Common Language Infrastructure (CLI)

Partition II:
Metadata Definition and Semantics

Table of contents

1 Introduction

This specification provides the normative description of the metadata: its physical layout (as a file format), its logical contents (as a set of tables and their relationships), and its semantics (as seen from a hypothetical assembler, *ilasm*).

2 Overview

This partition focuses on the semantics and the structure of metadata. The semantics of metadata, which dictate much of the operation of the VES, are described using the syntax of ILAsm, an assembly language for CIL. The ILAsm syntax itself (contained in clauses 5 through 21) is considered a normative part of this International Standard. (An implementation of an assembler for ILAsm is described in Partition VI.) The structure (both logical and physical) is covered in clauses 22 through 25.

[*Rationale:* An assembly language is really just syntax for specifying the metadata in a file, and the CIL instructions in that file. Specifying ILAsm provides a means of interchanging programs written directly for the CLI without the use of a higher-level language; it also provides a convenient way to express examples.

The semantics of the metadata can also be described independently of the actual format in which the metadata is stored. This point is important because the storage format as specified in clauses 22 through 25 is engineered to be efficient for both storage space and access time, but this comes at the cost of the simplicity desirable for describing its semantics. *end rationale]*

3 Validation and verification

Validation refers to the application of a set of tests on any file to check that the file's format, metadata, and CIL are self-consistent. These tests are intended to ensure that the file conforms to the normative requirements of this specification. When a conforming implementation of the CLI is presented with a non-conforming file, the behavior is unspecified.

Verification refers to the checking of both CIL and its related metadata to ensure that the CIL code sequences do not permit any access to memory outside the program's logical address space. In conjunction with the validation tests, verification ensures that the program cannot access memory or other resources to which it is not granted access.

Partition III specifies the rules for both correct and verifiable use of CIL instructions. Partition III also provides an informative description of rules for validating the internal consistency of metadata (the rules follow, albeit indirectly, from the specification in this Partition); it also contains a normative description of the verification algorithm. A mathematical proof of soundness of the underlying type system is possible, and provides the basis for the verification requirements. Aside from these rules, this standard leaves as unspecified:

- The time at which (if ever) such an algorithm should be performed.
- What a conforming implementation should do in the event of a verification failure.

The following figure makes this relationship clearer (see next paragraph for a description):

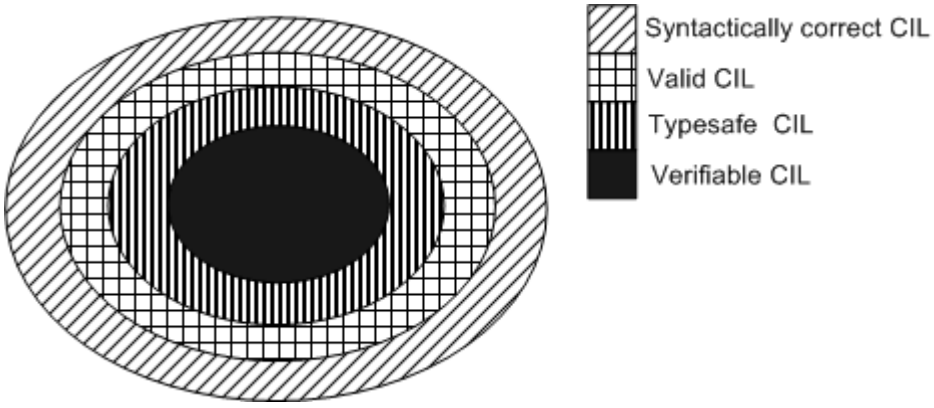


Figure 1: Relationship between correct and verifiable CIL

In the above figure, the outer circle contains all code permitted by the ILAsm syntax. The next inner circle represents all code that is correct CIL. The striped inner circle represents all type-safe code. Finally, the black innermost circle contains all code that is verifiable. (The difference between type-safe code and verifiable code is one of *provability*: code which passes the VES verification algorithm is, by-definition, *verifiable*; but that simple algorithm rejects certain code, even though a deeper analysis would reveal it as genuinely type-safe). Note that even if a program follows the syntax described in Partition VI, the code might still not be valid, because valid code shall adhere to restrictions presented in this Partition and in Partition III.

The verification process is very stringent. There are many programs that will pass validation, but will fail verification. The VES cannot guarantee that these programs do not access memory or resources to which they are not granted access. Nonetheless, they might have been correctly constructed so that they do not access these resources. It is thus a matter of trust, rather than mathematical proof, whether it is safe to run these programs. Ordinarily, a conforming implementation of the CLI can allow *unverifiable code* (valid code that does not pass verification) to be executed, although this can be subject to administrative trust controls that are not part of this standard. A conforming implementation of the CLI shall allow the execution of verifiable code, although this can be subject to additional implementation-specified trust controls.

4 Introductory examples

This clause and its subclauses contain only informative text.

4.1 "Hello world!"

To get the general feel of ILAsm, consider the following simple example, which prints the well known "Hello world!" salutation. The salutation is written by calling `WriteLine`, a static method found in the class `System.Console` that is part of the standard assembly `mscorlib` (see Partition IV). [*Example:*

```
.assembly extern mscorlib {}
.assembly hello {}
.method static public void main() cil managed

{ .entrypoint
  .maxstack 1
  ldstr "Hello world!"
  call void [mscorlib]System.Console::WriteLine(class System.String)
  ret
}
```

end example]

The `.assembly extern` declaration references an external assembly, `mscorlib`, which contains the definition of `System.Console`. The `.assembly` declaration in the second line declares the name of the assembly for this program. (Assemblies are the deployment unit for executable content for the CLI.) The `.method` declaration defines the global method `main`, the body of which follows, enclosed in braces. The first line in the body indicates that this method is the entry point for the assembly (`.entrypoint`), and the second line in the body specifies that it requires at most one stack slot (`.maxstack`).

Method `main` contains only three instructions: ldstr, call, and ret. The ldstr instruction pushes the string constant `"Hello world!"` onto the stack and the call instruction invokes `System.Console::WriteLine`, passing the string as its only argument. (Note that string literals in CIL are instances of the standard class `System.String`.) As shown, call instructions shall include the full signature of the called method. Finally, the last instruction, `ret`, returns from `main`.

4.2 Other examples

This Partition contains integrated examples for most features of the CLI metadata. Many subclauses conclude with an example showing a typical use of some feature. All these examples are written using the ILAsm assembly language. In addition, Partition VI contains a longer example of a program written in the ILAsm assembly language. All examples are, of course, informative only.

End informative text

5 General syntax

This clause describes aspects of the ILAsm syntax that are common to many parts of the grammar.

5.1 General syntax notation

This partition uses a modified form of the BNF syntax notation. The following is a brief summary of this notation.

