The .NET File Format

This article was released on www.codeproject.com.

First release: 11/12/2005. Last update: 12/10/2006.

Index

Introduction

The standards of the .NET format are public, you can find them on Microsoft and in your .NET SDK (look after "Partition II Metadata.doc"), but they are intended to be a more like a reference, not really a guide. So, the truth is that a description of the format can be useful. I mean there's a huge difference between having the WinNT.h and having the full explanation of structures and stuff. The documentation given by Microsoft has some explanations, but a lot of passages aren't very clear at all. Of course, it's required that you know quite well the PE File Format. If that's not the case, you should start with that first, otherwise you won't be able to make heads or tails of this article. A little warning: I'm not going to explain how to use the libraries given by Microsoft to access the .NET Format, I'm going to explain the format itself. This article is based on the Framework 2.0.

Getting Started

The only existing tool (at the moment) for viewing and editing the .NET format is my CFF Explorer. I'm sorry for the spam, but you need this tool to dig into the internal structures of the .NET format. I programmed it for this reason in the first place. The reference you could eventually need is the one I mentioned above, and you can find the includes in your Framework SDK "Include" directory (i.e. "C:\...\Microsoft.NET\SDK\v2.0\include").

.NET PE Files

Before we start with MetaData and other stuff, some small observations about .NET PEs are necessary. They all have for default three sections: .text, .reloc, .rsrc. The .text section contains the Import Table, the Import Address Table and the .NET Section. The .reloc is just there to relocate the address which the EntryPoint instruction jumps to (it's the only address contained the IAT). The IT counts just one imported module (mscoree.dll) and one imported function (_CorExeMain for executables and _CorDllMain for dynamic load libraries). The .rsrc section contains just the main icon for an executable, since all others resources are in the .NET Section. The sections flags are checked at runtime, if you change them the assembly won't start.

The .NET Directory

The obsolete COM Directory in PEs is now the .NET Directory (I call it this way). This sections starts with the COR20 structure, also known as CLI header:

```
// COM+ 2.0 header structure.
typedef struct IMAGE_COR20_HEADER
{
    // Header versioning
    DWORD                   cb;
    WORD                    MajorRuntimeVersion;
    WORD                    MinorRuntimeVersion;

    // Symbol table and startup information
    IMAGE_DATA_DIRECTORY    MetaData;
    DWORD                   Flags;
// DDBLD - Added next section to replace following lin
// DDBLD - Still verifying, since not in NT SDK
// DWORD EntryPointToken;

    // If COMIMAGE_FLAGS_NATIVE_ENTRYPOINT is not set, EntryPointToken represents a managed
entrypoint.
    // If COMIMAGE_FLAGS_NATIVE_ENTRYPOINT is set, EntryPointRVA represents an RVA to a native
entrypoint.
    union {
        DWORD               EntryPointToken;
        DWORD                   EntryPointRVA;
    };
// DDBLD - End of Added Area

    // Binding information
    IMAGE_DATA_DIRECTORY    Resources;
    IMAGE_DATA_DIRECTORY    StrongNameSignature;

    // Regular fixup and binding information
    IMAGE_DATA_DIRECTORY    CodeManagerTable;
    IMAGE_DATA_DIRECTORY    VTableFixups;
    IMAGE_DATA_DIRECTORY    ExportAddressTableJumps;

    // Precompiled image info (internal use only - set to zero)
    IMAGE_DATA_DIRECTORY    ManagedNativeHeader;

} IMAGE_COR20_HEADER, *PIMAGE_COR20_HEADER;
```

A brief description of the members:

cb Size of the structure.

MajorRuntimeVersion and **MinorRuntimeVersion** Version of the CLR Runtime.

MetaData A Data Directory giving RVA and Size of the MetaData.

Flags These are the supported flags:

```
typedef enum ReplacesCorHdrNumericDefines
{
// COM+ Header entry point flags.
    COMIMAGE_FLAGS_ILONLY               =0x00000001,
    COMIMAGE_FLAGS_32BITREQUIRED        =0x00000002,
    COMIMAGE_FLAGS_IL_LIBRARY           =0x00000004,
    COMIMAGE_FLAGS_STRONGNAMESIGNED     =0x00000008,
// DDBLD - Added Next Line - Still verifying general usage
    COMIMAGE_FLAGS_NATIVE_ENTRYPOINT    =0x00000010,
// DDBLD - End of Add
    COMIMAGE_FLAGS_TRACKDEBUGDATA       =0x00010000,

// Other kinds of flags follow

} ReplacesCorHdrNumericDefines;
```

EntryPointToken and **EntryPointRVA** Point to the EntryPoint method (the .NET one, this has nothing to do with the field in the Optional Header). Depending if native or IL it's a RVA or a Token. I'll explain in the MetaData paragraph what's a token.

Resources A Data Directory for the Resources. These resources are referenced in the MetaData.

StrongNameSignature A Data Directory for the Strong Name Signature. It's a signature to uniquely identify .NET Assemblies. This section is only present when the COMIMAGE_FLAGS_STRONGNAMESIGNED is set. It affects some fields in the MetaData as well (you'll see later). You can find detailed explanations about how the Strong Name Signature works and its implemenation over the internet.

CodeManagerTable Always 0.

VTableFixups I quote from the SDK: "Certain languages, which choose not to follow the common type system runtime model, may have virtual functions which need to be represented in a v-table. These v-tables are laid out by the compiler, not by the runtime. Finding the correct v-table slot and calling indirectly through the value held in that slot is also done by the compiler. The VtableFixups field in the runtime header contains the location and size of an array of Vtable Fixups (§14.5.1). V-tables shall be emitted into a read-write section of the PE file. Each entry in this array describes a contiguous array of v-table slots of the specified size. Each slot starts out initialized to the metadata token value for the method they need to call. At image load time, the runtime Loader will turn each entry into a pointer to machine code for the CPU and can be called directly.". And this is everything you'll find here about VTableFixups.

ExportAddressTableJumps Always 0.

ManagedNativeHeader Always 0 in normal .NET assemblies, only present in native images.

Ok now let's begin with the main subject.

The MetaData Section

This section begins with a header that I called MetaData Header (yes, I'm full of imagination indeed). Let's take a look at this header (since it's a dynamic header, it makes no sense declaring a structure, I'll just list the members):

Signature It's a simple dword-signature (similary to the ones you find in the Dos Header and the Optional Header). Anyway the value of this signature has to be 0x424A5342.

MajorVersion and **MinorVersion** Two word elements that are totally ignored by the loader. The value is 0x0001 for both.

Reserved A dword which value is always 0.

