AssemblyDelaySignAttribute**

# 9 Bitmasks

This section explains the various bitmasks used to define attributes of Types, Methods, Fields, etc. All of the enums described in this section are defined in **CorHdr.h**, which ships with the .NET SDK

## 9.1 Token Types [CorTokenType]

These are the values of the top byte in any metadata token that says what kind of token it is.  Unlike other lists in this spec, we includes the value assigned to each member:

```
mdtModule               = 0x00000000,       //
mdtTypeRef              = 0x01000000,       //
mdtTypeDef              = 0x02000000,       //
mdtFieldDef             = 0x04000000,       //
mdtMethodDef            = 0x06000000,       //
mdtParamDef             = 0x08000000,       //
mdtInterfaceImpl        = 0x09000000,       //
mdtMemberRef            = 0x0a000000,       //
mdtCustomAttribute      = 0x0c000000,       //
mdtPermission           = 0x0e000000,       //
mdtSignature            = 0x11000000,       //
mdtEvent                = 0x14000000,       //
mdtProperty             = 0x17000000,       //
mdtModuleRef            = 0x1a000000,       //
mdtTypeSpec             = 0x1b000000,       //
mdtAssembly             = 0x20000000,       //
mdtAssemblyRef          = 0x23000000,       //
mdtFile                 = 0x26000000,       //
mdtExportedType         = 0x27000000,       //
mdtManifestResource     = 0x28000000,       //
mdtString               = 0x70000000,       //
mdtName                 = 0x71000000,       //
mdtBaseType             = 0x72000000,
```

## 9.2 Scope Open Flags [CorOpenFlags]

These are used on IMetadataDispenser::OpenScope to specify the sort of access you want

```
ofRead      =   0x00000000,     // Open scope for read
ofWrite     =   0x00000001,     // Open scope for write.
ofCopyMemory =  0x00000002,     // Open scope with memory. Ask metadata to
                                // maintain its own copy of memory.
ofCacheImage =  0x00000004,     // EE maps but does not do relocations or
                                // verify image
ofNoTypeLib =   0x00000080,     // Don't OpenScope on a typelib.
```

## 9.3 Options for Size Calculation [CorSaveSize]

These are used on IMetaDataEmit::GetSaveSize to specify the sort of calculation you want

```
cssAccurate             = 0x0000,   // Find exact save size, accurate but slower.
cssQuick                = 0x0001,   // Estimate save size, may pad estimate,
                                    // but faster.
cssDiscardTransientCAs  = 0x0002,   // remove all of the CAs of discardable types
```

## 9.4 Flags for Types [CorTypeAttr]

You can define three kinds of **Type** in metadata – reference types (classes and interfaces), valuetypes (includes enums) and unmanaged valuetypes. You define any of those types using:

IMetaDataEmit::DefineTypeDef – to make the initial definition
IMetaDataEmit::SetTypeDefProps – to change the attributes for a previously-defined type

Both *DefineTypeDef* and *SetTypeDefProps* include a DWORD parameter, called *dwTypeDefFlags*, that is a bitmask of the *CorTypeAttr* enum. The individual bits within the *CorTypeAttr* enum are defined as follows:

```
// Use this mask to retrieve the type visibility information.
tdVisibilityMask     =   0x00000007,
tdNotPublic          =   0x00000000,  // Class is not public scope.
tdPublic             =   0x00000001,  // Class is public scope.
tdNestedPublic       =   0x00000002,  // Class is nested with public visibility
tdNestedPrivate      =   0x00000003,  // Class is nested with private visibility.
tdNestedFamily       =   0x00000004,  // Class is nested with family visibility.
tdNestedAssembly     =   0x00000005,  // Class is nested with assembly visibility.
tdNestedFamANDAssem  =   0x00000006,  // Class is nested with family and assembly
                                      // visibility.
tdNestedFamORAssem   =   0x00000007,  // Class is nested with family or assembly
                                      // visibility.
// Use this mask to retrieve class layout information
tdLayoutMask         =   0x00000018,
tdAutoLayout         =   0x00000000,     // Class fields are auto-laid out
tdSequentialLayout   =   0x00000008,     // Class fields are laid out sequentially
tdExplicitLayout     =   0x00000010,     // Layout is supplied explicitly

// Use this mask to retrieve class semantics information.
tdClassSemanticsMask    =   0x00000020,
tdClass                 =   0x00000000,     // Type is a class.
tdInterface             =   0x00000020,     // Type is an interface.

// Special semantics in addition to class semantics.
tdAbstract      =   0x00000080,     // Class is abstract
tdSealed        =   0x00000100,     // Class is concrete and may not be extended
tdSpecialName   =   0x00000400,     // Class name is special.  Name describes how.

// Implementation attributes.
tdImport                =   0x00001000,     // Class / interface is imported
tdSerializable          =   0x00002000,     // The class is Serializable.

// Use tdStringFormatMask to retrieve string information for native interop
tdStringFormatMask      =   0x00030000,
tdAnsiClass             =   0x00000000,     // LPTSTR is interpreted as ANSI in
this class
tdUnicodeClass          =   0x00010000,     // LPTSTR is interpreted as UNICODE
tdAutoClass             =   0x00020000,     // LPTSTR is interpreted automatically

tdBeforeFieldInit       =   0x00100000,     // Initialize the class any time
                                            // before first static field access.
// Flags reserved for runtime use.
tdReservedMask     =   0x00040800,
tdRTSpecialName    =   0x00000800,     // Runtime should check name encoding.
tdHasSecurity      =   0x00040000,     // Class has security associate with it.
```

Figure 1 shows, with a ✕ sign, which flags can be set for each kind or type-definition: class, interface, valuetype, and unmanaged valuetype. Conversely, the blank boxes show which settings are illegal. The table includes horizontal, shaded bands: these gather together flags that are mutually exclusive. Specifically:

- If defining a nested type or valuetype, you must set exactly one of the block of flags tdNestedPublic thru tdNestedFamOrAssem
- If defining a class, valuetype or unmanaged valuetype, you must set exactly one of tdAutoLayout, tdLayoutSequential or tdExplicitLayout

**Figure 1 – Legal Flag Combinations from CorTypeAttr**

| | Class | Interface | ValueType | Unmgd ValueType |
|---|---|---|---|---|
| **tdClass** | ✕ | | | |
| **tdInterface** | | ✕ | | |
| **tdNotPublic** | ✕ | ✕ | ✕ | ✕ |
| **tdPublic** | ✕ | ✕ | ✕ | ✕ |
| **tdNestedPublic** | ✕ | ✕ | ✕ | |
| **tdNestedPrivate** | ✕ | ✕ | ✕ | |
| **tdNestedFamily** | ✕ | ✕ | ✕ | |
| **tdNestedAssembly** | ✕ | ✕ | ✕ | |
| **tdNestedFamANDAssem** | ✕ | ✕ | ✕ | |
| **tdNestedFamOrAssem** | ✕ | ✕ | ✕ | |
| **tdAutoLayout** | ✕ | | ✕ | ✕ |
| **tdLayoutSequential** | ✕ | | ✕ | ✕ |
| **tdExplicitLayout** | ✕ | | ✕ | ✕ |
| **tdAbstract** | ✕ | ✕ | ✕ | ✕ |
| **tdSealed** | ✕ | | ✕ | ✕ |
| **tdSpecialName** | ✕ | ✕ | ✕ | ✕ |
| **tdRTSpecialName** | ✕ | ✕ | ✕ | ✕ |

Notes:

The runtime also takes note of each Type's inheritance chain to decide how to treat them –

- System.ValueType
- System.Enum
- System.MarshalByRefObject
- System.ContextBoundObject

## 9.5 Flags for Fields [CorFieldAttr]

Fields are defined using IMetadataEmit::DefineField.  The flags you can set are as follows:

```
// member access mask - Use this mask to retrieve accessibility information.
fdFieldAccessMask  =   0x0007,
fdPrivateScope     =   0x0000,  // Member not referenceable.
fdPrivate          =   0x0001,  // Accessible only by the parent type.
fdFamANDAssem      =   0x0002,  // Accessible by sub-types only in this Assembly.
fdAssembly         =   0x0003,  // Accessibly by anyone in the Assembly.
fdFamily           =   0x0004,  // Accessible only by type and sub-types.
fdFamORAssem       =   0x0005,  // Accessibly by sub-types anywhere, plus anyone
                                // in assembly.
fdPublic           =   0x0006,  // Accessibly by anyone who has visibility to
                                // this scope.
// field contract attributes.
fdStatic           =   0x0010,  // Defined on type, else per instance.
fdInitOnly         =   0x0020,  // Field may only be initialized, not written
                                // to after init.
fdLiteral          =   0x0040,  // Value is compile time constant.
fdNotSerialized    =   0x0080,  // Field does not have to be serialized when
                                // type is remoted.
fdSpecialName      =   0x0200,  // field is special.  Name describes how.

// interop attributes
fdPinvokeImpl      =   0x2000,  // Implementation is forwarded through pinvoke.

// Reserved flags for runtime use only.
fdReservedMask              =   0x9500,
fdRTSpecialName             =   0x0400,     // Runtime(metadata internal APIs)
should check name encoding.
fdHasFieldMarshal           =   0x1000,     // Field has marshalling information.
fdHasDefault                =   0x8000,     // Field has default.
fdHasFieldRVA               =   0x0100,     // Field has RVA.
```

## 9.6 Flags for Methods [CorMethodAttr]

Methods are defined using IMetadataEmit::DefineMethod.  The flags you can set are as follows:

```
// member access mask - Use this mask to retrieve accessibility information.
mdMemberAccessMask =   0x0007,
mdPrivateScope     =   0x0000,     // Member not referenceable.
mdPrivate          =   0x0001,     // Accessible only by the parent type.
mdFamANDAssem      =   0x0002,     // Accessible by sub-types only in this
                                   // Assembly.
mdAssem            =   0x0003,     // Accessibly by anyone in the Assembly.
mdFamily           =   0x0004,     // Accessible only by type and sub-types.
mdFamORAssem       =   0x0005,     // Accessibly by sub-types anywhere, plus
                                   // anyone in assembly.
mdPublic           =   0x0006,     // Accessibly by anyone who has visibility
                                   // to this scope.
// method contract attributes.
mdStatic           =   0x0010,     // Defined on type, else per instance.
mdFinal            =   0x0020,     // Method may not be overridden.
mdVirtual          =   0x0040,     // Method virtual.
mdHideBySig        =   0x0080,     // Method hides by name+sig, else just by name.

// vtable layout mask - Use this mask to retrieve vtable attributes.
mdVtableLayoutMask =   0x0100,
mdReuseSlot        =   0x0000,     // The default.
mdNewSlot          =   0x0100,     // Method always gets a new slot in the vtable.

// method implementation attributes.
mdAbstract         =   0x0400,     // Method does not provide an implementation.
mdSpecialName      =   0x0800,     // Method is special.  Name describes how.

// interop attributes
mdPinvokeImpl      =   0x2000,     // Implementation is forwarded through pinvoke.
mdUnmanagedExport  =   0x0008,     // Managed method exported via thunk to
                                   // unmanaged code.

// Reserved flags for runtime use only.
mdReservedMask     =   0xd000,
mdRTSpecialName    =   0x1000,     // Runtime should check name encoding.
mdHasSecurity      =   0x4000,     // Method has security associate with it.
mdRequireSecObject =   0x8000,     // Method calls another method containing
                                   // security code.
```

## 9.7 Flags for Method Parameters [CorParamAttr]

Method parameters are defined using IMetadataEmit::DefineParam and SetParamProps.  The flags you can set are as follows:

```
pdIn                     =   0x0001,     // Param is [In]
pdOut                    =   0x0002,     // Param is [out]
pdOptional               =   0x0010,     // Param is optional

// Reserved flags for Runtime use only.
pdReservedMask           =   0xf000,
pdHasDefault             =   0x1000,     // Param has default value.
pdHasFieldMarshal        =   0x2000,     // Param has FieldMarshal.
pdUnused                 =   0xcfe0,
```

## 9.8 Flags for Properties [CorPropertyAttr]

Properties are defined using IMetadataEmit::DefineProperty.  The flags you can set are as follows:

```
prSpecialName     =   0x0200,     // property is special.  Name describes how.

// Reserved flags for Runtime use only.
prReservedMask    =   0xf400,
prRTSpecialName   =   0x0400,     // Runtime(metadata internal APIs) should check
                                  // name encoding.
prHasDefault      =   0x1000,     // Property has default
```

```
    prUnused          =   0xe9ff,
```

## 9.9 Flags for Events [CorEventAttr]

Events are defined using IMetadataEmit::DefineEvent. The flags you can set are as follows:

```
    evSpecialName    =   0x0200,     // event is special.  Name describes how.

    // Reserved flags for Runtime use only.
    evReservedMask   =   0x0400,
    evRTSpecialName  =   0x0400,     // Runtime(metadata internal APIs) should
                                     // check name encoding.
```

## 9.10 Flags for MethodSemantics [CorMethodSemanticsAttr]

These flags describe the particular *role* played by each method defined (in a group) by a call to IMetaDataEmit::DefineProperty or to DefineEvent. They are derived from the way the methods were provided to the IMetaDataEmit::DefineProperty or DefineEvent call. This enumeration is used to return information from the IMetaDataImport::GetMethodSemantics call. Note that there is no corresponding *DefineMethodSemantics* call. The flags that can be set in the returned information are as follows:

```
        msSetter    =   0x0001,     // Setter for property
        msGetter    =   0x0002,     // Getter for property
        msOther     =   0x0004,     // other method for property or event
        msAddOn     =   0x0008,     // AddOn method for event
        msRemoveOn  =   0x0010,     // RemoveOn method for event
        msFire      =   0x0020,     // Fire method for event
```

## 9.11 Flags for Method Implementations [CorMethodImpl]

Method implementations are defined using IMetadataEmit::DefineMethod, DefineMethodImpl and SetRVA. The flags you can set are as follows:

```
    // code impl mask
    miCodeTypeMask      =   0x0003,   // Flags about code type.
    miIL                =   0x0000,   // Method impl is MSIL.
    miNative            =   0x0001,   // Method impl is native.
    miOPTIL             =   0x0002,   // Method impl is OPTIL
    miRuntime           =   0x0003,   // Method impl is provided by the runtime.

    // managed mask
    miManagedMask       =   0x0004,   // Flags specifying whether the code is managed
    or unmanaged.
    miUnmanaged         =   0x0004,   // Method impl is unmanaged, otherwise managed.
    miManaged           =   0x0000,   // Method impl is managed.

    // implementation info and interop
```

```
miForwardRef        =   0x0010,   // Indicates method is defined; used
                                  // primarily in merge scenarios.
miPreserveSig       =   0x0080,   // Indicates method sig is not to be mangled
                                  // to do HRESULT conversion.
miInternalCall      =   0x1000,   // Reserved for internal use.
miSynchronized      =   0x0020,   // Method is single threaded through the body.
miNoInlining        =   0x0008,   // Method may not be inlined.
miMaxMethodImplVal  =   0xffff,   // Range check value
```

## 9.12 Flags for Security [CorDeclSecurity]

Security attributes are declared using *IMetaDataEmit::DefineSecurityAttributeSet*. The flags you can set are listed below. Please see the Permissions spec for their meaning:

```
dclActionMask       =   0x000f,     // Mask allows growth of enum.
dclActionNil        =   0x0000,
dclRequest          =   0x0001,     //
dclDemand           =   0x0002,     //
dclAssert           =   0x0003,     //
dclDeny             =   0x0004,     //
dclPermitOnly       =   0x0005,     //
dclLinktimeCheck    =   0x0006,     //
dclInheritanceCheck =   0x0007,     //
dclRequestMinimum   =   0x0008,     //
dclRequestOptional  =   0x0009,     //
dclRequestRefuse    =   0x000a,     //
dclPrejitGrant      =   0x000b,     // Persisted grant set at prejit time
dclPrejitDenied     =   0x000c,     // Persisted denied set at prejit time
dclNonCasDemand     =   0x000d,     //
dclNonCasLinkDemand =   0x000e,
dclNonCasInheritance=   0x000f,
dclMaximumValue     =   0x000f,     // Maximum legal value
```

## 9.13 Struct for Field Offsets [COR_FIELD_OFFSET]

This struct is used by IMetaDataEmit::SetClassLayout. It has two fields, as follows:

| | |
|---|---|
| **mdFieldDef** | ridOfField; |
| **ULONG** | ulOffset; |

## 9.14 Typedef for Signatures [PCOR_SIGNATURE]

This type is used everywhere a metadata method takes a signature as an argument. In fact, it is simply a typedef for a pointer to an unsigned byte, so giving the definition doesn't help! However, for what it's worth, here's the definition:

```
typedef unsigned __int8     COR_SIGNATURE
typedef COR_SIGNATURE* PCOR_SIGNATURE
```

See section 10 for details on how signature 'blobs' should be formatted

## 9.15 Flags for PInvoke Interop [CorPinvokeMap]

Attributes that control how unmanaged methods are invoked, and how their arguments are mashalled via PInvoke, are defined using *IMetadataEmit::DefinePinvokeMap* or *SetPinvokeMap*. All of the flags below can be applied only to a method, never to a field. The flags you can set are as follows:

```
pmNoMangle          = 0x0001,   // Pinvoke is to use the member name as specified.

// Use this mask to retrieve the CharSet information.
pmCharSetMask       = 0x0006,
pmCharSetNotSpec    = 0x0000,
pmCharSetAnsi       = 0x0002,
pmCharSetUnicode    = 0x0004,
pmCharSetAuto       = 0x0006,
pmSupportsLastError = 0x0040,   // Information about target function. Not
                                // relevant for fields.
// None of the calling convention flags is relevant for fields.
pmCallConvMask      = 0x0700,
pmCallConvWinapi    = 0x0100,   // Pinvoke will use native callconv appropriate
                                // to target windows platform.
pmCallConvCdecl     = 0x0200,
pmCallConvStdcall   = 0x0300,
pmCallConvThiscall  = 0x0400,   // In M9, pinvoke will raise exception.
pmCallConvFastcall  = 0x0500,
```

Note that you can set only one of the calling convention flags

## 9.16 SetOptions: Duplicate Checking [CorCheckDuplicatesFor]

These flags are used in calling IMetadataDispenser::SetOption to control what checking the metadata API does for duplicates. The flags you can set in the bitmask are:

```
MDDupAll                = 0xffffffff,
MDDupENC                = MDDupAll,
MDNoDupChecks           = 0x00000000,
MDDupTypeDef            = 0x00000001,
MDDupInterfaceImpl      = 0x00000002,
MDDupMethodDef          = 0x00000004,
MDDupTypeRef            = 0x00000008,
MDDupMemberRef          = 0x00000010,
MDDupCustomAttribute    = 0x00000020,
MDDupParamDef           = 0x00000040,
MDDupPermission         = 0x00000080,
MDDupProperty           = 0x00000100,
MDDupEvent              = 0x00000200,
MDDupFieldDef           = 0x00000400,
MDDupSignature          = 0x00000800,
MDDupModuleRef          = 0x00001000,
MDDupTypeSpec           = 0x00002000,
MDDupImplMap            = 0x00004000,
MDDupAssemblyRef        = 0x00008000,
MDDupFile               = 0x00010000,
MDDupExportedType       = 0x00020000,
MDDupManifestResource   = 0x00040000,
// gap for debug junk
MDDupAssembly           = 0x10000000,

// This is the default behavior on metadata. It will check duplicates for
//  TypeRef, MemberRef, Signature, and TypeSpec
MDDupDefault = MDNoDupChecks | MDDupTypeRef | MDDupMemberRef |
               MDDupSignature | MDDupTypeSpec,
```

## 9.17 SetOptions: Ref-to-Def Optimizations [CorRefToDefCheck]

These flags are used in calling IMetadataDispenser::SetOption to control ref-to-def optimizations. The flags you can set in the bitmask are:

```
// default behavior is to always perform TypeRef to TypeDef and MemberRef
// to MethodDef/FieldDef optimization
MDRefToDefDefault       = 0x00000003,
MDRefToDefAll           = 0xffffffff,
MDRefToDefNone          = 0x00000000,
MDTypeRefToDef          = 0x00000001,
MDMemberRefToDef        = 0x00000002
```

