NGWS runtime

Metadata Interfaces

This document specifies the API for emitting and importing metadata. 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 Reflections interfaces, specified separately

This is preliminary documentation and subject to change

Last revised: 13 June 2000

1 Overview of the Metadata API

This document defines a set of APIs for *emitting* and *importing* metadata.

Metadata is used to describe, on the one hand, runtime types (classes and interfaces), fields and methods, and, on the other hand, internal implementation and layout information that is used by the runtime to JIT-compile IL, load classes, execute code, and interoperate with the COM classic or native world. This information is an integral part of every runtime component. Metadata is not embedded into the MSIL code that a compiler generates. So this information can be used by 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 classic COM, refer to the "COM Interoperability" 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 gathered together into the companion spec – "Metadata Structures"

1.1 Metadata Interfaces

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 interface to receive: The *emit* interface, used to write to a metadata scope, is *IMetadataEmit*. The *import* interface, which allows tools to read from a metadata scope, is *IMetadataImport*.

The metadata interfaces 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, documented as a separate specification. 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 VOS 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 VOS 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 IL 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 IL 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 IL 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 IL 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 VOS model, and the CLS may introduce higher-level abstractions that are layered over the VOS. 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 VOS 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 VOS type 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, IL 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* interfaces, which are managed versions of these unmanaged interfaces
- "PE File Format Extensions", of which the metadata binary is a part.
- "Virtual Object System", which defines the object model that underlies Runtime, its externalization in metadata, and its implications for the runtime.
- "Metadata Structures", which defines the binary encoding for signatures, custom attributes, etc
- 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.
- "COM Interoperability" and "Platform Invoke", which describe requirements for metadata to control how Runtime method invocations and field accesses are mapped onto underlying legacy services.

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. This feature is provided in methods such as DefineField, DefineParam and DefineProperty. 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)
- cbValue – count of the bytes 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 cbValue)
```

allows you to change *dwFieldFlags*, *dwDefType* and *pValue* (together with its new *cbValue*), 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 NGWS runtime 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 NGWS runtime 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 can be any string you want. It is for information only. It is not used by the Runtime

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 MergeEx

```
HRESULT MergeEx(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. *MergeEx* 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 MergeEndEx

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 MergeEndEx

```
HRESULT MergeEndEx( )
```

This routine triggers the actual merge of metadata, of all import scopes specified by preceding calls to *MergeEx* 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, *MergeEx* 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, *MergeEx* 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 values or attributes are merged only at the time we merge the item they are attached to. For example, custom values associated with a class will be merged when the class is first encountered. If custom values 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 and Custom Values

This section explains two closely-related topics – custom **attributes** and custom **values**. You use the same method, *DefineCustomAttribute*, to define both.

Custom attributes and custom values are just what they say – attributes or values you can attach to a programming element, such as a method or field. But these attributes or values 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.

A custom **value**, on the other hand, is a much simpler affair. Think of it as a triple of (tokenParent, tokenRef, opaque-blob) stored into metadata. Only the caller understands what that opaque-blob *means* and how it should be used; the runtime has no knowledge whatsoever of the its contents. The tokenRef is a way for the caller to associate a name string with the custom value. As before, the tokenParent specifies the metadata item that the opaque-blob is attached to.

Note: although we include descriptions for how to use Custom Values, it is likely we shall withdraw support for them before the product ships. Please therefore use only Custom Attributes

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 Using Custom Values

Note: although we include descriptions for how to use custom values, it is likely we shall withdraw support for them before the product ships. Please therefore use only custom attributes

The model for using custom values also has two steps. First, the compiler emits a TypeRef, in effect to record a name for the custom value. The call to DefineTypeRefByName returns the mdTypeRef token assigned – this is used as the tkAttrib value, in the next step.

Next, the compiler calls DefineCustomAttribute, specifying the required tkParent, tkAttrib (from the previous step), and the opaque value as the blob.

Note that the mdTypeRef token will never be resolved to a corresponding mdTypeDef token. Note also that only one current language provides syntax to allow a programmer to create a custom value, and that for one hard-wired case.

Custom value have the following characteristics:

- Simple in concept and low overhead
- User-selected string names may collide; unless a dev tool has sufficient embedded knowledge of the attributes to provide drop-down lists, etc., this approach is subject to spelling/format errors by developers
- The blob structure of the value is not self-describing and therefore cannot be interpreted by anyone except the definer of the blob
- There's no versioning support

3.2.3 DefineCustomAttribute

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

Defines a custom attribute-object, or a custom value, 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.