Length The length of the UTF string that follows (it's the version string, something like: "v1.1.4322"). The length has to be rounded up to a multiple of 4.

Version The string we just talked about.

Flags Reserved, this word is always 0.

Streams A word telling us the number of streams present in the MetaData.

StreamHeaders Every stream has a name, an offset and a size. The number of stream headers is given by the number above.

But what is a stream? A stream is a "section" in the MetaData which contains a specific kind of data. A stream header is made of 2 dwords (an Offset and a Size) and an Ascii string aligned to the next 4-byte boundary. Take a look:

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00074690                                        6C 00 00 00       l...
000746A0   FA 37 01 00 23 53 74 72  69 6E 67 73 00 00 00 00   ú7..#Strings....
000746B0   68 38 01 00 15 6F 00 00  23 55 53 00 80 A7 01 00   h8...o..#US.€§..
000746C0   94 B3 00 00 23 42 6C 6F  62 00 00 00 14 5B 02 00   ”³..#Blob....[..
000746D0   10 00 00 00 23 47 55 49  44 00 00 00 24 5B 02 00   ....#GUID...$[..
000746E0   EC 59 02 00 23 7E 00 00                            ìY..#~..
```

I marked every last byte of every Stream Header. As you can see, the string (plus its terminator) is always rounded up to 4. I think seeing it in an hex editor makes it easier to understand the disposition.

The offset in a Stream Header is not a File Offset, but an offset you have to add to the MetaData Header offset to obtain a File Offset. In other words the offset is relative to the start of the MetaData Section.

Default streams are:

#Strings An array of ascii strings. The strings in this stream are referenced by MetaData Tables (I'll explain later). These stream contains UTF8 strings with a null terminator byte.

#US Array of unicode strings. The name stands for User Strings, and these strings are referenced directly by code instructions (ldstr). This stream starts with a null byte exactly like the #Blob one. Each entry of this stream begins with a 7bit encoded integer which tells us the size of the following string (the size is in bytes, not characters). Moreover, there's an additional byte at the end of the string (so that every string size is odd and not even). This last byte tells the framework if any of the characters in the string has its high byte set or if the low byte is any of these particular values: 0x01–0x08, 0x0E–0x1F, 0x27, 0x2D.

#Blob Contains data referenced by MetaData Tables. I cannot talk about this stream before explaining the MetaData Tables. I mean you will understand this stream along with the MetaData Tables. I will spend a few words about this stream afterwards.

#GUID Contains 128bits long unique identifiers. Also referenced in MetaData Tables.

#~ The most important stream. It contains the MetaData Tables and is for this reason the main subject of this article. I cannot make a brief description of this stream: a new paragraph is necessary.

The MetaData Tables

Also knows as #~ Stream. This stream begins with a header I call "MetaData Tables Header", here are the members:

Reserved The first dword is always 0.

MajorVersion and **MinorVersion** Two bytes.

HeapOffsetSizes This field is very important, it's a byte that tells us the size that indexes into the "#String", "#GUID" and "#Blob" streams will have. I paste you the description and the bit mask from the SDK:

"The HeapSizes field is a bitvector that encodes how wide indexes into the various heaps are. If bit 0 is set, indexes into the "#String" heap are 4 bytes wide; if bit 1 is set, indexes into the "#GUID" heap are 4 bytes wide; if bit 2 is set, indexes into the "#Blob" heap are 4 bytes wide. Conversely, if the HeapSize bit for a particular heap is not set, indexes into that heap are 2 bytes wide."

Bit mask	**Description**
0x01	Size of "#String" stream >= 2^16.
0x02	Size of "#GUID" stream >= 2^16
0x04	Size of "#Blob" stream >= 2^16.

Reserved Byte, always 1.

Valid It's a bitmask-qword that tells us which MetaData Tables are present in the assembly. Of course, since this is a qword the maximum number of tables is 64. However, most of the tables aren't defined yet. So, the high bits of this qword are always 0.

Sorted Also a bitmask-qword. It tells us which tables are sorted.

Following there's an array of dwords with the number of rows for each present table. Ok this has to be explained. For every table there can be n rows. Let's say we have three tables: A, B and C. And the Valid mask tells us that the B table is not present, but A and C are. In this case there will be 2 dwords (not three), one for the rows in the A table and one for the C table. The B table rows are skypped since there is no B table in the assembly.

Following the array we find the actual MetaData Tables. The defined tables are:

```
00 - Module                 01 - TypeRef               02 - TypeDef
04 - Field                  06 - MethodDef             08 - Param
09 - InterfaceImpl          10 - MemberRef             11 - Constant
12 - CustomAttribute        13 - FieldMarshal          14 - DeclSecurity
15 - ClassLayout            16 - FieldLayout           17 - StandAloneSig
18 - EventMap               20 - Event                 21 - PropertyMap
23 - Property               24 - MethodSemantics       25 - MethodImpl
26 - ModuleRef              27 - TypeSpec              28 - ImplMap
29 - FieldRVA               32 - Assembly              33 - AssemblyProcessor
34 - AssemblyOS             35 - AssemblyRef           36 - AssemblyRefProcessor
37 - AssemblyRefOS          38 - File                  39 - ExportedType
40 - ManifestResource       41 - NestedClass           42 - GenericParam
44 - GenericParamConstraint
```

As you can see, some numbers are missing, that's because some tables, as I said before, are not defined yet. It's important you understand how the tables are stored. A table is made of an array of rows; a row is a structure (let's call it this way for the moment to make things easier). After the rows of a given table end, the rows of the next table follow. The problem with a row (remember, think of it like a structure) is that some of its fields aren't always of the same size and they change from assembly to assembly, so you have to calculate them dynamically. For example, I talked about the HeapOffsetSizes field and how it tells us the size that indexes into the "#String", "#GUID" and "#Blob" streams will have; this means if I have in a structure of one of these tables an index into the "#String" stream, its size is determined by HeapOffsetSizes, and so it could be a word or a dword. Of course that's not the only kind of index that can change of size, there are others. A very simple one to calculate is a direct index into another table. For example, the first element of a NestedClass row is an index into the TypeDef table, the size of this index depends on how much rows the TypeDef table counts: if the rows are > 0xFFFF, a dword is necessary to store the number, otherwise a word will do the job. The remaining indexes are the most annoying, they can index into a table or another. The Microsoft documentation is not so clear about this (at all), so I'll try to explain it in an easy way. Let's consider the TypeDefOrRefIndex, this is a kind of index that can either reference a row in the TypeRef table, in the TypeDef table or in the TypeSpec table. The low bits of the value tell us which table is being indexed and the remaining bits represent the actual index; since the choice is between 3 tables, it only takes 2 bits to encode the table for this kind of index. So if we have a word and the 2 low bits are reserved to encode the table that is being indexed, the remaining 14 bits can index a row in one of the three tables, but what if one of those 3 tables has more rows than a value of 14 bits can encode? Well, then a dword is needed. So, to get the size of an index like this it's necessary to compare the rows of each table it can reference, get the table with the biggest number of rows and then see if this number fits into the remaining bits of a word, if not, a dword is required. I paste you from the SDK the list of this kind of indexes and the values to encode the tables for each index type (which is the "Tag" column):

TypeDefOrRef: 2 bits to encode tag	**Tag**
TypeDef	0
TypeRef	1
TypeSpec	2

HasConstant: 2 bits to encode tag	**Tag**
FieldDef	0
ParamDef	1
Property	2

HasCustomAttribute: 5 bits to encode tag	**Tag**
MethodDef	0
FieldDef	1
TypeRef	2
TypeDef	3
ParamDef	4
InterfaceImpl	5
MemberRef	6
Module	7
Permission	8
Property	9
Event	10
StandAloneSig	11
ModuleRef	12
TypeSpec	13
Assembly	14
AssemblyRef	15
File	16
ExportedType	17
ManifestResource	18

HasFieldMarshall: 1 bit to encode tag	**Tag**
FieldDef	0
ParamDef	1

HasDeclSecurity: 2 bits to encode tag	**Tag**
TypeDef	0
MethodDef	1
Assembly	2

MemberRefParent: 3 bits to encode tag	**Tag**
TypeDef	0
TypeRef	1
ModuleRef	2
MethodDef	3
TypeSpec	4

HasSemantics: 1 bit to encode tag	**Tag**
Event	0
Property	1

MethodDefOrRef: 1 bit to encode tag	**Tag**
MethodDef	0
MemberRef	1

MemberForwarded: 1 bit to encode tag	**Tag**
FieldDef	0
MethodDef	1

Implementation: 2 bits to encode tag	**Tag**
File	0
AssemblyRef	1
ExportedType	

CustomAttributeType: 3 bits to encode tag	**Tag**
Not used	0
Not used	1
MethodDef	2
MemberRef	3
Not used	4

ResolutionScope: 2 bits to encode tag	**Tag**
Module	0
ModuleRef	1
AssemblyRef	2
TypeRef	3

Ok, I hope what I wrote above is clear. If not, you should try reading it again... And again. Now, I will list each table, give a brief description of it (and whatever I have to say in addition) and list (copy them form the SDK) its columns. Ah, before I forget, a token is a dword-value that represents a table and an index into that table. For example, the EntryPointToken 0x0600002C, references table 0x06 (MethodDef) and its row 0x2C. I think this is pretty simple to understand if you understood all the other stuff.

00 - Module Table

It's a one row table representing the current assembly.

Columns:

• Generation (2-byte value, reserved, shall be zero)
• Name (index into String heap)
• Mvid (index into Guid heap; simply a Guid used to distinguish between two versions of the same module)
• EncId (index into Guid heap, reserved, shall be zero)
• EncBaseId (index into Guid heap, reserved, shall be zero)

01 - TypeRef Table

Each row represents an imported class, its namespace and the assembly which contains it.

Columns:

• ResolutionScope (index into Module, ModuleRef, AssemblyRef or TypeRef tables, or null; more precisely, a ResolutionScope coded index)
• TypeName (index into String heap)
• TypeNamespace (index into String heap)

02 - TypeDef Table

Each row represents a class in the current assembly.

Columns:

• Flags (a 4-byte bitmask of type TypeAttributes)
• TypeName (index into String heap)
• TypeNamespace (index into String heap)
• Extends (index into TypeDef, TypeRef or TypeSpec table; more precisely, a TypeDefOrRef coded index)
• FieldList (index into Field table; it marks the first of a continguous run of Fields owned by this Type). The run continues to the smaller of:
 o the last row of the Field table
 o the next run of Fields, found by inspecting the FieldList of the next row in this TypeDef table

• MethodList (index into MethodDef table; it marks the first of a continguous run of Methods owned by this Type). The run continues to the smaller of:
  o the last row of the MethodDef table
  o the next run of Methods, found by inspecting the MethodList of the next row in this TypeDef table

Available flags are:

```
typedef enum CorTypeAttr
{
    // Use this mask to retrieve the type visibility information.
    tdVisibilityMask        =   0x00000007,
    tdNotPublic             =   0x00000000,     // Class is not public scope.
    tdPublic                =   0x00000001,     // Class is public scope.
    tdNestedPublic          =   0x00000002,     // Class is nested with public visibility.
    tdNestedPrivate         =   0x00000003,     // Class is nested with private visibility.
    tdNestedFamily          =   0x00000004,     // Class is nested with family visibility.
    tdNestedAssembly        =   0x00000005,     // Class is nested with assembly visibility.
    tdNestedFamANDAssem     =   0x00000006,     // Class is nested with family and assembly
visibility.
    tdNestedFamORAssem      =   0x00000007,     // Class is nested with family or assembly
visibility.

    // Use this mask to retrieve class layout information
    tdLayoutMask            =   0x00000018,
    tdAutoLayout            =   0x00000000,     // Class fields are auto-laid out
    tdSequentialLayout      =   0x00000008,     // Class fields are laid out sequentially
    tdExplicitLayout        =   0x00000010,     // Layout is supplied explicitly
    // end layout mask

    // Use this mask to retrieve class semantics information.
    tdClassSemanticsMask    =   0x00000060,
    tdClass                 =   0x00000000,     // Type is a class.
    tdInterface             =   0x00000020,     // Type is an interface.
    // end semantics mask

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

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

    // Use tdStringFormatMask to retrieve string information for native interop
    tdStringFormatMask      =   0x00030000,
    tdAnsiClass             =   0x00000000,     // LPTSTR is interpreted as ANSI in this class
    tdUnicodeClass          =   0x00010000,     // LPTSTR is interpreted as UNICODE
    tdAutoClass             =   0x00020000,     // LPTSTR is interpreted automatically
    tdCustomFormatClass     =   0x00030000,     // A non-standard encoding specified by
CustomFormatMask
    tdCustomFormatMask      =   0x00C00000,     // Use this mask to retrieve non-standard encoding
information for native interop. The meaning of the values of these 2 bits is unspecified.

    // end string format mask

    tdBeforeFieldInit       =   0x00100000,     // Initialize the class any time before first
static field access.
    tdForwarder             =   0x00200000,     // This ExportedType is a type forwarder.

