NGWS

Metadata Structures

This spec defines all of the data structures that a caller can pass to the Metadata API (bitmasks, signatures, custom attributes, marshalling descriptors)

This is preliminary documentation and subject to change

Last Updated: 8 June 2000

1 Introduction

This spec is a companion to the Metadata Interfaces spec. It describes data structures that you can emit into metadata – specifically

- bitmasks
- signatures
- custom attributes
- marshalling specs

The first is quite simple – you just need to know the names of the bits, what they mean, and what are the legal combinations. The others are moderately complex binary formats – each is defined in this spec via syntax charts and/or simple BNF grammars

2 Bitmasks

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

2.1 Token Types [CorTokenType]

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

mdtModule	0x00000000
mdtTypeRef	0x01000000
mdtTypeDef	0x02000000
mdtFieldDef	0x04000000
mdtMethodDef	0x06000000
mdtParamDef	0x08000000
mdtInterfaceImpl	0x09000000
mdtMemberRef	0x0a000000
mdtCustomAttribute	0x0c000000
mdtPermission	0x0e000000
mdtSignature	0x11000000
mdtEvent	0x14000000
mdtProperty	0x17000000
mdtModuleRef	0x1a000000
mdtTypeSpec	0x1b000000
mdtAssembly	0x21000000
mdtAssemblyRef	0x25000000
mdtFile	0x29000000
mdtComType	0x2a000000
mdtManifestResource	0x2b000000
mdtExecutionLocation	0x2d000000
mdtString	0x70000000
mdtName	0x71000000

2.2 Scope Open Flags [CorOpenFlags]

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

ofRead : open scope for read
ofWrite : open scope for write
ofCopyMemory : open cope with memory. Metadata keeps own copy
ofCacheImage : EE maps but does not perform relocs or verify image

2.3 Options for Size Calculation [CorSaveSize]

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

cssAccurate : find exact save size; accurate but slow
cssQuick : estimate save size; may pad estimate; but fast
cssDiscardTransientCAs : remove all Custom Attributes that are marked discardable

2.4 Flags for Types [CorTypeAttr]

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

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

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

Visibility : whether a type can be 'seen' outside of its assembly.

tdNotPublic : type cannot be seen outside of its assembly
tdPublic : type *can* be seen outside of its assembly

Accessibility of a nested class

tdNestedPrivate : class is nested. Accessible only by methods in its own, or its enclosing type
tdNestedFamily : class is nested. Accessible only by methods within its family; ie, its own type and any sub-types
tdNestedAssembly : class is nested. Accessible only by methods within its assembly
tdNestedFamANDAssem : class is nested. Accessible only by methods lying in the intersection of its family and assembly
tdNestedFamORAssem : class is nested. Accessibile only by methods lying in the union of its family and assembly

Layout of a class

tdAutoLayout : fields will be laid out at the whim of the runtime

tdLayoutSequential : fields will be laid out sequentially, in the order the fields were emitted to the metadata. You can control the gaps between fields by specifying a packing size (by a call to *SetClassLayout*)
tdExplicitLayout : fields will be laid out at the offsets specified (by a call to *SetClassLayout*)

Class semantics : these define the sort of type-definition.

tdClass : this is a class
tdInterface : this is an interface
tdValueType : this is a valuetype
tdUnmanagedValueType : is never allocated from the GC heap

Additional Class Semantics : these are used, in addition to the preceding "class semantic" flags, to refine what sort of type is being defined

tdAbstract : abstract (cannot be instantiate)
tdSealed : class cannot be derived-from
tdSpecialName : class is special : its name says how

Implementation Attributes

tdSerializable : class can be serialized

Interop Attributes

tdAnsiClass : strings are marshalled to unmanaged ANSI strings
tdUnicodeClass : strings are marshalled to unmanaged UNICODE strings
tdAutoClass : strings are marshalled to unmanaged ANSI or UNICODE, as determined by the platform, at runtime

Reserved for internal use : do not set these via the metadata APIs

tdRTSpecialName : class is treated specially by the runtime
tdImport : class or interface is defined in a type library
tdLateInit : class can be initialized lazily by runtime
tdHasSecurity : used internally

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

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

Figure 1 – Legal Flag Combinations from CorTypeAttr

	Class	Interface	ValueType	Unmgd ValueType
tdClass				
tdInterface				
tdValueType				
tdUnmanagedValueType				
tdNotPublic				
tdPublic				

tdNestedPublic	□	□	□	
tdNestedPrivate	□	□	□	
tdNestedFamily	□	□	□	
tdNestedAssembly	□	□	□	
tdNestedFamANDAssem	□	□	□	
tdNestedFamOrAssem	□	□	□	
tdAutoLayout	□		□	□
tdLayoutSequential	□		□	□
tdExplicitLayout	□		□	□
tdAbstract	□	□	□	□
tdSealed	□		□	□
tdSpecialName	□	□	□	□
tdRTSpecialName	□	□	□	□

Notes:

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

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

2.5 Flags for Fields [CorFieldAttr]

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

Field Accessibility

fdPublic : accessible by any methods
fdPrivate : accessible only by methods in its own type
fdFamily : accessible only by methods within its family; ie, its own type and any subtypes
fdAssembly : accessible only by methods within its assembly
fdFamANDAssem : accessible only by methods lying in the intersection of its family and assembly
fdFamORAssem : accessible only by methods lying in the union of its family and assembly
fdPrivateScope : field cannot be referenced (typically used by a compiler to mark a field which is a static local variable in a method)

Field *Contract*

fdStatic : field is defined for a type (else an instance field). Note this flag is encoded into a field signature; you don't need to specify both, but if you do, they should match
fdInitOnly : field value may be set only during initialize (in the class constructor). The runtime checks this behaviour; and JITs use this flag to optimize their code. Note that a field cannot be marked both fdInitOnly and fdLiteral (see next)
fdLiteral : value is compile-time constant. It might be optimized away by the compiler; in this case, the value is 'burned' into the IL stream, and no memory is allocated to hold this value. It is illegal to take the address of a literal field. Note that a field cannot be marked both fdLiteral and fdInitOnly. If a field is marked fdLiteral, it must also be marked fdStatic.
fdNotSerialized : field will not be serialized (unless class author implements ISerializable)
fdSpecialName : field is special : its name says in what way
fdPinvokeImpl : field is reached via PInvoke dispatch

Reserved for internal use : do not set these via the metadata APIs

fdRTSpecialName
fdHasFieldMarshal
fdHasSecurity
fdHasDefault
fdHasFieldRVA

2.6 Flags for Methods [CorMethodAttr]

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

Method Accessibility

mdPublic : callable by any method
mdPrivate : callable only by methods in its own, or its parent type
mdFamily : callable only by methods within its family; ie, its own type and any sub-types
mdAssem : callable only by methods within its assembly
mdFamANDAssem : callable only by methods lying in the intersection of its family and assembly
mdFamORAssem : callable only by methods lying in the union of its family and assembly
mdPrivateScope : method cannot be called (typically used by a compiler for a method whose scope is restricted to its compiland; eg C++ static global function)

Method *Contract*

mdStatic : method is defined for a type (else an instance method). Note this flag is encoded into a field signature; you don't need to specify both, but if you do, they should match
mdFinal : method may not be over-ridden by a sub-class. Mutually exclusive with mdAbstract
mdVirtual : method is virtual
mdAbstract : method has no implementation in this class. Mutually exclusive with mdFinal

mdHideBySig : method is hidden by name + signature, else just by name
mdUnmanagedExport : method is managed, but exported via the EAT, to unmanaged code (runtime inserts a marshalling thunk)
mdPinvokeImpl : method is called via PInvoke dispatch
mdSpecialName : method is special : its name says in what way. Used, for example, for operator overload

Vtable Layout

mdReuseSlot : reuse an existing slot. Note that if the superclass' declaration is deleted, and you have no references to that method in your implementation, then your declaration will be used to create the slot (may hide)
mdNewSlot : method always gets a new slot (hides)

Reserved for internal use : do not set these via the metadata APIs

mdRTSpecialName : method is treated specially by the runtime. If set, you must also set the mdSpecialName bit (eg ".ctor")
mdHasSecurity
mdRequireSecObject

2.7 Flags for Method Parameters [CorParamAttr]

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

Flags

pdIn : input parameter
pdOut : output parameter
pdLcid : LCID
pdRetVal : return value from a method
pdOptional : parameter is optional

Reserved for internal use : do not set these via the metadata APIs

pdHasDefault
pdHasFieldMarshal
pdReserved3
pdReserved4

2.8 Flags for Properties [CorPropertyAttr]

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

Flags

prSpecialName : property is special : its name says in what way. Used, for example, for operator overloading

Reserved for internal use : do not set these via the metadata APIs

prRTSpecialName : property is treated specially by the runtime. If set, you must also set the prSpecialName bit
prHasDefault
prReserved2 : reserved
prReserved3 : reserved

prReserved4 : reserved

2.9 Flags for Events [CorEventAttr]

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

evSpecialName : event is special : its name says in what way. Used, for example, for operator overloading
evRTSpecialName : event is treated specially by the runtime. If set, you must also set evSpecialName

2.10 Flags for MethodSemantics [CorMethodSemanticsAttr]

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

msSetter : the setter method for this property
msGetter : the getter method for this property
msOther : one of the 'other' methods defined for this property
msAddon : the AddOn method for the event
msRemoveOn : the RemoveOn method for the event
msFire : the Fire method for the event

2.11 Flags for Method Implementations [CorMethodImpl]

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

Method Implementation

miNative : implemented as native (machine) code. Mutually exclusive with miIL and miOPTIL
miIL : implemented as IL. Mutually exclusive with miNative and miOPTIL
miOPTIL : implemented as OPTIL. Mutually exclusive with miNative and miIL
miRuntime : implementation is provided by the runtime. For example, runtime supplies class initializer for a COM+ 1.0 class
miUnmanaged : implemented as unmanaged code
miManaged : implemented as managed code
miForwardRef : a forward reference (in C++) to a method whose implementation is provided in another module
miOLE : signature has been changed to return an HRESULT, with the real return value as a parameter
miSynchronized : method is single-threaded

miNoInlining : JIT is not allowed to inline this method
miOneWay : method returns void and all parameters are in-only. Can be executed synchronously or asynchronously with respect to the caller. On return, caller cannot assume the method has been executed yet