## 9.18 SetOptions: Token Remap Notification [CorNotificationForTokenMovement]

These flags are used in calling IMetadataDispenser::SetOption to specify which token remaps are notified to you. The flags you can set in the bitmask are:

```
// default behavior is to notify TypeRef, MethodDef, MemberRef, and
// FieldDef token remaps
MDNotifyDefault         = 0x0000000f,
MDNotifyAll             = 0xffffffff,
MDNotifyNone            = 0x00000000,
MDNotifyMethodDef       = 0x00000001,
MDNotifyMemberRef       = 0x00000002,
MDNotifyFieldDef        = 0x00000004,
MDNotifyTypeRef         = 0x00000008,

MDNotifyTypeDef         = 0x00000010,
MDNotifyParamDef        = 0x00000020,
MDNotifyInterfaceImpl   = 0x00000040,
MDNotifyProperty        = 0x00000080,
MDNotifyEvent           = 0x00000100,
MDNotifySignature       = 0x00000200,
MDNotifyTypeSpec        = 0x00000400,
MDNotifyCustomAttribute = 0x00000800,
MDNotifySecurityValue   = 0x00001000,
MDNotifyPermission      = 0x00002000,
MDNotifyModuleRef       = 0x00004000,

MDNotifyNameSpace       = 0x00008000,

MDNotifyAssemblyRef     = 0x01000000,
MDNotifyFile            = 0x02000000,
MDNotifyExportedType    = 0x04000000,
MDNotifyResource        = 0x08000000,
```

## 9.19 SetOptions: Edit & Continue [CorSetENC]

These flags are used in calling IMetadataDispenser::SetOption to specify options for your Edit And Continue scope. You can set just one of the following values – this is not a bitmask:

```
MDSetENCOn          = 0x00000001,   // Deprecated name.
MDSetENCOff         = 0x00000002,   // Deprecated name.
MDUpdateENC         = 0x00000001,   // ENC mode.  Tokens don't move; can be
updated.
MDUpdateFull        = 0x00000002,   // "Normal" update mode.
MDUpdateExtension   = 0x00000003,   // Extension mode.  Tokens don't move, adds
only.
MDUpdateIncremental = 0x00000004,   // Incremental compilation
MDUpdateMask        = 0x00000007,
MDUpdateDelta       = 0x00000008,   // If ENC on, save only deltas.
```

## 9.20 SetOptions: Out-of-Order Errors [CorErrorIfEmitOutOfOrder]

These flags are used in calling IMetadataDispenser::SetOption to specify which sorts of out-of-order emit 'errors' you are notified of.

```
MDErrorOutOfOrderDefault = 0x00000000,  // default not to generate any error
MDErrorOutOfOrderNone    = 0x00000000,  // do not generate error for out of
                                        // order emit
MDErrorOutOfOrderAll     = 0xffffffff,  // generate out of order emit for method,
                                        // field, param, property, and event
MDMethodOutOfOrder       = 0x00000001,  // generate error when methods are emitted
                                        // out of order
MDFieldOutOfOrder        = 0x00000002,  // generate error when fields are emitted
                                        // out of order
MDParamOutOfOrder        = 0x00000004,  // generate error when params are emitted
                                        // out of order
MDPropertyOutOfOrder     = 0x00000008,  // generate error when properties are
                                        // emitted out of order
MDEventOutOfOrder        = 0x00000010,  // generate error when events are emitted
                                        // out of order
```

## 9.21 SetOptions: Hide Deleted Tokens [CorImportOptions]

These flags are used in calling IMetadataDispenser::SetOption, in an Edit & Continue regime, to specify which sorts of deleted tokens are returned in enumerations.

```
MDImportOptionDefault       = 0x00000000,   // default to skip over deleted
                                            // records
MDImportOptionAll           = 0xFFFFFFFF,   // Enumerate everything
MDImportOptionAllTypeDefs   = 0x00000001,   // all of the typedefs including the
                                            // deleted typedef
MDImportOptionAllMethodDefs = 0x00000002,   // all of the methoddefs including the
                                            // deleted ones
MDImportOptionAllFieldDefs  = 0x00000004,   // all of the fielddefs including the
                                            // deleted ones
MDImportOptionAllProperties = 0x00000008,   // all of the properties including the
                                            // deleted ones
MDImportOptionAllEvents     = 0x00000010,   // all of the events including the
                                            // deleted ones
MDImportOptionAllCustomAttributes = 0x00000020, // all of the custom attributes
                                                // including the deleted ones
MDImportOptionAllExportedTypes  = 0x00000040,   // all of the ExportedTypes
                                                // including the deleted ones
```

## 9.22 Flags for Assemblies [CorAssemblyFlags]

Assemblies are defined using IMetadataEmit::DefineAssembly. The flags you can set are as follows:

```
afPublicKey             =   0x0001, // The assembly ref holds the full
                                    // (unhashed) public key.
afCompatibilityMask     =   0x0070,
afSideBySideCompatible  =   0x0000, // The assembly is side by side
                                    // compatible.
afNonSideBySideAppDomain=   0x0010, // The assembly cannot execute with other
                                    // versions if they are executing in the
                                    // same application domain.
afNonSideBySideProcess  =   0x0020, // The assembly cannot execute with other
                                    // versions if they are executing in the
                                    // same process.
afNonSideBySideMachine  =   0x0030, // The assembly cannot execute with other
                                    // versions if they are executing on the
                                    // same machine.
afEnableJITcompileTracking  =   0x8000, // From "DebuggableAttribute".
afDisableJITcompileOptimizer=   0x4000, // From "DebuggableAttribute".
```

## 9.23 Flags for Manifest Resources [CorManifestResourceFlags]

Manifest resources are defined using IMetadataEmit::DefineManifestResource. The flags you can set are as follows:

```
mrVisibilityMask =   0x0007,
mrPublic         =   0x0001,     // The Resource is exported from the Assembly.
mrPrivate        =   0x0002,     // The Resource is private to the Assembly.
```

## 9.24 Flags for Files [CorFileFlags]

File attributes are defined using IMetadataEmit::DefineFile. The flags you can set are as follows:

```
ffContainsMetaData      =   0x0000,     // This is not a resource file
ffContainsNoMetaData    =   0x0001,     // This is a resource file or other
                                        // non-metadata-containing file
```

## 9.25 Element Types in the runtime [CorElementType]

These element types are used in defining method and field signatures. Many of these require no explanation, and are simply listed by-name. See the Signatures Spec for more detail. The total list is:

```
ELEMENT_TYPE_END            = 0x0,
ELEMENT_TYPE_VOID           = 0x1,
ELEMENT_TYPE_BOOLEAN        = 0x2,
ELEMENT_TYPE_CHAR           = 0x3,
ELEMENT_TYPE_I1             = 0x4,
ELEMENT_TYPE_U1             = 0x5,
ELEMENT_TYPE_I2             = 0x6,
ELEMENT_TYPE_U2             = 0x7,
ELEMENT_TYPE_I4             = 0x8,
ELEMENT_TYPE_U4             = 0x9,
ELEMENT_TYPE_I8             = 0xa,
ELEMENT_TYPE_U8             = 0xb,
ELEMENT_TYPE_R4             = 0xc,
ELEMENT_TYPE_R8             = 0xd,
ELEMENT_TYPE_STRING         = 0xe,

// every type above PTR will be simple type
ELEMENT_TYPE_PTR            = 0xf,      // PTR <type>
ELEMENT_TYPE_BYREF          = 0x10,     // BYREF <type>

// Please use ELEMENT_TYPE_VALUETYPE. ELEMENT_TYPE_VALUECLASS is deprecated.
ELEMENT_TYPE_VALUETYPE      = 0x11,     // VALUETYPE <class Token>
ELEMENT_TYPE_CLASS          = 0x12,     // CLASS <class Token>
ELEMENT_TYPE_ARRAY          = 0x14,     // MDARRAY <type> <rank> <bcount>
                                        // <bound1> ... <lbcount> <lb1> ...
ELEMENT_TYPE_TYPEDBYREF     = 0x16,     // This is a simple type.
```

```
ELEMENT_TYPE_I              = 0x18,     // native integer size
ELEMENT_TYPE_U              = 0x19,     // native unsigned integer size
ELEMENT_TYPE_FNPTR          = 0x1B,     // FNPTR <complete sig for the function
                                        // including calling convention>
ELEMENT_TYPE_OBJECT         = 0x1C,     // Shortcut for System.Object
ELEMENT_TYPE_SZARRAY        = 0x1D,     // Shortcut for single dimension zero
                                        // lower bound array SZARRAY <type>
// This is only for binding
ELEMENT_TYPE_CMOD_REQD      = 0x1F,     // required C modifier : E_T_CMOD_REQD
                                        // <mdTypeRef/mdTypeDef>
ELEMENT_TYPE_CMOD_OPT       = 0x20,     // optional C modifier : E_T_CMOD_OPT
                                        // <mdTypeRef/mdTypeDef>

// This is for signatures generated internally (which will not be persisted in
// any way).
ELEMENT_TYPE_INTERNAL       = 0x21,     // INTERNAL <typehandle>

// Note that this is the max of base type excluding modifiers
ELEMENT_TYPE_MAX            = 0x22,     // first invalid element type
ELEMENT_TYPE_MODIFIER       = 0x40,
ELEMENT_TYPE_SENTINEL       = 0x01 | ELEMENT_TYPE_MODIFIER, // sentinel
                                                            // for varargs
ELEMENT_TYPE_PINNED         = 0x05 | ELEMENT_TYPE_MODIFIER,
```

## 9.26 Calling Conventions [CorCallingConvention]

These types are used in defining method and field signatures. They are used by the JIT to determine which sequence of machine code to generate. See the Signatures Spec for more detail.

```
IMAGE_CEE_CS_CALLCONV_DEFAULT   = 0x0,

IMAGE_CEE_CS_CALLCONV_VARARG    = 0x5,
IMAGE_CEE_CS_CALLCONV_FIELD     = 0x6,
IMAGE_CEE_CS_CALLCONV_LOCAL_SIG = 0x7,
IMAGE_CEE_CS_CALLCONV_PROPERTY  = 0x8,
IMAGE_CEE_CS_CALLCONV_UNMGD     = 0x9,
IMAGE_CEE_CS_CALLCONV_MAX       = 0x10,  // first invalid calling convention
// The high bits of the calling convention convey additional info
IMAGE_CEE_CS_CALLCONV_MASK      = 0x0f,  // Calling convention is bottom
                                         // 4 bits
IMAGE_CEE_CS_CALLCONV_HASTHIS   = 0x20,  // Top bit indicates a 'this'
                                         // parameter
IMAGE_CEE_CS_CALLCONV_EXPLICITTHIS = 0x40,   // This parameter is explicitly
                                             // in  the signature
```

## 9.27 Unmanaged Calling Conventions [CorUnmanagedCallingConvention]

These types are used in defining method signatures. They are used by the JIT to determine which sequence of machine code to generate. Each is self-describing:

```
IMAGE_CEE_UNMANAGED_CALLCONV_C         = 0x1,
IMAGE_CEE_UNMANAGED_CALLCONV_STDCALL   = 0x2,
IMAGE_CEE_UNMANAGED_CALLCONV_THISCALL  = 0x3,
IMAGE_CEE_UNMANAGED_CALLCONV_FASTCALL  = 0x4,

IMAGE_CEE_CS_CALLCONV_C         = IMAGE_CEE_UNMANAGED_CALLCONV_C,
IMAGE_CEE_CS_CALLCONV_STDCALL   = IMAGE_CEE_UNMANAGED_CALLCONV_STDCALL,
IMAGE_CEE_CS_CALLCONV_THISCALL  = IMAGE_CEE_UNMANAGED_CALLCONV_THISCALL,
IMAGE_CEE_CS_CALLCONV_FASTCALL  = IMAGE_CEE_UNMANAGED_CALLCONV_FASTCALL,
```

Note that FASTCALL is defined only as a placeholder for the future; the CLR does **not** support Fast calls in V1

## 9.28 Argument Types [CorArgType]

These types are used in defining method signatures. See section 10 for more detail

```
IMAGE_CEE_CS_END        = 0x0,
IMAGE_CEE_CS_VOID       = 0x1,
IMAGE_CEE_CS_I4         = 0x2,
IMAGE_CEE_CS_I8         = 0x3,
IMAGE_CEE_CS_R4         = 0x4,
IMAGE_CEE_CS_R8         = 0x5,
IMAGE_CEE_CS_PTR        = 0x6,
IMAGE_CEE_CS_OBJECT     = 0x7,
IMAGE_CEE_CS_STRUCT4    = 0x8,
IMAGE_CEE_CS_STRUCT32   = 0x9,
IMAGE_CEE_CS_BYVALUE    = 0xA,
```

## 9.29 Native Types [CorNativeType]

These are used to define rules when marshalling method arguments between managed and unmanaged code, for example, in the IMetaDataEmit::SetFieldMarshal method. See the DataTypeMarshaling spec for details.

```
NATIVE_TYPE_END         = 0x0,    //DEPRECATED
NATIVE_TYPE_VOID        = 0x1,    //DEPRECATED
NATIVE_TYPE_BOOLEAN     = 0x2,    // (4 byte boolean value: TRUE = non-zero,
                                  // FALSE = 0)
NATIVE_TYPE_I1          = 0x3,
NATIVE_TYPE_U1          = 0x4,
NATIVE_TYPE_I2          = 0x5,
NATIVE_TYPE_U2          = 0x6,
NATIVE_TYPE_I4          = 0x7,
NATIVE_TYPE_U4          = 0x8,
NATIVE_TYPE_I8          = 0x9,
NATIVE_TYPE_U8          = 0xa,
NATIVE_TYPE_R4          = 0xb,
NATIVE_TYPE_R8          = 0xc,
NATIVE_TYPE_SYSCHAR     = 0xd,    //DEPRECATED
NATIVE_TYPE_VARIANT     = 0xe,    //DEPRECATED
NATIVE_TYPE_CURRENCY    = 0xf,
NATIVE_TYPE_PTR         = 0x10,   //DEPRECATED

NATIVE_TYPE_DECIMAL     = 0x11,   //DEPRECATED
```

```
NATIVE_TYPE_DATE        = 0x12,   //DEPRECATED
NATIVE_TYPE_BSTR        = 0x13,
NATIVE_TYPE_LPSTR       = 0x14,
NATIVE_TYPE_LPWSTR      = 0x15,
NATIVE_TYPE_LPTSTR      = 0x16,
NATIVE_TYPE_FIXEDSYSSTRING  = 0x17,
NATIVE_TYPE_OBJECTREF   = 0x18,   //DEPRECATED
NATIVE_TYPE_IUNKNOWN    = 0x19,
NATIVE_TYPE_IDISPATCH   = 0x1a,
NATIVE_TYPE_STRUCT      = 0x1b,
NATIVE_TYPE_INTF        = 0x1c,
NATIVE_TYPE_SAFEARRAY   = 0x1d,
NATIVE_TYPE_FIXEDARRAY  = 0x1e,
NATIVE_TYPE_INT         = 0x1f,
NATIVE_TYPE_UINT        = 0x20,

NATIVE_TYPE_NESTEDSTRUCT  = 0x21, //DEPRECATED (use NATIVE_TYPE_STRUCT)
NATIVE_TYPE_BYVALSTR    = 0x22,
NATIVE_TYPE_ANSIBSTR    = 0x23,
NATIVE_TYPE_TBSTR       = 0x24, // select BSTR or ANSIBSTR depending on
                                // platform
NATIVE_TYPE_VARIANTBOOL = 0x25, // (2-byte boolean value: TRUE = -1,
                                // FALSE = 0)
NATIVE_TYPE_FUNC        = 0x26,
NATIVE_TYPE_ASANY       = 0x28,
NATIVE_TYPE_ARRAY       = 0x2a,
NATIVE_TYPE_LPSTRUCT    = 0x2b,
NATIVE_TYPE_CUSTOMMARSHALER = 0x2c,  // Custom marshaler native type. This
// must be followed by a string of the following format:
// "Native type name/0Custom marshaler type name/0Optional cookie/0"
// Or
// "{Native type GUID}/0Custom marshaler type name/0Optional cookie/0"
NATIVE_TYPE_ERROR       = 0x2d, // This native type coupled with
                                // ELEMENT_TYPE_I4 will map to VT_HRESULT
NATIVE_TYPE_MAX         = 0x50, // first invalid element type
```

# 10 Signatures

The word *signature* is conventionally used to describe the type info for a function or method – that's to say, the type of each of its parameters, and the type of its return value. Within metadata, we extend the use of the word *signature* to also describe the type info for fields, properties and local variables. Each Signature is stored as a (counted) byte array in the Blob heap. There are five sorts of Signature, as follows:

- MethodDefSig
- MethodRefSig – differs from a MethodDefSig only for VARARG calls
- FieldSig
- PropertySig
- LocalVarSig

You can tell which sort of Signature blob you are looking at from the value of its leading byte (see later)

This section defines the binary blob format for each sort of Signature. For the most part, we use syntax diagrams (hopefully easier to understand than formal XML or EBNF)

Note that Signatures are compressed before being stored into the blob heap. It's actually the compiler or code generator who is responsible for compressing them, before passing them into the metadata engine. However, all compilers use the same small family of helper functions, defined in Cor.h, to do this task –

- CorSigCompressData / CorSigUncompressData
- CorSigCompressSignedInt / CorSigUncompressSignedInt
- CorSigCompressToken / CorSigUncompressToken

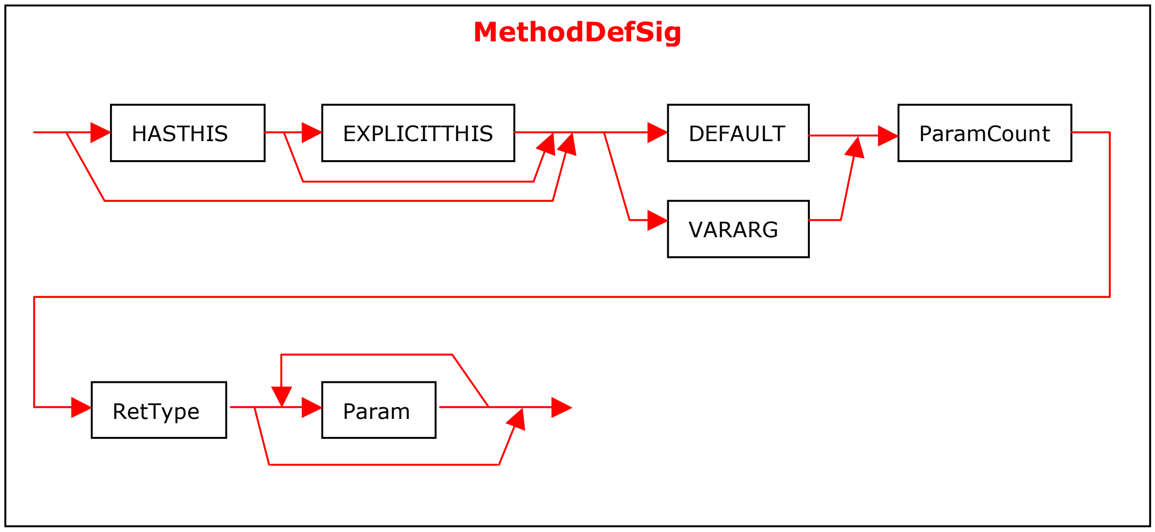
In order to uncompress a value in a Signature, you must know (from its position in the Signature) whether to call *CorSigUncompressData, CorSigUncompressSignedInt* or *CorSigUncompressToken*

Signatures include two *modifiers* called:

- ELEMENT_TYPE_BYREF – such an element points to a data item which may be allocated from the GC heap, or from elsewhere. It may point to the start of an object, or to the interior of an object. Either way, the GC is notified of its existence; if it actually points into the heap, then GC knows to update its value if it moves the object pointed-to during a garbage collection. This modifier can only occur in the definition of Param (section 10.10) or RetType (section 10.11). It may **not** occur within the definition of a Field (section 10.4) [conceptually you could imagine a runtime that *did* support BYREF fields, but ours doesn't – BYREFs, especially those that point into the interior of an object in the GC heap, are expensive to track – since there's no very strong requirement for BYREF fields, we excluded them]
- ELEMENT_TYPE_PTR – such an element points to a data item which is not allocated from the GC heap. The GC is not notified of its existence. This modifier can occur in the definition of Param (section 10.10) or RetType (section 10.11) or Field (section 10.4)

## 10.1 MethodDefSig

A MethodDefSig is indexed by the Method.Signature column. It captures the *signature* of a method or global function. The syntax chart for a MethodDefSig looks like this:



This chart uses the following abbreviations:

|  |  |  |
|---|---|---|
| • HASTHIS | for | IMAGE_CEE_CS_CALLCONV_HASTHIS |
| • EXPLICITTHIS | for | IMAGE_CEE_CS_CALLCONV_EXPLICITTHIS |
| • DEFAULT | for | IMAGE_CEE_CS_CALLCONV_DEFAULT |
| • VARARG | for | IMAGE_CEE_CS_CALLCONV_VARARG |

The first byte of a Signature is composed of two nybbles: the high nybble holds the HASTHIS or EXPLICITTHIS (or no) modifier; the low nybble holds the calling convention – DEFAULT or VARARG. (Strictly speaking, a compiler composes the value as described, but then calls the *CorSigCompressData* helper function in Cor.h to compress it into 1, 2 or 4 bytes, as required – with the definitions in force today, this *always* results in a 1-byte item)

*ParamCount* is an integer that holds the number of parameters (0 or more). It can be any number between 0 and 0x1FFF.FFFF The compiler compresses it too, using *CorSigCompressData*, before storing into the blob (*ParamCount* counts just the method parameters – it does not include the method's return type)

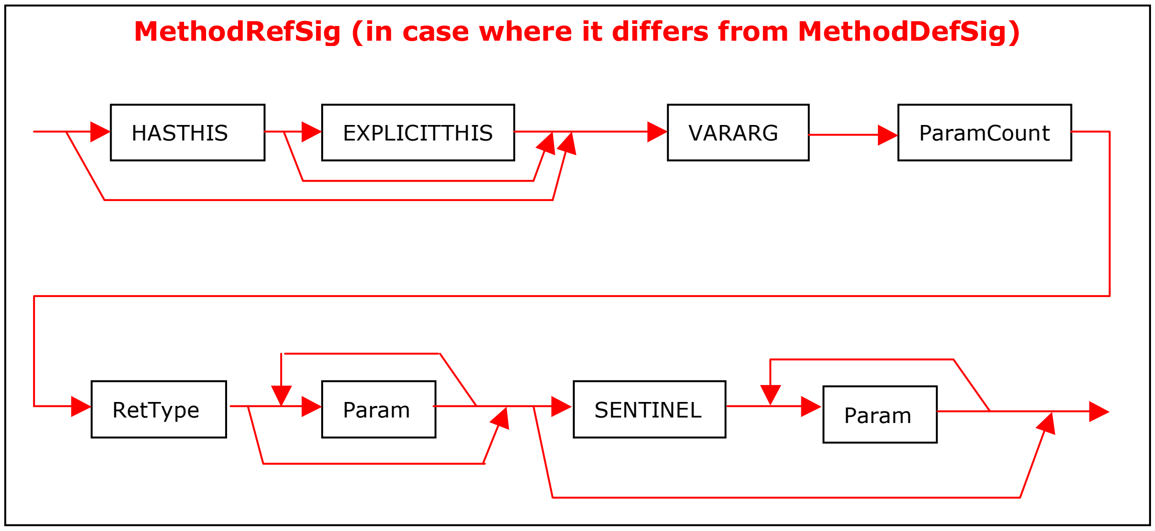
The *RetType* item describes the type of the method's return value (see later)

The *Param* item describes the type of each of the method's parameters (see later). There must be *ParamCount* instances of the *Param* item.