Terminals are written in a constant-width font (e.g., `.assembly`, `extern`, and `float64`); however, terminals consisting solely of punctuation characters are enclosed in single quotes (e.g., `':'`, `'['`, and `'('`). The names of syntax categories are capitalized and italicized (e.g. *ClassDecl*) and shall be replaced by actual instances of the category. Items placed in [ ] brackets (e.g., [*Filename*] and [*Float*]), are optional, and any item followed by * (e.g., *HexByte** and [`'.'` *Id*]*) can appear zero or more times. The character "|" means that the items on either side of it are acceptable (e.g., `true` | `false`). The options are sorted in alphabetical order (to be more specific: in ASCII order, and case-insensitive). If a rule starts with an optional term, the optional term is not considered for sorting purposes.

ILAsm is a case-sensitive language. All terminals shall be used with the same case as specified in this clause.

[*Example:* A grammar such as

Top ::= *Int32* | `float` *Float* | `floats` [*Float* [`','` *Float*]*] | `else` *QSTRING*

would consider all of the following to be valid:

```
12
float 3
float -4.3e7
floats
floats 2.4
floats 2.4, 3.7
else "Something \t weird"
```

but all of the following to be invalid:

```
else 3
3, 4
float 4.3, 2.4
float else
stuff
```

end example]

5.2 Basic syntax categories

These categories are used to describe syntactic constraints on the input intended to convey logical restrictions on the information encoded in the metadata.

Int32 is either a decimal number or "0x" followed by a hexadecimal number, and shall be represented in 32 bits. [*Note*: ILAsm has no concept of 8- or 16-bit integer constants. Instead, situations requiring such a constant (such as int8(...) and int16(...) in §16.2) accept an *Int32* instead, and use only the least-significant bytes. *end note*]

Int64 is either a decimal number or "0x" followed by a hexadecimal number, and shall be represented in 64 bits.

HexByte is a hexadecimal number that is a pair of characters from the set 0–9, a–f, and A–F.

RealNumber is any syntactic representation for a floating-point number that is distinct from that for all other syntax categories. In this partition, a period (.) is used to separate the integer and fractional parts, and "e" or "E" separates the mantissa from the exponent. Either of the period or the mantissa separator (but not both) can be omitted.

[*Note:* A complete assembler might also provide syntax for infinities and NaNs. *end note*]

QSTRING is a string surrounded by double quote (″) marks. Within the quoted string the character "\" can be used as an escape character, with "\t" representing a tab character, "\n" representing a newline character, and "\" followed by three octal digits representing a byte with that value. The "+" operator can be used to concatenate string literals. This way, a long string can be broken across multiple lines by using "+" and a new string on each line. An alternative is to use "\" as the last character in a line, in which case, that character and the line break following it are not entered into the generated string. Any white space characters (space, line-feed, carriage-return, and tab) between the "\" and the first non-white space character on the next line are ignored. [*Note:* To include a double quote character in a *QSTRING*, use an octal escape sequence. *end note*]

[*Example:* The following result in strings that are equivalent to `"Hello World from CIL!"`:

```
ldstr "Hello " + "World " +
"from CIL!"
```

and

```
ldstr "Hello World\
       \040from CIL!"
```

end example]

[*Note:* A complete assembler will need to deal with the full set of issues required to support Unicode encodings, see Partition I (especially CLS Rule 4). *end note*]

SQSTRING is just like *QSTRING* except that the former uses single quote (′) marks instead of double quote. [*Note:* To include a single quote character in an *SQSTRING*, use an octal escape sequence. *end note*]

ID is a contiguous string of characters which starts with either an alphabetic character (A–Z, a–z) or one of "_", "$", "@", "`" (grave accent), or "?", and is followed by any number of alphanumeric characters (A–Z, a–z, 0–9) or the characters "_", "$", "@", "`" (grave accent), and "?". An *ID* is used in only two ways:

- As a label of a CIL instruction (§5.4).
- As an *Id* (§5.3).

5.3 Identifiers

Identifiers are used to name entities. Simple identifiers are equivalent to an *ID*. However, the ILAsm syntax allows the use of any identifier that can be formed using the Unicode character set (see Partition I). To achieve this, an identifier shall be placed within single quotation marks. This is summarized in the following grammar.

Id ::=
ID
| *SQSTRING*

A keyword shall only be used as an identifier if that keyword appears in single quotes (see Partition VI for a list of all keywords).

Several *Id*s can be combined to form a larger *Id*, by separating adjacent pairs with a dot (`.`). An *Id* formed in this way is called a *DottedName*.

DottedName ::= *Id* [`.` *Id*]*

[*Rationale: DottedName* is provided for convenience, since "." can be included in an *Id* using the *SQSTRING* syntax. *DottedName* is used in the grammar where "`.`" is considered a common character (e.g., in fully qualified type names) *end rationale*]

[*Example:* The following are simple identifiers:

```
A  Test   $Test   @Foo?   ?_X_   MyType`1
```

The following are identifiers in single quotes:

```
'Weird Identifier'   'Odd\102Char'   'Embedded\nReturn'
```

The following are dotted names:

```
System.Console  'My Project'.'My Component'.'My Name'  System.IComparable`1
```

end example]

5.4 Labels and lists of labels

Labels are provided as a programming convenience; they represent a number that is encoded in the metadata. The value represented by a label is typically an offset in bytes from the beginning of the current method, although the precise encoding differs depending on where in the logical metadata structure or CIL stream the label occurs. For details of how labels are encoded in the metadata, see clauses 22 through 25; for their encoding in CIL instructions see Partition III.

A simple label is a special name that represents an address. Syntactically, a label is equivalent to an *Id*. Thus, labels can be single quoted and can contain Unicode characters.

A list of labels is comma separated, and can be any combination of simple labels.

LabelOrOffset ::= *Id*
Labels ::= *LabelOrOffset* [',' *LabelOrOffset*]*

[*Note:* In a real assembler the syntax for *LabelOrOffset* might allow the direct specification of a number rather than requiring symbolic labels. e*nd note*]

ILAsm distinguishes between two kinds of labels: code labels and data labels. Code labels are followed by a colon (":") and represent the address of an instruction to be executed. Code labels appear before an instruction and they represent the address of the instruction that immediately follows the label. A particular code label name shall not be declared more than once in a method.

In contrast to code labels, data labels specify the location of a piece of data and do not include the colon character. A data label shall not be used as a code label, and a code label shall not be used as a data label. A particular data label name shall not be declared more than once in a module.

CodeLabel ::= *Id* ':'
DataLabel ::= *Id*

[*Example:* The following defines a code label, `ldstr_label`, that represents the address of the ldstr instruction:

```
ldstr_label: ldstr  "A label"
```

end example]

5.5 Lists of hex bytes

A list of bytes consists simply of one or more hexbytes.

Bytes ::= *HexByte* [*HexByte**]

5.6 Floating-point numbers

There are two different ways to specify a floating-point number:

1. As a *RealNumber*.
2. By using the keyword `float32` or `float64`, followed by an integer in parentheses, where the integer value is the binary representation of the desired floating-point number. For example, `float32(1)` results in the 4-byte value 1.401298E-45, while `float64(1)` results in the 8-byte value 4.94065645841247E-324.