Reserved for internal use : do not set this flag via the metadata APIs

miInternalCall : reserved (indicates a fast call within minimal, or no, stack frame)

2.12 Flags for Security [CorDeclSecurity]

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

dclActionNil :
dclRequest :
dclDemand :
dclAssert :
dclDeny :
dclPermitOnly :
dclLinktimeCheck :
dclInheritanceCheck :
dclRequestMinimum :
dclRequestOptional :
dclRequestRefuse :
dclPrejitGrant :
dclPrejitDenied :

2.13 Struct for Field Offsets [COR_FIELD_OFFSET]

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

mdFieldDef ridOfField;
ULONG ulOffset;

2.14 Typedef for Signatures [PCOR_SIGNATURE]

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

typedef unsigned __int8 COR_SIGNATURE

typedef COR_SIGNATURE* PCOR_SIGNATURE

See section 3 for details on how signature 'blobs' should be formatted

2.15 Flags for PInvoke Interop [CorPinvokeMap]

Attributes that control how unmanaged methods are invoked, and how their arguments are mashalled via PInvoke, are defined using

IMetadataEmit::DefinePinvokeMap or *SetPinvokeMap*. All of the flags below can be applied only to a method, never to a field. The flags you can set are as follows:

Flags

pmNoMangle : directs PInvoke to take the name precisely as specified; it will not peform a fuzzy match on the name (eg specify Foo, but look for FooA, FooW, Foo). Can be applied only to methods.
pmCharSetNotSpec : no character set specified for marshalling
pmCharSetAnsi : marshal managed Strings to ASCII strings
pmCharSetUnicode : marshal managed Strings to Unicode strings
pmCharSetAuto : marshal managed Strings to ASCII or Unicode, as determined by current platform. Note, this is determined at *compile* time, not runtime.
pmPinvokeOLE : returns an HRESULT
pmSupportsLastError : save last error encountered whilst executing unmanaged code: can be interrogated later

Calling Convention Flags

pmCallConvWinapi : will use the calling convention for the actual windows platform; this is determined at *run* time
pmCallConvCdecl : use CDECL
pmCallConvStdcall : use STDCALL
pmCallConvThiscall : not supported
pmCallConvFastcall : not supported

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

2.16 SetOptions: Duplicate Checking [CorCheckDuplicatesFor]

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

MDNoDupChecks
MDDupTypeDef
MDDupInterfaceImpl
MDDupMethodDef
MDDupTypeRef
MDDupMemberRef
MDDupMethodImpl
MDDupCustomValue
MDDupCustomAttribute
MDDupParamDef
MDDupPermission
MDDupProperty
MDDupEvent
MDDupFieldDef
MDDupSignature
MDDupModuleRef
MDDupTypeSpec
MDDupImplMap
MDDupOrdinalMap

MDDupAssemblyRef
MDDupFile
MDDupComType
MDDupManifestResource
MDDupExecutionLocation
MDDupDefault : the default, set to MDNoDupChecks | MDDupTypeRef | MDDupMemberRef | MDDupSignature | MDDupTypeSpec
MDDupAll : set all bits on
MDDupENC : default for Edit & Continue – same as **MDDupAll**

2.17 SetOptions: Ref-to-Def Optimizations [CorRefToDefCheck]

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

MDRefToDefNone
MDTypeRefToDef
MDMemberRefToDef
MDRefToDefDefault : default, = MDTypeRefToDef | MDMemberRefToDef
MDRefToDefAll : set all bits on

2.18 SetOptions: Token Remap Notification [CorNotificationForTokenMovement]

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

MDNotifyNone
MDNotifyMethodDef
MDNotifyMemberRef
MDNotifyFieldDef
MDNotifyTypeRef
MDNotifyTypeDef
MDNotifyParamDef
MDNotifyMethodImpl
MDNotifyInterfaceImpl
MDNotifyProperty
MDNotifyEvent
MDNotifySignature
MDNotifyTypeSpec
MDNotifyCustomValue
MDNotifyCustomAttribute
MDNotifySecurityValue
MDNotifyPermission
MDNotifyModuleRef
MDNotifyNameSpace
MDNotifyDebugTokens : covers all debug tokens
MDNotifyAssemblyRef
MDNotifyFile
MDNotifyComType
MDNotifyResource

MDNotifyExecutionLocation
MDNotifyDefault : MDNotifyTypeRef | MDNotifyMethodDef | MDNotifyMemberRef | MDNotifyFieldDef
MDNotifyAll : set all bits on

2.19 SetOptions: Edit & Continue [CorSetENC]

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

MDUpdateENC : ENC mode. Tokens don't move; can be updated
MDUpdateFull : normal update mode
MDUpdateExtension : extension mode. Tokens don't move, adds only
MDUpdateIncremental : incremental compilation
MDUpdateDelta : if ENC on, save only deltas

2.20 SetOptions: Out-of-Order Errors [CorErrorIfEmitOutOfOrder]

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

MDErrorOutOfOrderNone : do not generate any errors for out of order emit
MDMethodOutOfOrder : generate error when methods are emitted out of order
MDFieldOutOfOrder : generate error when fields are emitted out of order
MDParamOutOfOrder : generate error when params are emitted out of order
MDPropertyOutOfOrder : generate error when properties are emitted out of order
MDEventOutOfOrder : generate error when events are emitted out of order
MDErrorOutOfOrderDefault : default = do not generate any errors
MDErrorOutOfOrderAll : set all bits on

2.21 SetOptions: Hide Deleted Tokens [CorImportOptions]

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

MDImportOptionAllTypeDefs : all TypeDefs
MDImportOptionAllMethodDefs : all MethodDefs
MDImportOptionAllFieldDefs : all FieldDefs
MDImportOptionAllProperties : all Properties
MDImportOptionAllEvents : all Events
MDImportOptionAllCustomAttributes : all CustomAttributes
MDImportOptionAllComTypes : all ComTypes
MDImportOptionDefault : default is none
MDImportOptionAll : set all bits on

2.22 Flags for Assemblies [CorAssemblyFlags]

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

afImplicitComTypes : ComType definitions are implicit within the files
afImplicitResources : resource definitions are implicit within the files
afSideBySideCompatible: assembly is side by side compatible
afNonSideBySideAppDomain : assembly cannot execute with other versions if they are executing in the same application domain
afNonSideBySideProcess : assembly cannot execute with other versions if they are executing in the same process
afNonSideBySideMachine : assembly cannot execute with other versions if they are executing on the same machine

2.23 Flags for Assembly Reference [CorAssemblyRefFlags]

Assembly references are defined using IMetadataEmit::DefineAssemblyRef. The flags you can set are as follows:

arFullOriginator : assembly ref holds the full (undotted) originator

2.24 Flags for Manifest Resources [CorManifestResourceFlags]

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

mrPublic : the resource is exported from the assembly

mrPrivate : the resource is private to the assembly

2.25 Flags for Files [CorFileFlags]

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

ffWriteable : the file is writeable post-build

ffContainsNoMetaData : the file contains no metadata

2.26 Element Types in the runtime [CorElementType]

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

'Simple' Types

ELEMENT_TYPE_END : used to terminate arrays of info in the metadata API

ELEMENT_TYPE_VOID
ELEMENT_TYPE_BOOLEAN
ELEMENT_TYPE_CHAR
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

'Non-Simple' Types

ELEMENT_TYPE_PTR
ELEMENT_TYPE_BYREF
ELEMENT_TYPE_VALUETYPE
ELEMENT_TYPE_CLASS
ELEMENT_TYPE_ARRAY : the most general array – multi-dimensional, with lower and upper bounds
ELEMENT_TYPE_COPYCTOR : copy-construct the argument
ELEMENT_TYPE_TYPEDBYREF
ELEMENT_TYPE_VALUEARRAY
ELEMENT_TYPE_I : native integer – will JIT to the platform's 'natural' size
ELEMENT_TYPE_U : native unsigned integer – will JIT to the platform's 'natural' size
ELEMENT_TYPE_R : native real – will JIT JIT to the platform's 'natural' size
ELEMENT_TYPE_FNPTR : function pointer
ELEMENT_TYPE_OBJECT : a shortcut for System.Object
ELEMENT_TYPE_SZARRAY : single dimension array with zero lower bound
ELEMENT_TYPE_GENERICARRAY : 'open' array – no rank or dimensions information

Modifiers

ELEMENT_TYPE_CMOD_REQD : required NGWS modifier; if a compiler imports a type with this modifier set, it should only use the type if it 'understands' the required semantic of the language that defined the type
ELEMENT_TYPE_CMOD_OPT : optional NGWS modifier; if a compiler imports a type with this modifier set, it is OK to use
ELEMENT_TYPE_MODIFIER : set this bit, together with either of the following:
- **ELEMENT_TYPE_SENTINEL** : sentinel to mark end of predefined arguments in a varargs method signature
- **ELEMENT_TYPE_PINNED** : object is pinned against garbage reclamation

2.27 Calling Conventions [CorCallingConvention]

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

Calling Conventions

IMAGE_CEE_CS_CALLCONV_DEFAULT : use default calling convention; determined at runtime
IMAGE_CEE_CS_CALLCONV_VARARG : C's "vararg" (variable number of arguments)
IMAGE_CEE_CS_CALLCONV_FIELD : denotes this signature is a field, not a method
IMAGE_CEE_CS_CALLCONV_LOCAL_SIG : field is a method-local variable
IMAGE_CEE_CS_CALLCONV_PROPERTY : 'field' is a property
IMAGE_CEE_CS_CALLCONV_UNMGD : calls unmanaged code

Modifier Bits : 'or' these bits into the previous values, if required (actually two bits in the high nybble of the calling convention byte)

IMAGE_CEE_CS_CALLCONV_HASTHIS : JIT a 'this' argument for this method
IMAGE_CEE_CS_CALLCONV_EXPLICITTHIS : this parameter is explicitly in the signature

2.28 Unmanaged Calling Conventions [CorUnmanagedCallingConvention]

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

IMAGE_CEE_UNMANAGED_CALLCONV_C
IMAGE_CEE_UNMANAGED_CALLCONV_STDCALL
IMAGE_CEE_UNMANAGED_CALLCONV_THISCALL
IMAGE_CEE_UNMANAGED_CALLCONV_FASTCALL

2.29 Argument Types [CorArgType]

These types are used in defining method signatures. See section 3 for more detail