    // Flags reserved for runtime use.
    tdReservedMask          =   0x00040800,
    tdRTSpecialName         =   0x00000800,     // Runtime should check name encoding.
    tdHasSecurity           =   0x00040000,     // Class has security associate with it.
} CorTypeAttr;
```

04 - Field Table

Each row represents a field in a TypeDef class. The fields of one class are not stored casually: after the fields of one class end, the fields of the next class begin.

Columns:

• Flags (a 2-byte bitmask of type FieldAttributes)
• Name (index into String heap)
• Signature (index into Blob heap)

Available flags are:

```
typedef enum CorFieldAttr
{
    // member access mask - Use this mask to retrieve accessibility information.
    fdFieldAccessMask           =   0x0007,
    fdPrivateScope              =   0x0000,     // Member not referenceable.
    fdPrivate                   =   0x0001,     // Accessible only by the parent type.
    fdFamANDAssem               =   0x0002,     // Accessible by sub-types only in this Assembly.
    fdAssembly                  =   0x0003,     // Accessibly by anyone in the Assembly.
    fdFamily                    =   0x0004,     // Accessible only by type and sub-types.
    fdFamORAssem                =   0x0005,     // Accessibly by sub-types anywhere, plus anyone
in assembly.
    fdPublic                    =   0x0006,     // Accessibly by anyone who has visibility to this
scope.
    // end member access mask

    // field contract attributes.
    fdStatic                    =   0x0010,     // Defined on type, else per instance.
    fdInitOnly                  =   0x0020,     // Field may only be initialized, not written to
after init.
    fdLiteral                   =   0x0040,     // Value is compile time constant.
    fdNotSerialized             =   0x0080,     // Field does not have to be serialized when type
is remoted.

    fdSpecialName               =   0x0200,     // field is special. Name describes how.

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

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

06 - MethodDef Table

Each row represents a method in a specific class. The methods sequence follows the same logic of the fields one.

Columns:

- RVA (a 4-byte constant)
- ImplFlags (a 2-byte bitmask of type MethodImplAttributes)
- Flags (a 2-byte bitmask of type MethodAttribute)
- Name (index into String heap)
- Signature (index into Blob heap)
- ParamList (index into Param table). It marks the first of a contiguous run of Parameters owned by this method. The run continues to the smaller of:
    - o the last row of the Param table
    - o the next run of Parameters, found by inspecting the ParamList of the next row in the MethodDef table

Available flags are:

```
typedef enum CorMethodAttr
{
    // member access mask - Use this mask to retrieve accessibility information.
    mdMemberAccessMask          =   0x0007,
    mdPrivateScope              =   0x0000,     // Member not referenceable.
    mdPrivate                   =   0x0001,     // Accessible only by the parent type.
    mdFamANDAssem               =   0x0002,     // Accessible by sub-types only in this Assembly.
    mdAssem                     =   0x0003,     // Accessibly by anyone in the Assembly.
    mdFamily                    =   0x0004,     // Accessible only by type and sub-types.
    mdFamORAssem                =   0x0005,     // Accessibly by sub-types anywhere, plus anyone
in assembly.
    mdPublic                    =   0x0006,     // Accessibly by anyone who has visibility to this
scope.
    // end member access mask

    // method contract attributes.
    mdStatic                    =   0x0010,     // Defined on type, else per instance.
    mdFinal                     =   0x0020,     // Method may not be overridden.
    mdVirtual                   =   0x0040,     // Method virtual.
    mdHideBySig                 =   0x0080,     // Method hides by name+sig, else just by name.
    // vtable layout mask - Use this mask to retrieve vtable attributes.
    mdVtableLayoutMask          =   0x0100,
    mdReuseSlot                 =   0x0000,     // The default.
    mdNewSlot                   =   0x0100,     // Method always gets a new slot in the vtable.
    // end vtable layout mask

    // method implementation attributes.
    mdCheckAccessOnOverride     =   0x0200,     // Overridability is the same as the visibility.
    mdAbstract                  =   0x0400,     // Method does not provide an implementation.
    mdSpecialName               =   0x0800,     // Method is special. Name describes how.

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

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

} CorMethodAttr;

typedef enum CorMethodImpl
{
    // code impl mask
    miCodeTypeMask      =   0x0003,   // Flags about code type.
    miIL                =   0x0000,   // Method impl is IL.
    miNative            =   0x0001,   // Method impl is native.
    miOPTIL             =   0x0002,   // Method impl is OPTIL
    miRuntime           =   0x0003,   // Method impl is provided by the runtime.
    // end code impl mask

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

    // implementation info and interop
    miForwardRef        =   0x0010,   // Indicates method is defined; used primarily in merge
scenarios.
    miPreserveSig       =   0x0080,   // Indicates method sig is not to be mangled to do HRESULT
conversion.

    miInternalCall      =   0x1000,   // Reserved for internal use.

    miSynchronized      =   0x0020,   // Method is single threaded through the body.
    miNoInlining        =   0x0008,   // Method may not be inlined.
    miMaxMethodImplVal  =   0xffff,   // Range check value
} CorMethodImpl;
```

The RVA points to the method body, I'll explain the format of that later. The Signature gives information about the method declaration, remember that data stored in the #Blob stream follows 7bit encoding/decoding rules.

08 - Param Table

Each row represents a method's param.

Columns:

- Flags (a 2-byte bitmask of type ParamAttributes)
- Sequence (a 2-byte constant)
- Name (index into String heap)

Available flags are:

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

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

    pdUnused                       =   0xcfe0,
} CorParamAttr;
```

09 - InterfaceImpl Table

Each row tells the framework a class that implements a specific interface.

Columns:

• Class (index into the TypeDef table)
• Interface (index into the TypeDef, TypeRef or TypeSpec table; more precisely, a TypeDefOrRef coded index)

10 - MemberRef Table

Also known as MethodRef table. Each row represents an imported method.

Columns:

• Class (index into the TypeRef, ModuleRef, MethodDef, TypeSpec or TypeDef tables; more precisely, a MemberRefParent coded index)
• Name (index into String heap)
• Signature (index into Blob heap)

11 - Constant Table

Each row represents a constant value for a Param, Field or Property.

Columns:

• Type (a 1-byte constant, followed by a 1-byte padding zero).
• Parent (index into the Param or Field or Property table; more precisely, a HasConstant coded index)
• Value (index into Blob heap)

12 - CustomAttribute Table

I think the best description is given by the SDK: "The CustomAttribute table stores data that can be used to instantiate a Custom Attribute (more precisely, an object of the specified Custom Attribute class) at runtime. The column called Type is slightly misleading – it actually indexes a constructor method – the owner of that constructor method is the Type of the Custom Attribute."

Columns:

• Parent (index into any metadata table, except the CustomAttribute table itself; more precisely, a HasCustomAttribute coded index)
• Type (index into the MethodDef or MethodRef table; more precisely, a CustomAttributeType coded index)
• Value (index into Blob heap)

13 - FieldMarshal Table

Each row tells the way a Param or Field should be threated when called from/to unmanaged code.

Columns:

• Parent (index into Field or Param table; more precisely, a HasFieldMarshal coded index)
• NativeType (index into the Blob heap)

14 - DeclSecurity Table

Security attributes attached to a class, method or assembly.

Columns:

• Action (2-byte value)
• Parent (index into the TypeDef, MethodDef or Assembly table; more precisely, a HasDeclSecurity coded index)
• PermissionSet (index into Blob heap)

15 - ClassLayout Table

Remember "#pragma pack(n)" for VC++? Well, this is kind of the same thing for .NET. It's useful when handing something from managed to unmanaged code.

Columns:

• PackingSize (a 2-byte constant)
• ClassSize (a 4-byte constant)
• Parent (index into TypeDef table)

16 - FieldLayout Table

Related with the ClassLayout.

Columns:

• Offset (a 4-byte constant)
• Field (index into the Field table)

17 - StandAloneSig Table

Each row represents a signature that isn't referenced by any other table.

Columns:

• Signature (index into the Blob heap)

18 - EventMap Table

List of events for a specific class.

Columns:

• Parent (index into the TypeDef table)
• EventList (index into Event table). It marks the first of a contiguous run of Events owned by this Type. The run continues to the smaller of:
  o the last row of the Event table
  o the next run of Events, found by inspecting the EventList of the next row in the EventMap table

20 - Event Table

Each row represents an event.

Columns:

• EventFlags (a 2-byte bitmask of type EventAttribute)
• Name (index into String heap)
• EventType (index into TypeDef, TypeRef or TypeSpec tables; more precisely, a TypeDefOrRef coded index) [this corresponds to the Type of the Event; it is not the Type that owns this event]

Available flags are:

```
typedef enum CorEventAttr
{
    evSpecialName           =   0x0200,     // event is special. Name describes how.

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

21 - PropertyMap Table

List of Properties owned by a specific class.

Columns:

• Parent (index into the TypeDef table)
• PropertyList (index into Property table). It marks the first of a contiguous run of Properties owned by Parent. The run continues to the smaller of:
  o the last row of the Property table
  o the next run of Properties, found by inspecting the PropertyList of the next row in this PropertyMap table

23 - Property Table

Each row represents a property.

Columns:

• Flags (a 2-byte bitmask of type PropertyAttributes)
• Name (index into String heap)

• Type (index into Blob heap) [the name of this column is misleading. It does not index a TypeDef or TypeRef table – instead it indexes the signature in the Blob heap of the Property)

Available flags are:

```
typedef enum CorPropertyAttr
{
    prSpecialName           =   0x0200,     // property is special. Name describes how.

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

    prUnused                =   0xe9ff,
} CorPropertyAttr;
```

24 - MethodSemantics Table

Links Events and Properties to specific methods. For example one Event can be associated to more methods. A property uses this table to associate get/set methods.

Columns:

• Semantics (a 2-byte bitmask of type MethodSemanticsAttributes)
• Method (index into the MethodDef table)
• Association (index into the Event or Property table; more precisely, a HasSemantics coded index)

Available flags are:

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

25 - MethodImpl Table

I quote: "MethodImpls let a compiler override the default inheritance rules provided by the CLI. Their original use was to allow a class “C”, that inherited method “Foo” from interfaces I and J, to provide implementations for both methods (rather than have only one slot for “Foo” in its vtable). But MethodImpls can be used for other reasons too, limited only by the compiler writer’s ingenuity within the constraints defined in the Validation rules below.".

Columns:

• Class (index into TypeDef table)
• MethodBody (index into MethodDef or MemberRef table; more precisely, a MethodDefOrRef coded index)
• MethodDeclaration (index into MethodDef or MemberRef table; more precisely, a MethodDefOrRef coded index)

26 - ModuleRef Table

Each row represents a reference to an external module.

Columns:

• Name (index into String heap)

27 - TypeSpec Table

Each row represents a specification for a TypeDef or TypeRef. The only column indexes a token in the #Blob stream.

Columns:

• Signature (index into the Blob heap)

28 - ImplMap Table

I quote: "The ImplMap table holds information about unmanaged methods that can be reached from managed code, using PInvoke dispatch.
Each row of the ImplMap table associates a row in the MethodDef table (MemberForwarded) with the name of a routine (ImportName) in some unmanaged DLL (ImportScope).". This means all the unmanaged functions used by the assembly are listed here.

Columns:

• MappingFlags (a 2-byte bitmask of type PInvokeAttributes)
• MemberForwarded (index into the Field or MethodDef table; more precisely, a MemberForwarded coded index. However, it only ever indexes the MethodDef table, since Field export is not supported)
• ImportName (index into the String heap)
• ImportScope (index into the ModuleRef table)

Available flags are:

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

    // Use this mask to retrieve the CharSet information.
    pmCharSetMask       = 0x0006,
    pmCharSetNotSpec    = 0x0000,
    pmCharSetAnsi       = 0x0002,
    pmCharSetUnicode    = 0x0004,
    pmCharSetAuto       = 0x0006,


    pmBestFitUseAssem   = 0x0000,
    pmBestFitEnabled    = 0x0010,
    pmBestFitDisabled   = 0x0020,
    pmBestFitMask       = 0x0030,

    pmThrowOnUnmappableCharUseAssem   = 0x0000,
    pmThrowOnUnmappableCharEnabled    = 0x1000,
    pmThrowOnUnmappableCharDisabled   = 0x2000,
    pmThrowOnUnmappableCharMask       = 0x3000,

    pmSupportsLastError = 0x0040,   // Information about target function. Not relevant for fields.