## 10.2 MethodRefSig

A MethodRefSig is indexed by the MemberRef.Signature column. This provides the *callsite* Signature for a method. Normally, this callsite Signature must match exactly the Signature specified in the definition of the target method. For example, if a method Foo is defined that takes two uint32s and returns void; then any callsite must index a signature that takes exactly two uint32s and returns void. In this case, the syntax chart for a MethodRefSig is identical with that for a MethodDefSig – see section 10.1

The Signature at a callsite differs from that at its definition, only for a method with the VARARG calling convention. In this case, the callsite Signature is extended to include info about the extra VARARG arguments (for example, corresponding to the "…" in C syntax). The syntax chart for this case is:



This chart uses the following abbreviations:

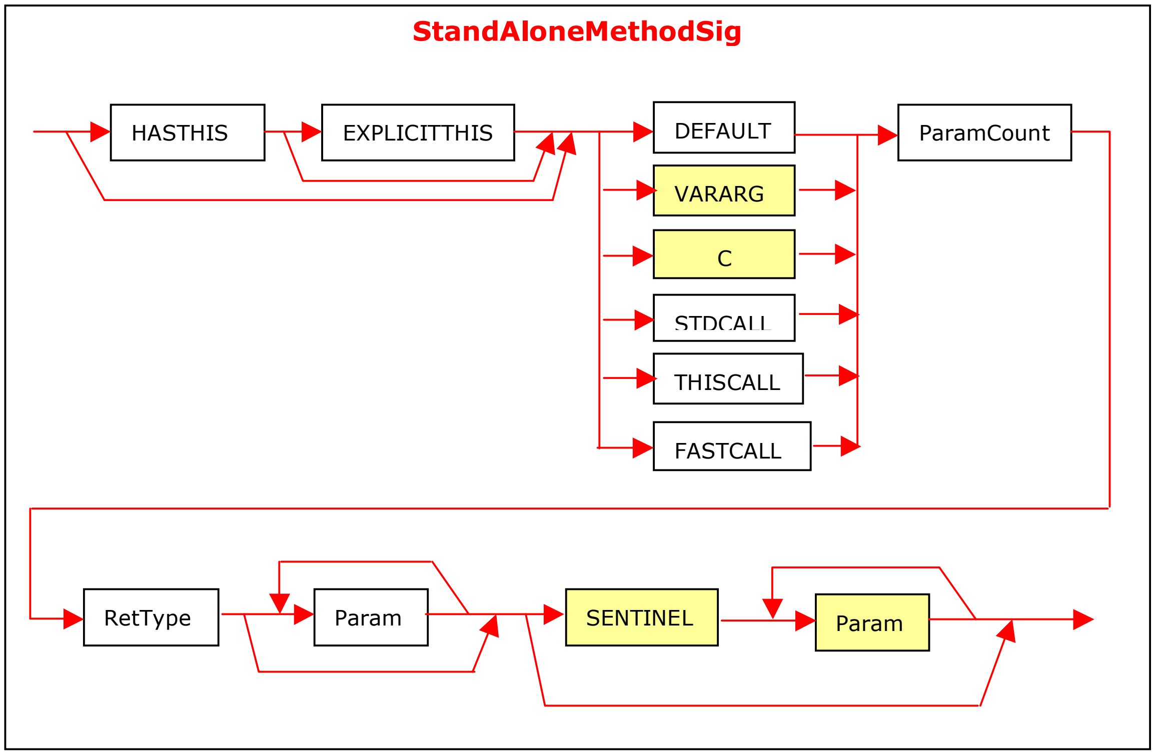
| | | |
|---|---|---|
| • HASTHIS | for | IMAGE_CEE_CS_CALLCONV_HASTHIS |
| • EXPLICITTHIS | for | IMAGE_CEE_CS_CALLCONV_EXPLICITTHIS |
| • VARARG | for | IMAGE_CEE_CS_CALLCONV_VARARG |
| • SENTINEL | for | ELEMENT_TYPE_SENTINEL |

This starts just like the MethodDefSig for a VARARG method (see section 10.1). But we then append an ELEMENT_TYPE_SENTINEL token, followed by extra *Param* items to describe the extra VARARG arguments. Note that the *ParamCount* item must tell us the total number of *Param* items in the Signature – so it includes items both *before* and *after* the SENTINEL byte.

In the unusual case that a callsite supplies no extra arguments, the signature should **not** include a SENTINEL (this is the route shown by the lower arrow that bypasses SENTINEL and goes to the end of the MethodRefSig definition)

## 10.3 StandAloneMethodSig

A StandAloneMethodSig is indexed by the StandAloneSig.Signature column. It is typically created as preparation for executing a *calli* instruction. It is very similar to a MethodRefSig, in that it represents a callsite signature, but its calling convention may specify an unmanaged target (the *calli* instruction invokes either managed, or unmanaged code). Its syntax chart looks like this:



This chart uses the following abbreviations:

| | | |
|---|---|---|
| • HASTHIS | for | IMAGE_CEE_CS_CALLCONV_HASTHIS |
| • EXPLICITTHIS | for | IMAGE_CEE_CS_CALLCONV_EXPLICITTHIS |
| • DEFAULT | for | IMAGE_CEE_CS_CALLCONV_DEFAULT |
| • VARARG | for | IMAGE_CEE_CS_CALLCONV_VARARG |
| • C | for | IMAGE_CEE_CS_CALLCONV_C |
| • STDCALL | for | IMAGE_CEE_CS_CALLCONV_STDCALL |
| • THISCALL | for | IMAGE_CEE_CS_CALLCONV_THISCALL |
| • FASTCALL | for | IMAGE_CEE_CS_CALLCONV_FASTCALL |
| • SENTINEL | for | ELEMENT_TYPE_SENTINEL |

(Note that FASTCALL is defined only as a placeholder for the future; the CLR does **not** support Fast calls in V1)

This is the most complex of the various method signatures. We have combined two separate charts into one, using shading. Thus, for the following calling conventions:

DEFAULT (managed)
STDCALL, THISCALL and FASTCALL (unmanaged)

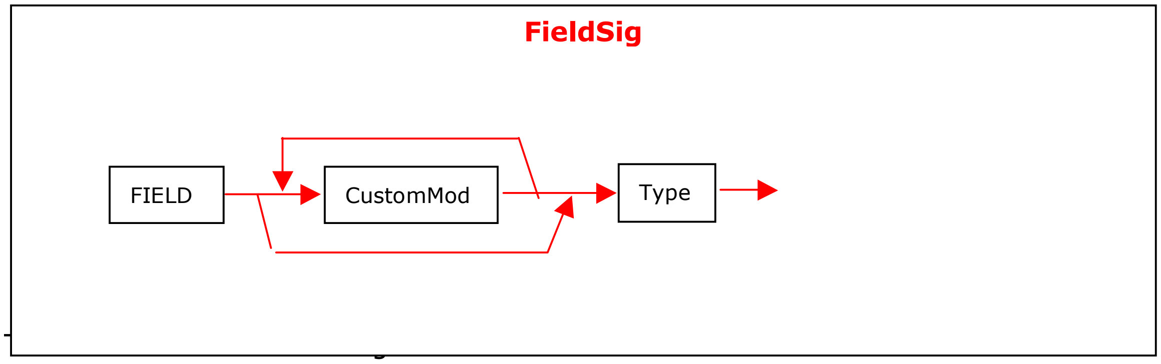
the signature ends just before the SENTINEL item (these are all non vararg signatures). However, for the managed and unmanaged vararg calling conventions:

VARARG (managed)
C (unmanaged)

the signature can include the SENTINEL and final Param items (it doesn't have to). These options are what is intended by the shading of boxes in the syntax chart

## 10.4 FieldSig

A FieldSig is indexed by the Field.Signature column, or by the MemberRef.Signature column (in the case where it specifies a reference to a field, not a method, of course). The Signature captures the field's definition. The field may be a static or instance field in a class, or it may be a global variable. The syntax chart for a FieldSig looks like this:



- FIELD for IMAGE_CEE_CS_CALLCONV_FIELD

*CustomMod* is defined in section 10.7. *Type* is defined in section 10.12

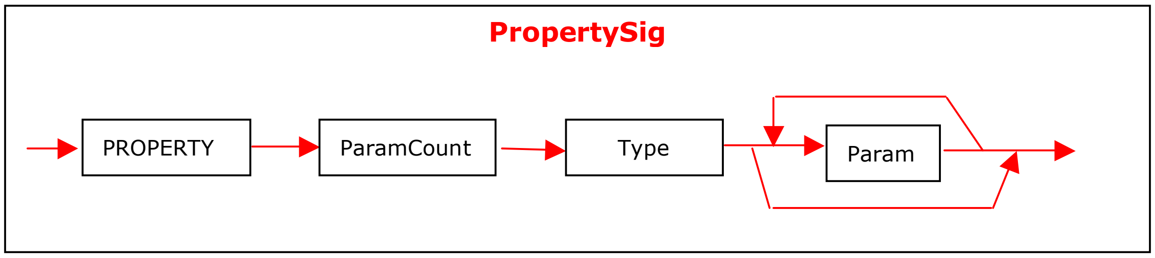
## 10.5 PropertySig

A PropertySig is indexed by the Property.Type column. It captures the type info for a Property – that's to say:

- how many parameters are supplied to its *setter* or *getter* methods
- the base type of the Property – the type returned by its *getter* method
- type info for each parameter in its *setter* or *getter* methods – that's to say, the index parameters

Note that there is no requirement that a Property have *setter, getter* or *other* methods (though having *none* amounts to useless construct). So the signature supplied must be syntactically valid, and *should* follow the suggestions above. However, this is not checked or enforced. The Property signature may be used by browsers, compilers or other tools, but is not used by the CLR itself.

The syntax chart for a PropertySig looks like this:



This chart uses the following abbreviations:

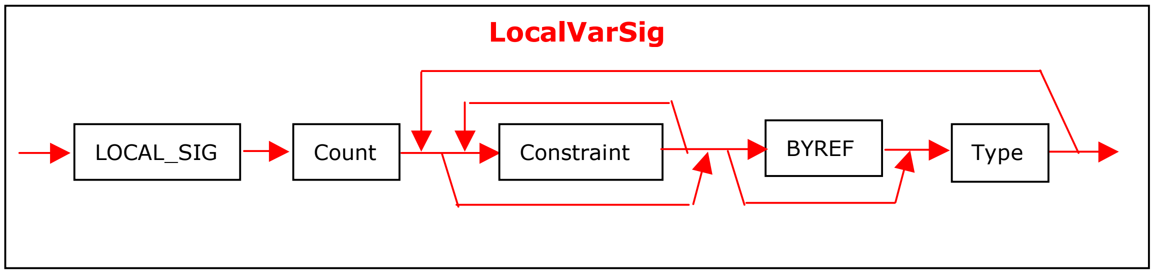
| | | |
|---|---|---|
| • PROPERTY | for | IMAGE_CEE_CS_CALLCONV_PROPERTY |

*Type* specifies the type returned by the *Getter* method for this property. *Type* is defined in section 10.12. *Param* is defined in section 10.10

*ParamCount* is an integer that holds the number of index parameters in the *setter* or *getter* methods (0 or more). It can be any number between 0 and 0x1FFF.FFFF The compiler compresses it, using *CorSigCompressData*, before storing into the blob (it almost inevitably ends up as a single byte) (*ParamCount* counts just the method parameters – it does not include the method's base type of the Property -- often termed the *this* pointer)

## 10.6 LocalVarSig

A LocalVarSig is indexed by the StandAloneSig.Signature column. It captures the type of all the local variables in a method. Its syntax chart looks like this:



This chart uses the following abbreviations:

| | | |
|---|---|---|
| • LOCAL_SIG | for | IMAGE_CEE_CS_CALLCONV_LOCAL_SIG |
| • BYREF | for | ELEMENT_TYPE_BYREF |

*Constraint* is defined in section 10.9 *Type* is defined in section 10.12

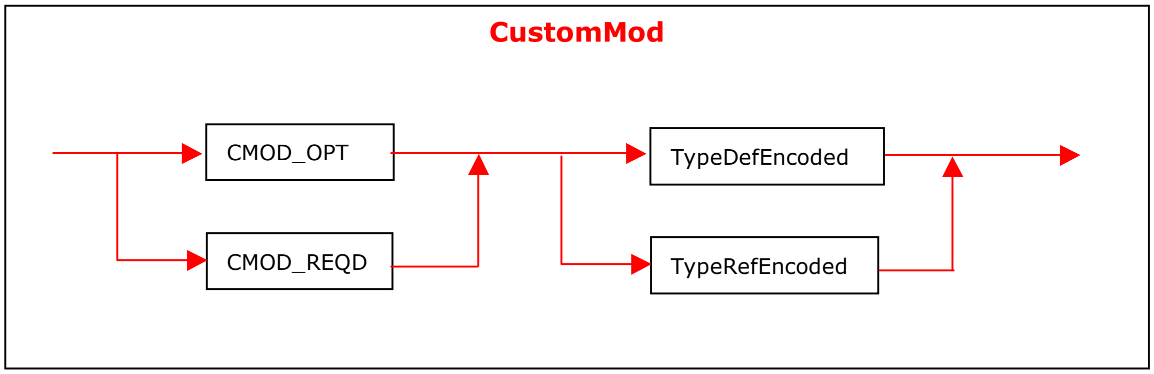
*Count* is an unsigned integer that holds the number of local variables. It can be any number between 1 and 0xFFFE. The compiler compresses it, using *CorSigCompressData*, before storing into the blob (it almost always compresses into one byte)

There must be *Count* instances of the *Constraint**/BYREF?/*Type* chain in the LocalVarSig

A LocalVarSig is created by Compilers and other code generators. For example, ILASM generates a LocalVarSig in response to the .locals directive

## 10.7 CustomMod

The *CustomMod* (custom modifier) item in Signatures has a syntax chart like this:



This chart uses the following abbreviations:

| | | |
|---|---|---|
| • CMOD_OPT | for | ELEMENT_TYPE_CMOD_OPT |
| • CMOD_REQD | for | ELEMENT_TYPE_CMOD_REQD |

The CMOD_OPT or CMOD_REQD value is compressed using *CorSigCompressData* – their values today are small numbers, so they always compress to a single byte.

Be careful not to confuse "Custom Modifiers" with "Custom Attributes". Both terms are used, but refer to totally different items: a custom modifier occurs as a particular item within a signature blob – as its name suggest, it *modifies* the meaning of the signature. On the other hand, a custom attribute describes a runtime object that is *hung off* another object, method, field, etc – custom attributes are stored within metadata as blobs with their own detailed layout (see section 11)

The CMOD_OPT or CMOD_REQD is followed by a metadata token that indexes a row in the *TypeDef* table or the *TypeRef* table. However, these tokens are encoded and compressed – see section 10.8 for details

If the CustomModifier is tagged CMOD_OPT, then any importing compiler can freely ignore it entirely. Conversely, if the CustomModifier is tagged CMOD_REQD, any importing compiler must 'understand' the semantic implied by this CustomModifier in order to reference the surrounding Signature.

A typical use for a *CustomModifier* is for VISUAL C++. NET to denote a method parameter as *const*. It does this using a CMOD_OPT, followed by a *TypeRef* to Microsoft.VisualC.IsConstModifier (defined in Microsoft.VisualC.DLL)

VISUAL C++ .NET also uses a *CustomModifier* (embedded within a *RetType* – see section 10.11) to mark the native calling convention of a function. Of course, if that routine is implemented as managed code, this info is not used. But if it turns out to be implemented as unmanaged code, it becomes crucial, so that automatically generated thunks marshal the arguments correctly. This technique is used in IJW ("It Just Works") scenarios. Strictly speaking, such a custom modifier does not apply only to the *RetType*, it really applies to the whole function. However, attaching it to the *RetType* proved a convenient carrier. In these cases, the *TypeRef* following the CMOD_OPT is to one of *CallConvCdecl*, *CallConvStdcall*, *CallConvThiscall* or *CallConvFastcall*. These are all defined in the namespace System.Runtime.InteropServices. Finally, note that *CallConvFastcall* is defined only as a placeholder for the future; the CLR does **not** support Fast calls in V1.

## 10.8 TypeDefEncoded and TypeRefEncoded

These items are compact ways to store a TypeDef or TypeRef token in a Signature.

Consider a regular TypeRef token, such as 0x01000012. The top byte of 0x01 tells us this is a TypeRef token (see the CorTokenType enum in CorHdr.h). The lower 3 bytes (0x000012) index row number 0x12 in the TypeRef table

The encoded version of this TypeRef token is made up as follows:

a) encode the table that this token indexes as the least significant 2 bits. The bit values to use are defined in Cor.h, as follows:

```
const static mdToken g_tkCorEncodeToken[4] = {mdtTypeDef,
mdtTypeRef, mdtTypeSpec, mdtBaseType};
```

b) shift the 3-byte row index (0x000012 in our example) left by 2 bits and OR into the 2-bit encoding from step a)

c) call *CorSigCompressData* on the resulting value

For our example, we end up with the following encoded value:

```
a)  encoded = g_tkCorEncodToken[1] = 0b0001
b)  encoded = ( 0x000012 << 2 ) |  0x01
            = 0x48 | 0x01
            = 0x49
c)  encoded = CorSigCompressData (0x49)
            = 0x49
```

So, instead of the original, regular TypeRef token value of 0x01000012, requiring 4 bytes of space in the Signature blob, we encode it as a single byte.