IMAGE_CEE_CS_END
IMAGE_CEE_CS_VOID
IMAGE_CEE_CS_I4
IMAGE_CEE_CS_I8
IMAGE_CEE_CS_R4
IMAGE_CEE_CS_R8
IMAGE_CEE_CS_PTR
IMAGE_CEE_CS_OBJECT
IMAGE_CEE_CS_STRUCT4
IMAGE_CEE_CS_STRUCT32
IMAGE_CEE_CS_BYVALUE

2.30 Native Types [CorNativeType]

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

NATIVE_TYPE_BOOLEAN : 4 byte boolean value: TRUE = non-zero, FALSE = 0
NATIVE_TYPE_I1
NATIVE_TYPE_U1
NATIVE_TYPE_I2
NATIVE_TYPE_U2
NATIVE_TYPE_I4
NATIVE_TYPE_U4
NATIVE_TYPE_I8
NATIVE_TYPE_U8
NATIVE_TYPE_R4
NATIVE_TYPE_R8
NATIVE_TYPE_BSTR : Basic string
NATIVE_TYPE_LPSTR : ASCII string
NATIVE_TYPE_LPWSTR : Unicode string
NATIVE_TYPE_LPTSTR : choose LPSTR or LPWSTR, depending on compile-time platform
NATIVE_TYPE_FIXEDSYSSTRING : string in a fixed-length buffer
NATIVE_TYPE_STRUCT : C-style struct
NATIVE_TYPE_INTF : COM interface
NATIVE_TYPE_SAFEARRAY : OLE automation safe array
NATIVE_TYPE_FIXEDARRAY : fixed-length array
NATIVE_TYPE_INT : native integer – will JIT to the platform's 'natural' size
NATIVE_TYPE_UINT : native unsigned integer – will JIT to the platform's 'natural' size
NATIVE_TYPE_BYVALSTR : used only by Visual Basic
NATIVE_TYPE_ANSIBSTR : length-prefixed ASCII string
NATIVE_TYPE_TBSTR : choose BSTR or ANSIBSTR, depending on compile-time platform
NATIVE_TYPE_VARIANTBOOL : 2-byte boolean value: TRUE = -1, FALSE = 0
NATIVE_TYPE_FUNC
NATIVE_TYPE_LPVOID : blind pointer (no deep marshaling)
NATIVE_TYPE_ASANY
NATIVE_TYPE_R : native real – will JIT to the platform's 'natural' size
NATIVE_TYPE_ARRAY
NATIVE_TYPE_LPSTRUCT : pointer to a C-style struct
NATIVE_TYPE_CUSTOMMARSHALER : custom marshaler native type.

---

3 Signatures

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

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

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

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

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

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

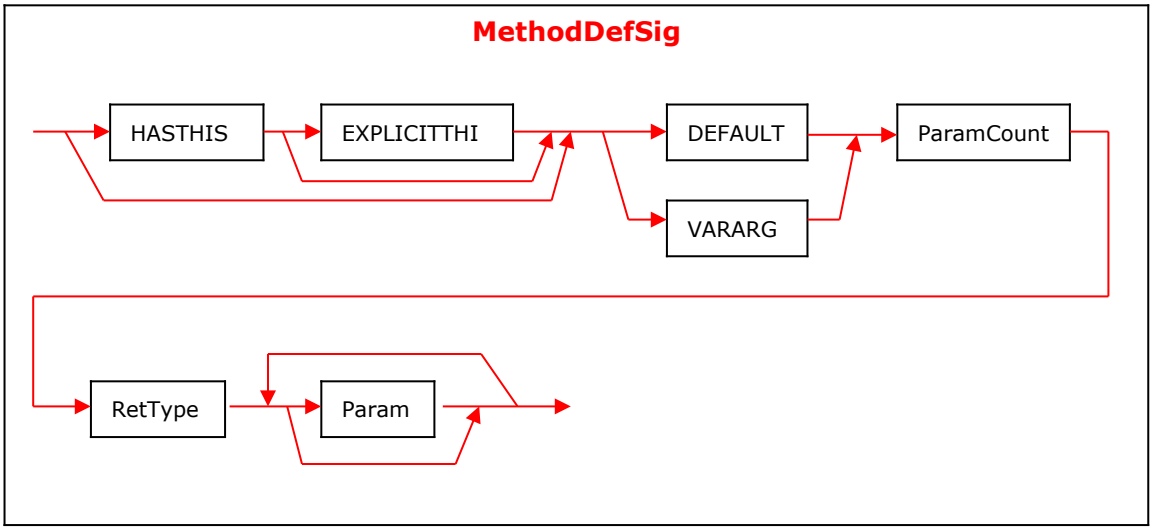
(Note that *CorSigCompressSignedInt* is not currently used to build in Signatures). In order to uncompress a value in a Signature, you must know (from its position in the Signature) whether to call *CorSigUncompressData* or *CorSigUncompressToken*

Signatures include two *modifiers* called:

- ELEMENT_TYPE_BYREF – this element 'points' to data item which may be allocated from the GC heap, or from elsewhere. It may 'point' to the start of an object, or to the interior of an object. Either way, the GC is notified of its existence; if it actually 'points' into the heap, then GC knows to update its value if it moves the object pointed-to during a garbage collection. This modifier can only occur in the definition of Param (section 3.10) or RetType (section 3.11). It may **not** occur within the definition of a Field (section 3.4) [conceptually you could imagine a runtime that *did* support BYREF fields, but ours doesn't – BYREFs, especially those that point into the interior of an object in the GC heap, are expensive to track – since there's no very strong requirement for BYREF fields, we excluded them]
- ELEMENT_TYPE_PTR – this element 'points' to a data item which is not allocated from the GC heap. This modifier can occur in the definition of Param (section 3.10) or RetType (section 3.11) or Field (section 3.4)

3.1 MethodDefSig

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



This chart uses the following abbreviations:

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

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

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

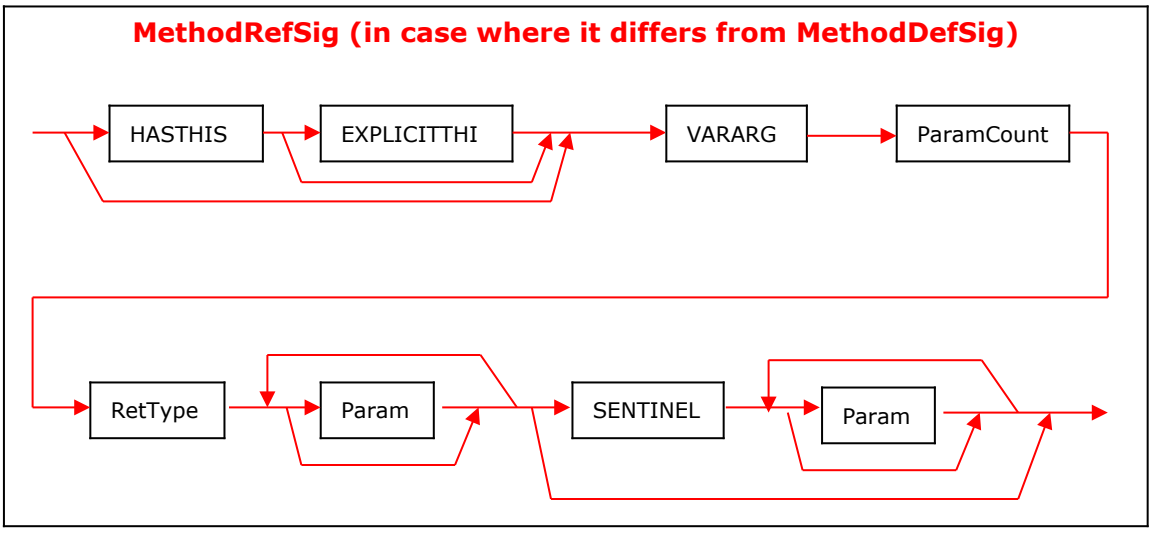
The *RetType* item describes the type of the method's return value (see later)

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

3.2 MethodRefSig

A MethodRefSig is indexed by the MemberRef.Signature column. This provides the *callsite* Signature for a method. Normally, this callsite Signature must match exactly the Signature specified in the definition of the target method. For example, if a method Foo is defined that takes two uint32s and returns void; then any callsite must index a signature that takes exactly two uint32s and returns void. In this case, the syntax chart for a MethodRefSig is identical with that for a MethodDefSig – see section 3.1

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



This chart uses the following abbreviations:

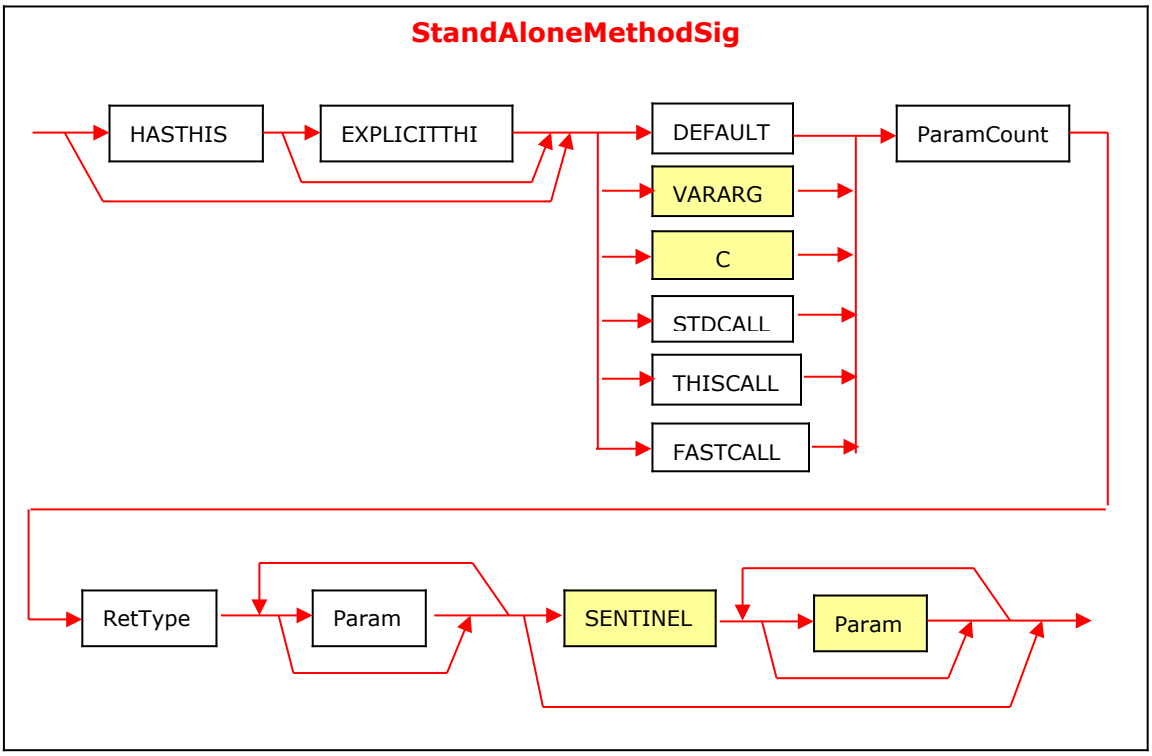
- HASTHIS for IMAGE_CEE_CS_CALLCONV_HASTHIS
- EXPLICITTHIS for IMAGE_CEE_CS_CALLCONV_EXPLICITTHIS
- VARARG for IMAGE_CEE_CS_CALLCONV_VARARG
- SENTINEL for ELEMENT_TYPE_SENTINEL

This starts just like the MethodDefSig for a VARARG method (see section 3.1). But we then append an ELEMENT_TYPE_SENTINEL token, followed by extra *Param* items to describe the extra VARARGE arguments. Note that the *ParamCount* item must tell us the total number of *Param* items in the Signature – before and after the SENTINEL byte.