Float32 ::=

RealNumber
| `float32` `'('` *Int32* `')'`
Float64 `::=`
RealNumber
| `float64` `'('` *Int64* `')'`

[*Example:*

```
5.5
1.1e10
float64(128)  // note: this results in an 8-byte value whose bits are the same
              // as those for the integer value 128.
```

end example]

5.7 Source line information

The metadata does not encode information about the lexical scope of variables or the mapping from source line numbers to CIL instructions. Nonetheless, it is useful to specify an assembler syntax for providing this information for use in creating alternate encodings of the information.

`.line` takes a line number, optionally followed by a column number (preceded by a colon), optionally followed by a single-quoted string that specifies the name of the file to which the line number is referring:

ExternSourceDecl `::=` `.line` *Int32* [`':'` *Int32*] [*SQSTRING*]

5.8 File names

Some grammar elements require that a file name be supplied. A file name is like any other name where "." is considered a normal constituent character. The specific syntax for file names follows the specifications of the underlying operating system.

Filename `::=`	**Clause**
DottedName	5.3

5.9 Attributes and metadata

Attributes of types and their members attach descriptive information to their definition. The most common attributes are predefined and have a specific encoding in the metadata associated with them (§23). In addition, the metadata provides a way of attaching user-defined attributes to metadata, using several different encodings.

From a syntactic point of view, there are several ways for specifying attributes in ILAsm:

- Using special syntax built into ILAsm. For example, the keyword `private` in a *ClassAttr* specifies that the visibility attribute on a type shall be set to allow access only within the defining assembly.
- Using a general-purpose syntax in ILAsm. The non-terminal *CustomDecl* describes this grammar (§21). For some attributes, called *pseudo-custom attributes*, this grammar actually results in setting special encodings within the metadata (§21.2.1).
- Security attributes are treated specially. There is special syntax in ILAsm that allows the XML representing security attributes to be described directly (§20). While all other attributes defined either in the standard library or by user-provided extension are encoded in the metadata using one common mechanism described in §22.10, security attributes (distinguished by the fact that they inherit, directly or indirectly from `System.Security.Permissions.SecurityAttribute`, see Partition IV) shall be encoded as described in §22.11.

5.10 *ilasm* source files

An input to *ilasm* is a sequence of top-level declarations, defined as follows:

ILFile ::=	**Reference**
*Decl**	5.10

The complete grammar for a top-level declaration is shown below. The reference subclauses contain details of the corresponding productions of this grammar. These productions begin with a name having a '.' prefix. Such a name is referred to as a *directive*.

Decl ::=	**Reference**
.assembly *DottedName* '{' *AsmDecl** '}'	6.2
| .assembly extern *DottedName* '{' *AsmRefDecl** '}'	6.3
| .class *ClassHeader* '{' *ClassMember** '}'	10
| .class extern *ExportAttr DottedName* '{' *ExternClassDecl** '}'	6.7
| .corflags *Int32*	6.2
| .custom *CustomDecl*	21
| .data *DataDecl*	16.3.1
| .field *FieldDecl*	16
| .file [nometadata] *Filename* .hash '=' '(' *Bytes* ')' [.entrypoint]	6.2.3
| .method *MethodHeader* '{' *MethodBodyItem** '}'	15
| .module [*Filename*]	6.4
| .module extern *Filename*	6.5
| .mresource [public | private] *DottedName* '{' *ManResDecl** '}'	6.2.2
| .subsystem *Int32*	6.2
| .vtfixup *VTFixupDecl*	15.5.1
| *ExternSourceDecl*	5.7
| *SecurityDecl*	20

6 Assemblies, manifests and modules

Assemblies and modules are grouping constructs, each playing a different role in the CLI.

An *assembly* is a set of one or more files deployed as a unit. An assembly always contains a *manifest* that specifies (§6.1):

- Version, name, culture, and security requirements for the assembly.
- Which other files, if any, belong to the assembly, along with a cryptographic hash of each file. The manifest itself resides in the metadata part of a file, and that file is always part of the assembly.
- The types defined in other files of the assembly that are to be exported from the assembly. Types defined in the same file as the manifest are exported based on attributes of the type itself.
- Optionally, a digital signature for the manifest itself, and the public key used to compute it.

A *module* is a single file containing executable content in the format specified here. If the module contains a manifest then it also specifies the modules (including itself) that constitute the assembly. An assembly shall contain only one manifest amongst all its constituent files. For an assembly that is to be executed (rather than simply being dynamically loaded) the manifest shall reside in the module that contains the entry point.

While some programming languages introduce the concept of a *namespace*, the only support in the CLI for this concept is as a metadata encoding technique. Type names are always specified by their full name relative to the assembly in which they are defined.

6.1 Overview of modules, assemblies, and files

This subclause contains informative text only.

Consider the following figure:

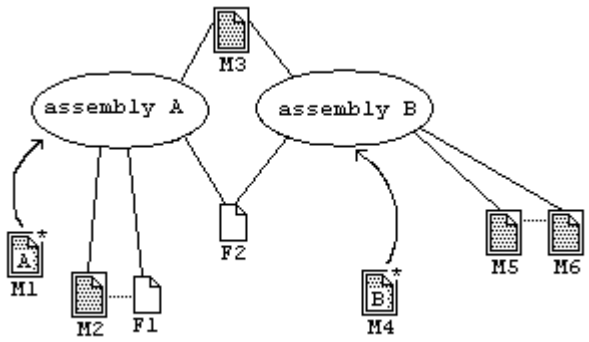


Figure 2: References to Modules and Files

Eight files are shown, each with its name written below it. The six files that each declare a module have an additional border around them, and their names begin with M. The other two files have a name beginning with F. These files can be resource files (such as bitmaps) or other files that do not contain CIL code.

Files M1 and M4 declare an assembly in addition to the module declaration, namely assemblies A and B, respectively. The assembly declaration in M1 and M4 references other modules, shown with straight lines. For example, assembly A references M2 and M3, and assembly B references M3 and M5. Thus, both assemblies reference M3.

Usually, a module belongs only to one assembly, but it is possible to share it across assemblies. When assembly A is loaded at runtime, an instance of M3 will be loaded for it. When assembly B is loaded into the same application domain, possibly simultaneously with assembly A, M3 will be shared for both assemblies. Both assemblies also reference F2, for which similar rules apply.

The module M2 references F1, shown by dotted lines. As a consequence, F1 will be loaded as part of assembly A, when A is executed. Thus, the file reference shall also appear with the assembly declaration. Similarly, M5 references another module, M6, which becomes part of B when B is executed. It follows that assembly B shall also have a module reference to M6.