Note that there are two helper functions in Cor.h – *CorSigCompressToken* and *CorSigUncompressToken* that combine these steps together (encoding the target table type and compressing)

## 10.9 Constraint

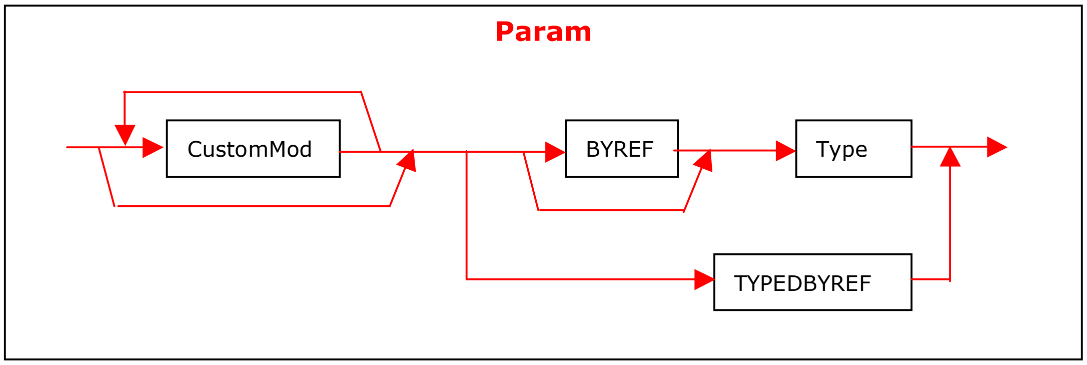
The *Constraint* item in Signatures currently has only one possible value – ELEMENT_TYPE_PINNED, which specifies that the target type is pinned in the runtime heap, and will not be moved by the actions of garbage collection. Note that the Compiler calls *CorCompressData* to compress the value for *Modifier* before inserting into the Signature blob; but today's value is small enough that it compresses to a single byte.

A *Constraint* can only be applied within a LocalVarSig (not a FieldSig). The Type of the local variable must either be a reference type (in other words, it *points* to the actual variable – for example, an Object, or a String); or it must include the BYREF item. The reason is that local variables are allocated on the runtime stack – they are never allocated from the runtime heap; so unless the local variable *points* at an object allocated in the GC heap, pinning makes no sense.

[Note: in previous versions, *Constraint* could also include a VOLATILE value. However, this constraint was removed from the Signature – compilers instead issue MSIL instructions that indicate the target variable is volatile]

## 10.10 Param

The *Param* (parameter) item in Signatures has a syntax chart like this:



This chart uses the following abbreviations:

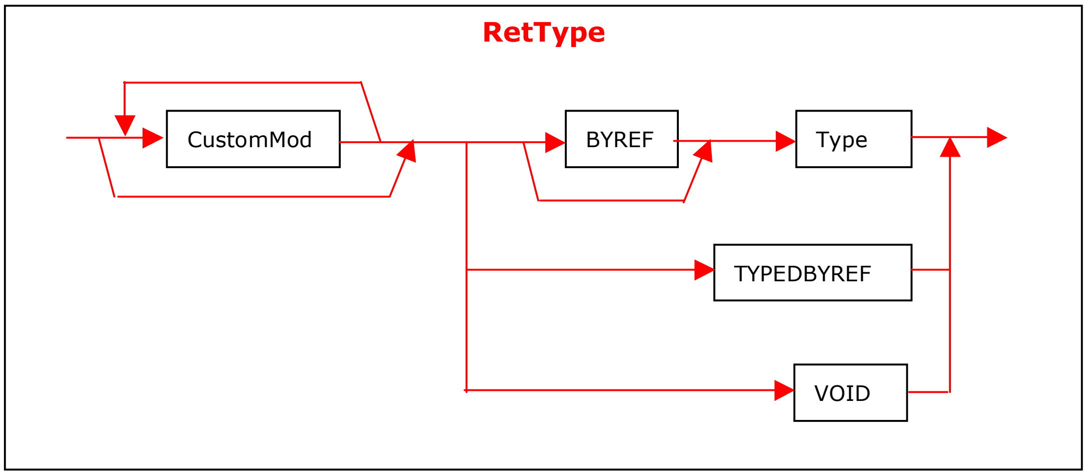
| | | |
|---|---|---|
| BYREF | for | ELEMENT_TYPE_BYREF |
| TYPEDBYREF | for | ELEMENT_TYPE_TYPEDBYREF |

*CustomMod* is defined in section 10.7. *Type* is defined in section 10.12

A TYPEDBYREF is a simple structure of two DWORDs – one indicates the type of the parameter, the other, its value. This struct is pushed on the stack by the caller. So, only at runtime, is the type of the parameter actually provided. TYPEDBYREF was originally introduced to support VB's "refany" argument-passing technique

## 10.11 RetType

The *RetType* (return type) item in Signatures has a syntax chart like this:



*RetType* is identical to *Param* except for one extra possibility, that it can include the type VOID. This chart uses the following abbreviations:

| | | |
|---|---|---|
| BYREF | for | ELEMENT_TYPE_BYREF |
| TYPEDBYREF | for | ELEMENT_TYPE_TYPEDBYREF (see section 10.10) |
| VOID | for | ELEMENT_TYPE_VOID |

*CustomMod* is defined in section 10.7. *Type* is defined in section 10.12

Note that a *CustomMod* is used by the VC compiler to record the native calling convention of the method – see section 10.7.

## 10.12 Type

The *Type* item in Signatures can be quite complicated. Below is a simple EBNF grammar for *Type.* As usual, "|" separates alternatives, "*" denotes zero or more occurrences, "?" denotes zero or one occurrence. Note that the last three productions are all recursive: PTR and SZARRAY are left-recursive, whilst ARRAY is right-recursive.

```
Type :=    Intrinsic
          | VALUETYPE          TypeDefOrRefEncoded
          | CLASS              TypeDefOrRefEncoded
          | STRING
          | OBJECT
          | PTR                CustomMod*  VOID
          | PTR                CustomMod*  Type
          | FNPTR              MethodDefSig
          | FNPTR              MethodRefSig
          | ARRAY              Type        ArrayShape
          | SZARRAY            CustomMod*  Type
```

For compactness, we have missed out the ELEMENT_TYPE_ prefixes in this list. So, for example, "CLASS" is shorthand for ELEMENT_TYPE_CLASS (see the CorElementType enum defined in CorHdr.h)

### 10.12.1 Intrinsic

This represents the set of simple value types provided by the runtime. They are defined as follows:

BOOLEAN | CHAR | I1 | U1 | I2 | U2 | I4 | U4 | I8 | U8 | R4 | R8 | I | U

However, CLS does not support this full range of intrinsic types – it excludes those in listed in the CLS rule below

### 10.12.2 ARRAY Type ArrayShape

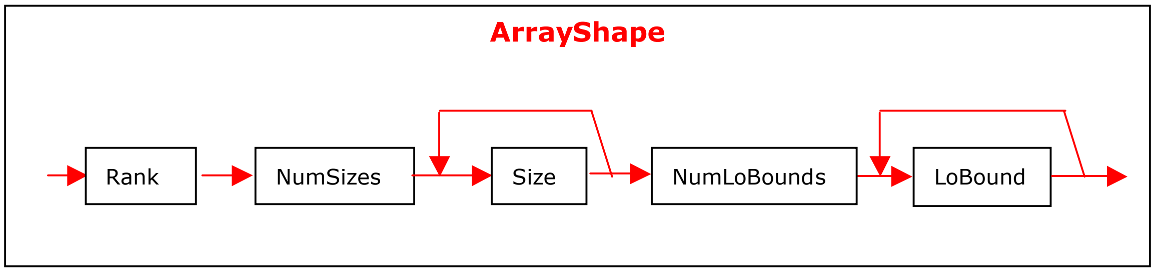
The ARRAY production describes the most general definition of an array – multi-dimensional, specifying size and lower bounds for each dimension.

### 10.12.3 SZARRAY CustomMod* Type

The SZARRAY production describes a frequently-used, special-case of ARRAY – that's to say, a single-dimension (rank 1) array, with a zero lower bound, and no specified size

## 10.13 ArrayShape

An ArrayShape has the following syntax chart:



*Rank* is an integer (compressed using *CorSigCompressData*) that specifies the number of dimensions in the array (must be 1 or more). *NumSizes* is a compressed integer that says how many dimensions have specified sizes (it must be 0 or more). *Size* is a compressed integer specifying the size of that dimension – the sequence starts at the first dimension, and goes on for a total of *NumSizes* items. Similarly, *NumLoBounds* is a compressed integer that says how many dimensions have specified lower bounds (it must be 0 or more). And *LoBound* is a compressed integer specifying the lower bound of that dimension – the sequence starts at the first dimension, and goes on for a total of *NumLoBounds* items. Note that you cannot 'skip' dimensions in these two sequences – but you are allowed to specify less than all *Rank* dimensions. Here are a few examples, all for element type I4:

| | Type | Rank | NumSizes | Size* | NumLoBounds | LoBound* |
|---|---|---|---|---|---|---|
| [0..2] | I4 | 1 | 1 | 3 | 0 | |
| [,,,,,,] | I4 | 6 | 0 | | | |
| [0..3, 0..2,,,,] | I4 | 6 | 2 | 4 3 | 0 | |
| [1..2, 6..8] | I4 | 2 | 2 | 2 3 | 2 | 1 6 |
| [5, 3..5, , ] | I4 | 3 | 2 | 5 3 | 2 | 0 3 |

Note that definitions can nest, since the Type may itself be an array

## 10.14 Short Form Signatures

The general specification for signatures leaves some leeway in how to encode certain items. For example, it appears legal to encode a String as either

- long-form: ( ELEMENT_TYPE_CLASS, TypeRef-to-System.String )
- short-form: ELEMENT_TYPE_STRING

Only the short form is valid. Below is a list of all possible long-form and short-form items. (As usual, for compactness, we miss out the ELEMENT_TYPE_ prefix – so VALUETYPE is short for ELEMENT_TYPE_VALUETYPE)

| **Long Form** | | **Short Form** |
|---|---|---|
| **Prefix** | **TypeRef to:** | |
| CLASS | System.String | STRING |
| CLASS | System.Object | OBJECT |
| VALUETYPE | System.Void | VOID |
| VALUETYPE | System.Boolean | BOOLEAN |
| VALUETYPE | System.Char | CHAR |
| VALUETYPE | System.Byte | U1 |
| VALUETYPE | System.SByte | I1 |
| VALUETYPE | System.Int16 | I2 |
| VALUETYPE | System.UInt16 | U2 |
| VALUETYPE | System.Int32 | I4 |
| VALUETYPE | System.UInt32 | U4 |
| VALUETYPE | System.Int64 | I8 |
| VALUETYPE | System.UInt64 | U8 |
| VALUETYPE | System.IntPtr | I |
| VALUETYPE | System.UIntPtr | U |
| VALUETYPE | System.TypedReference | TYPEDBYREF |

Note: arrays must be encoded in signatures using one of ELEMENT_TYPE_ARRAY or ELEMENT_TYPE_SZARRAY. There is no long form involving a TypeRef to System.Array

# 11 Attributes

Programmers can attach "custom attributes" to a programming element, such as a method or field. Each such "custom attribute" is defined, by the programmer, as a regular Type to metadata.

A "custom attribute" within metadata is a triple of (tokenParent, tokenMethod, blob) stored into metadata. The blob holds the arguments to the class constructor method specified by tokenMethod. The runtime (strictly speaking, Reflection) has a full understanding of the contents of this blob; on request, it will instantiate the AttributeObject that the blob represents, attaching it to the item whose token is tokenParent.

## *11.1 Using Attributes*

There are two steps in defining and using new Attributes. First, the programmer defines an AttributeClass, and the language emits that definition into the metadata, just as it would for any regular class. Here is an example of defining an AttributeClass, called Who, in C#:

```
public class Who {
    string name;
    string date;
    public Who(string n, string d) {name = n; date = d;}
}
```

Second, the programmer defines an instance of that AttributeClass (let's call it an AttributeObject) and attaches it to some programming element. Here is an example of defining two *Who* AttributeObjects and attaching them to the classes, Television and Refrigerator. Note that we define the AttributeObjects by providing literal string arguments to the *Who* constructor method:

```
[Who("Joe",  "Jan-2001")] class Television { . . . }
[Who("Bill", "Dec-2000")] class Refrigerator { . . . }
```

At runtime, you can instantiate the corresponding *Who* AttributeObjects, using Reflection. Here's the code for Television's Who:

```
System.Reflection.MemberInfo mi = typeof(Television);
object[] atts = mi.GetCustomAttributes(typeof(Who));
Who wt = (Who)atts[0];
```

Note that AttributeClasses are not distinguished in any way by the runtime – their definition within metadata looks just like any regular Type definition. Our use therefore of the name "AttributeClass" rather than just "Class" is simply aimed to help understanding. (CLS and .net framework impose restrictions in their design patterns, insisting that AttributeClasses all descend from System.Attribute, but the CLR itself does not care).

AttributeObjects can be attached to any item that has a metadata token: mdField, mdTypeDef, mdTypeRef, mdMethod, mdParameter, etc. Duplicates are supported, such that a given programming element may well have multiple AttributeObjects of the same AttributeClass attached to it. [so, in the example above, class Television might have two Who AttributeObjects – if the Television class were jointly invented by "Joe" and "Mary"]

It is legal to attach an AttributeObject to an AttributeClass. But we disallow attaching an AttributeObject to an AttributeObject.

AttributeClasses have the following characteristics:

- Require up-front design before AttributeObjects can be emitted
- Capitalize on the runtime infrastructure for class identity, structure, and versioning
- Allow tools, services, and third parties (the primary customers for this mechanism) to extend the types of information that may be carried in metadata without having to depend on the runtime to maintain and version that information
- Although each language or tool will provide a language-specific syntax and conventions for using Attributes, the self-describing nature of these Attributes will enable tools to provide drop-down lists and other developer aids
- Runtime Reflection support browsing over these Attributes, since they are self-describing.

## 11.2 Persisted Format of an AttributeObject

The data required to instantiate an AttributeObject is saved into metadata in three parts:

- Prolog
- Constructor arguments
- Named Fields or Properties

Each separate constructor argument, named field and named property is written into metadata using the same format (nearly) as used by the .NET binary serializer. [We make a few optimizations that avoid repeating information that already exists elsewhere in the metadata]

We define the format of these ‘pickled’ AttributeObjects, for only a subset of argument types – ints, chars, strings, and so on (see later).

It might help to have an example in mind, as we discuss the formats. Here is a simple one, written in C#

```
public class Attrib {
    public readonly string Name;
    public object   Whim;
    public int Depth { get{...}; set{...} }
    public Attrib(string n)        { this.Name = n; }
    public Attrib(string n, int d) { this.Name = n; this.Depth = d; }
}
[Attrib(“Monday”)]                        class Ex1 { ... }
[Attrib(“Tuesday”, 2)]                    class Ex2 { ... }
[Attrib(“Friday”,  Whim=42]               class Ex3 { ... }
[Attrib(“Green”,   Depth=3, Whim=”yellow”) class Ex4 { ... }
```

This example defines an AttributeClass called Attrib, with two fields – Name and Whim, and one property, Depth. It defines two constructors – the first takes one argument; the second takes two.

Following the definition of Attrib we show it used to attribute four classes called Ex1 through Ex4. Ex1 is hooked to an Attrib object using the single-argument constructor. Ex2 is hooked to an Attrib object using the two-argument constructor.

Ex3 is hooked to an Attrib object using a constructor which takes the one-argument constructor, and sets the named field Whim. The outcome of this is to instantiate an Attrib object with Name of "Friday" and Whim (an object field) holding the integer value 42. Finally, Ex4 is hooked to an Attrib object using the one-argument constructor, augmented by values for the Depth property and the Whim field.

Note that any class may have multiple AttributeObjects 'hooked' to it. These can be of different types, or even of the same type.

All binary data is persisted in **little-endian** format (least signficant bytes come first in the file). The format for floats and doubles is IEEE-754. For 8-byte doubles, the more-significant 4 bytes is emitted *after* the less-significant 4 bytes. There is just one exception to the little-endian rule – the "PackedLen" count that precedes a string – a one-two-or-four byte item – is always encoded big-endian.

Note that, if the constructor method takes no arguments, and you don't want to specify any extra named fields or properties, you can omit the blob entirely.

## 11.3 Prolog

The prolog simply identifies the blob that follows. It consists of a two-byte ID. In the first release, set this to the value 1.

The prolog is obviously a hedge against future extensions to this blob format.

## 11.4 Constructor Arguments

We define a new enumeration, SERIALIZATION_TYPE_, which specifies the types of data item in the blob. Where members correspond directly to runtime ELEMENT_TYPE_'s, we use the same name and value. Where members correspond to specific serialization types, we choose a value beyond the range used by the ELEMENT_TYPE_ enum. (See later for detailed list)

This spec provides a blow-by-blow account of how to serialize the following subset:

| | |
|---|---|
| SERIALIZATION_TYPE_BOOLEAN | SERIALIZATION_TYPE_CHAR |
| SERIALIZATION_TYPE_I1 | SERIALIZATION_TYPE_U1 |
| SERIALIZATION_TYPE_I2 | SERIALIZATION_TYPE_U2 |
| SERIALIZATION_TYPE_I4 | SERIALIZATION_TYPE_U4 |
| SERIALIZATION_TYPE_I8 | SERIALIZATION_TYPE_U8 |
| SERIALIZATION_TYPE_R4 | SERIALIZATION_TYPE_R8 |
| SERIALIZATION_TYPE_STRING | SERIALIZATION_TYPE_TYPE |
| SERIALIZATION_TYPE_FIELD | SERIALIZATION_TYPE_PROPERTY |

Also, a one-dimensional, zero-based array (SZARRAY) of any of those types.

The signature for any class constructor is stored in metadata, and indexed via its MethodDef. This signature specifies the number, order and type of each parameter. Therefore, we store the actual arguments into the PE file as dense binary, with no type descriptions and with no alignment packing. For each argument, emit the following data:

- For BOOLEAN, a single bytes, with False = 0 and True = 1
- For the intrinsics CHAR thru R8 in the table above, just their value (in their full field width)
- For STRING, a count of the number of **bytes** in the string (after encoding) followed immediately by the characters of the string in UTF8 format. (The

count is encoded as a "PackedLen" – see below details) Note that the count represents the overall length, in bytes, of the UTF8 sequence. In general, this is not the same as the number of UTF8 characters, since different UTF8 characters can occupy between 1 and 3 bytes
- For TYPE, a string that describes the type – see later for details
- For TAGGED_OBJECT,
- For ENUM, the actual value of its underlying type. [As a specific example, a particular Enum might use 4-byte integers as its underlying type, and should therefore be saved as a SERIALIZATION_TYPE_I4 value]
- For SZARRAY, the number of elements as an I4, followed by the value (in its full field width) of each element

If the AttributeClass provides several constructors, overload resolution to the appropriate MethodDef or MethodRef must be done at compile time (ie, no late-binding). This design does not perform automatic widening when we instantiate an AttributeObject via Reflection (for example, store 16 bit integer, but widen to signature's parameter type of 32 bits)

For the length-in-bytes of a UTF8 string, we use the standard 1,2 or 4 byte "PackedLen" encoding used within metadata (see the description of helper routine *CorSigCompressData* in section 10):

- If the length-in-bytes lies between 0 (0x00) and 127 (0x7F), inclusive, encode as a one-byte integer (bit #7 is obviously clear, integer held in bits #6 thru #0)
- If the length-in-bytes lies between 2^8 (0x80) and 2^14 – 1 (0x3FFF), inclusive, encode as a two-byte integer with bit #15 set, bit #14 clear (integer held in bits #13 thru #0)
- Otherwise, encode as a 4-byte integer, with bit #31 set, bit #30 set, bit #29 clear (integer held n bits #28 thru #0)
- A null string should be represented with the reserved single byte 0xFF, and no following data. (The value of 0xFF is a reserved value in metadata's count prefix)

The table below shows several examples. The first column shows an example count value (one-byte, two-byte and three-byte). The second column shows the corresponding size, expressed as a normal integer.

The table below shows several examples. The first column gives a value, expressed in familiar (C-like) hex notation . The second column shows the corresponding, compressed result, as it would appear in a PE file, with successive bytes of the result lying at successively higher byte offsets within the file. (This is the opposite order from how regular binary integers are laid out in a PE file)

| Original Value | Compressed Representation |
|---|---|
| 0x03 | 03 |
| 0x7F | 7F (7 bits set) |
| 0x80 | 8080 |
| 0x2E57 | BE57 |
| 0x3FFF | BFFF |
| 0x4000 | C000 4000 |
| 0x1FFF FFFF | DFFF FFFF |

Thus, by examining the most significant bits (the first ones encountered in a PE file) of a "compressed" field, you can determine whether it occupies 1, 2 or 4 bytes, as well as its value. For this to work, the "compressed" value, as explained above, is stored in **big-endian** order - most significant byte at the smallest offset within the file.