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

3.3 StandAloneMethodSig

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



This chart uses the following abbreviations:

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

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

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

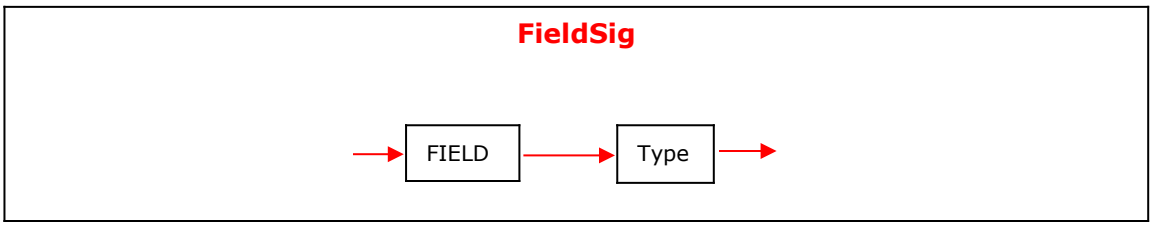
the signature ends just before the SENTINEL item (these are all non vararg signatures). However, for the managed and unmanaged vararg calling conventions:

VARARG (managed)
C (unmanaged)

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

3.4 FieldSig

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



This chart uses the following abbreviations:

- FIELD for IMAGE_CEE_CS_CALLCONV_FIELD

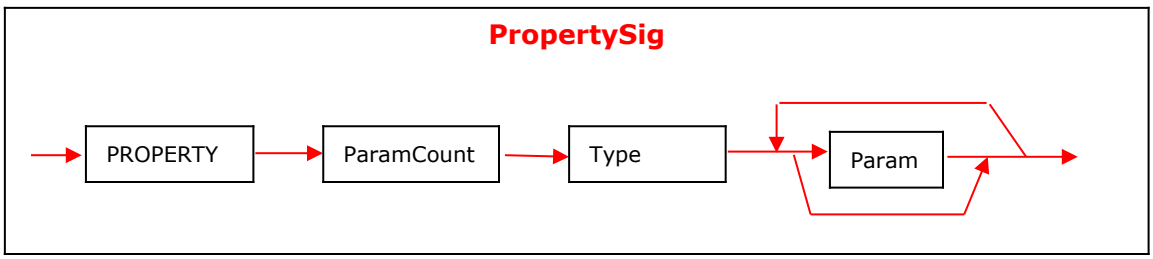
Type is defined in section 3.12

3.5 PropertySig

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

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

The syntax chart for a PropertySig looks like this:



This chart uses the following abbreviations:

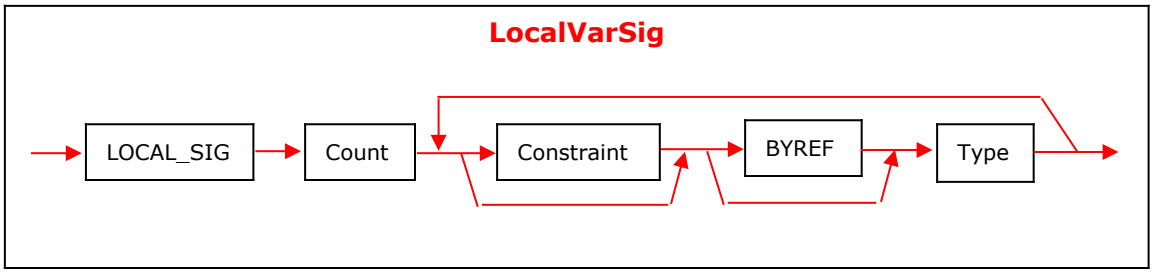
- PROPERTY for IMAGE_CEE_CS_CALLCONV_PROPERTY

Type specifies the type returned by the *Getter* method for this property. *Type* is defined in section 3.12. *Param* is defined in section 3.10

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

3.6 LocalVarSig

A LocalVarSig is indexed by the StandAloneSig.Signature column. It captures the type of all the local variables in a method. Its syntax chart looks like this:



This chart uses the following abbreviations:

- LOCAL_SIG for IMAGE_CEE_CS_CALLCONV_LOCAL_SIG
- BYREF for ELEMENT_TYPE_BYREF

Constraint is defined in section 3.9 *Type* is defined in section 3.12

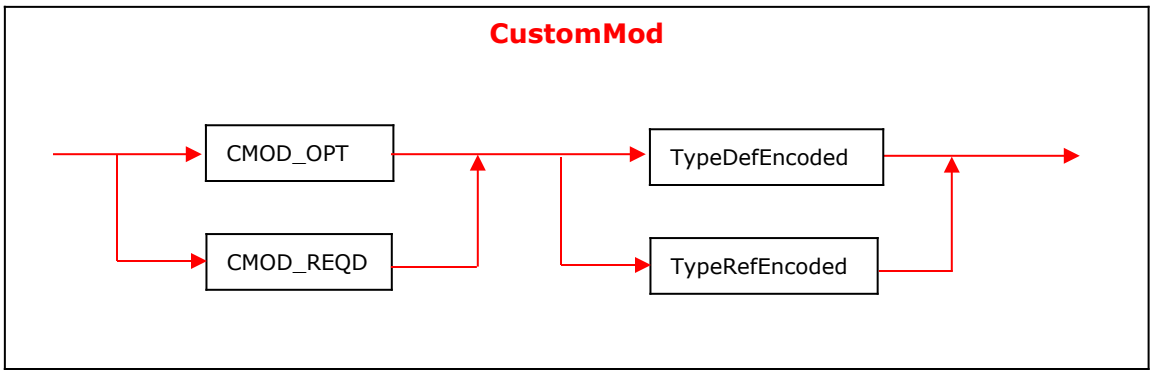
Count is an unsigned integer that holds the number of local variables. It can be any number between 1 and 0xFFFF (constrained by the IL instruction set). The compiler compresses it, using *CorSigCompressData*, before storing into the blob (it almost always compresses into one byte)

There must be *Count* instances of the *Constraint**-BYREF?-*Type* chain in the LocalVarSig

A *LocalVarSig* is created by Compilers and other code generators. For example, ILASM generates a *LocalVarSig* in response to the .locals directive

3.7 CustomMod

The *CustomMod* (custom modifier) item in Signatures has a syntax chart like this:



This chart uses the following abbreviations:

- CMOD_OPT for ELEMENT_TYPE_CMOD_OPT
- CMOD_REQD for ELEMENT_TYPE_CMOD_REQD

The CMOD_OPT or CMOD_REQD value is compressed using *CorSigCompressData* – their values today are small numbers, so they always compress to a single byte.

This item is followed by an metadata token that indexes a row in the *TypeDef* table or the *TypeRef* table. However, these tokens are encoded and compressed – see section 3.8 for details

If the CustomModifier is tagged CMOD_OPT, then any importing compiler can freely ignore it entirely. Conversely, if the CustomModifier is tagged CMOD_REQD, any importing compiler must 'understand' the semantic implied by this CustomModifier in order to reference the surrounding Signature.

A typical use for a CustomModifier is for VC++ to tag a *const* parameter to a method

3.8 TypeDefEncoded and TypeRefEncoded

These items are compact ways to store a TypeDef or TypeRef token in a Signature.

Consider a regular TypeRef token, such as 0x01000012. The top byte of 0x01 tells us this is a TypeRef token (see the CorTokenType enum in CorHdr.h). The lower 3 bytes (0x000012) index row number 0x12 in the TypeRef table

The encoded version of this TypeRef token is made up as follows:

a) encode the table that this token indexes as the least significant 2 bits. The bit values to use are defined in Cor.h, as follows:

```
const static mdToken g_tkCorEncodeToken[4] = {mdtTypeDef,
mdtTypeRef, mdtTypeSpec, mdtBaseType};
```

b) shift the 3-byte row index (0x000012 in our example) left by 2 bits and OR into the 2-bit encoding from step a)

c) call *CorSigCompressData* on the resulting value

For our example, we end up with the following encoded value:

```
a)  encoded = g_tkCorEncodToken[1] = 0b0001
b)  encoded = ( 0x000012 << 2 ) |  0x01
            = 0x48 | 0x01
            = 0x49
c)  encoded = CorSigCompressData (0x49)
            = 0x49
```

So, instead of the original, regular TypeRef token value of 0x01000012, requiring 4 bytes of space in the Signature blob, we encode it as a single byte.