If *tkCtor* is an mdMethodDef or mdMemberRef, then you are defining a custom attribute, and *tkCtor* is the token that identifies the constructor method to execute to create the custom attribute-object. Alternatively, if *tkCtor* is an mdTypeRef, then you are defining a custom value (yes, *tkCtor* is not an appropriate name in this case, but as noted above, our focus is on custom attributes)

The format of *pBlob* for defining a custom attribute is defined in the "Metadata Structures" 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.

3.2.4 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, CLASSVERSION *pVer,
        DWORD dwTypeDefFlags, mdToken tkExtends,
        mdToken rtkImplements[], mdTypeDef *ptd)
```

Defines a type. A flag in *dwTypeDefFlags* specifies whether the type being created is a VOS Reference Type (class or interface) or a VOS 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 Interop spec for more details.

in/out	Parameter	Description	Required?
in	wzName	Name of type in Unicode	yes
in	pVer	Version number. Not checked. Rejected if specified for an interface	no
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, CLASSVERSION *pVer,
        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	pVer	Version number	no
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).

Refer to the "Metadata Structures" 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 classic 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 interoperability. They have no impact on managed clients that may be using the interface.

The format of the signature blob is specified in a separate "Metadata Structures" 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 cbValue,
        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).

The format of the signature blob is specified in "Metadata Structures". 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 IL. 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 cbValue 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	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 for dwDefType.

3.4.4 SetFieldProps

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

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	cbValue	Size in bytes 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, CLASSVERSION *pVer,
```

```
        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	pVer	Version number. Not checked. Rejected if specified for an interface	no
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 cbValue, 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 cbValue 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	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. 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 cbValue)
```

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	cbValue	Size in bytes 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 Interoperability" 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,
        mdExecutionLocation tkExec, 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*, *tkExec* 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	tkExec	Token for ExecutionLocation for Assembly pAssemImport	yes
in	pImport	Metadata scope (module) holding target Type	yes
in	tdImport	TypeRef token for target Type within pImport scope	yes
in out	pAssemEmit ptr	Assembly scope for output 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,
        mdExecutionLocation tkExec, 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	tkExec	Token for ExecutionLocation for Assembly pAssemImport	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 cbValue,
        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 cbValue 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	cbValue	Count of bytes 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 cbValue,
        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 cbValue 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	cbValue	Count of bytes 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 can 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, ComType 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.

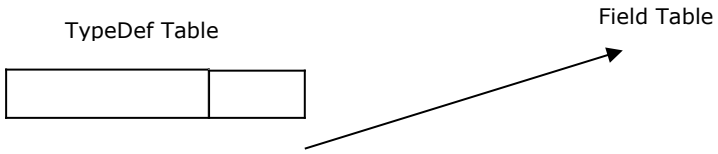
in/out	Parameter	Description	Required?
in	tk	Token to delete	yes

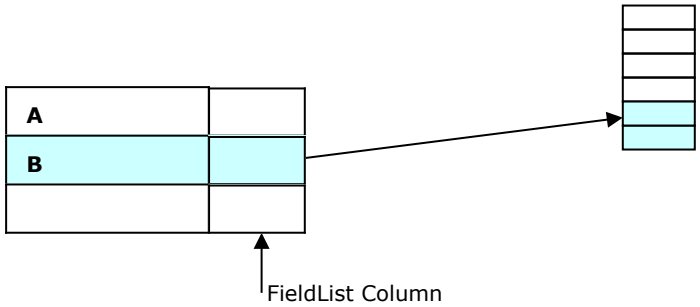
3.9 Order of Emission

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

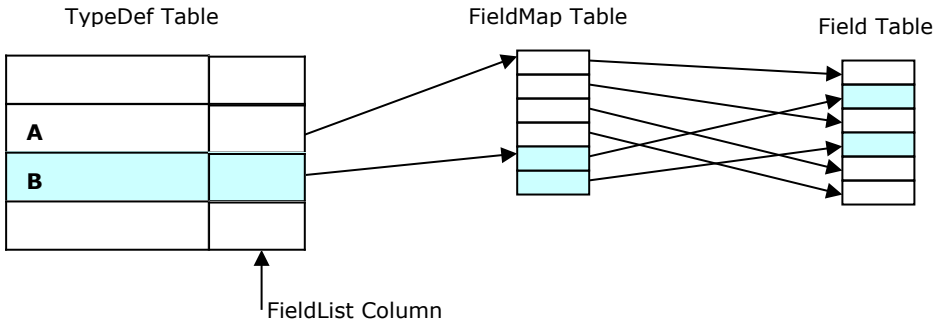
The reason for these rules is illustrated by the picture below –