There is clearly scope to compact the above binary format, in the same way that existing metadata structures have been optimized to avoid "bloat". Possible techniques are legion. The first release of the runtime does **not** include any such optimizations (except for "PackedLen")

## 11.5 Constructor Arguments – Example 1

```
Foo (int a, char[] b, String c);

int a = 7;

char[] b = new char[] {'A', 'B', 'C', 'D'};

String c = "Today";

Foo (a, b, c);
```

Note that this example snippet uses a language that stores each "char" as a two-byte Unicode character (contrast with C++ single-byte "char"). The arguments to the *Foo* constructor would be encoded as follows:

```
0100 07000000 04000000 41424344 05 546F646179 0000
```

We start with the Prolog – a 2-byte value of 1. Next comes the first argument – a 4-byte value of 7. The second argument, a 4-element char array, is represented by a 4-byte count-of-array-elements with value 4, followed by the four ASCII characters A thru D (each "char" element starts as a 2-byte Unicode value, but is compressed into a single byte when converted into Utf8). The third argument consists of the UTF8-encoded string "Today"; its length in bytes (5) fits into a single count byte, followed by 5 characters, each encoded into a single byte. [I have added whitespace for clarity – it's not really there of course]). The last value is a two-byte value of zero, giving the total number of named fields and named properties (see later). Note that the display of bytes is the same as they would appear in memory – each byte occupies the next highest address in memory.

Note that you can specify an array of length zero – simply provide the 4-byte count-of-array-elements as 0x00000000. (Technically, this is different from a null array – we don't specify an encoding for this case)

## 11.6 Constructor Arguments – Example 2

```
Enum Colors {Red, Green, Blue};

Bar (object a, Colors b, bool[] c);

object a = "Hello";

Colors b = Colors.Green;

bool[] b = new bool[] {false, true, true};
```

```
Bar (a, b, c);
```

The arguments to the Bar constructor would be encoded as follows:

```
0100 0E 05 48656C6C6F 01000000 03000000 00 01 01 0000
```

The Prolog is followed by the first argument, a 1-byte ELEMENT_TYPE_STRING (0x0E) followed by 5-byte string for “Hello” – a one-byte count, plus 5 bytes of UTF8 encoded characters. (Note that Reflection knows from the signature for the Bar constructor, that its first argument is an object – we need only note that this argument is persisted as a string. The second argument is an enumeration with a 4-byte integer base type; we serialize Green as its value (of 1). The third argument is a 3-element BOOLEAN array – so we have a 4-byte element count with value 3, followed by 3 bytes for each boolean value, in order (False = 0, True = 1). (Recall that CLR BOOLEANs are stored with one byte per element). The last value is a two-byte value of zero, giving the total number of named fields and named properties (see later)

## 11.7 Constructor Arguments – Example 3

```
Zog (Object[] a, short[] b);
Object[] a = new Object[] {123, “Hello”, 11.0};
short[] b = new short[] {42, 7};
Zog (a, b);
```

Note: this example is contrived – I can’t find a CLR language that lets me do this, but if you can, then the blob it emits should be as follows:

```
0100 03000000 08 7B000000 0E 05 48656C6C6F 0D BA5E353F40100C49
02000000 2A00 0700
0000
```

The first argument is an object array with 3 elements; so we start with a 4-byte element count with value 3. Element 0 of the object array is an integer, which is encoded with a single-byte tag value of 08 (SERIALIZATION_TYPE_I4), followed by its 4-byte value (123 decimal, 7B hex). Element 1 of the object array is a String, encoded with a single-byte tag value of 0E (SERIALIZATION_TYPE_STRING), and followed by the byte-count of 05 and the UTF8 string for “Hello”. Element 2 of the object array is a double, so it starts with a single-byte tag value 0D (SERIALIZATION_TYPE_R8), and follows with the 8-byte binary floating-point representation for 11.0

The second argument is an array of 2 shorts. We start with a 4-byte count of elements. Then follows two shorts – 42 decimal (2A hex) and 7 decimal. The last value is a two-byte value of zero, giving the total number of named fields and named properties (see later).

## 11.8 Constructor Arguments – Example 4

An Attribute constructor may have an System.Type argument. In this case, the Attribute blob should contain the stringified, fully-qualified name of the target type. For example:

```
MyAttrib(Type t);
```

```
MyAttrib (typeof(System.Diagnostics.StackFrame));
```

The argument is persisted as a UTF8 string, described in previous examples. The format matches that of Reflection's AssemblyName. In this case, the string that's persisted is: "mscorlib, System.Diagnostics.StackFrame"

## 11.9 Named Fields and Properties

Named fields and properties are optional components for specifying an AttributeObject. We allow them to be specified in any order (languages may choose to impose tighter constraints). Therefore, the persisted blob defines each named field or property by recording a quad giving {FieldOrProperty, name, type, value}, in the obvious way.

We include Field-or-Property, as well as type, so that we can, at instantiation time, perform overload resolution of the named field or property.

We start with a 2-byte count specifying the total number of named fields and properties to follow. This count must always be supplied – if there are no named fields or properties for this AttributeObject, the count must be zero.

Whether each item is a field or property is specified with the one-byte tag SERIALIZATION_TYPE_FIELD or SERIALIZATION_TYPE_PROPERTY. The field or property name is encoded as a string – compacted byte-count plus UTF8 sequence. The type is encoded as its corresponding SERIALIZATION_TYPE_ member. Its value is similarly encoded exactly as described already.

## 11.10 Named Field – Example

```
using System;
using System.Reflection;
public class Who : Attribute {
   public string name;
   public string date;
   public string comment;
   public Who(string n, string d) {name = n; date = d;}
}
[Who("Joe", "Jan-2001", comment="Revisit")]
class Television { }

class Test {
   public static void Main() {
      MemberInfo mi = typeof(Television);
      object[] atts = mi.GetCustomAttributes(typeof(Who));
      Who w = (Who) atts[0];
      Console.WriteLine("w.comment = " + w.comment);
   }
}
```

The arguments to the Attrib constructor would be encoded as follows:

```
0100                           // prolog
03 4a6f65                      // "Joe"
08 4a616e2d32303031            // "Jan-2001"
0100                           // number of 'named' args
53                             // SERIALIZATION_TYPE_FIELD
0e 07636f6d6d656e74            // "comment"
07 52657669736974              // "Revisit"
```

## 11.11 General Case (not supported in V1)

This spec documents a subset of the general .NET binary serialization format, as an aid for compilers who wish to serialize AttributeObjects 'by-hand'. So, the format for saving AttributeObjects is piece-wise identical to the format used by CLR serialization to persist objects. That's to say, if you look at the binary layout for any constructor argument, it is identical (ok, we've included a couple of optimizations) to how it would look in a binary-serialized stream.

[The general-case serialized object includes extra fields. For example, each serialized object is assigned an ObjectID to support references to it from other objects in the graph. This is omitted for Strings in the serialized constructor arguments]

So, what if an attribute-class actually defined an argument of some user-defined class, Quix? How does the general serialized format look? The answer is, as you would expect, that the Quix object, as an argument to the attribute-class constructor, is persisted into the metadata, in the same format as if it had been instantiated as a regular common type system object and serialized.

Suppose the following example:

```
public class Attrib : System.Attribute {
    public readonly string Name;
    public Attrib(string n, Quix q) { . . . }
}
// Instantiate and setup aQuix
[Attrib("Friday", aQuix)] class Ex5 { . . . }
```

The arguments to the Attrib constructor would be encoded as follows:

```
0100 06 467269646179 0100 11 . . . .
```

We start with the Prolog – a 2-byte with value 1. Next comes the first argument – the String "Friday". Next we have the serialized aQuix – SERIALIZATION_TYPE_CLASS, then the binary blob for its field values.

Serializing arbitrary object graphs is clearly more complex than the subset of cases we have described above. Whilst the general case is addressed in the Spec for Binary Format Serialization, we will call out one aspect, that could arise in this last example. The Attrib constructor expects a Quix object; however, at compile time, it could be given an instance of a class derived from Quix. In this case, we need to include instance-type information, rather than just declaration-type information.

## 11.12 Arguments of type "Type"

As noted above, it is possible for a user to define an attribute constructor method that takes an argument of type "Type". What is stored into the custom attribute blob is a “stringified” representation of the type’s name. This section details that format.

Note that it is the Reflection component of the CLR that *understands* and uses custom attribute blobs at runtime, in order to instantiate AttributeObjects. The format of the type string is therefore made to agree with regular Reflection conventions.

The format, inside a custom attribute blob, for a .ctor argument of type **System.Type**, is a string that contains the fully-qualified name of the type. Prototypically – “Ozzy.OutBack.Kangaroo+Wallaby, MyAssembly”

However, Reflection, under certain conditions, tolerates a less-than-fully-qualified name: if you supply less info, Reflection will conduct a search for the Type string you provide. Here are the rules Reflection follows --

- You must always specify full namespace and name -- eg "a.b.c" If that's all you specify, Reflection looks for "a.b.c" throughout all the modules that comprise the current assembly (where "current" is the one that contains the CA blob we are parsing). If we find "a.b.c" in the current assembly, we're done. If not, then, *as a special case*, we look in MSCORLIB.DLL
- If the target Type (eg "x.y.z") is defined in a different assembly from the current one, and from MSCORLIB.DLL, then you **must** specify the full type name and assembly -- eg "x.y.z, MyAssembly"
- You can add even more info, to control exact versions, locale, etc -- in addition to the "x.y.z, MyAssembly". For example:

  x.y.z, MyAssembly, SN=a5d015c7d5a0b012, Loc=en, Ver=1.2.3.4
- If the target Type is an array, then emit the array dimensions in the opposite order from how they are represented in typical high-level languages. For example, if the user declares an array: int32 [,,,,] [ ] [,,] then it should be emitted into the custom attribute blob with dimensions *backwards* – that’s to say: int32 [,,][][,,,,]

(Note, in particular, that there's never any need to specify which module within an assembly holds the Type you're interested in)

## 11.13 SERIALIZATION_TYPE_ enum

| | |
|---|---|
| SERIALIZATION_TYPE_BOOLEAN | = ELEMENT_TYPE_BOOLEAN |
| SERIALIZATION_TYPE_CHAR | = ELEMENT_TYPE_CHAR |
| SERIALIZATION_TYPE_I1 | = ELEMENT_TYPE_I1 |
| SERIALIZATION_TYPE_U1 | = ELEMENT_TYPE_U1 |
| SERIALIZATION_TYPE_I2 | = ELEMENT_TYPE_I2 |
| SERIALIZATION_TYPE_U2 | = ELEMENT_TYPE_U2 |
| SERIALIZATION_TYPE_I4 | = ELEMENT_TYPE_I4 |
| SERIALIZATION_TYPE_U4 | = ELEMENT_TYPE_U4 |
| SERIALIZATION_TYPE_I8 | = ELEMENT_TYPE_I8 |
| SERIALIZATION_TYPE_U8 | = ELEMENT_TYPE_U8 |
| SERIALIZATION_TYPE_R4 | = ELEMENT_TYPE_R4 |
| SERIALIZATION_TYPE_R8 | = ELEMENT_TYPE_R8 |
| SERIALIZATION_TYPE_STRING | = ELEMENT_TYPE_STRING |
| SERIALIZATION_TYPE_TYPE | = 0x50 |
| SERIALIZATION_TYPE_FIELD | = 0x53 |

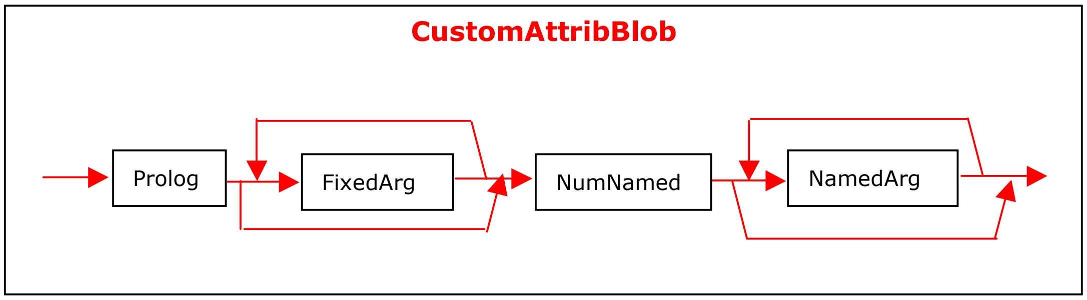
```
SERIALIZATION_TYPE_PROPERTY   = 0x54
```

# 12 CustomAttributes – Syntax

This section describes the layout of Custom Attribute blobs, stored in metadata, and already described in section 11. It makes no sense unless you have read that section (and even then).

Throughout this section, we use a shorthand: all upper-case names should be prefixed by SERIALIZATION_TYPE_. So for example, STRING is shorthand for SERIALIZATION_TYPE_STRING.

A Custom Attribute blob has the following syntax chart:

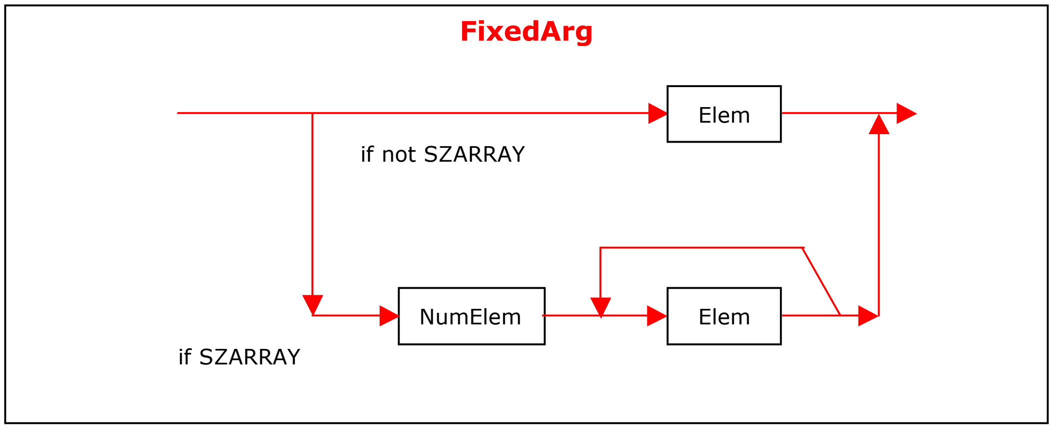


All binary values are stored in little-endian format (except PackedLen items – used only as counts for the number of bytes to follow in a Utf8 string)

*CustomAttribBlob* starts with a *Prolog* – a U2, with value 0x0001

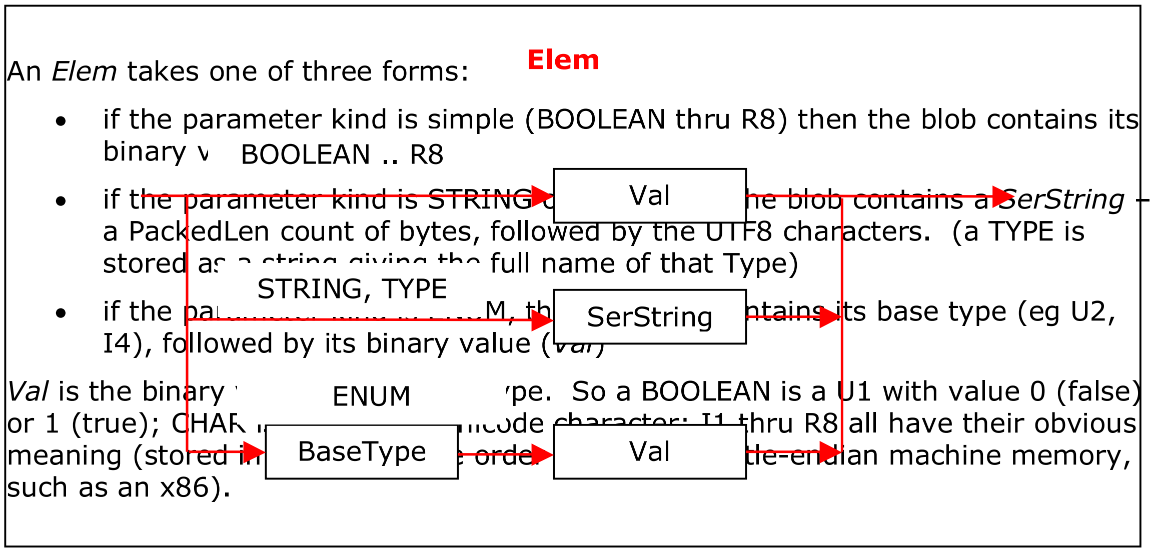
Next comes a description of the fixed arguments for the constructor method. Their number and type can be found by examining that constructor's MethodDef; this info is therefore *not* repeated in the *CustomAttribBlob* itself. As the syntax chart shows, there can be zero or more *FixedArgs*. (note that VARARG constructor methods are not allowed in the definition of Custom Attributes)

Next is a description of the optional "named" fields and properties. This starts with *NumNamed* – a U2 giving the number of "named" properties or fields that follow. Note that *NumNamed* must always be present. If its value is zero, there are no "named" properties or fields to follow (and of course, in this case, the CustomAttrib must end immediately after *NumNamed*) In the case where *NumNamed* is non-zero, it is followed by *NumNamed* repeats of *NamedArg*



The format for each *FixedArg* depends upon whether that argument is single, or an SZARRAY – this is shown in the upper and lower paths, respectively, of the syntax chart. So each *FixedArg* is either a single *Elem*, or *NumElem* repeats of *Elem*.

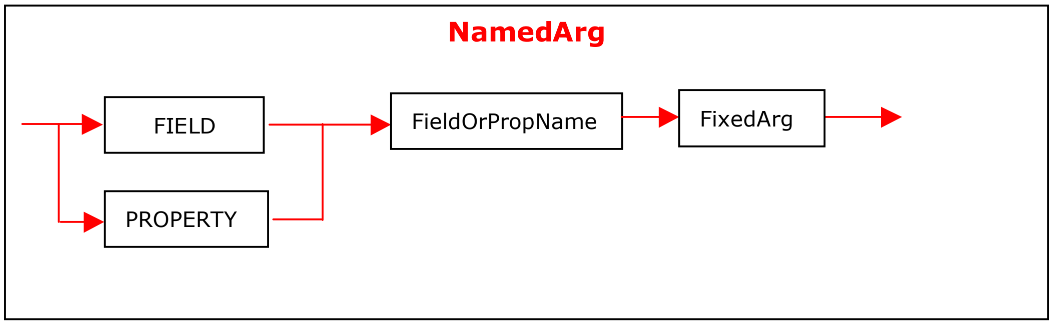
*NumElem* is a U4 specifying the number of elements in the SZARRAY



An *Elem* takes one of three forms:

- if the parameter kind is simple (BOOLEAN thru R8) then the blob contains its binary v[illegible]
- if the parameter kind is STRING o[illegible]he blob contains a *SerString* – a PackedLen count of bytes, followed by the UTF8 characters. (a TYPE is stored as a string giving the full name of that Type)
- if the pa[illegible]M, th[illegible]ntains its base type (eg U2, I4), followed by its binary value (*Val*)

*Val* is the binary [illegible]pe. So a BOOLEAN is a U1 with value 0 (false) or 1 (true); CHAR [illegible]nicode character; I1 thru R8 all have their obvious meaning (stored in[illegible] order[illegible]tle-endian machine memory, such as an x86).



A *NamedArg* is simply a *FixedArg* (discussed above) preceded by information to identify which field or property it represents. The *FieldOrPropName* is that name, stored, as usual, as a *SerString*.

FIELD is the single byte SERIALIZATION_TYPE_FIELD

PROPERTY is the single byte SERIALIZATION_TYPE_PROPERTY

The *SerString* used to encode an argument of type Type includes the namespace and type, followed optionally by the assembly where it is defined, its version, culture and public key token. If the assembly name is missed out, Reflection looks first in this assembly, and then the system assembly.

Here are some examples. Each shows the (attribute) class definition, its use, and a hex dump of the resulting blob. It also shows the text of the corresponding SerString embedded in the blob:

```
class TypeAtt : Attribute { public TypeAtt(Type t) { } }
[TypeAtt(typeof(short))]
0100                       // Prolog
0c                         // # bytes in string
53797374656d2e496e743136   // "System.Int16"
0000
```

```
class TypeAtt : Attribute { public TypeAtt(Type t) { } }
[TypeAtt(typeof(System.Drawing.Brush))]
0100 69                                     // Prolog
69                                          // # bytes in string
53797374656d2e44726177696e672e42727573682c // "System.Drawing.Brush,"
53797374656d2e44726177696e672c             // "System.Drawing,"
2056657273696f6e3d312e302e323431312e302c   // " Version=1.0.2411.0,"
2043756c747572653d6e65757472616c2c         // " Culture=neutral,"
205075626c69634b6579546f6b656e3d           // " PublicKeyToken="
62303366356637663131643530613361           // "b03f5f7f11d50a3a"
0000                                        // NumNamed
```

```
class TypeAtt : Attribute { public TypeAtt(Type t) { } }
[TypeAtt(typeof(int[,,,][][,]))]
0100                                        // Prolog
16                                          // # bytes in string
53797374656d2e496e743332                    // "System.Int32"
5b2c5d5b5d5b2c2c2c5d                        // "[,][][,,,]"
0000                                        // NumNamed
```

Note in this last case that arrays must be emitted *backwards* compared with their order in the source program of typical programming languages

# 13 Marshalling Descriptor

A Marshalling Descriptor is like a signature – it's a blob of binary data. It describes how a field or parameter (which, as usual, covers the method return, as parameter number 0) should be marshalled when calling to or from unmanaged coded via PInvoke dispatch or IJW ("It Just Works") thunking.