Note that there are two helper functions in Cor.h – *CorSigCompressToken* and *CorSigUncompressToken* that combine these steps together (encoding the target table type and compressing)

3.9 Constraint

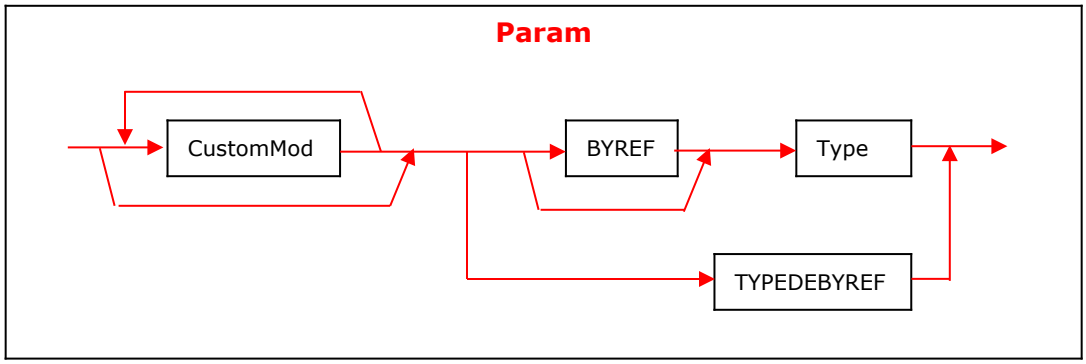
The *Constraint* item in Signatures currently has only one possible value – ELEMENT_TYPE_PINNED, which specifies that the target type is pinned in the runtime heap, and will not be moved by the actions of garbage collection. Note that the Compiler calls *CorCompressData* to compress the value for *Modifier* before inserting into the Signature blob; but today's value is small enough that it compresses to a single byte.

A *Constraint* can only be applied within a LocalVarSig (not a FieldSig). The Type of the local variable must either be a reference type (in other words, it *points* to the actual variable – for example, an Object, or a String); or it must include the BYREF item. The reason is that local variables are allocated on the runtime stack – they are never allocated from the runtime heap; so unless the local variable *points* at an object allocated in the GC heap, pinning makes no sense.

[Note: in previous versions, *Constraint* could also include a VOLATILE value. However, this constraint was removed from the Signature – compilers instead issue IL instructions that indicate the target variable is volatile]

3.10 Param

The *Param* (parameter) item in Signatures has a syntax chart like this:



This chart uses the following abbreviations:

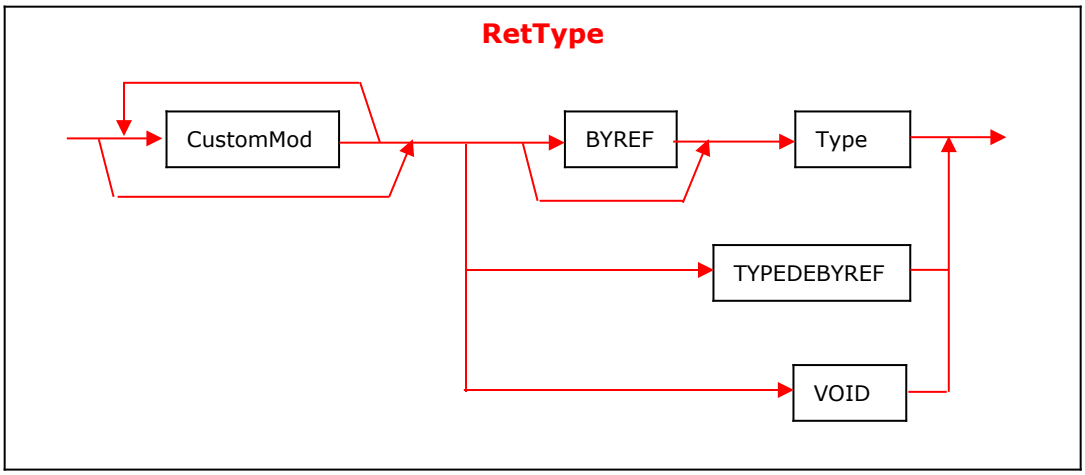
- BYREF for ELEMENT_TYPE_BYREF
- TYPEDBYREF for ELEMENT_TYPE_TYPEDBYREF

CustomMod is defined in section 3.7. *Type* is defined in section 3.12

A TYPEDBYREF is a simple structure of two DWORDs – one indicates the type of the parameter, the other, its value. This struct is pushed on the stack by the caller. So, only at runtime, is the type of the parameter actually provided. TYPEDBYREF was originally introduced to support VB's "refany" argument-passing technique

3.11 RetType

The *RetType* (return type) item in Signatures has a syntax chart like this:



RetType is identical to *Param* except for one extra possibility, that it can include the type VOID. This chart uses the following abbreviations:

- BYREF for ELEMENT_TYPE_BYREF
- TYPEDBYREF for ELEMENT_TYPE_TYPEDBYREF (see section 3.10)
- VOID for ELEMENT_TYPE_VOID

CustomMod is defined in section 3.7. *Type* is defined in section 3.12

3.12 Type

The *Type* item in Signatures can be quite complicated. Below is a simple EBNF grammar for *Type.* As usual, "|" separates alternatives, "*" denotes zero or more occurrences, "?" denotes zero or one occurrence. Note that the last four productions are all recursive: PTR, GENERICARRAY and SZARRAY are left-recursive, whilst ARRAY is right-recursive.

Type := Intrinsic
| VALUETYPE TypeDefOrRefEncoded
| CLASS TypeDefOrRefEncoded
| STRING
| OBJECT
| PTR CustomMod* VOID
| FNPTR MethodDefSig
| FNPTR MethodRefSig
| PTR CustomMod* Type
| ARRAY Type ArrayShape
| GENERICARRAY CustomMod* Type
| SZARRAY CustomMod* Type

For compactness, we have missed out the ELEMENT_TYPE_ prefixes in this list. So, for example, "CLASS" is shorthand for ELEMENT_TYPE_CLASS (see the CorElementType enum defined in CorHdr.h)

3.12.1 Intrinsic

This represents the set of simple value types provided by the runtime. They are defined as follows:

BOOLEAN | CHAR | I1 | U1 | I2 | U2 | I4 | U4 | I8 | U8 | R4 | R8 | I | U | R

However, CLS (Common Language Subset) does not support this full range of intrinsic types – it excludes those in listed in the CLS rule below

3.12.2ARRAY Type ArrayShape

The ARRAY production describes the most general definition of an array – multi-dimensional, specifying size and lower bounds for each dimension. There are two specialized versions of ARRAY – SZARRAY and GENERICARRAY. Compilers *must* specify these specialized versions when possible to do so

3.12.3GENERICARRAY CustomMod* Type

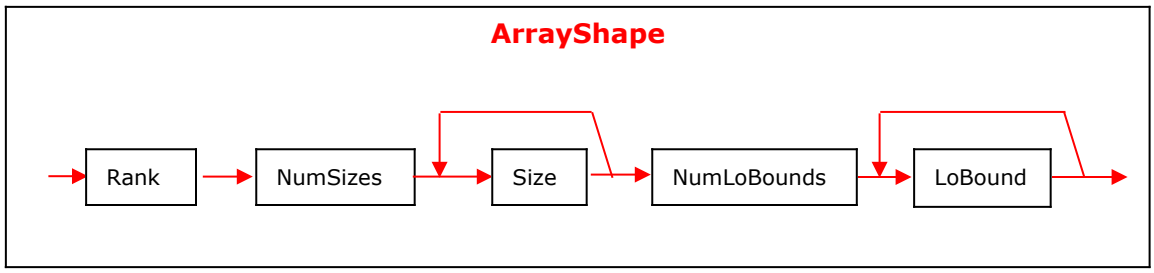
The GENERICARRAY production describes an infrequently-used, special-case of ARRAY – that's to say, one whose element type is known, but nothing else – no rank, sizes or bounds. (This signature is emitted by C# for an "int[?]" array)

3.12.4SZARRAY CustomMod* Type

The SZARRAY production describes a frequently-used, special-case of ARRAY – that's to say, a single-dimension (rank 1) array, with a zero lower bound, and no specified size

3.13 ArrayShape

An ArrayShape has the following syntax chart:



Rank is an integer (compressed using *CorSigCompressData*) that specifies the number of dimensions in the array (must be 1 or more). *NumSizes* is a compressed integer that says how many dimensions have specified sizes (it must be 0 or more). *Size* is a compressed integer specifying the size of that dimension – the sequence starts at the first dimension, and goes on for a total of *NumSizes* items. Similarly, *NumLoBounds* is a compressed integer that says how many dimensions have specified lower bounds (it must be 0 or more). And *LoBound* is a compressed integer specifying the lower bound of that dimension – the sequence starts at the first dimension, and goes on for a total of *NumLoBounds* items. Note that you cannot 'skip' dimensions in these two sequences – but you are allowed to specify less than all *Rank* dimensions. Here are a few examples, all for element type I4:

	Type	Rank	NumSizes	Size*	NumLoBounds	LoBound*
[0..2]	I4	1	1	3	0	
[,,,,,,]	I4	6	0			
[0..3, 0..2,,,,]	I4	6	2	4 3	0	
[1..2, 6..8]	I4	2	2	2 3	2	1 6
[5, 3..5, ,]	I4	3	2	5 3	2	0 3

Note that definitions can nest, since the Type may itself be an array

Note: the runtime cares only about *Rank* when checking for a signature match; it ignores any dimension sizes or lower bounds. For the first release, we recommend that all languages emit an ARRAY Signature with *NumSizes* = *NumLoBounds* = 0.

3.14 Short Form Signatures

The general specification for signatures leaves some leeway in how to encode certain items. For example, it appears legal to encode a String as either

- long-form: (ELEMENT_TYPE_CLASS, TypeRef-to-System.String)
- short-form: ELEMENT_TYPE_STRING

Only the short form is valid. Below is a list of all possible long-form and short-form items. (As usual, for compactness, we miss out the ELEMENT_TYPE_ prefix – so VALUETYPE is short for ELEMENT_TYPE_VALUETYPE)

Long Form		**Short Form**
Prefix	**TypeRef to:**	
CLASS	System.String	STRING
CLASS	System.Object	OBJECT
VALUETYPE	System.Void	VOID
VALUETYPE	System.Boolean	BOOLEAN
VALUETYPE	System.Char	CHAR
VALUETYPE	System.Byte	U1
VALUETYPE	System.SByte	I1
VALUETYPE	System.Int16	I2
VALUETYPE	System.UInt16	U2
VALUETYPE	System.Int32	I4
VALUETYPE	System.UInt32	U4
VALUETYPE	System.Int64	I8
VALUETYPE	System.UInt64	U8
VALUETYPE	System.SysInt	I
VALUETYPE	System.SysUInt	U
VALUETYPE	System.SingleResult	R

Note: arrays must be encoded in signatures using one of ELEMENT_TYPE_ARRAY, ELEMENT_TYPE_SZARRAY or ELEMENT_TYPE_GENERICARRAY. There is no long form involving a TypeRef to System.Array

4 Custom Attributes

Programmers can attach CustomAttributes a programming element, such as a method or field. Each CustomAttribute is defined, by the programmer, as a regular Type to Metadata.