End informative text

6.2 Defining an assembly

An assembly is specified as a module that contains a manifest in the metadata; see §22.2. The information for the manifest is created from the following portions of the grammar:

Decl ::=	**Clause**
.assembly *DottedName* '{' *AsmDecl** '}'	6.2
| .assembly extern *DottedName* '{' *AsmRefDecl** '}'	6.3
| .corflags *Int32*	6.2
| .file [nometadata] *Filename* .hash '=' '(' *Bytes* ')' [.entrypoint]	6.2.3
| .module extern *Filename*	6.5
| .mresource [public|private] *DottedName* '{' *ManResDecl** '}'	6.2.2
| .subsystem *Int32*	6.2
| …	

The .assembly directive declares the manifest and specifies to which assembly the current module belongs. A module shall contain at most one .assembly directive. The *DottedName* specifies the name of the assembly. [*Note*: The standard library assemblies are described in Partition IV. *end note*])

[*Note:* Since some platforms treat names in a case-insensitive manner, two assemblies that have names that differ only in case should not be declared. *end note*]

The .corflags directive sets a field in the CLI header of the output PE file (see §25.3.3.1). A conforming implementation of the CLI shall expect this field's value to be 1. For backwards compatibility, the three least-significant bits are reserved. Future versions of this standard might provide definitions for values between 8 and 65,535. Experimental and non-standard uses should thus use values greater than 65,535.

The .subsystem directive is used only when the assembly is executed directly (as opposed to its being used as a library for another program). This directive specifies the kind of application environment required for the program, by storing the specified value in the PE file header (see §25.2.2). While any 32-bit integer value can be supplied, a conforming implementation of the CLI need only respect the following two values:

- If the value is 2, the program should be run using whatever conventions are appropriate for an application that has a graphical user interface.
- If the value is 3, the program should be run using whatever conventions are appropriate for an application that has a direct console attached.

[*Example:*

```
.assembly CountDown
{ .hash algorithm 32772
  .ver 1:0:0:0
}
.file Counter.dll .hash = (BA D9 7D 77 31 1C 85 4C 26 9C 49 E7
02 BE E7 52 3A CB 17 AF)
```

end example]

6.2.1 Information about the assembly (*AsmDecl*)

The following grammar shows the information that can be specified about an assembly:

AsmDecl ::=	**Description**	**Clause**
.custom *CustomDecl*	Custom attributes	21
| .hash algorithm *Int32*	Hash algorithm used in the .file directive	6.2.1.1
| .culture *QSTRING*	Culture for which this assembly is built	6.2.1.2
| .publickey '=' '(' *Bytes* ')'	The originator's public key.	6.2.1.3
| .ver *Int32* ':' *Int32* ':' *Int32* ':' *Int32*	Major version, minor version, build, and revision	6.2.1.4
| *SecurityDecl*	Permissions needed, desired, or prohibited	20

6.2.1.1 Hash algorithm

AsmDecl ::= .hash algorithm *Int32* | …

When an assembly consists of more than one file (see §6.2.3), the manifest for the assembly specifies both the name and cryptographic hash of the contents of each file other than its own. The algorithm used to compute the hash can be specified, and shall be the same for all files included in the assembly. All values are reserved for future use, and conforming implementations of the CLI shall use the SHA-1 (see FIPS 180-1 in Partition I, 3) hash function and shall specify this algorithm by using a value of 32772 (0x8004).

[*Rationale:* SHA-1 was chosen as the best widely available technology at the time of standardization (see *Partition I*). A single algorithm was chosen since all conforming implementations of the CLI would be required to implement all algorithms to ensure portability of executable images.*end rationale*]

6.2.1.2 Culture

AsmDecl ::= .culture *QSTRING* | …

When present, this indicates that the assembly has been customized for a specific culture. The strings that shall be used here are those specified in Partition IV as acceptable with the class `System.Globalization.CultureInfo`. When used for comparison between an assembly reference and an assembly definition these strings shall be compared in a case-insensitive manner. (See §23.1.3.)

[*Note:* The culture names follow the IETF RFC1766 names. The format is "<language>-<country/region>", where <language> is a lowercase two-letter code in ISO 639-1. <country/region> is an uppercase two-letter code in ISO 3166. *end note*]

6.2.1.3 Originator's public key

AsmDecl ::= .publickey '=' '(' *Bytes* ')' | …

The CLI metadata allows the producer of an assembly to compute a cryptographic hash of that assembly (using the SHA-1 hash function) and then to encrypt it using the RSA algorithm (see Partition I) and a public/private key pair of the producer's choosing. The results of this (an "SHA-1/RSA digital signature") can then be stored in the metadata (§25.3.3) along with the public part of the key pair required by the RSA algorithm. The .publickey directive is used to specify the public key that was used to compute the signature. To calculate the hash, the signature is zeroed, the hash calculated, and then the result is stored into the signature.

All of the assemblies in the Standard Library (see Partition IV) use the public key 00 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00. This key is known as the *Standard Public Key* in this standard.

A reference to an assembly (§6.3) captures some of this information at compile time. At runtime, the information contained in the assembly reference can be combined with the information from the manifest of the assembly located at runtime to ensure that the same private key was used to create both the assembly seen when the reference was created (compile time) and when it is resolved (runtime).

The Strong Name (SN) signing process uses standard hash and cipher algorithms for Strong name signing. An SHA-1 hash over most of the PE file is generated. That hash value is RSA-signed with the SN private key. For verification purposes the public key is stored into the PE file as well as the signed hash value.

Except for the following, all portions of the PE File are hashed:

- *The Authenticode Signature entry*: PE files can be authenticode signed. The authenticode signature is contained in the 8-byte entry at offset 128 of the PE Header Data Directory ("Certificate Table" in §25.2.3.3) and the contents of the PE File in the range specified by this directory entry. [*Note*: In a conforming PE File, this entry shall be zero. *end note*]
- *The Strong Name Blob*: The 8-byte entry at offset 32 of the CLI Header ("StrongNameSignature" in §25.3.3) and the contents of the hash data contained at this RVA in the PE File. If the 8-byte entry is 0, there is no associated strong name signature.
- *The PE Header Checksum*: The 4-byte entry at offset 64 of the PE Header Windows NT-Specific Fields ("File Checksum" in §25.2.3.2). [*Note*: In a conforming PE File, this entry shall be zero. *end note*]

6.2.1.4 Version numbers

AsmDecl `::= .ver` *Int32* `':'` *Int32* `':'` *Int32* `':'` *Int32* | …

The version number of an assembly is specified as four 32-bit integers. This version number shall be captured at compile time and used as part of all references to the assembly within the compiled module.

All standardized assemblies shall have the last two 32-bit integers set to 0. This standard places no other requirement on the use of the version numbers, although individual implementers are urged to avoid setting both of the last two 32-bit integers to 0 to avoid a possible collision with future versions of this standard.

Future versions of this standard shall change one or both of the first two 32-bit integers specified for a standardized assembly if any additional functionality is added or any additional features of the VES are required to implement it. Furthermore, future versions of this standard shall change one or both of the first two 32-bit integers specified for the **mscorlib** assembly so that its version number can be used (if desired) to distinguish between different versions of the Execution Engine required to run programs.