The blob has the following format –

*MarshalSpec* **:==**
 *NativeInstrinsic*
 | CUSTOMMARSHALLER Guid UnmanagedType ManagedType Cookie
 | FIXEDARRAY NumElem ArrayElemType
 | SAFEARRAY SafeArrayElemType
 | ARRAY ArrayElemType ParamNum ElemMult NumElem

*NativeInstrinsic* :==
 BOOLEAN | I1 | U1 | I2 | U2 | I4 | U4 | I8 | U8 | R4 | R8
 | BSTR | LPSTR | LPWSTR | LPTSTR | FIXEDSYSSTRING | STRUCT
 | INTF |FIXEDARRAY | INT | UINT | BYVALSTR | ANSIBSTR | TBSTR
 | VARIANTBOOL | FUNC | LPVOID | ASANY | LPSTRUCT | ERROR | MAX

For compactness, we have omitted the NATIVE_TYPE_ prefixes in the above lists. So, for example, "ARRAY" is shorthand for NATIVE_TYPE_ARRAY (see the CorNativeType enum defined in CorHdr.h) Note that *NativeIntrinsic* excludes those elements of the CorNativeType enum commented as "deprecated"

*Guid* is a counted-Utf8 string – eg "{90883F05-3D28-11D2-8F17-00A0C9A6186D}" – it must include leading **{** and trailing **}** and be exactly 38 characters long

*UnmanagedType* is a counted-Utf8 string – eg "Point"

*ManagedType* is a counted-Utf8 string – eg "System.Util.MyGeometry" – it must be the fully-qualified name (namespace and name) of a managed Type defined within the current assembly (that Type must implement ICustomMarshaller, and provides a "to" and "from" marshalling method)

*Cookie* is a counted-Utf8 string – eg "123" – an empty string is allowed

*NumElem* is an integer that tells us how many elements are in the array

*ArrayElemType* :==
 *NativeInstrinsic* | BOOLEAN | I1 | U1 | I2 | U2
 | I4 | U4 | I8 | U8 | R4 | R8 | BSTR | LPSTR | LPWSTR | LPTSTR
 | FIXEDSYSSTRING | STRUCT | INTF | INT | UINT | BYVALSTR
 | ANSIBSTR | TBSTR | VARIANTBOOL | FUNC | LPVOID | ASANY
 | LPSTRUCT | ERROR | MAX
The value MAX is used to indicate "no info"

*SafeArrayElemType* :== I2 | I4 | R4 | R8 | CY | DATE | BSTR | DISPATCH |
 | ERROR | BOOL | VARIANT | UNKNOWN | DECIMAL | I1 | UI1 | UI2
 | UI4 | INT | UINT
where each is prefixed by VT_. Note that these VT_xxx form a subset of the standard OLE constants (defined, for example, in the file WType.h that ships with Visual Studio, installed to the default directory "Program Files\Microsoft Visual Studion\VC98\Include")

*ParamNum* is an integer, which says which parameter in the method call provides the number of elements in the array – see below

*ElemMult* is an integer (says by what factor to multiply – see below)

For example, in the method declaration:

```
    Foo (int ar1[], int size1, byte ar2[], int size2)
```

The *ar1* parameter might own a row in the *FieldMarshal* table, which indexes a *MarshalSpec* in the Blob heap with the format:

```
    ARRAY  MAX  2  1  0
```

This says the parameter is marshalled to a NATIVE_TYPE_ARRAY. There is no additional info about the type of each element (signified by that NATIVE_TYPE_MAX). The value of *ParamNum* is 2, which tells us that parameter number 2 in the method (the one called "size1") will tell us the number of elements in the actual array – let's suppose its value on a particular call were 42. The value of *ElemMult* is 1. The value of *NumElem* is 0. The calculated total size, in bytes, of the array is given by the formula:

```
if ParamNum == 0
    SizeInBytes = NumElem * sizeof (elem)
else
    SizeInBytes = ( @ParamNum * ElemMult  +  NumElem ) * sizeof (elem)
endif
```

We have used the syntax "@*ParamNum*" to denote the value passed in for parameter number *ParamNum* – it would be 42 in this example. The size of each element is calculated from the metadata for the *ar1* parameter in *Foo*'s signature – an ELEMENT_TYPE_I4 of size 4 bytes.

Note that, just as in signature blobs, every simple scalar, such as integers or Utf8 byte-counts, are stored in compressed format, using the *CorSigCompressData* helper routines (see section 10 for details)

# 14 Metadata Specific to PInvoke

Platform Invocation Services, abbreviated "PInvoke", allows managed code to call unmanaged functions that are implemented in a DLL.  PInvoke takes care of finding and invoking the correct function, as well as marshalling its managed arguments to and from their unmanaged counterparts (integers, strings, arrays, structures, etc).

PInvoke was intended primarily to allow managed code to call existing, unmanaged code, typically written in C.  A good example is the several thousand functions that comprise the Win32 API.

As mentioned above, PInvoke marshals function arguments between managed and unmanaged code.  For simple data types (bytes, integers, floats, etc), or arrays of those simple types, marshalling is straightforward.  Even for strings, so long as you specify whether the unmanaged code expects an Ansi string, a Unicode string, or a BSTR, marshalling is again without problems.

But marshalling of structured arguments presents a problem.  (Structured types are also known as structs, records, aggregates, etc, depending upon which source language we are discussing.  We shall call them "structs"  in this spec).   Given free-rein, the runtime will lay out the fields of a managed struct in the 'most efficient' way.  What is 'most efficient'?  Well, it includes making garbage collection fast and space-efficient.  It can also take account of access patterns.  The point is, that the runtime's choice of layout will rarely match what unmanaged code (typically C) expects, and has hard-wired into its machine code as fixed offsets – where fields of a struct are laid out in the lexical order they were defined in the source code.

[As an aside, you might wonder how user's managed code can ever 'find' the right fields in a class which the runtime lays out in memory at its own whim.  The answer is that field access within MSIL is done via metadata tokens; in effect, these provide the 'name' of the field to be accessed, rather than its predefined byte offset within the managed struct]

So, somehow, at runtime, PInvoke must 'manufacture' and hand over a struct, holding fields in the exact order and size that unmanaged code expects.  The way it does this is firstly to disallow runtime's normal freedom for how it lays out managed structs (classes or valuetypes); instead, it directs the runtime to lay the struct out in managed memory in the way most-nearly expected by the unmananaged user of this struct.  We call such an item a "formatted type".

For many cases, we can achieve an exact, byte-by-byte, match between the managed object and the struct the unmanaged code expects; in these cases, we say the managed and unmanaged struct are "isomorphic".  When PInvoke calls the unmanaged code, it can either pin the managed object (so that it will not be moved by garbage collection), and hand a pointer to the managed code; or it can allocate some memory (unmanaged heap or stack) and do a fast, 'blind', byte-by-byte copy from the managed isomorphic object.  Either technique results in low overhead.

But there are some cases (non-isomorphic), where PInvoke must carry out marshalling – copying and reformatting of data – at runtime.   This is slower than if the struc were isomorphic.  The common cases which destroy isomorphism include:

- managed string is Unicode, but the unmanaged code expects Ansi
- managed argument or field is boolean; this occupies 1 byte in managed memory, but 2 or 4 bytes in unmanaged structures

A programmer can avoid the inefficiency incurred with managed boolean fields by declaring them as 2-byte or 4-byte integers instead.

In all other respects, except its predefined field layout in memory, a "formatted" object looks just like a regular managed object. In particular, managed code can read and write all its fields with MSIL instructions.

When it comes to call an unmanaged function, PInvoke locates the DLL where it lives, loads that DLL into process memory, finds the function address in memory, pushes its arguments onto the stack (marshalling if required) and transfers control to the address for the unmanaged code. If the arguments are isomorphic, then no marshalling is required.

## 14.1 Overview of PInvoke Metadata

This document specifies what information a tool or compiler must emit into metadata to describe how PInvoke should call an unmanaged function from the Runtime. This information includes the location of the target function (which DLL it lives in) and its signature (number of arguments, their type, and any function return type).

Each compiler provides a construct for its users to decorate methods and arguments with the required PInvoke information. For example, Managed Extensions for C++ provides the "sysimport" attribute, whilst Visual Basic provides the "DECLARE" statement. The compiler parses the decoration and emits the corresponding language-neutral metadata that will be used by PInvoke.

Information required by PInvoke falls into three kinds:

- Define the CLR method that corresponds to the unmanaged function. This includes its name, location, arguments and return type
- Where the unmanaged code expects a struct argument, define a CLR class that corresponds to the unmanaged struct – its fields, layout and alignment
- Where the default marshalling provided by PInvoke is not what you want, override with a different marshalling behaviour

Building PInvoke metadata can be quite simple. Here is an example (the source language doesn't matter; its intent should be clear):

```
class C {
    [sysimport(dll = "user32.dll")]
    public static extern int MessageBoxA(int h, string m, string c, int type);
    public static int Main() {
        return MessageBoxA(0, "Hello World!", "Caption", 0);
    }
}
```

To build the corresponding PInvoke metadata, you need only call *DefineMethod*, *DefinePinvokeMap* and *DefineParam* for each parameter. (Individual compilers may choose to structure their definitions differently – a *DefineMethod* followed by a *SetMethodProps*, for example, but the suggested sequence is possible)

On the other hand, building PInvoke metadata can also come quite involved, as witnessed by the size of this spec, and the other specs, listed below, that support it. This happens if your unmanaged function accepts struct arguments, and requires non-default marshalling. Here is a second, more complicated example:

```
[sysstruct(format=ClassFormat.Auto)]
public class LOGFONT {
   public const int LF_FACESIZE = 32;
   public int lfHeight;
   public int lfWidth;
   public int lfEscapement;
   public int lfOrientation;
   public int lfWeight;
   public byte lfItalic;
   public byte lfUnderline;
   public byte lfStrikeOut;
   public byte lfCharSet;
   public byte lfOutPrecision;
   public byte lfClipPrecision;
   public byte lfQuality;
   public byte lfPitchAndFamily;
   [nativetype(NativeType.FixedSysString, size=LF_FACESIZE)]
   public string lfFaceName;
};

class C {
   [sysimport(dll="gdi32.dll",charset=CharacterSet.Auto)]
   public static extern int CreateFontIndirect(
      [in, nativetype(NativeType.NativeTypePtr)]
      LOGFONT lplf   // characteristics
   );
   public static void Main() {
      LOGFONT lf = new LOGFONT();
      lf.lfHeight = 9;
      lf.lfFaceName = "Arial";
      int i = CreateFontIndirectA(lf);
      Console.WriteLine(i);
   }
}
```

To build the PInvoke metadata for this example requires calls to most of the following routines in the *IMetaDataEmit* interface:

*DefineMethod*: Define a method, with its CLR method signature

*DefinePinvokeMap:* Specify PInvoke info for a method

*DefineTypeDef*: Define a CLR class or valuetype, used as an argument to a PInvoke-dispatched function

*DefineField*: Define a data field within a class

*SetClassLayout*: Supply additional info on the class layout, such as field packing

*SetFieldMarshal:* Supply non-default marshaling for a function argument, function return value, or a field within a struct

For more info on specific areas touched upon in this spec, see the following documents:

- Platform Invoke Usage Guide for an overview of PInvoke from the user's perspective
- DataTypeMarshaling for details of all the field marshalling supported by Pinvoke (much of it shared with COM integration)

## 14.2 PInvoke Metadata for Methods

You must define a managed method, that describes the target unmanaged function you wish to reach via PInvoke. You may include several methods in a given class that describe unmanaged functions, or you can define a separate class and method for each unmanaged function; the choice is yours.

In the descriptions that follow, all metadata methods are defined on the *IMetaDataEmit* interface.

## 14.3 DefineMethod for PInvoke

For each unmanaged function you want to call via PInvoke, you must define a managed method, that describes that target unmanaged function. For this, use *DefineMethod.* This routine is used to define all managed methods to the metadata. However, when used for methods that match to PInvoke-dispatched native functions, some of the arguments have particular restrictions. These are listed in the next table.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | td | TypeDef token of parent | no |
| in | wzName | Member name in Unicode | yes |
| in | dwMethodFlags | Member attributes | yes |
| in | pvSig | Method signature | yes |
| in | cbSig | Count of bytes in pvSig | yes |
| in | ulCodeRVA | Address of code | must be 0 |
| in | dwImplFlags | Implementation flags for method | no, may be all 1s |
| out | pmd | Member token | |

*dwMethodFlags* is a bitmask from the *CorMethodAttr* enum in CorHdr.h. You must: set *mdStatic*; clear *mdSynchronized*; clear *mdAbstract*

*pvSig* must be a valid CLR method signature. Each parameter must be a valid managed (as opposed to unmanaged) data type. See earlier chapters of this spec for how to compose a CLR method signature; take special note of the *CorCallingConvention* enum in CorHdr.h

*ulCodeRVA* must be zero

*dwImplFlags* is a bitmask from the *CorMethodImpl* enum in CorHdr.h. You must: set *miNative*; set *miUnmanaged*

## 14.4 DefineMethodImpl for PInvoke

If you are defining the implementation for a method that is defined by an interface, you use *DefineMethodImpl.* This routine accepts only a subset of what *DefineMethod* accepts, because some of the inherited information cannot be changed (for example, the name of the method). Whilst this is used for regular managed methods, we do not support its use for PInvoke.

## 14.5 DefinePinvokeMap for PInvoke

Use *DefinePinvokeMap* to provide further information about a method already defined by the *DefineMethod* call above.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | tk | Token for target method – a MethodDef or MethodImpl | yes |
| in | dwMappingFlags | Flags used by Pinvoke to do the mapping | yes |
| in | wzImportName | Name of target export method in unmanaged DLL | no |
| in | mdImportDLL | mdModuleRef token for target DLL | yes |

*dwMappingFlags* is a bitmask from the *CorPinvokeMap* enum in CorHdr.h. You can set the following flags:

- *pmNoMangle* – if set, function name is used as-is in searching the target native DLL (ie, no fuzzy matching)
- *pmCharSetAnsi*, *pmCharSetUnicode*, *pmCharSetAuto* – set one as appropriate
- *pmSupportLastError* – if set, user can query last error set within the unmanaged method

## 14.6 SetPinvokeMap for PInvoke

Use *SetPinvokeMap* to provide further information, or change the information, you supplied in an earlier call to *DefinePinvokeMap*. The arguments, their meanings and restrictions are exactly as for *DefinePinvokeMap*, above.

## 14.7 Method Signatures for PInvoke

The call to *DefineMethod* includes an argument, called pvSig, that takes the signature of the method. This blob specifies the method's signature – the type for each argument, and for the return type, if any. The format of this blob is defined below. This section summarizes details of the signature that are specific to its use for PInvoke:

- All data types must be managed data types, even though they end up, after PInvoke dispatch, as arguments to an unmanaged function
- PInvoke provides default, automatic marshaling of simple (non-struct) arguments, and of simple fields within struct arguments. The defaults are chosen using heuristics about the managed data type declaration, target platform, and method-level ansi/unicode/auto attribute. (This default marshaling can be over-ridden if required – see *SetFieldMarshal*)
- In the method signature, a struct argument should be declared as a CLR class or valuetype that carries layout information (what we called a "formatted type" in the discussion above).

## 14.8 PInvoke Metadata for Function Parameters

You should specify the *direction* of each parameter to an unmanaged function. That's to say, whether it is an in, out, or inout parameter. In the cases where a copy of the corresponding argument is made for the unmanaged code to access (typically, for a non-isomorphic struct passed by-reference), the setting of these flags is important. In these cases, PInvoke does the following:

- **in**: make a copy of the managed struct for the unmanaged code to access. This struct is **not** copied back to the managed caller
- **out**: create a freshly-initialized, unmanaged struct for the unmanaged code to access. This struct **is** copied back to the managed caller
- **inout**: make a copy of the managed struct for the unmanaged code to access. This struct **is** copied back to the managed caller

Note that where a struct is isomorphic, but specified only as **in** or as **out**, PInvoke may, for reasons of efficiency, pin the managed struct and pass a reference to that struct to the unmanaged code. In such cases, the behaviour that results will be as if you had asked for **inout**.

## 14.9 DefineParam for PInvoke

To specify each parameter's *direction*, call *DefineParam*. Do not specify a default value – *dwDefType*, *pValue* or *cbValue*.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | md | Token for the method whose parameter is being defined | yes |
| in | ulParamSeq | Parameter sequence number | yes |
| in | wzName | Name of parameter in Unicode | no |
| in | dwParamFlags | Flags for parameter | no |
| in | dwDefType | ELEMENT_TYPE_* for the constant value | no |
| in | pValue | Constant value for parameter | no |
| in | cbValue | Size in bytes of pValue | no |
| out | ppd | ParamDef token assigned | |

*ulParamSeq* specifies the parameter sequence number, starting at 1. Use a value of 0 to mean the method return value

*wzName* is the name to give the parameter. If you specify null, this argument is ignored

*dwParamFlags* is a bitmask from the *CorParamAttr* enumeration in CorHdr.h. Set the *pdIn* and/or *pdOut* bits in this mask

## 14.10 SetParamProps for PInvoke

As an alternative to *DefineParam*, you may use *SetParamProps*. This is an unlikely scenario, except perhaps during an incremental compilation session. However, for the record, here is the detail – the same restrictions apply as for *DefineParam*

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | pd | Token for target parameter | yes |
| in | wzName | Name of parameter in Unicode | no |
| in | dwParamFlags | Flags for parameter | no |
| in | dwDefType | ELEMENT_TYPE_* for the constant value | no |
| in | pValue | Constant value for parameter | no |
| in | cbValue | Size in bytes of pValue | no |

## 14.11 PInvoke Metadata for Struct Arguments

As mentioned previously, a PInvoke-called function can accept struct arguments. Such arguments are expressed as classes or valuetypes that include layout information ("formatted types"). This section describes details of how to specify layout in the *DefineTypeDef* call you make to define those classes.

## 14.12 DefineTypeDef for PInvoke

Although supported, it is unlikely that a compiler will define methods or properties for a "formatted type" – that's to say, for a managed class or valuetype, whose purpose is to describe a matching unmanaged struct argument for a PInvoke-called function. Routinely, the type definition will include only fields – no methods, no properties, no superclass, no interfaces-to-implement. With these simplifications, the arguments to *DefineTypeDef* for a PInvoke struct, are as follows:

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | wzName | Name of type in Unicode | yes |
| in | dwTypeDefFlags | Typedef attributes | yes |
| in | tkExtends | Token of the superclass. Specify as zero | yes |
| in | rtkImplements[] | Array of tokens specifying the interfaces that this class or interface implements (inherits via interface inheritence). Specify as null | no |
| out | ptd | TypeDef token assigned | |

*dwTypeDefFlags* is a bitmask from the *CorTypeAttr* enum in CorHdr.h. You must set either *tdLayoutSequential*, or *tdExplicitLayout* (not both). You should set *tdAnsiClass*, *tdUnicodeClass* or *tdAutoClass*.

If your struct has no unions, then set *tdLayoutSequential*, and, if necessary, call *SetClassLayout* to provide more details. If you are in the unfortunate position that your struct includes unions (sometimes called overlays, depending upon source language), or your struct includes weird padding between fields, then you must set *tdExplicitLayout*, and follow with a call to *SetClassLayout* to provide more details.