A CustomAttribute within metadata is a triple of (tokenParent, tokenMethod, blob) stored into metadata. The blob holds the arguments to the class constructor method specified by tokenMethod. The runtime 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.

4.1 Using Custom Attributes

The model for using CustomAttributes 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 we disallow attaching a custom-attribute object to a custom attribute-object.

CustomAttributes 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.

4.2 Persisted Format of an Attribute-Object

The data required to instantiate an object of an attribute-class is saved into Metadata in three parts:

- Prolog
- Constructor arguments
- Named Fields or Properties

Each constructor argument, each named field and each named property is written into metadata just as if it had been saved, using the NGWS binary serializer. [We make a few optimizations that avoid duplicating information that already exists elsewhere in the metadata]

In order to help compilers emit arguments, named fields and named properties, without using NGWS serialization, we specify how to serialize a chosen subset of VOS objects – the specific subset that compilers have requested for custom attributes.

It might help to have an example in mind, as we discuss the formats. Here is a simple one, written in C#

```
[attribute (VOSElementtype.All) ]
public class Attrib {
    public readonly string Name;
    public variant Whim;
    public int Depth { get{...}; set{...} }
    public Attrib(string n)         { this.Name = n; }
    public Attrib(string n, int d) { this.Name = n; this.Depth = d; }
}
[Attrib("Monday")]                          class Ex1 { . . . }
[Attrib("Tuesday", 2)]                      class Ex2 { . . . }
[Attrib("Friday", Whim=42]                  class Ex3 { . . . }
[Attrib("Green", Depth=3, Whim="yellow")] class Ex4 { . . . }
```

This example defines an attribute-class called Attrib, with two fields – Name and Whim, and one property, Depth. It defines two constructors – the first takes one positional argument; the second takes two.

Following the definition of Attrib we show it used to attribute four classes called Ex1 through Ex4. Ex1 is hooked to an Attrib object using the single-argument constructor. Ex2 is hooked to an Attrib object using the two-argument constructor. Ex3 is hooked to an Attrib object using a constructor which takes the one-argument constructor, and sets the named field Whim. The outcome of this is to instantiate an Attrib object with Name of "Friday" and Whim (a variant field) holding the integer value 42. Finally, Ex4 is hooked to an Attrib object using the one-argument constructor, augmented by values for the Depth property and the Whim field.

Note any class may have multiple attribute-objects 'hooked' to it. These can be of different types, or even of the same type.

All binary data is persisted in **little-endian** format (least signficant bytes come first in the file). The format for floats and doubles is IEEE-754. For 8-byte doubles, the more-significant 4 bytes is emitted *after* the less-significant 4 bytes. There is just one exception to the little-endian rule – the "PackedLen" count that precedes a string – a one-two-or-four byte item – is always encoded big-endian.

Note that, if the constructor method takes no arguments, and you don't want to specify any extra named fields or properties, you can omit the blob entirely.

4.3 Prolog

The prolog simply identifies the blob that follows. It consists of a two-byte ID. In the first release, set this to the value 1.

The prolog is obviously a hedge against future extensions to this blob format.

4.4 Constructor Arguments

We define a new enumeration, SERIALIZATION_TYPE_, which specifies data types. Where members correspond directly to runtime ELEMENT_TYPE_'s, we use the same name and value. Where members correspond to specific serialization types, we choose a value beyond the range used by the ELEMENT_TYPE_ enum. (See later for detailed list)

This spec provides a blow-by-blow account of how to serialize the following subset:

SERIALIZATION_TYPE_BOOLEAN	SERIALIZATION_TYPE_CHAR
SERIALIZATION_TYPE_I1	SERIALIZATION_TYPE_U1
SERIALIZATION_TYPE_I2	SERIALIZATION_TYPE_U2
SERIALIZATION_TYPE_I4	SERIALIZATION_TYPE_U4
SERIALIZATION_TYPE_I8	SERIALIZATION_TYPE_U8
SERIALIZATION_TYPE_R4	SERIALIZATION_TYPE_R8
SERIALIZATION_TYPE_STRING	SERIALIZATION_TYPE_TYPE

plus (a subset of) VARIANT. Also, a one-dimensional, zero-based array (SZARRAY) of any of those types. (The subset of VARIANT excludes DateTime, TimeSpan, Decimal, Currency and Object)

The signature for a class constructor will be stored in metadata, as a MethodDef or MethodRef. This specifies the number, order and type of each parameter. Therefore, we store the actual arguments into the PE file as dense binary, with no type descriptions and with no alignment packing. For each argument, emit the following data:

- For intrinsics, just their value (in their full field width)
- For STRING, a count of the number of **bytes** in the string (after encoding) followed immediately by the characters of the string in UTF8 format. (The count is encoded as a "PackedLen" – see below details) Note that the count represents the overall length, in bytes, of the UTF8 sequence. In general, this is not the same as the number of UTF8 characters, since different UTF8 characters can occupy between 1 and 3 bytes
- For VARIANT, a one-byte tag, defining which type this instance of the Variant corresponds to, followed by its actual value. Again, for this spec, we limit attention to those Variant types in the SERIALIZATION_TYPE_ list above

- For SZARRAY, the number of elements as an I4, followed by the value (in its full field width) of each element

For arguments of type System.Enum, emit the actual value of their underlying type. [As a specific example, a particular Enum might use 4-byte integers as its underlying type, and should therefore be saved as a SERIALIZATION_TYPE_I4 value]

For arguments of System.Type, emit the actual type as a String, including the full assembly name of the defining module

Note that SERIALIZATION_TYPE_BOOLEAN items are encoded in a single byte, with False = 0 and True = 1. [This contrasts with how NGWS lays out Booleans in memory – a single Boolean occupies 4 bytes, whereas each element of a Boolean array occupies just 1 byte]

If the attribute-class provides several constructors, overload resolution to the appropriate MethodDef or MethodRef must be done at compile time (ie, no late-binding). Runtime cannot therefore perform automatic widening (for example, store 16 bit integer, but widen to signature's parameter type of 32 bits)

For the length-in-bytes of a UTF8 string, we use the standard 1,2 or 4 byte "PackedLen" encoding used within Metadata (see the description of helper routine *CorSigCompressData* in section 3):

- If the length-in-bytes lies between 0 and 127, encode as a one-byte integer (bit #7 is obviously clear, integer held in bits #6 thru #0)
- If the length-in-bytes lies between 2^8 and 2^14 encode as a two-byte integer with bit #15 set, bit #14 clear (integer held in bits #13 thru #0)
- Otherwise, encode as a 4-byte integer, with bit #31 set, bit #30 set, bit #29 clear (integer held n bits #28 thru #0)
- A null string should be represented with the reserved single byte 0xFF, and no following data. (The value of 0xFF is a reserved value in Metadata's count prefix)

The table below shows several examples. The first column shows an example count value (one-byte, two-byte and three-byte). The second column shows the corresponding size, expressed as a normal integer.

Metadata Count Value	Corresponding Size
0x03	0x03
0x7F	0x7F (7 bits set)
0x8080	0x80
0x8081	0x81
0x83FF	0x3FF (14 bits set)
0xC0008400	0x8400
0xDFFFFFFF	0x1FFFFFFF (29 bits set)

Thus, by examining the most significant bits of a "PackedLen" field, code can determine whether it occupies 1, 2 or 4 bytes, as well as its value. For this to work, the "PackedLen" is stored in **big-endian** order – most significant byte at the smallest offset within the file. [see CPackedLen::GetLength and CPackedLen::PutLength methods in the Lightning source tree at $/Com99/Src/Utilcode/StgPooli.cpp code for details]

There is clearly scope to compact the above binary format, in the same way that existing metadata structures have been optimized to avoid "bloat". Possible

techniques are legion. The first release of the runtime does **not** include any such optimizations (except for "PackedLen")

4.5 Constructor Arguments – Example 1

```
Foo (int a, char[] b, String c);

int a = 7;

char[] b = new char[] {'A', 'B', 'C', 'D'};

String c = "Today";

Foo (a, b, c);
```

Note that this example snippet uses a language that stores each "char" as a two-byte Unicode character (contrast with C++ single-byte "char"). The arguments to the *Foo* constructor would be encoded as follows:

```
0100 07000000 04000000 41424344 05 546F646179 0000
```

We start with the Prolog – a 2-byte value of 1. Next comes the first argument – a 4-byte value of 7. The second argument, a 4-element char array, is represented by a 4-byte count-of-array-elements with value 4, followed by the four ASCII characters A thru D (each "char" element starts as a 2-byte Unicode value, but is compressed into a single byte when converted into Utf8). The third argument consists of the UTF8-encoded string "Today"; its length in bytes (5) fits into a single count byte, followed by 5 characters, each encoded into a single byte. [I have added whitespace for clarity – it's not really there of course]). The last value is a two-byte value of zero, giving the total number of named fields and named properties (see later). Note that the display of bytes is the same as they would appear in memory – each byte occupies the next highest address in memory

4.6 Constructor Arguments – Example 2

```
Enum Colors {Red, Green, Blue};

Bar (Variant a, Colors b, bool[] c);

Variant a = "Hello";

Colors b = Colors.Green;

bool[] b = new bool[] {false, true, true};

Bar (a, b, c);
```

The arguments to the Bar constructor would be encoded as follows:

```
0100 0E 05 48656C6C6F 01000000 03000000 00 01 01 0000
```

The Prolog is followed by the first argument, a VARIANT; it starts with a single-byte tag value 0x0E (SERIALIZATION_TYPE_STRING), and follows with a 5-byte string for "Hello" – a one-byte count, plus 5 bytes of UTF8 encoded characters. The second argument is an enumeration with a 4-byte integer base type; we serialize Green as its value (of 1). The third argument is a 3-element BOOLEAN array – so we have a 4-byte element count with value 3, followed by 3 bytes for each boolean value, in order (False = 0, True = 1). (Recall that BOOLEAN arrays are stored with one byte per element. This contrasts with a simple BOOLEAN, which is stored as a 4-byte

quantity). The last value is a two-byte value of zero, giving the total number of named fields and named properties (see later).