[*Note:* A conforming implementation can ignore version numbers entirely, or it can require that they match precisely when binding a reference, or it can exhibit any other behavior deemed appropriate. By convention:

1. The first of these 32-bit integers is considered to be the major version number, and assemblies with the same name, but different major versions, are not interchangeable. This would be appropriate, for example, for a major rewrite of a product where backwards compatibility cannot be assumed.
2. The second of these 32-bit integers is considered to be the minor version number, and assemblies with the same name and major version, but different minor versions, indicate significant enhancements, but with the intention of being backwards compatible. This would be appropriate, for example, on a "point release" of a product or a fully backward compatible new version of a product.
3. The third of these 32-bit integers is considered to be the build number, and assemblies that differ only by build number are intended to represent a recompilation from the same source. This would be appropriate, for example, because of processor, platform, or compiler changes.
4. The fourth of these 32-bit integers is considered to be the revision number, and assemblies with the same name, major and minor version number, but different revisions, are intended to be fully interchangeable. This would be appropriate, for example, to fix a security hole in a previously released assembly.

end note]

6.2.2 Manifest resources

A *manifest resource* is simply a named item of data associated with an assembly. A manifest resource is introduced using the `.mresource` directive, which adds the manifest resource to the assembly manifest begun by a preceding `.assembly` declaration.

Decl ::=	**Clause**
`.mresource [public| private]` *DottedName* `'{'` *ManResDecl** `'}'`	
| …	5.10

If the manifest resource is declared `public`, it is exported from the assembly. If it is declared `private`, it is not exported, in which case, it is only available from within the assembly. The *DottedName* is the name of the resource.

ManResDecl ::=	**Description**	**Clause**
`.assembly extern` *DottedName*	Manifest resource is in external assembly with name *DottedName*.	6.3
| `.custom` *CustomDecl*	Custom attribute.	21
| `.file` *DottedName* `at` *Int32*	Manifest resource is in file *DottedName* at byte offset *Int32*.	

For a resource stored in a file that is not a module (for example, an attached text file), the file shall be declared in the manifest using a separate (top-level) `.file` declaration (see §6.2.3) and the byte offset shall be zero. A resource that is defined in another assembly is referenced using `.assembly extern`, which requires that the assembly has been defined in a separate (top-level) `.assembly extern` directive (§6.3).

6.2.3 Associating files with an assembly

Assemblies can be associated with other files (such as documentation and other files that are used during execution). The declaration `.file` is used to add a reference to such a file to the manifest of the assembly: (See §22.19)

Decl ::=	**Clause**
`.file [nometadata]` *Filename* `.hash '=' '('` *Bytes* `')' [.entrypoint]`	
| …	5.10

The attribute `nometadata` is specified if the file is not a module according to this specification. Files that are marked as `nometadata` can have any format; they are considered pure data files.

The *Bytes* after the `.hash` specify a hash value computed for the file. The VES shall recompute this hash value prior to accessing this file and if the two do not match, the behavior is unspecified. The algorithm used to calculate this hash value is specified with `.hash algorithm` (§6.2.1.1).

If specified, the `.entrypoint` directive indicates that the entrypoint of a multi-module assembly is contained in this file.

6.3 Referencing assemblies

Decl ::=	**Clause**
`.assembly extern` *DottedName* `[as` *DottedName* `] '{'` *AsmRefDecl** `'}'`	
| …	5.10

An assembly mediates all accesses to other assemblies from the files that it contains. This is done through the metadata by requiring that the manifest for the executing assembly contain a declaration for any assembly referenced by the executing code. A top-level `.assembly extern` declaration is used for this purpose. The optional **as** clause provides an alias, which allows ILAsm to address external assemblies that have the same name, but differing in version, culture, etc.

The dotted name used in `.assembly extern` shall exactly match the name of the assembly as declared with an `.assembly` directive, in a case-sensitive manner. (So, even though an assembly might be stored within a file, within a file system that is case-insensitive, the names stored internally within metadata are case-sensitive, and shall match exactly.)

AsmRefDecl ::=	**Description**	**Clause**
`.hash '=' '(' Bytes ')'`	Hash of referenced assembly	6.2.3
`| .custom` *CustomDecl*	Custom attributes	21
`| .culture` *QSTRING*	Culture of the referenced assembly	6.2.1.2
`| .publickeytoken '=' '(' Bytes ')'`	The low 8 bytes of the SHA-1 hash of the originator's public key.	6.3
`| .publickey '=' '(' Bytes ')'`	The originator's full public key	6.2.1.3
`| .ver` *Int32* `':'` *Int32* `':'` *Int32* `':'` *Int32*	Major version, minor version, build, and revision	6.2.1.4

These declarations are the same as those for `.assembly` declarations (§6.2.1), except for the addition of `.publickeytoken`. This declaration is used to store the low 8 bytes of the SHA-1 hash of the originator's public key in the assembly reference, rather than the full public key.

An assembly reference can store either a full public key or an 8-byte "public key token." Either can be used to validate that the same private key used to sign the assembly at compile time also signed the assembly used at runtime. Neither is required to be present, and while both can be stored, this is not useful.

A conforming implementation of the CLI need not perform this validation, but it is permitted to do so, and it can refuse to load an assembly for which the validation fails. A conforming implementation of the CLI can also refuse to permit access to an assembly unless the assembly reference contains either the public key or the public key token. A conforming implementation of the CLI shall make the same access decision independent of whether a public key or a token is used.

[*Rationale*: The public key or public key token stored in an assembly reference is used to ensure that the assembly being referenced and the assembly actually used at runtime were produced by an entity in possession of the same private key, and can therefore be assumed to have been intended for the same purpose. While the full public key is cryptographically safer, it requires more storage in the reference. The use of the public key token reduces the space required to store the reference while only weakening the validation process slightly. *end rationale*]

[*Note*: To validate that an assembly's contents have not been tampered with since it was created, the full public key in the assembly's own identity is used, not the public key or public key token stored in a reference to the assembly. *end note*]

[*Example:*

```
.assembly extern MyComponents
{ .publickeytoken = (BB AA BB EE 11 22 33 00)
  .hash = (2A 71 E9 47 F5 15 E6 07 35 E4 CB E3 B4 A1 D3 7F 7F A0 9C 24)
  .ver 2:10:2002:0
}
```

end example]

6.4 Declaring modules

All CIL files are modules and are referenced by a logical name carried in the metadata rather than by their file name. See §22.30.

Decl ::=	**Clause**
| `.module` *Filename*	
| …	5.10

[*Example:*

```
.module CountDown.exe
```

end example]

6.5 Referencing modules

When an item is in the current assembly, but is part of a module other than the one containing the manifest, the defining module shall be declared in the manifest of the assembly using the `.module extern` directive. The name used in the `.module extern` directive of the referencing assembly shall exactly match the name used in the `.module` directive (§6.4) of the defining module. See §22.31.