The string formatting flags say how managed strings (which are always encoded in Unicode) should be marshalled to and from unmanaged code:

- *tdAnsiClass* – PInvoke will marshal to unmanaged Ansi

- *tdUnicodeClass* – PInvoke will pin, or copy, to unmanaged Unicode (no format change of the individual characters required)
- *tdAutoClass* – PInvoke will choose *tdAnsiClass* or *tdUnicodeClass*, by inspecting which platform it is being executed upon

## 14.13 DefineField for PInvoke

Having defined the struct using *DefineTypeDef*, the next step is to define each field in the struct, using *DefineField.* Just follow the usual rules for using *DefineField*; there are no special rules to apply just because these are fields of a struct that will be used for PInvoke. As a reminder, here are the arguments for the *Definefield* method:

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | td | Typedef token for the enclosing class | yes |
| in | wzName | Field name in Unicode | yes |
| in | dwFieldFlags | Field attributes | yes |
| in | pvSig | Field signature as a blob | yes |
| in | cbSig | Count of bytes in pvSig | yes |
| in | dwDefType | ELEMENT_TYPE_* for the constant value | no |
| in | pValue | Constant value for field | no |
| in | cbValue | Size in bytes of pValue | no |
| out | pmd | FieldDef token assigned | |

*dwFieldFlags* is a bitmask from the *CorFieldAttr* enumeration in CorHdr.h

*dwDefType* is a value from the *CorElementType* enumeration in CorHdr.h. If you do not want to define any constant value for this field, supply a value of ELEMENT_TYPE_END

## 14.14 SetClassLayout for PInvoke (Sequential)

If you told *DefineTypeDef* that your struct was tdLayoutSequential, then you should call *SetClassLayout* to further define the field layout.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | td | Token for the class being laid out | yes |
| in | dwPackSize | Packing size: 1, 2, 4, 8 or 16 bytes | no |
| in | rFieldOffsets | Array of mdFieldDef / ululByteOffset values for each field. Specify as zero | no |
| in | ulClassSize | Overall size of these class objects, in bytes | no |

*dwPackSize* is the packing size between adjacent fields. For each field in sequence, the runtime looks at its size, and current offset within the struct. It lays the field down to start at its natural offset, or the pack size, whichever results in the *smaller* offset. This matches precisely the semantics of the C and C++ #pragma pack compiler directive

*rFieldOffsets* is not required in this instance. Specify it as zero

*ulClassSize* is optional. If you specify this argument, then PInvoke will marshal this struct argument by making a blind, byte-by-byte copy of the managed object. [This technique is used by Visual C++]

## 14.15 SetClassLayout for PInvoke (Explicit)

If you told *DefineTypeDef* that your struct was tdExplicitLayout, then you must call *SetClassLayout* to further define the field layout.

| in/out | Parameter | Description | Required? |
|---|---|---|---|
| in | td | Token for the class being laid out | yes |
| in | dwPackSize | Packing size. Specify as zero | no |
| in | rFieldOffsets | Array of mdFieldDef / ululByteOffset values for each field on the class for which sequence or offset information is specified. Terminate array with mdTokenNil. | no |
| in | ulClassSize | Overall size of these class objects. Specify as zero | no |

*rFieldOffsets* is an array of *COR_FIELD_OFFSET*s. The *COR_FIELD_OFFSET* struct is defined in CorHdr.h, but repeated here for convenience:

```
typedef struct COR_FIELD_OFFSET {
    mdFieldDef  tokField;
    ULONG       ulOffset;
} COR_FIELD_OFFSET;
```

The *tokField* is the token for the target field; the *ulOffset* is the byte offset within the struct at which it starts. The struct is assumed to start at offset 0. (So, if you specify just one field, 4 bytes wide, with a *ulOffset* of 1000, then you create a managed struct that is 1004 bytes long). Terminate the *rFieldOffsets[]* array with a field token of *mdTokenNil*.

## 14.16 PInvoke Metadata for Explicit Marshalling

If the default marshaling provided by PInvoke is just what you need, then you can skip this section. However, if you want to specify non-default marshaling for any of the following items:

- function return value
- function argument
- field within a struct that is a function argument

then you must specify the requested behaviour using *SetFieldMarshal*.

Both PInvoke and COM integration provide marshaling of data between managed and unmanaged code. And for most data types, they share the same marshalling code. The marshalling behaviour is specified in the DataTypeMarshaling spec. Please consult this spec for details of PInvoke's default marshaling (in most cases the same as for COM integration), as well as the valid alternatives you may specify for non-default marshalling.

## 14.17 SetFieldMarshal for PInvoke

For each item that requires non-default marshaling, call *SetFieldMarshal* and specify which native type the item should be marshalled to.

| **in/out** | **Parameter** | **Description** | **Required?** |
|---|---|---|---|
| in | tk | Token for target item | yes |
| in | pvUnmgdType | Signature for unmanaged type | yes |
| in | cbUnmgdType | Count of bytes in pvUnmgdType | yes |

## 14.18 PInvoke Custom Attributes

Compilers can alternatively set marshaling information by emitting certain pre-defined Custom Attributes (eg the *MarshalAsAttribute*). This enables compilers to use their generic code, that parses and handles all Custom Attributes, to be used to direct the operation of the runtime for PInvoke marshalling.

Each compiler is free to choose which way it emits metadata – depending upon the tradeoff it chooses among the following factors:

1. compile-time speed and efficiency
2. ease of use
3. good argument checking
4. good isolation or *genericity*
5. whether the compiler itself is written in managed or unmanaged code

Broadly speaking, using unmanaged metadata-emit APIs is good for 1. Using unmanaged metadata-emit APIs, together with pre-defined Custom Attributes provides some level of 4, at the cost of slowing compilation. Use of Reflection Emit (only works for case 5) is good for 2, 3 and 4.

# 15 Minimal Metadata

*Most* of the information captured in CLR metadata is required in order for CLR to successfully execute the accompanying code. But not all. There are a few items in metadata that are optional – compilers can emit those items, or not. If they choose not to emit them, CLR can still execute the accompanying code; but you lose information that can be helpful in various scenarios – design-time browsing of type information, runtime serialization, remote invocation of methods, runtime debug support, runtime Reflection of type information, runtime profiling of a managed application, 'clean' disassembly of an image back to MSIL, COM interop, etc

There are two reasons why a compiler may choose not to emit such optional items of metadata:

- Avoid wasting space – disk space, memory and comms bandwidth
- Make the results of disassembling an EXE or DLL image less understandable to a human reader

This MiniSpec explains which parts of metadata can be omitted, and the consequences if compilers do so. It does not discuss implementation in any depth – only enough to explain space consumption, both for *Define* methods, and for subsequent *Set* methods.

Note that compilers should consider very carefully the benefits of omitting any metadata, or of providing a compiler switch that allows their users to do so. It might offer particular advantages for a 'pre-cooked' custom application – a proto-typical case would be that of an embedded application (eg a cellphone, or an automobile) that performs a dedicated, pre-defined, realtime task, where it is known ahead of time that the system will not need to support browsing, Reflection, serialization, or whatever.

However, omitting any metadata for regular desktop or server applications removes many of the advantages CLR has to offer (see above). This is **not** a step to be taken lightly.

Finally, a summary of this chapter is: there are no miracles! About the only item a compiler might contemplate *not* emitting is the definition of method parameters. Almost everything else is required. There are no massive savings lurking in the fine-print. The area most likely to reap rewards is to avoid inadvertently creating *remap* tables – see section 15.5.

## 15.1 Space Saving

When you store metadata in an image, it can use up three sorts of space:

- Disk space, for the saved EXE or DLL file
- Memory, when that metadata is imported by another compiler, or imported by the CLR itself at execution time
- Bandwidth, when transferring the image between computers, for mobile code scenarios

As you will see from this chapter, the total possible savings to be made by not emitting optional items of metadata is really very small, in comparison with the total size of a typical image. By contrast, you can make significant savings by being careful over *when* you emit information, rather than *what* information you emit – this is due to how metadata silently creates intermediate *remap* tables (see section 15.5)

This MiniSpec explains how much space is consumed, in the persisted image, for each item of metadata. And, by implication, of the savings to be made by *not* emitting those items. The savings depend upon where the item is stored, as follows:

### 15.1.1String Heap

This is where we keep program identifiers – names of classes, fields, methods, etc. Strings are stored as null-terminated Utf8 strings. So, if your identifiers use simple, ASCII characters, each letter takes up one byte in the heap. So, a field name like "Weight" consumes 7 bytes in the heap (6 for "Weight" plus one for the terminating "\0").

But, recall that we detect duplicates and store only one instance of each string. So if a compiland uses "Weight" in two different declarations, it is stored only once in the String heap.

### 15.1.2Blob Heap

This is where we keep genuine binary data that can legitimately embed nulls – for example, method and field signatures. The blob is stored as a size (number of bytes that follow, encoded as a *PackedLen*), followed by the byte array that represents the blob itself. *PackedLen* is a compressed integer, explained in detail elsewhere – see section 11.4. In summary, it occupies:

- 1 byte, if the length-in-bytes of the blob lies between 0 and 127
- 2 bytes, if the length-in-bytes of the blob lies between 2^8 and 2^14
- 4 bytes, otherwise

Like other heaps, we detect duplicates and store only one instance of each blob

### 15.1.3UserString Heap

This is where we keep user-defined strings. A typical example is a literal string like "HelloWorld", passed as an argument to a Console.WriteLine routine.

UserStrings are stored in their own Blob heap. They are stored in Unicode, exactly as provided by the compiler (via a call to *DefineUserString)* plus one trailing byte that records whether the string includes any characters that would prevent us doing a fast compare (specifically, a comma, a hyphen, or any character with code-point value >= 0x80). And like the regular String heap, we prepend a *PackedLen* size.

Like other heaps, we detect duplicates and store only one instance of each UserString.

### 15.1.4Metadata Tables

Metadata stores its information in tables – two-dimensional arrays. For example, the TypeDef table stores all the Types defined within a module. It has several columns, for the TypeDef's *Name, Flags, FieldList* and so on. Each Type definition corresponds to a row in that table. Each column stores a 2-byte or 4-byte value. That value is either a constant (eg a *Flags* bitmask), or an index into another metadata table, or one of the metadata heaps.  We decide at save-time, whether each column needs to be 2-byte or 4-byte by examining the largest value it holds.

Although we squeeze the stored *width* of each column at save-time, each table has a fixed *number* of columns.  That's to say, even if all the values for a certain column were zero in a given table (highly unlikely), we still store that entire column of zeroes; we do not squeeze it out altogether.

So, omitting optional metadata has two effects on the space taken up by metadata tables:

- each item not emitted saves a row in one (and sometimes more) metadata tables
- if the total number of rows in a given metadata table thereby falls below a threshold, it may trigger a column-width compression to 2 bytes

In the details that follow, we say how many columns relevant metadata tables hold.  So, if a given item of metadata creates a new row in, say, a 3-column table, then you know it uses up between 6 and 12 bytes of space in the persisted image (6 bytes if all 3 columns have values that can be encoded in 2-bytes, or 12 bytes if all 3 columns have values that require 4 bytes to encode those values)

## 15.2 Obfuscation

Many software providers are concerned that purchasers of their product might be able to reverse-engineer its algorithms or data structures, by disassembling, or decompiling, the binary images they ship.  This is perceived as a threat to safeguarding their Intellectual Property.  Where the software is shipped as conventional, binary images, this task is widely viewed as just taking too much effort to repay its cost.  But the task is made considerably easier by any system, CLR included, that embeds type information into the image, and includes code in any intermediate language, that is essentially unoptimized (and therefore easier to 'understand')

There are numerous techniques available to hide, or obfuscate both code, and its accompanying type information, to make harder the task of deciphering its algorithms and data structures.  This starts with the obvious, like giving all program identifiers 'nonsense' or hard-to-understand names, and advances through more innovative techniques to confuse the 'reader'.  There's also a long history of obfuscator tools, and de-obfuscator tools – each leap-frogging the other in their attempts to conceal/reveal the 'secrets' of shipped images.  This MiniSpec does not address the area of 'active' obfuscation.

However, not emitting optional items of metadata helps conceal an image's 'secrets' too, by making the results of disassembly or decompilation, perhaps a little harder to understand – a kind of 'passive' obfuscation.  This MiniSpec will help compilers understand what can be achieved in this direction.  But it's left up to each reader to decide how much each omission achieves as a step towards obfuscation.

## 15.3 Define and Set

You can use a *Define* method to emit metadata.  Subsequently, you can use a companion *Set* method to modify the information you already emitted.  If the information changed by the *Set* call is stored in a metadata table – typically a *Flags* field, then the *Set* call updates the in-memory row of that table, and consumes no extra space. However, if the information to be changed lies in one of the heaps, we create the new info in the heap, and update indexes in metadata tables to point to that new info. But, we do not delete the old heap info – that's to say, the metadata engine does *not* shift down all subsequent items in the heap to squeeze that dead info out of existence. (Of course, the info would only be candidate for such deletion if it were not referenced from elsewhere – recall that we share duplicate info in heaps)

This strategy works fine for 'well-behaved' compilers, that emit metadata via *Define* calls, once they have gathered all the information required from their source files. But it's certainly possible for a naive compiler to waste lots of space in metadata by injudicious use of *Defines* followed by *Sets.*

## 15.4 One Big Module

In principle, by merging lots of separate modules into one big module, it should be possible to eliminate all *internal* identifier name strings (stored in the String heap) – that's to say, the names of all identifiers that are defined within our one big module.

In the simplest case, think of building one big DLL from a single source file, rather than 5 small DLLs from 5 small source files. The resulting MSIL code embeds metadata *def* tokens (TypeDef, FieldDef, MethodDef, etc) and does not require CLR to perform runtime string-matching to resolve any references. (it still needs to retain names for references to any 'external' modules, of course).

This strategy, if supported, would clearly save considerable space, and provide some minimal level of obfuscation. (On the downside, it would render the output of browsers, debuggers, or Reflection queries, virtually useless)

However, this scenario is not supported: CLR itself uses these text names to build hash tables for fast runtime lookup of information – even though it turns out that we never *actually* need subsequently to lookup that info. This is the situation for V1; it *might* be reviewed post V1.

On a related topic, a compiler might imagine it could save space in metadata by avoiding emitting Refs (TypeRef, MemberRef) to any items that in fact are defined in the same module – simply work out the correct Def and emit that instead. This strategy is always possible, but at extra cost in the buffering and complexity of the compiler. However, the metadata engine already performs automatic Ref-to-Def folding at save time, so any such attempt by the compiler would be wasted effort!

[Ref-to-Def folding describes the situation where a compiler emits references (TypeRef, MemberRef) to items that are actually defined, earlier or later during compilation, within the same module. The metadata engine replaces those Refs with their corresponding Defs, and discards the now-redundant Refs. The same holds true when the VC Linker calls metadata's Merge() routine to combine the two or more OBJs into a single image]

## 15.5 Order of Emission

If you emit metadata in a certain order, then the metadata engine can store the resulting data in a compact form. If you break these ordering constraints, then everything still works, but the metadata engine silently creates intermediate *remap* tables. These take up extra space in the PE file, and are slower to query at runtime. Each *remap* table has a single column (2 or 4 bytes wide, as required – see section 15.1.4).

Note that you need only emit one single item out-of-order to trigger creating a *remap* table. For example, if you emit a single Field definition out-of-order, the metadata engine creates a *remap* table with one row for each Field that is defined in the entire module.

The reasons for, and rules surrounding, order of emission, are explained in section 3.9. The following list summarizes the cost in disk space of emitting out-of-order:

- Field : *remap* table with one row for each Field (includes any global variables) defined in the module
- Method : *remap* table with one row for each Method (includes any global functions) defined in the module
- Param : *remap* table with one row for each Param defined in the module
- Property : *remap* table with one row for each Property defined in the module
- Event : *remap* table with one row for each Event defined in the module

Note that this single factor – avoiding emitting metadata definitions out-of-order – likely saves more space than all other factors put together.

## 15.6 Properties and Events

A Property definition is recorded in metadata as essentially a collection of Methods, associated with a given Type definition – and not much else. Moreover, the CLR itself does not *understand* Properties – it operates upon the specific Methods that go to make up the Property. From most perspectives, you can think of Property definitions within metadata as syntactic sugar, generated by compilers, and used by browsers. This is not to disparage their usefulness – it just highlights that the abstraction really is built at a higher level, without much help from metadata or the CLR.

The same arguments apply to Events.

So, if a given compiler system were intent on creating an executable image, that consumes the very minimum of metadata, it could choose not to emit Property or Event metadata; instead it would directly reference the Methods that comprise each Property or Event.

Clearly, browsers or Reflection would not report any Properties or Events – instead, just a large number of methods with unusual names (set_foo, get_foo, etc). But the savings in metadata space do mount up, as follows:

Each Type that defines a Property consumes one row of the 2-column “PropertyMap” table

Each Property defined for a given Type consumes one row of the 4-column “Property” table, plus its name (String heap), plus its signature (Blob heap)

Each Method defined for a given Property (*setter, getter, OtherMethods[]*) consumes one row of the 3-column “MethodSemantics” table

This is in addition to the space consumed for each Method – that’s to say, the space consumed if the Method were a regular, non-Property-owned, method.

For events, the corresponds savings are similar:

Each Type that defines an Event consumes one row of the 2-column “EventMap” table

Each Event defined for a given Type consumes one row of the 3-column “Event” table, plus its name (String heap)

Each Method defined for a given Event (*AddOn, RemoveOn, Fire* and *OtherMethods[]*) consumes one row of the 3-column “MethodSemantics” table

This is in addition to the space consumed for each Method – that’s to say, the space consumed if the Method were a regular, non-Event-owned, method.

## 15.7 NGen

CLR supports ‘pre-compiling’ of assemblies, using the Ngen tool (Native Code Generator). That’s to say, rather than JIT-compile each method from MSIL to native code, on-demand at runtime, we compile all of the MSIL throughout the files that comprise the assembly, and save them to disk. Then, at runtime, we can simply executed the pre-compiled, native code.

How does Ngen affect the “minimal metadata” picture?

The answer is (at least for V1 of CLR) – not at all. The entire content of metadata in the original files is kept, without change, in the original file.

## 15.8 Details

This section defines the space used to store each item of metadata, and says which are optional. Methods are described in the same order as they are documented in the Metadata API spec.

### 15.8.1DefineAssembly

```
STDAPI DefineAssembly(const void *pbOriginator,
                      ULONG cbOriginator,
                      ULONG ulHashAlgId,
                      LPCWSTR szName,
                      const ASSEMBLYMETADATA *pMetaData,
                      DWORD dwAssemblyFlags,
                      mdAssembly *pma)
```

A call to *DefineAssembly* creates a row in the 10-column “Assembly” table. In addition:

*pbOriginator/cbOriginator* are stored in the Blob heap (see section15.1.2)

*szName* is stored in the String heap (see section 15.1.1)

The *fields of pMetaData* are disposed as follows:

- *szLocale* stored in String heap (see section 15.1.1)
- *rProcessor/ulProcessor* stored into successive rows of the 1-column “AssemblyProcessor” table
- *rOS/ulOS* – an array of OSINFOs (with fields *dwOSPlatformId, dwOSMajorVersion, dwOSMinorVersion)* – stored into successive rows of the 3-column “AssemblyOS” table

- *szConfiguration* stored in the String heap (see section 15.1.1)

The following arguments are informational, and are ignored for execution: *pMetaData->rProcessor, pMetaData->rOS, pMetaData->szConfiguration.*

The *pbOriginator/cbOriginator* is required if this assembly is to be stored in the shared Assembly cache cache.

Recall that binding of Assembly*Ref*s to Assembly*Def*s depends upon the fields supplied via the following arguments:

- Assembly name = *szName*
- Public Key = *pbOriginator/cbOriginator* (if present)
- Version = *pMetaData->usMajorVersion, usMinorVersion, usRevisionNumber, usBuildNumber*
- Locale = *pMetaData->szLocale/cbLocal*

### 15.8.2DefineFile

```
STDAPI DefineFile(LPCWSTR szName,
                  const void *pbHashValue,
                  ULONG cbHashValue,
                  DWORD dwFileFlags,
                  mdFile *pmf)
```

Each call to *DefineFile* creates a row in the 3-column "File" table.  In addition:

*szName* is stored in the String heap (see section 15.1.1)

*pbHashValue/cbHashValue* are stored in the Blob heap (see section15.1.2)

All arguments are required and used for execution.

### 15.8.3DefineExportedType

```
STDAPI DefineExportedType(LPCWSTR szName,
                     mdToken tkImplementation,
                     mdTypeDef tkTypeDef,
                     DWORD dwExportedTypeFlags,
                     mdExportedType *pmdct)
```

Each call to *DefineFile* creates a row in the 5-column "ExportedType" table.  In addition:

*szName* is stored in the String heap (see section 15.1.1)

### 15.8.4DefineManifestResource

```
STDAPI DefineManifestResource(LPCWSTR szName,
                              LPCWSTR szDescription,
                              mdToken tkImplementation,
                              DWORD dwOffset,
                              DWORD dwResourceFlags,
                              mdManifestResource *pmmr)
```

Each call to *DefineManifestResource* creates a row in the 7-column "ManifestResource" table. In addition:

*szName, szDescription, szMIMEType* and *szLocale* are stored in the String heap (see section 15.1.1)

*szDescription* is informational and ignored for execution.