4.7 Constructor Arguments – Example 3

```
Zog (Variant[] a, short[] b);

Variant[] a = new Variant[] {123, "Hello", 11.0};

short[] b = new short[] {42, 7};

Zog (a, b);
```

The arguments to the Zog constructor would be encoded as follows:

```
0100 03000000 08 7B000000 0E 05 48656C6C6F 0D BA5E353F40100C49 02000000
2A00 0700 0000
```

The first argument is a VARIANT array with 3 elements; so we start with a 4-byte element count with value 3. Element 0 of the VARIANT array is an integer, which is encoded with a single-byte tag value of 08 (SERIALIZATION_TYPE_I4), followed by its 4-byte value (123 decimal, 7B hex). Element 1 of the VARIANT array is a String, which is encoded with a single-byte tag value of 0E (SERIALIZATION_TYPE_STRING), followed by the byte-count of 05 and the UTF8 string for "Hello". Element 2 of the VARIANT array is a double, so it starts with a single-byte tag value 0D (SERIALIZATION_TYPE_R8), and follows with the 8-byte binary floating-point representation for 11.0

The second argument is a short array with 2 elements. We start with a 4-byte count of elements. Then follows two shorts – 42 decimal (2A hex) and 7 decimal. The last value is a two-byte value of zero, giving the total number of named fields and named properties (see later).

4.8 Named Fields and Properties

Named fields and properties are optional components for specifying an attribute-object. We allow them to be specified in any order (languages may choose to impose tighter constraints). Therefore, the serialized format defines each named field or property by recording a quad giving {FieldOrProperty, type, name, value}, in the obvious way.

We include Field-or-Property, as well as type, so that we can, at instantiation time, perform overload resolution of the named field or property.

We start with a 2-byte count specifying the total number of named fields and properties to follow. This count must always be supplied – if there are none, the count must be zero.

Whether each item is a field or property is specified with the one-byte tag SERIALIZATION_TYPE_FIELD or SERIALIZATION_TYPE_PROPERTY. The field or property name is encoded as a string – compacted byte-count plus UTF8 sequence. The type is encoded as its corresponding SERIALIZATION_TYPE_ member. Its value is similarly encoded exactly as described already – name (string) and value. The name is encoded as a String, defined above (compacted byte-count, followed by a UTF8 sequence). The value too is encoded exactly as defined before.

4.9 Named Field – Example

```
 [Attrib("Friday", Whim=42)] class Ex3 { . . . }
```

The arguments to the Attrib constructor would be encoded as follows:

```
0100 06 467269646179 0100 53 51 04 5768696D 08 2A000000
```

We start with the Prolog – a 2-byte value of 1. Next comes the positional argument – the String "Friday". Next we have a 2-byte count with value 1, telling us there is one named item to follow. Next comes the constant SERIALIZATION_TYPE_FIELD. Next its type, SERIALIZATION_TYPE_VARIANT. Next its name ("Whim"). Finally its actual type (SERIALIZATION_TYPE_I4) and its 4-byte value of 42 (2A hex).

4.10 General Case

This spec documents a subset of the general NGWS binary serialization format, as an aid for compilers who wish to serialize objects 'by-hand'. So, the format for saving attribute-objects is piece-wise identical to the format used for serializing VOS objects. That's to say, if you look at the binary layout for any constructor argument, it is identical (ok, we've included a couple of optimizations) to how it would look in a binary-serialized stream.

[The general-case serialized object includes extra fields. For example, each serialized object is assigned an ObjectID to support references to it from other objects in the graph. This is omitted for Strings in the serialized constructor arguments]

So, what if an attribute-class actually defined an argument of some user-defined class, Quix? How does the general serialized format look? The answer is, as you would expect, that the Quix object, as an argument to the attribute-class constructor, is persisted into the metadata, in the same format as if it had been instantiated as a regular VOS object and serialized.

Suppose the following example:

```
[attribute(VOSElementtype.All)]
public class Attrib {
    public readonly string Name;
    public variant Whim;
    public int Depth;
    public Attrib(string n, Quix q) { . . . }
}
// Instantiate and setup aQuix
[Attrib("Friday", aQuix)] class Ex5 { . . . }
```

The arguments to the Attrib constructor would be encoded as follows:

```
0100 06 467269646179 0100 11 . . . .
```

We start with the Prolog – a 2-byte with value 1. Next comes the first argument – the String "Friday". Next we have the serialized aQuix – SERIALIZATION_TYPE_CLASS, then the binary blob for its field values.

Serializing arbitrary object graphs is clearly more complex than the subset of cases we have described above. Whilst the general case is addressed in the Spec for Binary Format Serialization, we will call out one aspect, that could arise in this last example. The Attrib constructor expects a Quix object; however, at compile time, it could be given an instance of a class derived from Quix. In this case, we need to include instance-type information, rather than just declaration-type information. [In fact, if you look back at Variant examples, this same two-level typing occurs there too]

Note: saving attribute-objects by serializing the object is **not** supported in the first release of the runtime

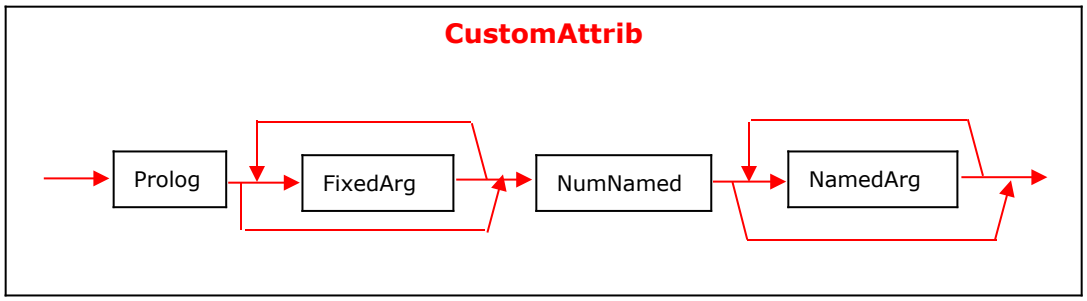
4.11 SERIALIZATION_TYPE_ enum

```
SERIALIZATION_TYPE_BOOLEAN    = ELEMENT_TYPE_BOOLEAN
SERIALIZATION_TYPE_CHAR       = ELEMENT_TYPE_CHAR
SERIALIZATION_TYPE_I1         = ELEMENT_TYPE_I1
SERIALIZATION_TYPE_U1         = ELEMENT_TYPE_U1
SERIALIZATION_TYPE_I2         = ELEMENT_TYPE_I2
SERIALIZATION_TYPE_U2         = ELEMENT_TYPE_U2
SERIALIZATION_TYPE_I4         = ELEMENT_TYPE_I4
SERIALIZATION_TYPE_U4         = ELEMENT_TYPE_U4
SERIALIZATION_TYPE_I8         = ELEMENT_TYPE_I8
SERIALIZATION_TYPE_U8         = ELEMENT_TYPE_U8
SERIALIZATION_TYPE_R4         = ELEMENT_TYPE_R4
SERIALIZATION_TYPE_R8         = ELEMENT_TYPE_R8
SERIALIZATION_TYPE_STRING     = ELEMENT_TYPE_STRING
SERIALIZATION_TYPE_VALUETYPE  = ELEMENT_TYPE_VALUETYPE
SERIALIZATION_TYPE_CLASS      = ELEMENT_TYPE_CLASS
SERIALIZATION_TYPE_SZARRAY    = ELEMENT_TYPE_SZARRAY
SERIALIZATION_TYPE_ARRAY      = ELEMENT_TYPE_ARRAY
SERIALIZATION_TYPE_TYPE       = 0x50
SERIALIZATION_TYPE_VARIANT    = 0x51
SERIALIZATION_TYPE_FIELD      = 0x53
SERIALIZATION_TYPE_PROPERTY   = 0x54
SERIALIZATION_TYPE_ENUM       = 0x55
```

5 CustomAttributes – Syntax

This section summarizes the syntax charts for defining CustomAttribute objects, detailed in section 4. It makes no sense unless you have read that section (and even then).

A valid Custom Attribute has the following syntax chart:

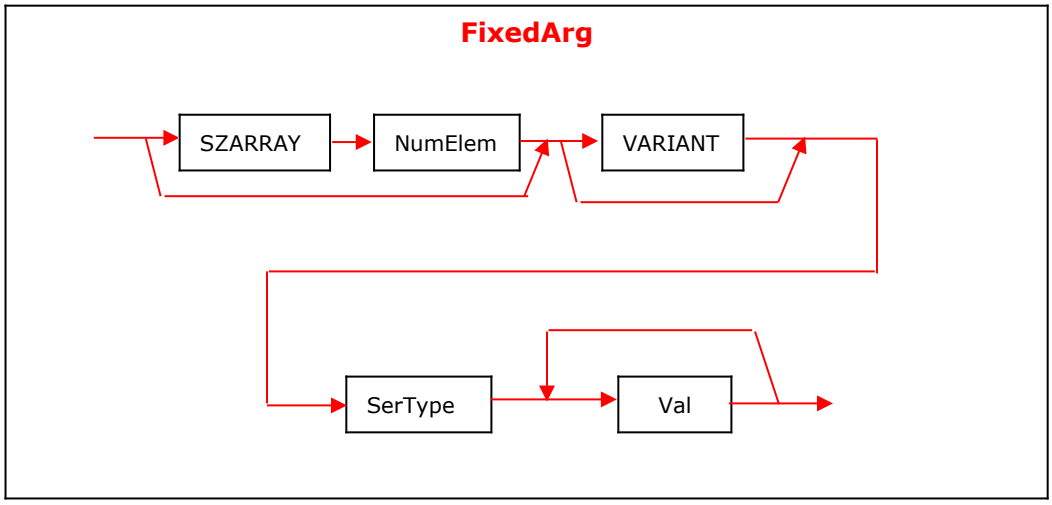


All binary values are stored in little-endian format (except PackedLen items – used only as counts for the number of bytes to follow in a Utf8 string)