    // None of the calling convention flags is relevant for fields.
    pmCallConvMask      = 0x0700,
    pmCallConvWinapi    = 0x0100,   // Pinvoke will use native callconv appropriate to target
windows platform.
    pmCallConvCdecl     = 0x0200,
    pmCallConvStdcall   = 0x0300,
    pmCallConvThiscall  = 0x0400,   // In M9, pinvoke will raise exception.
    pmCallConvFastcall  = 0x0500,

    pmMaxValue          = 0xFFFF,
} CorPinvokeMap;
```

29 - FieldRVA Table

Each row is an extension for a Field table. The RVA in this table gives the location of the inital value for a Field.

Columns:

• RVA (a 4-byte constant)
• Field (index into Field table)

32 - Assembly Table

It's a one-row table. It stores information about the current assembly.

Columns:

• HashAlgId (a 4-byte constant of type AssemblyHashAlgorithm)
• MajorVersion, MinorVersion, BuildNumber, RevisionNumber (2-byte constants)
• Flags (a 4-byte bitmask of type AssemblyFlags)
• PublicKey (index into Blob heap)
• Name (index into String heap)
• Culture (index into String heap)

Available flags are:

```
typedef enum CorAssemblyFlags
{
    afPublicKey             =   0x0001,     // The assembly ref holds the full (unhashed) public
key.

    afPA_None               =   0x0000,     // Processor Architecture unspecified
    afPA_MSIL               =   0x0010,     // Processor Architecture: neutral (PE32)
    afPA_x86                =   0x0020,     // Processor Architecture: x86 (PE32)
    afPA_IA64               =   0x0030,     // Processor Architecture: Itanium (PE32+)
    afPA_AMD64              =   0x0040,     // Processor Architecture: AMD X64 (PE32+)
    afPA_Specified          =   0x0080,     // Propagate PA flags to AssemblyRef record
    afPA_Mask               =   0x0070,     // Bits describing the processor architecture
    afPA_FullMask           =   0x00F0,     // Bits describing the PA incl. Specified
    afPA_Shift              =   0x0004,     // NOT A FLAG, shift count in PA flags <--> index
conversion

    afEnableJITcompileTracking  =   0x8000, // From "DebuggableAttribute".
    afDisableJITcompileOptimizer=   0x4000, // From "DebuggableAttribute".

    afRetargetable          =   0x0100,     // The assembly can be retargeted (at runtime) to an
                                            // assembly from a different publisher.
} CorAssemblyFlags;
```

The PublicKey is != 0, only if the StrongName Signature is present and the afPublicKey flag is set.

33 - AssemblyProcessor Table

This table is ignored by the CLI and shouldn't be present in an assembly.

Columns:

- Processor (a 4-byte constant)

34 - AssemblyOS Table

This table is ignored by the CLI and shouldn't be present in an assembly.

Columns:

- OSPlatformID (a 4-byte constant)
- OSMajorVersion (a 4-byte constant)
- OSMinorVersion (a 4-byte constant)

35 - AssemblyRef Table

Each row references an external assembly.

Columns:

- MajorVersion, MinorVersion, BuildNumber, RevisionNumber (2-byte constants)
- Flags (a 4-byte bitmask of type AssemblyFlags)
- PublicKeyOrToken (index into Blob heap – the public key or token that identifies the author of this Assembly)
- Name (index into String heap)
- Culture (index into String heap)
- HashValue (index into Blob heap)

The flags are the same ones of the Assembly table.

36 - AssemblyRefProcessor Table

This table is ignored by the CLI and shouldn't be present in an assembly.

Columns:

- Processor (4-byte constant)
- AssemblyRef (index into the AssemblyRef table)

37 - AssemblyRefOS Table

This table is ignored by the CLI and shouldn't be present in an assembly.

Columns:

- OSPlatformId (4-byte constant)
- OSMajorVersion (4-byte constant)
- OSMinorVersion (4-byte constant)
- AssemblyRef (index into the AssemblyRef table)

38 - File Table

Each row references an external file.

Columns:

• Flags (a 4-byte bitmask of type FileAttributes)
• Name (index into String heap)
• HashValue (index into Blob heap)

Available flags are:

```
typedef enum CorFileFlags
{
    ffContainsMetaData      =   0x0000,     // This is not a resource file
    ffContainsNoMetaData    =   0x0001,     // This is a resource file or other non-metadata-
containing file
} CorFileFlags;
```

39 - ExportedType Table

I quote: "The ExportedType table holds a row for each type, defined within other modules of this Assembly, that is exported out of this Assembly. In essence, it stores TypeDef row numbers of all types that are marked public in other modules that this Assembly comprises.". Be careful, this doesn't mean that when an assembly uses a class contained in my assembly I export that type. In fact, I haven't seen yet this table in an assembly.

Columns:

• Flags (a 4-byte bitmask of type TypeAttributes)
• TypeDefId (4-byte index into a TypeDef table of another module in this Assembly). This field is used as a hint only. If the entry in the target TypeDef table matches the TypeName and TypeNamespace entries in this table, resolution has succeeded. But if there is a mismatch, the CLI shall fall back to a search of the target TypeDef table
• TypeName (index into the String heap)
• TypeNamespace (index into the String heap)
• Implementation. This can be an index (more precisely, an Implementation coded index) into one of 2 tables, as follows:
    o File table, where that entry says which module in the current assembly holds the TypeDef
    o ExportedType table, where that entry is the enclosing Type of the current nested Type

The flags are the same ones of the TypeDef.

40 - ManifestResource Table

Each row references an internal or external resource.

Columns:

• Offset (a 4-byte constant)
• Flags (a 4-byte bitmask of type ManifestResourceAttributes)
• Name (index into the String heap)
• Implementation (index into File table, or AssemblyRef table, or null; more precisely, an Implementation coded index)

Available flags are:

```
typedef enum CorManifestResourceFlags
{
    mrVisibilityMask        =   0x0007,
    mrPublic                =   0x0001,     // The Resource is exported from the Assembly.
    mrPrivate               =   0x0002,     // The Resource is private to the Assembly.
} CorManifestResourceFlags;
```

If the Implementation index is 0, then the referenced resource is internal. We obtain the File Offset of the resource by adding the converted Resources RVA (the one in the CLI Header) to the offset present in this table. I wrote an article you can either find on NTCore or codeproject about Manifest Resources, anyway I quote some parts from the other article to give at least a brief explanation, since this section is absolutely undocumented. There are different kinds of resources referenced by this table, and not all of them can be threated in the same way. Reading a bitmap, for example, is very simple: every Manifest Resource begins with a dword that tells us the size of the actual embedded resource... And that's it... After that, we have our bitmap. Ok, but what about those ".resources" files? For every dialog in a .NET Assembly there is one, this means every resource of a dialog is contained in the dialog's own ".resources" file.

A very brief description of ".resources" files format: "The first dword is a signature which has to be 0xBEEFCACE, otherwise the resources file has to be considered as invalid. Second dword contains the number of readers for this resources file, don't worry, it's something we don't have to talk about... Framework stuff. Third dword is the size of reader types This number is only good for us to skip the string (or strings) that follows, which is something like: "System.Resources.ResourceReader, mscorlibsSystem.Resources.RuntimeResourceSet, mscorlib, Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089". It tells the framework the reader to use for this resources file.

Ok, now we got to the interesting part. The next dword tells us the version of the resources file (existing versions are 1 and 2). After the version, another dword gives the number of actual resources in the file. Another dword follows and gives the number of resource types.

To gather the additional information we need, we have to skip the resource types. For each type there's a 7bit encoded integer who gives the size of the string that follows. To decode these kind of integers you have to read every byte until you find one which hasn't the highest bit set and make some additional operations to obtain the final value... For the moment let's just stick to the format. After having skipped the types we have to align our position to an 8 byte base. Then we have a dword * NumberOfResources and each dword contains the hash of a resource. Then we have the same amount of dwords, this time with the offsets of the resource names. Another important dword follows: the Data Section Offset. We need this offset to retrieve resources offsets. After this dword we have the resources names. Well, actually it's not just the names (I just call it this way), every name (7bit encoded integer + unicode string) is followed by a dword, an offset which you can add to the DataSection offset to retrieve the resource offset. The first thing we find, given a resource offset, is a 7bit encoded integer, which is the type index for the current resource.".

If you're interested in this subject, check out that other article I wrote, since there you can find code that maybe helps you understand better.

41 - NestedClass Table

Each row represents a nested class. You know what a nested class is, right?

The columns are of course only two.

Columns:

• NestedClass (index into the TypeDef table)
• EnclosingClass (index into the TypeDef table)

42 - GenericParam Table

I quote: "The GenericParam table stores the generic parameters used in generic type definitions and generic methoddefinitions. These generic parameters can be constrained (i.e., generic arguments shall extend some class and/or implement certain interfaces) or unconstrained.".

Columns:

• Number (the 2-byte index of the generic parameter, numbered left-to-right, from zero)
• Flags (a 2-byte bitmask of type GenericParamAttributes)
• Owner (an index into the TypeDef or MethodDef table, specifying the Type or Method to which this generic parameter applies; more precisely, a TypeOrMethodDef coded index)
• Name (a non-null index into the String heap, giving the name for the generic parameter. This is purely descriptive and is used only by source language compilers and by Reflection)

Available flags are:

```
typedef enum CorGenericParamAttr
{
    // Variance of type parameters, only applicable to generic parameters
    // for generic interfaces and delegates
    gpVarianceMask          =   0x0003,
    gpNonVariant            =   0x0000,
    gpCovariant             =   0x0001,
    gpContravariant         =   0x0002,

    // Special constraints, applicable to any type parameters
    gpSpecialConstraintMask =  0x001C,
    gpNoSpecialConstraint   =   0x0000,
    gpReferenceTypeConstraint = 0x0004,      // type argument must be a reference type
    gpNotNullableValueTypeConstraint   =   0x0008, // type argument must be a value type but not
Nullable
    gpDefaultConstructorConstraint = 0x0010, // type argument must have a public default
constructor
} CorGenericParamAttr;
```

44 - GenericParamConstraint Table

I quote: "The GenericParamConstraint table records the constraints for each generic parameter. Each generic parameter can be constrained to derive from zero or one class. Each generic parameter can be constrained to implement zero or more interfaces. Conceptually, each row in the GenericParamConstraint table is ‘owned’ by a row in the GenericParam table. All rows in the GenericParamConstraint table for a given Owner shall refer to distinct constraints.".

The columns needed are, of course, only two

Columns:

• Owner (an index into the GenericParam table, specifying to which generic parameter this row refers)
• Constraint (an index into the TypeDef, TypeRef, or TypeSpec tables, specifying from which class this generic parameter is constrained to derive; or which interface this generic parameter is constrained to implement; more precisely, a TypeDefOrRef coded index)

Ok that's all about MetaData tables. The last thing I have to explain, as I promised, is the Method format.

Methods

Every method contained in an assembly is referenced in the MethodDef table, the RVA tells us where the method is. The method body is made of three or at least two parts:

- A header, which can be a Fat or a Tiny one.

- The code. The code size is specified in the header.

- Extra Sections. These sections are not always present, the header tells us if they are. Those sections can store different kinds of data, but for now they are only used to store Exception Sections. Those sections sepcify try/catch handlers in the code.

The first byte of the method tells us the type of header used. If the method uses a tiny header the CorILMethod_TinyFormat (0x02) flag will be set otherwise the CorILMethod_FatFormat (0x03) flag. If the tiny header is used, the 2 low bits are reserved for flags (header type) and the rest specify the code size. Of course a tiny header can only be used if the code size is less than 64 bytes. In addition it can't be used if maxstack > 8 or local variables or exceptions (extra sections) are present. In all these other cases the fat header is used:

Offset	Size	Field	Description
0	12 (bits)	Flags	Flags (CorILMethod_FatFormat shall be set in bits 0:1).
12 (bits)	4 (bits)	Size	Size of this header expressed as the count of 4-byte integers occupied (currently 3).
2	2	MaxStack	Maximum number of items on the operand stack.
4	4	CodeSize	Size in bytes of the actual method body
8	4	LocalVarSigTok	Meta Data token for a signature describing the layout of the local variables for the method. 0 means there are no local variables present. This field references a stand-alone signature in the MetaData tables, which references an entry in the #Blob stream.

The available flags are:

Flag	Value	Description
CorILMethod_FatFormat	0x3	Method header is fat.
CorILMethod_TinyFormat	0x2	Method header is tiny.
CorILMethod_MoreSects	0x8	More sections follow after this header.
CorILMethod_InitLocals	0x10	Call default constructor on all local variables.

This means that when the CorILMethod_MoreSects is set, extra sections follow the method. To reach the first extra section we have to add the size of the header to the code size and to the file offset of the method, then aligne to the next 4-byte boundary.

Extra sections can have a Fat (1 byte flags, 3 bytes size) or a Small header (1 byte flags, 1 byte size); the size includes the header size. The type of header and the type of section is specified in the first byte, of course:

Flag	Value	Description
CorILMethod_Sect_EHTable	0x1	Exception handling data.
CorILMethod_Sect_OptILTable	0x2	Reserved, shall be 0.
CorILMethod_Sect_FatFormat	0x40	Data format is of the fat variety, meaning there is a 3-byte length. If not set, the header is small with a 1-byte length
CorILMethod_Sect_MoreSects	0x80	Another data section occurs after this current section

No other types than the exception handling sections are declared (this doesn't mean you shouldn't check the CorILMethod_Sect_EHTable flag). So if the section is small it will be:

Offset	Size	Field	Description
0	1	Kind	Flags as described above.
1	1	DataSize	Size of the data for the block, including the header, say *n*12+4.*
2	2	Reserved	Padding, always 0.
4	*n*	Clauses	*n* small exception clauses.

Otherwise:

Offset	Size	Field	Description
0	1	Kind	Which type of exception block is being used
1	3	DataSize	Size of the data for the block, including the header, say *n*24+4.*
4	*n*	Clauses	*n* fat exception clauses.

The number of the clauses is given byte the DataSize. I mean you have to subtract the size of the header and then divide by the size of a Fat/Small exception clause (this, of course, depends on the kind of header). The small one:

Offset	Size	Field	Description
0	2	Flags	Flags, see below.
2	2	TryOffset	Offset in bytes of try block from start of the header.
4	1	TryLength	Length in bytes of the try block
5	2	HandlerOffset	Location of the handler for this try block
7	1	HandlerLength	Size of the handler code in bytes
8	4	ClassToken	Meta data token for a type-based exception handler
8	4	FilterOffset	Offset in method body for filter-based exception handler

And the fat one:

Offset	Size	Field	Description
0	4	Flags	Flags, see below.
4	4	TryOffset	Offset in bytes of try block from start of the header.
8	4	TryLength	Length in bytes of the try block
12	4	HandlerOffset	Location of the handler for this try block
16	4	HandlerLength	Size of the handler code in bytes
20	4	ClassToken	Meta data token for a type-based exception handler
20	4	FilterOffset	Offset in method body for filter-based exception handler

Available flags are:

Flag	Value	Description
COR_ILEXCEPTION_CLAUSE_EXCEPTION	0x0000	A typed exception clause
COR_ILEXCEPTION_CLAUSE_FILTER	0x0001	An exception filter and handler clause
COR_ILEXCEPTION_CLAUSE_FINALLY	0x0002	A finally clause
COR_ILEXCEPTION_CLAUSE_FAULT	0x0004	Fault clause (finally that is called on exception only)

Ok, that's all.

The #Blob Stream

This stream contains different things as you might have already noticed going through the MetaData tables, but the only thing in this stream which is a bit difficult to understand are signatures. Every signature referenced by the MetaData tables is contained in this stream. What does a signature stand for? For instance, it could tell us the declaration of a method (be it a defined method or a referenced one), this meaning the parameters and the return type of that method. The various kind of signatures are: MethodDefSig, MethodRefSig, FieldSig, PropertySig, LocalVarSig, TypeSpec, MethodSpec.

When I first wrote this article (11/12/2005) the .NET reference content about signatures was a little bit poor, but now, after the April 2006 release, it has a very rich and full explanation about this subject. So there's no point in explaining the schema of every kind of signatures, you can find those in the reference. I'll explain one type of signature and I think it should be enough to understand the logic behind all of them. Let's take for instance the LocalVarSig signature. If there's a method like:

```
private void f()
{
    int A = 3;
    bool B = false;

    if (A == 3 && B == false)
    {
        // I had to put the "if" clause, because without it the local variables
        // would have been discarded for being unreferenced
    }
}
```

The signature for the local variables in this method would be: 04 07 02 08 02. Keep in mind that every integer in the Blob is encoded but that in this case we won't notice it because the signature is made of values which are all less than 0x80. The first byte stands for the number of bytes of the signature, which in this case are 4 (keep in mind that this length is encoded). So, the actual signature begins form the second byte and its first member is 0x07, which is the LOCAL_SIG (it stands for the .locals directive). Every LocalVarSig starts with this directive. The second byte (or the third one, depending if you count the signature size) is the number of local variables (or, technically speaking, types) for this method, in our case there are only two types and so the value of this field is 0x02. The bytes which follow are the types themselves. You can find a list of possible kind of types either in the SDK reference or in the CorHdr.h. I will paste the enum from the header, because it seems to me that the brief description next to the values is even easier to understand than the one in the reference.

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

    // every type above PTR will be simple type

    ELEMENT_TYPE_PTR            = 0xf,      // PTR
    ELEMENT_TYPE_BYREF          = 0x10,     // BYREF

    // Please use ELEMENT_TYPE_VALUETYPE. ELEMENT_TYPE_VALUECLASS is deprecated.
    ELEMENT_TYPE_VALUETYPE      = 0x11,     // VALUETYPE

    ELEMENT_TYPE_CLASS          = 0x12,     // CLASS
    ELEMENT_TYPE_VAR            = 0x13,     // a class type variable VAR
    ELEMENT_TYPE_ARRAY          = 0x14,     // MDARRAY     ...   ...

    ELEMENT_TYPE_GENERICINST    = 0x15,     // GENERICINST    ...
    ELEMENT_TYPE_TYPEDBYREF     = 0x16,     // TYPEDREF  (it takes no args) a typed referece to some other type

    ELEMENT_TYPE_I              = 0x18,     // native integer size
    ELEMENT_TYPE_U              = 0x19,     // native unsigned integer size
    ELEMENT_TYPE_FNPTR          = 0x1B,     // FNPTR

    ELEMENT_TYPE_OBJECT         = 0x1C,     // Shortcut for System.Object
    ELEMENT_TYPE_SZARRAY        = 0x1D,     // Shortcut for single dimension zero lower bound array
                                            // SZARRAY
    ELEMENT_TYPE_MVAR           = 0x1e,     // a method type variable MVAR

    // This is only for binding
    ELEMENT_TYPE_CMOD_REQD      = 0x1F,     // required C modifier : E_T_CMOD_REQD
    ELEMENT_TYPE_CMOD_OPT       = 0x20,     // optional C modifier : E_T_CMOD_OPT

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

    // Note that this is the max of base type excluding modifiers
    ELEMENT_TYPE_MAX            = 0x22,     // first invalid element type
    ELEMENT_TYPE_MODIFIER       = 0x40,
    ELEMENT_TYPE_SENTINEL       = 0x01 | ELEMENT_TYPE_MODIFIER, // sentinel for varargs
    ELEMENT_TYPE_PINNED         = 0x05 | ELEMENT_TYPE_MODIFIER,
    ELEMENT_TYPE_R4_HFA         = 0x06 | ELEMENT_TYPE_MODIFIER, // used only internally for R4 HFA types

    ELEMENT_TYPE_R8_HFA         = 0x07 | ELEMENT_TYPE_MODIFIER, // used only internally for R8 HFA types

} CorElementType;
```

The two type bytes in the signature are 0x08 and 0x02, which translated are: ELEMENT_TYPE_I4 and ELEMENT_TYPE_BOOLEAN. As you can see these are exactly the local variables we had in our method: a 32bit unsigned integer and a boolean. Of course, I haven't told you everything about signatures, but you should have understood the way they work.

Conclusions

I hope this article gave you an easy insight into the .NET file format.

Daniel Pistelli