Decl ::=	**Clause**
| `.module extern` *Filename*	
| …	5.10

[*Example:*

```
.module extern Counter.dll
```

end example]

6.6 Declarations inside a module or assembly

Declarations inside a module or assembly are specified by the following grammar. More information on each option can be found in the corresponding clause or subclause.

Decl ::=	**Clause**
| `.class` *ClassHeader* `'{'` *ClassMember** `'}'`	10
| `.custom` *CustomDecl*	21
| `.data` *DataDecl*	16.3.1
| `.field` *FieldDecl*	16
| `.method` *MethodHeader* `'{'` *MethodBodyItem** `'}'`	15
| *ExternSourceDecl*	5.7
| *SecurityDecl*	20
| …	

6.7 Exported type definitions

The manifest module, of which there can only be one per assembly, includes the `.assembly` directive. To export a type defined in any other module of an assembly requires an entry in the assembly's manifest. The following grammar is used to construct such an entry in the manifest:

Decl ::=	**Clause**
.class extern *ExportAttr DottedName* '{' *ExternClassDecl** '}'	
| …	

ExternClassDecl ::=	**Clause**
.file *DottedName*	
| .class extern *DottedName*	
| .custom *CustomDecl*	21

The *ExportAttr* value shall be either `public` or `nested public` and shall match the visibility of the type.

For example, suppose an assembly consists of two modules, A.EXE and B.DLL. A.EXE contains the manifest. A public class `Foo` is defined in B.DLL. In order to export it—that is, to make it visible by, and usable from, other assemblies—a `.class extern` directive shall be included in A.EXE. Conversely, a public class `Bar` defined in A.EXE does not need any `.class extern` directive.

[*Rationale:* Tools should be able to retrieve a single module, the manifest module, to determine the complete set of types defined by the assembly. Therefore, information from other modules within the assembly is replicated in the manifest module. By convention, the manifest module is also known as the assembly. *end rationale*]

6.8 Type forwarders

A *type forwarder* indicates that a type originally in this assembly is now located in a different assembly, the VES shall resolve references for the type to the other assembly. The type forwarding information is stored in the *ExportedType* table (§**Error! Reference source not found.**). The following grammar is used to construct the entry in the *ExportedType* table:

Decl ::=	**Clause**
.class extern forwarder *DottedName* '{'.assembly extern *DottedName* '}'	
| …	

[*Rationale:* Type forwarders allow assemblies which reference the original assembly for the type to function correctly without recompilation if the type is moved to another assembly. *end rationale*]

7 Types and signatures

The metadata provides mechanisms to both define and reference types. §10 describes the metadata associated with a type definition, regardless of whether the type is an interface, class, or value type. The mechanism used to reference types is divided into two parts:

- A logical description of user-defined types that are referenced, but (typically) not defined in the current module. This is stored in a table in the metadata (§22.38).
- A *signature* that encodes one or more type references, along with a variety of modifiers. The grammar non-terminal *Type* describes an individual entry in a signature. The encoding of a signature is specified in §23.1.16.

7.1 Types

The following grammar completely specifies all built-in types (including pointer types) of the CLI system. It also shows the syntax for user defined types that can be defined in the CLI system:

Type ::=	**Description**	**Clause**
'!' *Int32*	Generic parameter in a type definition, accessed by index from 0	9.1
| '!!' *Int32*	Generic parameter in a method definition, accessed by index from 0	9.2
| bool	Boolean	7.2
| char	16-bit Unicode code point	7.2
| class *TypeReference*	User defined reference type	7.3
| float32	32-bit floating-point number	7.2
| float64	64-bit floating-point number	7.2
| int8	Signed 8-bit integer	7.2
| int16	Signed 16-bit integer	7.2
| int32	Signed 32-bit integer	7.2
| int64	Signed 64-bit integer	7.2
| method *CallConv Type* '*' '(' *Parameters* ')'	Method pointer	14.5
| native int	32- or 64-bit signed integer whose size is platform-specific	7.2
| native unsigned int	32- or 64-bit unsigned integer whose size is platform-specific	7.2
| object	See System.Object in Partition IV	
| string	See System.String in Partition IV	
| *Type* '&'	Managed pointer to *Type*. *Type* shall not be a managed pointer type or typedref	14.4
| *Type* '*'	Unmanaged pointer to *Type*	14.4
| *Type* '<' *GenArgs* '>'	Instantiation of generic type	9.4

Type ::=	**Description**	**Clause**
| *Type* '[' [*Bound* [',' *Bound*]*] ']'	Array of *Type* with optional rank (number of dimensions) and bounds.	14.1and 14.2
| *Type* modopt '(' *TypeReference* ')'	Custom modifier that can be ignored by the caller.	7.1.1
| *Type* modreq '(' *TypeReference* ')'	Custom modifier that the caller shall understand.	7.1.1
| *Type* pinned	For local variables only. The garbage collector shall not move the referenced value.	7.1.2
| typedref	Typed reference (i.e., a value of type System.TypedReference), created by mkrefany and used by refanytype or refanyval.	7.2
| valuetype *TypeReference*	(Unboxed) user defined value type	13
| unsigned int8	Unsigned 8-bit integer	7.2
| unsigned int16	Unsigned 16-bit integer	7.2
| unsigned int32	Unsigned 32-bit integer	7.2
| unsigned int64	Unsigned 64-bit integer	7.2
| void	No type. Only allowed as a return type or as part of void *	7.2

In several situations the grammar permits the use of a slightly simpler representation for specifying types; e.g., "System.GC" can be used instead of "class System.GC". Such representations are called *type specifications*:

TypeSpec ::=	**Clause**
'[' [.module] *DottedName* ']'	7.3
| *TypeReference*	7.2
| *Type*	7.1

7.1.1 modreq and modopt

Custom modifiers, defined using modreq ("required modifier") and modopt ("optional modifier"), are similar to custom attributes (§21) except that modifiers are part of a signature rather than being attached to a declaration. Each modifer associates a type reference with an item in the signature.

The CLI itself shall treat required and optional modifiers in the same manner. Two signatures that differ only by the addition of a custom modifier (required or optional) shall not be considered to match. Custom modifiers have no other effect on the operation of the VES.

[*Rationale:* The distinction between required and optional modifiers is important to tools other than the CLI that deal with the metadata, typically compilers and program analysers. A required modifier indicates that there is a special semantics to the modified item that should not be ignored, while an optional modifier can simply be ignored.

For example, the const qualifier in the C programming language can be modelled with an optional modifier since the caller of a method that has a const-qualified parameter need not treat it in any special way. On the other hand, a parameter that shall be copy-constructed in C++ shall be marked with a required custom attribute since it is the caller who makes the copy. *end rationale*]

7.1.2 pinned

The signature encoding for `pinned` shall appear only in signatures that describe local variables (§15.4.1.3). While a method with a pinned local variable is executing, the VES shall not relocate the object to which the local refers. That is, if the implementation of the CLI uses a garbage collector that moves objects, the collector shall not move objects that are referenced by an active pinned local variable.