CustomAttrib starts with a *Prolog* – a U2, with value 0x0001

Next comes a description of the fixed arguments for the constructor method. Their number and type is found by examining that constructor method's MethodDef; this info is *not* repeated in the *CustomAttrib* itself. As the syntax chart shows, there can be zero or more *FixedArg*s. (note that VARARG constructor methods are not allowed in the definition of Custom Attributes)

Next is a description of the optional "named" fields and properties. This starts with *NumNamed* – a U2 giving the number of "named" properties or fields that follow. Note that *NumNamed* must always be present. If its value is zero, there are no "named" properties or fields to follow (and of course, in this case, the CustomAttrib must end immediately after *NumNamed*) In the case where *NumNamed* is non-zero, it is followed by *NumNamed* repeats of *NamedArgs*



SZARRAY is the single byte SERIALIZATION_TYPE_SZARRAY

NumElem is a U4 specifying the number of elements in the SZARRAY

VARIANT is the single byte SERIALIZATION_TYPE_VARIANT

(Note, as the syntax chart shows, that each FixedArg can be an SZARRAY of *SerType-Val*s, or an SZARRAY of VARIANTs of *SerType-Val*s or just a regular *SerType-Val*)

A *SerType* is defined as one of:

BOOLEAN | CHAR | I1 | U1 | I2 | U2 | I4 | U4 | I8 | U8 | R4 | R8

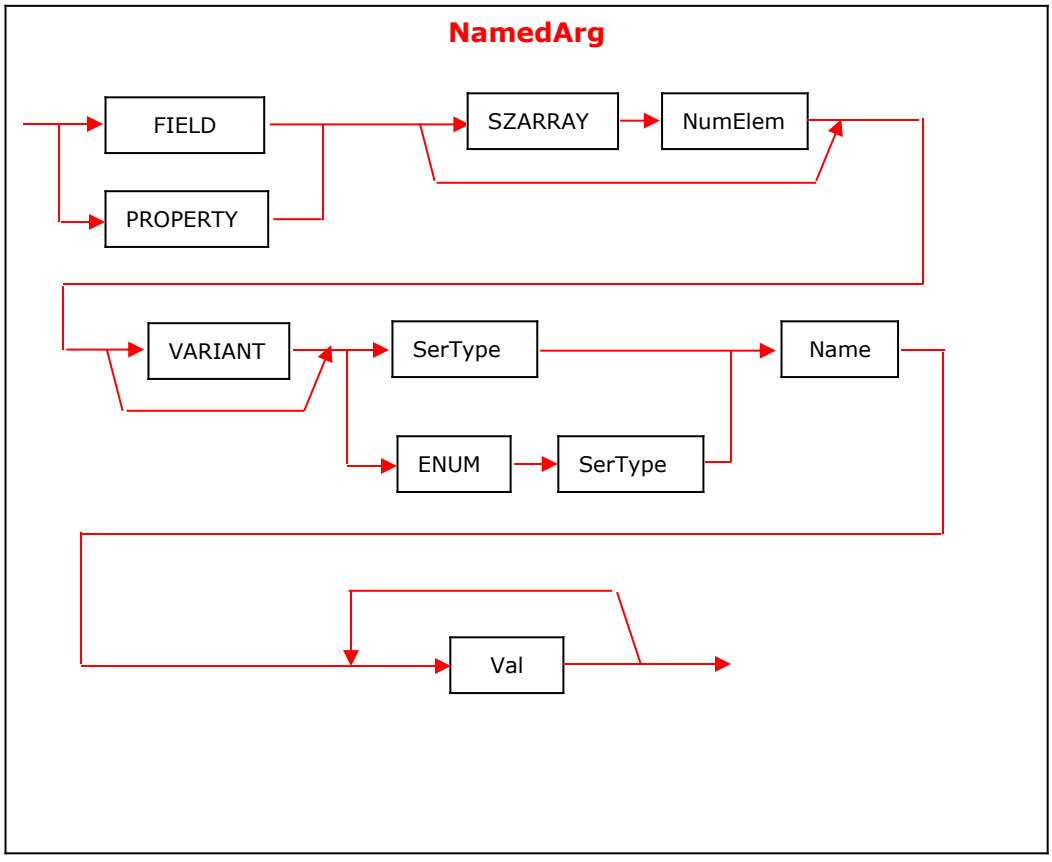
| STRING | TYPE

where we have omitted the SERIALIZATION_TYPE_ prefix for briefness. So, for example, "STRING" is short for SERIALIZATION_TYPE_STRING.

Val is the binary representation for each of those *SerType*s. So, BOOLEAN is a U1 with value 0 (false) or 1 (true); CHAR is a two-byte unicode character; I1 thru R8 all have their obvious meaning (stored in the same byte order as held in a little-endian machine memory, such as an x86); STRING and TYPE are a little more complicated, as follows:

For STRING, the following *Val* item is a PackedLen value for the number of bytes in the string, followed by the string, encoded in Utf8.

The *Val* item following a TYPE is the same as for STRING – that's because we persist a TYPE as its stringified type name (including the defining assembly name)



FIELD is the single byte SERIALIZATION_TYPE_FIELD

PROPERTY is the single byte SERIALIZATION_TYPE_PROPERTY

SZARRAY is the single byte SERIALIZATION_TYPE_SZARRAY

NumElem is a U4 specifying the number of elements in the SZARRAY

VARIANT is the single byte SERIALIZATION_TYPE_VARIANT

SerType was defined above. This represents the simple type of the FIELD or PROPERTY; if the previous VARIANT item were included, then it represents the base type of that VARIANT

ENUM is the single byte SERIALIZATION_TYPE_ENUM. This is followed by a *SerType* giving the base type of this enum. (this choice allows for an attribute class that defines, for example, two fields with the same name – one with type I4, the other with type enum, represented as I4s)

Name is the name of this field or name – just its simple name within the attribute class (which we know, via the metadata token for the constructor method). It is encoded like all other names – PackedLen byte count of the follow-on Utf8 string.

Val was defined above. This is repeated *NumElem* times

---

6 Marshalling Descriptor

A Marshalling Descriptor is like a signature – it's a blob of binary data. It describes how a field or parameter (which, as usual, covers the method return, as parameter number 0) should be marshalled when calling to or from unmanaged coded via PInvoke dispatch or IJW ("It Just Works") thunking.

The blob has the following format –

MarshalSpec **:==**
NativeInstrinsic
| CUSTOMMARSHALLER Guid UnmanagedType ManagedType Cookie
| FIXEDARRAY NumElem ArrayElemType
| SAFEARRAY SafeArrayElemType
| ARRAY ArrayElemType ParamNum ElemMult NumElem

NativeInstrinsic :==
BOOLEAN | I1 | U1 | I2 | U2 | I4 | U4 | I8 | U8 | R4 | R8
| BSTR | LPSTR | LPWSTR | LPTSTR | FIXEDSYSSTRING | STRUCT
| INTF |FIXEDARRAY | INT | UINT | BYVALSTR | ANSIBSTR | TBSTR
| VARIANTBOOL | FUNC | LPVOID | ASANY | R | LPSTRUCT | ERROR | MAX

For compactness, we have omitted the NATIVE_TYPE_ prefixes in the above lists. So, for example, "ARRAY" is shorthand for NATIVE_TYPE_ARRAY (see the CorNativeType enum defined in CorHdr.h) Note that *NativeIntrinsic* excludes those elements of the CorNativeType enum commented as "deprecated"

Guid is a counted-Utf8 string – eg "{90883F05-3D28-11D2-8F17-00A0C9A6186D}" – it must include leading **{** and trailing **}** and be exactly 38 characters long

UnmanagedType is a counted-Utf8 string – eg "Point"

ManagedType is a counted-Utf8 string – eg "System.Util.MyGeometry" – it must be the fully-qualified name (namespace and name) of a managed Type defined within the current Assembly (that Type must implement ICustomMarshaller, and provides a "to" and "from" marshalling method)

Cookie is a counted-Utf8 string – eg "123" – an empty string is allowed

NumElem is an integer that tells us how many elements are in the array

ArrayElemType :==
NativeInstrinsic | BOOLEAN | I1 | U1 | I2 | U2
| I4 | U4 | I8 | U8 | R4 | R8 | BSTR | LPSTR | LPWSTR | LPTSTR
| FIXEDSYSSTRING | STRUCT | INTF | INT | UINT | BYVALSTR
| ANSIBSTR | TBSTR | VARIANTBOOL | FUNC | LPVOID | ASANY
| R | LPSTRUCT | ERROR | MAX
The value MAX is used to indicate "no info"

SafeArrayElemType :== I2 | I4 | R4 | R8 | CY | DATE | BSTR | DISPATCH |
| ERROR | BOOL | VARIANT | UNKNOWN | DECIMAL | I1 | UI1 | UI2
| UI4 | INT | UINT
where each is prefixed by VT_. Note that these VT_xxx form a subset of the standard OLE constants (defined, for example, in the file WType.h that ships with Visual studio, installed to the default directory "Program Files\Microsoft Visual Studion\VC98\Include")

ParamNum is an integer, which says which parameter in the method call provides the number of elements in the array – see below

ElemMult is an integer (says by what factor to multiply – see below)

For example, in the method declaration:

Foo (int ar1[], int size1, byte ar2[], int size2)

The *ar1* parameter might own a row in the *FieldMarshal* table, which indexes a *MarshalSpec* in the Blob heap with the format:

ARRAY MAX 2 1 0

This says the parameter is marshalled to a NATIVE_TYPE_ARRAY. There is no additional info about the type of each element (signified by that NATIVE_TYPE_MAX). The value of *ParamNum* is 2, which tells us that parameter number 2 in the method (the one called "size1") will tell us the number of elements in the actual array – let's suppose its value on a particular call were 42. The value of *ElemMult* is 1. The value of *NumElem* is 0. The calculated total size, in bytes, of the array is given by the formula:

if *ParamNum* == 0
SizeInBytes = *NumElem* * sizeof (elem)
else
SizeInBytes = (@P*aramNum* * *ElemMult* + *NumElem*) * sizeof (elem)
endif

We have used the syntax "@*ParamNum*" to denote the value passed in for parameter number *ParamNum* – it would be 42 in this example. The size of each element is calculated from the metadata for the *ar1* parameter in *Foo*'s signature – an ELEMENT_TYPE_I4 of size 4 bytes.

Note that, just as in signature blobs, every simple scalar, such as integers or Utf8 byte-counts, are stored in compressed format, using the *CorSigCompressData* helper routines (see section 3 for details)