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 IL 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 dim = 5;      // dimension of array
HCORENUM  enr = 0;      // enumerator
mdTypeDef toks[dim];    // array to hold returned tokens
ULONG     count;        // count of tokens returned
HRESULT   h;
h = pImp->EnumTypeDefs(&enr, toks, dim, &count);
while(count > 0) {
    for(int i = 0; i < count; i++) cout << toks[i] << " ";
    h = pImp->EnumTypeDefs(&enr, toks, dim, &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	Dimension 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	Dimension 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	Dimension 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	Dimension 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	Dimension 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	Dimension 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	Dimension 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	Dimension 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	Dimension 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	Dimension 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	Dimension 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 *t*d

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	Dimension 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	Dimension 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	Dimension 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	Dimension 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	Dimension 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	Dimension of rTokens [] array	yes
out	pcTokens	Number of tokens actually returned	

4.1.22EnumCustomAttributes

```
HRESULT EnumCustomAttributes (HCORENUM *phEnum, mdToken tk,
        mdToken tkType, mdCustomValue 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	Dimension 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, or custom value, 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. If you want to enumerate custom values, then supply *tkType* as the mdTypeRef token with which that custom value was defined. *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	Dimension 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	Dimension 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	Dimension 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 in the Metadata-Structures 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	Biffer 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, CLASSVERSION *pver,
        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	pver	Version number.	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 caOrCv,
        mdToken *ptkOwner, mdToken *ptkType,
        void const **ppBlob, ULONG *pcbBlob)
```

Returns information about a custom attribute or custom value.

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 (see "Metadata Structures" for how a custom attribute is stored)

A custom value is also stored as a blob, but its format is understood only by its caller.

in/out	Parameter	Description	Required?
in	caOrCv	Token for required custom attribute or custom value	yes
out	ptkOwner	Token for owner	
out	ptkType	Token for custom value or attribute	no
out	ppBlob	Pointer to blob for custom attribute or value	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, or custom value, is attached to. A custom attribute can be attached to any sort of owner, with the sole exception of an mdCustomAttribute. A custom value can be attached to any sort of owner.

If caOrCv is a custom attribute, then *ptkType* is the mdMethodDef or mdMemberRef token for its constructor method. If caOrCv is a custom value, then *ptkType* is an mdTypeRef (that need not resolve to a matching TypeDef)

4.3.7 GetCustomAttributeByName

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

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

In the case of a custom **value**, the name is simply the name you gave it via your call to DefineTypeRef. In the case of a genuine custom **attribute**, the name is the name of the attribute class (see Section 3.2.3)

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.

Note: if you need to determine whether this custom 'blob' is a custom **value** or a genuine custom **attribute**, you need to check its token; that's to say, the token

supplied as the *tkAttrib* argument in the DefineCustomAttribute call that created it. However, this token is *not* returned from GetCustomAttributeByName. Use either GetCustomAttributeProps, or EnumCustomAttributes instead.

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 CorParamAttr enum in CorHdr.h.

4.3.11 GetParamProps

```
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	Dimension of rOtherMethods array	no
out	pcOtherMethods	Number of other methods actually returned	no

4.3.21 GetMethodSemantics

```
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

4.3.22 GetClassLayout

```
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	Dimension 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

5 Appendix – 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 NGWS SDK. In particular, those code paths corresponding to the "-raw" and "-heaps" command line switches to that tool.

6 Appendix – 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? [This was considered for a previous design]

7 Appendix – 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 Execution Engine 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 Execution Engine:

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. Lightning 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.

[Note that a previous proposed design for nested types *did* require that names of nested types be module-wide unique. Since it would be unreasonable for the language to pass this burden upwards to its users, it required the language to generate a uniqufied name for each nested type. A *de facto* standard of emerged earlier during development of concatenating the type names together, with a $ separator character. This would generate *AA$BB* as the name for the inner class in the last example. Whilst this worked OK for each language individually, it required that the mangling scheme be agreed across all languages that used the Runtime and wanted to share code]

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 ABC or similar. With this new proposal, we don't mangle names. And we circumvent the view that a TypeRef for ABC refers to a global class with a name of ABC]

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 Appendix – '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 Frameworks 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.**MarshalAsAttribute**
System.InteropServices.**PreserveSigAttribute**
System.InteropServices.**InAttribute**
System.InteropServices.**OutAttribute**

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

For a definition of these PCAs, see the online doc for Base 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 Base 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.**DebuggableAttribute**
System.Diagnostics.**DebuggerHiddenAttribute**
System.Diagnostics.**DebuggerStepThroughAttribute**
System.Diagnostics.**DebuggableAmbivalentAttribute**

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**