[*Rationale:* If unmanaged pointers are used to dereference managed objects, these objects shall be pinned. This happens, for example, when a managed object is passed to a method designed to operate with unmanaged data. *end rationale*]

7.2 Built-in types

The CLI built-in types have corresponding value types defined in the Base Class Library. They shall be referenced in signatures only using their special encodings (i.e., not using the general purpose `valuetype` *TypeReference* syntax). Partition I specifies the built-in types.

7.3 References to user-defined types (*TypeReference*)

User-defined types are referenced either using their full name and a resolution scope or, if one is available in the same module, a type definition (§10).

A *TypeReference* is used to capture the full name and resolution scope:

TypeReference ::=
[*ResolutionScope*] *DottedName* [`'/'` *DottedName*]*

ResolutionScope ::=
`'['` `.module` *Filename* `']'`
| `'['` *AssemblyRefName* `']'`

AssemblyRefName ::=	**Clause**
DottedName	5.1

The following resolution scopes are specified for un-nested types:

- **Current module (and, hence, assembly)**. This is the most common case and is the default if no resolution scope is specified. The type shall be resolved to a definition only if the definition occurs in the same module as the reference.

[*Note:* A type reference that refers to a type in the same module and assembly is better represented using a type definition. Where this is not possible (e.g., when referencing a nested type that has `compilercontrolled` accessibility) or convenient (e.g., in some one-pass compilers) a type reference is equivalent and can be used. *end note*]

- **Different module, current assembly**. The resolution scope shall be a module reference syntactically represented using the notation `[.module` *Filename*`]`. The type shall be resolved to a definition only if the referenced module (§6.4) and type (§6.7) have been declared by the current assembly and hence have entries in the assembly's manifest. Note that in this case the manifest is not physically stored with the referencing module.
- **Different assembly**. The resolution scope shall be an assembly reference syntactically represented using the notation `[`*AssemblyRefName*`]`. The referenced assembly shall be declared in the manifest for the current assembly (§6.3), the type shall be declared in the referenced assembly's manifest, and the type shall be marked as exported from that assembly (§6.7 and §10.1.1).

- For nested types, the resolution scope is always the enclosing type. **(**See §10.6). This is indicated syntactically by using a slash ("/") to separate the enclosing type name from the nested type's name.

[*Example:* The type `System.Console` defined in the base class library (found in the assembly named *mscorlib*):

```
.assembly extern mscorlib { }
.class [mscorlib]System.Console
```

A reference to the type named `C.D` in the module named *x* in the current assembly:

```
.module extern x
.class [.module x]C.D
```

A reference to the type named `C` nested inside of the type named `Foo.Bar` in another assembly, named *MyAssembly*:

```
.assembly extern MyAssembly { }
.class [MyAssembly]Foo.Bar/C
```

end example]

7.4 Native data types

Some implementations of the CLI will be hosted on top of existing operating systems or runtime platforms that specify data types required to perform certain functions. The metadata allows interaction with these *native data types* by specifying how the built-in and user-defined types of the CLI are to be *marshalled* to and from native data types. This marshalling information can be specified (using the keyword `marshal`) for

- the return type of a method, indicating that a native data type is actually returned and shall be marshalled back into the specified CLI data type
- a parameter to a method, indicating that the CLI data type provided by the caller shall be marshalled into the specified native data type. (If the parameter is passed by reference, the updated value shall be marshalled back from the native data type into the CLI data type when the call is completed.)
- a field of a user-defined type, indicating that any attempt to pass the object in which it occurs, to platform methods shall make a copy of the object, replacing the field by the specified native data type. (If the object is passed by reference, then the updated value shall be marshalled back when the call is completed.)

The following table lists all native types supported by the CLI, and provides a description for each of them. (A more complete description can be found in Partition IV in the definition of the enum `System.Runtime.Interopservices.UnmanagedType`, which provides the actual values used to encode these types.) All encoding values in the range 0–63, inclusive, are reserved for backward compatibility with existing implementations of the CLI. Values in the range 64–127 are reserved for future use in this and related Standards.

NativeType `::=`	**Description**	**Name in the class library enum type `UnmanagedType`**
`'[' ']'`	Native array. Type and size are determined at runtime from the actual marshaled array.	`LPArray`
`| bool`	Boolean. 4-byte integer value where any non-zero value represents TRUE, and 0 represents FALSE.	`Bool`
`| float32`	32-bit floating-point number.	`R4`
`| float64`	64-bit floating-point number.	`R8`

NativeType ::=	**Description**	**Name in the class library enum type** `UnmanagedType`
| [unsigned] int	Signed or unsigned integer, sized to hold a pointer on the platform	`SysUInt` or `SysInt`
| [unsigned] int8	Signed or unsigned 8-bit integer	`U1` or `I1`
| [unsigned] int16	Signed or unsigned 16-bit integer	`U2` or `I2`
| [unsigned] int32	Signed or unsigned 32-bit integer	`U4` or `I4`
| [unsigned] int64	Signed or unsigned 64-bit integer	`U8` or `I8`
| lpstr	A pointer to a null-terminated array of ANSI characters. The code page is implementation-specific.	`LPStr`
| lpwstr	A pointer to a null-terminated array of Unicode characters. The character encoding is implementation-specific.	`LPWStr`
| method	A function pointer.	`FunctionPtr`
| *NativeType* '[' ']'	Array of *NativeType*. The length is determined at runtime by the size of the actual marshaled array.	`LPArray`
| *NativeType* '[' *Int32* ']'	Array of *NativeType* of length *Int32*.	`LPArray`
| *NativeType* '[' '+' *Int32* ']'	Array of *NativeType* with runtime supplied element size. The *Int32* specifies a parameter to the current method (counting from parameter number 0) that, at runtime, will contain the size of an element of the array in bytes. Can only be applied to methods, not fields.	`LPArray`
| *NativeType* '[' *Int32* '+' *Int32* ']'	Array of *NativeType* with runtime supplied element size. The first *Int32* specifies the number of elements in the array. The second *Int32* specifies which parameter to the current method (counting from parameter number 0) will specify the additional number of elements in the array. Can only be applied to methods, not fields.	`LPArray`

[*Example:*

```
.method int32 M1( int32 marshal(int32), bool[] marshal(bool[5]) )
```

Method M1 takes two arguments: an `int32`, and an array of 5 `bool`s.

```
.method int32 M2( int32 marshal(int32), bool[] marshal(bool[+1]) )
```

Method M2 takes two arguments: an `int32`, and an array of `bool`s: the number of elements in that array is given by the value of the first parameter.

```
.method int32 M3( int32 marshal(int32), bool[] marshal(bool[7+1]) )
```

Method M3 takes two arguments: an `int32`, and an array of `bool`s: the number of elements in that array is given as 7 plus the value of the first parameter. *end example*]

8 Visibility, accessibility and hiding

Partition I specifies visibility and accessibility. In addition to these attributes, the metadata stores information about method name hiding. *Hiding* controls which method names inherited from a base type are available for compile-time name binding.