### 15.8.5SetModuleProps

```
HRESULT SetModuleProps(LPCWSTR wzName)
```

A call to *SetModuleProps* updates the automatically-created, single row in the 3-column "Module" table. In addition:

*wzName* is stored in the String heap (see section 15.1.1).

*wzName* is not required for execution. If you *do* supply a name, then CLR checks that the name matches the name supplied in any ModuleRefs elsewhere in the Assembly (and this check is a case-sensitive match). If you do *not* supply a name, then ModuleRef lookups are based solely on the name by which the module is known to the Assembly cache resolving service – typically, the name of the file in which it is held within a Win32 file system (and these matches are made case-blind)

### 15.8.6DefineCustomAttribute

```
HRESULT DefineCustomAttribute(mdToken tkOwner,
                              mdToken tkAttrib,
                              void const *pBlob,
                              ULONG cbBlob,
                              mdCustomAttribute *pca)
```

Each call to *DefineCustomAttribute* creates a row in the 3-column "CustomAttribute" table. In addition:

*pBlob/cbBlob* are stored in the Blob heap (see section 15.1.2). If the constructor method has no arguments (that's to say, its simple absence or presence is all that's required to convey what's required), then you don't need to emit a Blob, even one of length zero.

### *15.8.7SetCustomAttributeValue*

```
HRESULT SetCustomAttributeValue(mdCustomAttribute pca,
                                void const *pBlob,
                                ULONG cbBlob)
```

Each call to *SetCustomAttribute* updates a row in the 3-column "CustomAttribute" table. In addition:

*pBlob/cbBlob* are stored in the Blob heap (see section 15.1.2). Metadata will not reclaim the space occupied in the heap by a previously-defined *pBlob/cbBlob* (see section 15.3)

### 15.8.8DefineTypeDef

```
HRESULT DefineTypeDef(LPCWSTR wzName,
                      DWORD dwTypeDefFlags,
                      mdToken tkExtends,
                      mdToken rtkImplements[],
                      mdTypeDef *ptd)
```

Each call to *DefineTypeDef* creates a row in the 9-column “TypeDef” table.  In addition:

*wzName* is stored in the String heap (see section 15.1.1).  You *must* supply a non-null *wzName* (see section 15.4)

### 15.8.9SetTypeDefProps

```
HRESULT SetTypeDefProps(mdTypeDef td,
                        DWORD dwTypeDefFlags,
                        mdToken tkExtends,
                        DWORD mdToken rtkImplements[])
```

Each call to *SetTypeDefProps* updates a row in the 9-column “TypeDef” table.  In addition:

### 15.8.10 DefineMethod

```
HRESULT DefineMethod(mdTypeDef td,
                     LPCWSTR wzName,
                     DWORD dwMethodFlags,
                     PCCOR_SIGNATURE pvSig,
                     ULONG cbSig,
                     ULONG ulCodeRVA,
                     DWORD dwImplFlags,
                     mdMethodDef *pmd)
```

Each call to *DefineMethod* creates a row in the 6-column “Method” table.  In addition:

*wzName* is stored in the String heap (see section 15.1.1).  You *must* supply a non-null *wzName* (see section 15.4)

*pvSig/cbSig* are stored in the Blob heap (see section 15.1.2)

### 15.8.11 SetMethodProps

```
HRESULT SetMethodProps(mdMethodDef md,
                       DWORD dwMethodFlags,
                       ULONG ulCodeRVA,
                       DWORD dwImplFlags)
```

Each call to *SetMethodProps* updates a row in the 6-column "Method" table. No other space is consumed.

## 15.8.12 DefineField

```
HRESULT DefineField(mdTypeDef td,
                    LPCWSTR wzName,
                    DWORD dwFieldFlags,
                    PCCOR_SIGNATURE pvSig,
                    ULONG cbSig,
                    DWORD dwDefType,
                    void const *pValue,
                    ULONG cchValue,
                    mdFieldDef *pmd)
```

Each call to *DefineField* creates a row in the 3-column "Field" table. In addition:

*wzName* is stored in the String heap (see section 15.1.1). You *must* supply a non-null *wzName* (see section 15.4)

*pvSig/cbSig* are stored in the Blob heap (see section 15.1.2)

*pValue/cchValue* are stored in the Blob heap (see section 15.1.2). These are entirely optional (used to record a default value for a field – that value can be inspected at compile-time). If you choose to store such a value, it consumes space in both the Blob heap, as well as one row in the 3-column "Constant" table.

## 15.8.13 SetFieldProps

```
HRESULT SetFieldProps(mdFieldDef fd,
                      DWORD dwFieldFlags,
                      DWORD dwDefType,
                      void const *pValue,
                      ULONG cchValue)
```

Each call to *SetFieldProps* updates a row in the 3-column "Field" table. In addition:

*pValue/cchValue* are stored in the Blob heap (see section 15.1.2). Metadata will not reclaim the space occupied in the heap by a previously-defined *pValue/cchValue* (see section 15.3)

## 15.8.14 DefineNestedType

```
HRESULT DefineNestedType(LPCWSTR wzName,
                         DWORD dwTypeDefFlags,
                         mdToken tkExtends,
                         mdToken rtkImplements[],
                         mdTypeDef tdEncloser,
                         mdTypeDef *ptd)
```

Each call to *DefineNestedType* creates a row in the 9-column "TypeDef" table. It also creates a row in the 2-column "NestedClass" table. It differs from the *DefineType* method only in its *tdEncloser* argument (which ends up as one column of the "NestedClass" table). See *DefineTypeDef* (section 15.8.8) for further details on the space consumed by the other arguments.

### 15.8.15 DefineParam

```
HRESULT DefineParam(mdMethodDef md,
                    ULONG ulParamSeq,
                    LPCWSTR wzName,
                    DWORD dwParamFlags,
                    DWORD dwDefType,
                    void const *pValue,
                    ULONG cchValue,
                    mdParamDef *ppd)
```

This is one of the few genuine cases where you can freely omit metadata, yet the code will still execute successfully!

Each call to *DefineParam* creates a row in the 3-column "Param" table. In addition:

*wzName* is stored in the String heap (see section 15.1.1). It is informational and not used for execution.

*pValue/cchValue* are stored in the Blob heap (see section 15.1.2). These are entirely optional (used to record a default value for the parameter – that value can be inspected at compile-time). If you choose to store such a value, it consumes space in both the Blob heap, as well as one row in the 3-column "Constant" table.

### 15.8.16 SetParamProps

```
HRESULT SetParamProps(mdParamDef pd,
                      LPCWSTR wzName,
                      DWORD dwParamFlags,
                      DWORD dwDefType,
                      void const *pValue,
                      ULONG cchValue)
```

Each call to *SetParamProps* updates a row in the 3-column "Param" table. In addition:

*wzName* is stored in the String heap (see section 15.1.1). Metadata will not reclaim the space occupied in the heap by a previously-defined *wzName* (see section 15.3)

*pValue/cchValue* are stored in the Blob heap (see section 15.1.2). Metadata will not reclaim the space occupied in the heap by a previously-defined *pValue/cchValue* (see section 15.3)

### 15.8.17 DefineMethodImpl

```
HRESULT DefineMethodImpl(mdTypeDef td,
                         mdToken tkBody,
                         mdToken tkDecl)
```

Each call to *DefineMethodImpl* creates a row in the 3-column "MethodImpl" table.

### 15.8.18 SetRVA

```
HRESULT SetRVA(mdMethodDef md,
               ULONG ulRVA)
```

Each call to *SetRVA* updates a row in the 6-column "Method" table.  It consumes no extra space.

### 15.8.19 SetFieldRVA

```
HRESULT SetFieldRVA(mdFieldDef fd,
                    ULONG ulRVA)
```

It is rare to assign a field an RVA.  So we don't waste space by defining a column for RVA in the 3-column "Field" table.  So, when you call *SetFieldRVA,* it creates a row in the 2-column "FieldRVA" table.  Subsequent calls to *SetFieldRVA* for the same field simply update that row, without consuming extra space (though they do indicate an insane compiler-writer)

### 15.8.20 DefinePinvokeMap

```
HRESULT DefinePinvokeMap(mdToken tk,
                         DWORD dwMappingFlags,
                         LPCWSTR wzImportName,
                         mdModuleRef mrImportDLL)
```

You should define PInvoke information only if the target method is definitely unmanaged, and to be reached via PInvoke dispatch.  Otherwise, such definition creates redundant metadata (which is not detected as such)

Each call to *DefinePinvokeMap* creates a row in the 4-column "ImplMap" table.  In addition:

*wzImportName* is stored in the String heap (see section 15.1.1).  You *must* supply a non-null *wzImportName* (see section 15.4) – it represents the name of the target, unmanaged function (or a string-encoded number like "#124" for import-by-ordinal)

### 15.8.21 SetPinvokeMap

```
HRESULT SetPinvokeMap(mdToken tk,
                      DWORD dwMappingFlags,
                      LPCWSTR wzImportName,
                        mdModuleRef mrImportDLL)
```

Each call to *SetPinvokeMap* updates a row in the 4-column "Param" table. In addition:

*wzImportName* is stored in the String heap (see section 15.1.1). Metadata will not reclaim the space occupied in the heap by a previously-defined *wzImportName* (see section 15.3)

### 15.8.22 SetFieldMarshal

```
HRESULT SetFieldMarshal(mdToken tk,
                        PCCOR_SIGNATURE pvUnmgdType,
                        ULONG cbUnmgdType)
```

When you call *SetFieldMarshal* it creates a row in the 2-column "FieldMarshal" table. Subsequent calls to *SetFieldMarshal* for the same field simply update that row, without consuming extra space (this would be an unusual action for any compiler). In addition:

*pvUnmgdType/cbUnmgdType* are stored in the Blob heap (see section 15.1.2). Recall that metadata will not reclaim the space occupied in the heap by a previously-defined *pvUnmgdType/cbUnmgeType* (see section 15.3)

### 15.8.23 DefineAssemblyRef

```
STDAPI DefineAssemblyRef(const void *pbOriginator,
       ULONG cbOriginator,
       LPCWSTR szName,
       const ASSEMBLYMETADATA *pMetaData,
       const void *pbHashValue,
       ULONG cbHashValue,
       DWORD dwAssemblyRefFlags,
       mdAssemblyRef *pmar)
```

Each call to *DefineAssemblyRef* creates a row in the 10-column "AssemblyRef" metadata table. In addition:

*pbOriginator/cbOriginator* are stored in the Blob heap (see section15.1.2)

*szName* is stored in the String heap (see section 15.1.1)

The *fields of pMetaData* are disposed as follows:

- *szLocale* stored in String heap (see section 15.1.1)
- *rProcessor/ulProcessor* stored into successive rows of the 1-column "AssemblyRefProcessor" metadata table
- *rOS/ulOS* – an array of OSINFOs (with fields *dwOSPlatformId, dwOSMajorVersion, dwOSMinorVersion)* – stored into successive rows of the 3-column AssemblyRefOS" metadata table
- *szConfiguration* stored in the String heap (see section 15.1.1)

*pbHashValue/cbHashValue* are stored in the Blob heap (see section 15.1.2)

The following arguments are informational, and are ignored for execution: *pMetaData->rProcessor, pMetaData->rOS, pMetaData->szConfiguration, pbHashValue/cbHashValue.*

## 15.8.24 DefineTypeRefByName

```
HRESULT DefineTypeRefByName(mdToken tkResScope,
                            LPCWSTR wzName,
                            mdTypeRef *ptr)
```

Each call to *DefineTypeRefByName* creates a row in the 3-column “TypeRef” table. In addition:

*wzName* is stored in the String heap (see section 15.1.1). You *must* supply a non-null *wzName* (see section 15.4) – it is the *essence* of a TypeRef

## 15.8.25 DefineImportType

```
HRESULT DefineImportType(IMetaDataAssemblyImport *pAssemImport,
                         const void *pbHashValue,
                         ULONG cbHashValue,
                         IMetaDataImport *pImport,
                         mdTypeDef tdImport,
                         IMetaDataAssemblyEmit *pAssemEmit,
                         mdTypeRef *ptr)
```

This method looks up the metadata in another module and uses it to create a TypeRef in the current scope. It consumes space in exactly the same manner as a call to *DefineTypeRefByName* (see section 15.8.24)

## 15.8.26 DefineMemberRef

```
HRESULT DefineMemberRef(mdToken tkImport,
                        LPCWSTR wzName,
                        PCCOR_SIGNATURE pvSig,
                        ULONG cbSig,
                        mdMemberRef *pmr)
```

Each call to *DefineMemberRef* creates a row in the 3-column “MemberRef” table. In addition:

*wzName* is stored in the String heap (see section 15.1.1). You *must* supply a non-null *wzName* (see section 15.4) – it is essential to a MemberRef

*pvSig/cbSig* are stored in the Blob heap (see section 15.1.2)

## 15.8.27 DefineImportMember

```
HRESULT DefineImportMember(IMetaDataAssemblyImport *pAssemImport,
        const void *pbHashValue,
        ULONG cbHashValue,
        IMetaDataImport *pImport,
        mdToken mbMember,
        IMetaDataAssemblyEmit *pAssemEmit,
        mdToken tkParent,
        mdMemberRef *pmr)
```

This method looks up the metadata in another module and uses it to create a MemberRef in the current scope. It consumes space in exactly the same manner as a call to *DefineMemberRef* (see section 15.8.26)

## 15.8.28 DefineModuleRef

```
HRESULT DefineModuleRef(LPCWSTR wzName,
                        mdModuleRef *pmur)
```

Each call to *DefineModuleRef* creates a row in the 1-column "ModuleRef" table. In addition:

*wzName* is stored in the String heap (see section 15.1.1). You *must* supply a non-null *wzName* (see section 15.4) – it is essential to a ModuleRef

## 15.8.29 SetParent

```
HRESULT SetParent(mdMemberRef mr,
                  mdToken tk)
```

This method changes updates a row in the 3-column "MemberRef" table. It therefore consumes no extra space (see section 15.1.4)

## 15.8.30 DefineProperty

```
HRESULT DefineProperty(mdTypeDef td,
                       LPCWSTR wzProperty,
                       DWORD dwPropFlags,
                       PCCOR_SIGNATURE pvSig,
                       ULONG cbSig,
                       DWORD dwDefType,
                       void const *pValue,
                       ULONG cchValue,
                       mdMethodDef mdSetter,
                       mdMethodDef mdGetter,
                       mdMethodDef rmdOtherMethods[],
                       mdFieldDef fdBackingField,
                       mdProperty *pmdProp)
```

Each call to *DefineProperty* creates a row in the 2-column "PropertyMap" table.

Each call to *DefineProperty* also creates a row in the 4-column "Property" table.

Each call to *DefineProperty* also creates rows in the 3-column "MethodSemantics" table. It amounts to one row for the *mdGetter* argument; one for the *mdSetter* argument; and one for each element of the *rmdOtherMethods* argument.

In addition:

*wzProperty* is stored in the String heap (see section 15.1.1). You *must* supply a non-null *wzProperty* (see section 15.4)

*pvSig/cbSig* are stored in the Blob heap (see section 15.1.2) and are required.

*pValue/cchValue* are stored in the Blob heap (see section 15.1.2). These are entirely optional (used to record a default value for the property – that value can be inspected at compile-time). If you choose to store such a value, it consumes space in both the Blob heap, as well as one row in the 3-column "Constant" table.

### 15.8.31 SetPropertyProps

```
HRESULT SetPropertyProps(mdProperty pr,
                         DWORD dwPropFlags,
                         DWORD dwDefType,
                         void const *pValue,
                         ULONG cchValue,
                         mdMethodDef mdSetter,
                         mdMethodDef mdGetter,
                         mdMethodDef rmdOtherMethods[],
                         mdFieldDef fdBackingField)
```

You consume extra space (beyond that already consumed by the *DefineProperty* call for this Property), as follows:

If this call to *SetPropertyProps* supplies *dwDefType, pValue, cchValue* for the first time for this Property (ie, you did not supply them in any previous calls to *DefineProperty* or *SetPropertyProps)*, then, it creates a row in the 3-column "Constant" table, and stores *pValue/cchValue* in the Blob heap. Contrariwise, if you already supplied *dwDefType, pValue, cchValue* for this property, it simply consumes space in the Blob heap for the new *pValue/cchValue*.

If this call to *SetPropertyProps* supplies *mdSetter, mdGetter* or any *rmdOtherMethods* for the first time for this Property (ie, you did not supply them in any previous calls to *DefineProperty* or *SetPropertyProps)*, then, it creates a row in 3-column "MethodSemantics" table for each new method.

Changes to the value of *dwPropFlags* or *fdBackingField* simply updates a column in an existing table, consuming no extra space.

### 15.8.32 DefineEvent

```
HRESULT DefineEvent(mdTypeDef td,
        LPCWSTR wzEvent,
        DWORD dwEventFlags,
        mdToken tkEventType,
        mdMethodDef mdAddOn,
        mdMethodDef mdRemoveOn,
        mdMethodDef mdFire,
        mdMethodDef rmdOtherMethod[],
        mdEvent *pmdEvent)
```

Events are stored using tables and techniques very similar to Properties (see section 15.8.30).

Each call to *DefineEvent* creates one row in the 2-column “EventMap” table.

Each call to *DefineEvent* also creates a row in the 4-column “Event” table.

Each call to *DefineEvent* also creates rows in the 3-column “MethodSemantics” table. This amounts to one row for the *mdAddOn* method; one for the *mdRemoveOn* method; one for the *mdFire* method; and one for each element of the *rmdOtherMethod* argument.

In addition:

*wzEvent* is stored in the String heap (see section 15.1.1). You *must* supply a non-null *wzEvent* (see section 15.4)

### 15.8.33 SetEventProps

```
HRESULT DefineEvent(mdEvent ev,
                    DWORD dwEventFlags,
                    mdToken tkEventType,
                    mdMethodDef mdAddOn,
                    mdMethodDef mdRemoveOn,
                    mdMethodDef mdFire,
                    mdMethodDef rmdOtherMethod[])
```

You consume extra space (beyond that already consumed by the *DefineEvent* call for this Event), as follows:

If this call to *SetEventProps* supplies *mdAddOn, mdRemoveOn* or any *rmdOtherMethod* for the first time for this Event (ie, you did not supply them in any previous calls to *DefineEvent* or *SetEventProps)*, then, it creates a row in 3-column “MethodSemantics” table for each new method.

Changes to the value of *dwEventFlags* or *tkEventType* simply updates a column in an existing table, consuming no extra space.

### 15.8.34 SetClassLayout

```
HRESULT SetClassLayout(mdTypeDef td,
                       DWORD dwPackSize,
                       COR_FIELD_OFFSET rFieldOffsets[],
                       ULONG ulClassSize)
```

Each call to *SetClassLayout* creates a row in the 2-column “ClassLayout” table.

If you supply *rFieldOffsets* then the call also creates one row in the 2-column "FieldLayout" table, for each element in the *rFieldOffsets* array. (The two parts of each COR_FIELD_OFFSET supply the values for those two columns).

Calling this method to re-set any values consumes no extra space, so long as those values were previously defined; on the contrary, if you supply more elements in the rFieldOffsets array, each new element consumes one (2-column) row in the "FieldLayout" table.

## 15.8.35 GetTokenFromSig

```
HRESULT GetTokenFromSig(PCCOR_SIGNATURE pvSig,
                        ULONG cbSig,
                        mdSignature *pmsig)
```

Each call to *GetTokenFromSig* creates a row in the 1-column "StandAloneSig" table. In addition:

*pvSig/cbSig* are stored in the Blob heap (see section 15.1.2)

## 15.8.36 DefineUserString

```
HRESULT DefineUserString(LPCWSTR wzString,
                         ULONG cchString,
                         mdString *pstk)
```

Each call to *DefineUserString* stores *wsString/cchString* into the UserString heap (see section 15.1.3). Unlike almost every other call to define something in metadata, it does *not* create a row in any metadata table. The token handed back (in *pstk*) indexes the string directly in the heap.

## 15.8.37 DeleteToken

```
HRESULT DeleteToken(mdToken tk)
```

Recall that you can 'delete' only the following kinds of token – TypeDef, MethodDef, FieldDef, Event, Property, ExportedType and CustomAttribute.

Calling *DeleteToken* marks corresponding rows in metadata tables as 'deleted'. But it reclaims no physical space in-memory. It doesn't even reclaim that 'deleted' space when it saves the in-memory metadata is saved to disk. This is because if we were to compress out such 'deleted' rows in a table, it would cause a "token remap" – metadata tokens are, in effect, simply row numbers in their corresponding metadata table. But the compiler may have already made use of these assigned tokens (for example, embedding them into the MSIL stream it generates); if we were to change their values, the compiler would have to participate in that "token remap" – which most compilers avoid, like the plague.