8.1 Visibility of top-level types and accessibility of nested types

Visibility is attached only to top-level types, and there are only two possibilities: visible to types within the same assembly, or visible to types regardless of assembly. For nested types (i.e., types that are members of another type) the nested type has an *accessibility* that further refines the set of methods that can reference the type. A nested type can have any of the seven accessibility modes (see Partition I), but has no direct visibility attribute of its own, using the visibility of its enclosing type instead.

Because the visibility of a top-level type controls the visibility of the names of all of its members, a nested type cannot be more visible than the type in which it is nested. That is, if the enclosing type is visible only within an assembly then a nested type with `public` accessibility is still only available within that assembly. By contrast, a nested type that has `assembly` accessibility is restricted to use within the assembly even if the enclosing type is visible outside the assembly.

To make the encoding of all types consistent and compact, the visibility of a top-level type and the accessibility of a nested type are encoded using the same mechanism in the logical model of §23.1.15.

8.2 Accessibility

Accessibility is encoded directly in the metadata (see §22.26 for an example).

8.3 Hiding

Hiding is a compile-time concept that applies to individual methods of a type. The CTS specifies two mechanisms for hiding, specified by a single bit:

- *hide-by-name*, meaning that the introduction of a name in a given type hides all inherited members of the same kind with the same name.
- *hide-by-name-and-sig*, meaning that the introduction of a name in a given type hides any inherited member of the same kind, but with precisely the same type (in the case of nested types and fields) or signature (in the case of methods, properties, and events).

There is no runtime support for hiding. A conforming implementation of the CLI treats all references as though the names were marked hide-by-name-and-sig**.** Compilers that desire the effect of hide-by-name can do so by marking method definitions with the `newslot` attribute (§15.4.2.3) and correctly choosing the type used to resolve a method reference (§15.1.3).

9 Generics

As mentioned in Partition I, generics allows a whole family of types and methods to be defined using a pattern, which includes placeholders called *generic parameters*. These generic parameters are replaced, as required, by specific types, to instantiate whichever member of the family is actually required. For example, `class List<T>{}`, represents a whole family of possible *List*s; `List<string>`, `List<int>` and `List<Button>` are three possible instantiations; however, as we'll see below, the CLS-compliant names of these types are really `class List`1<T>{}`, `List`1<string>`, `List`1<int>`, and `List`1<Button>`.

A generic type consists of a name followed by a `<…>`-delimited list of generic parameters, as in `C<T>`. Two or more generic types shall not be defined with the same name, but different numbers of generic parameters, in the same scope. However, to allow such overloading on generic arity at the source language level, CLS Rule 43 is defined to map generic type names to unique CIL names. That Rule states that the CLS-compliant name of a type `C` having one or more generic parameters, shall have a suffix of the form `` `n ``, where *n* is a decimal integer constant (without leading zeros) representing the number of generic parameters that `C` has. For example: the types `C`, `C<T>`, and `C<K,V>` have CLS-compliant names of `C`, `C`1<T>`, and `C`2<K,V>`, respectively. [*Note*: The names of all standard library types are CLS-compliant; e.g., `System.Collections.Generic.IEnumerable`1<T>`. *end note*]

Before generics is discussed in detail, here are the definitions of some new terms:

- `public class List`1<T> {}` is a *generic type definition*.
- `<T>` is a generic parameter list, and `T` is a generic parameter.
- `List`1<T>` is a *generic type*; it is sometimes termed a *generic type*, or *open generic type* because it has at least one generic parameter. This partition will use the term *open type*.
- `List`1<int>` is a *closed generic type* because it has no unbound generic parameters. (It is sometimes called an *instantiated* generic type or a generic type *instantiation*). This partition will use the term *closed type*.
- Note that generics includes generic types which are neither strictly *open* nor strictly *closed*; e.g., the base class B, in: `.public class D`1<V> extends B`2<!0,int32> {}`, given `.public class B`2<T,U> {}`.
- If a distinction need be made between generic types and ordinary types, the latter are referred to as *non-generic types*.
- `<int>` is a generic argument list, and `int` is a generic argument.
- This standard maintains the distinction between generic parameters and generic arguments. If at all possible, use the phrase "`int` is the type used for generic parameter T*"* when speaking of `List`1<int>`. (In Reflection, this is sometimes referred to as "T is bound to `int`")
- `"(C1, …, Cn) T"` is a *generic parameter constraint* on the generic parameter `T`.

[*Note*: Conside the following definition:

```
class C`2<(I1,I2) S, (Base,I3) T> { … }
```

This denotes a class called `C`, with two generic parameters, `S` and `T`. `S` is constrained to implement two interfaces, `I1` and `I2`. `T` is constrained to derive from the class `Base`, and also to implement the interface `I3`. *end note*]

Within a generic type definition, its generic parameters are referred to by their index. Generic parameter zero is referred to as `!0`, generic parameter one as `!1`, and so on. Similarly, within the body of a generic method definition, its generic parameters are referred to by their index; generic parameter zero is referred to as `!!0`, generic parameter one as `!!1`, and so on.

9.1 Generic type definitions

A generic type definition is one that includes generic parameters. Each such generic parameter can have a name and an optional set of constraints—types with which generic arguments shall be assignment-compatible. Optional variance notation is also permitted (§10.1.7). (For an explanation of the `!` and `!!` notation used below, see §9.4) The generic parameter is in scope in the declarations of:

- its constraints (e.g., `.class … C`1<(class IComparable`1<!0>) T>`)
- any base class from which the type-under-definition derives (e.g., `.class … MultiSet`1<T> extends class Set`1<!0[]>`)
- any interfaces that the type-under-definition implements (e.g., `.class … Hashtable`2<K,D> implements class IDictionary`2<!0,!1>`)
- all members (instance and static fields, methods, constructors, properties and events) except nested classes. [*Note*: C# allows generic parameters from an enclosing class to be used in a nested class, but adds any required extra generic parameters to the nested class definition in metadata. *end note*]

A generic type definition can include static, instance, and virtual methods.

Generic type definitions are subject to the following restrictions:

- A generic parameter, on its own, cannot be used to specify the base class, or any implemented interfaces. So, for example, `.class … G`1<T> extends !0` is invalid. However, it is valid for the base class, or interfaces, to use that generic parameter when nested within another generic type. For example, `.class … G`1<T> extends class H`1<!0>` and `.class … G`1<T> extends class B`2<!0,int32>` are valid.

  [*Rationale*: This permits checking that generic types are valid at definition time rather than at instantiation time. e.g., in `.class … G`1<T> extends !0`, we do not know what methods would override what others because no information is available about the base class; indeed, we do not even know whether `T` is a class: it might be an array or an interface. Similarly, for `.class … C`2<(!1)T,U>` where we are in the same situation of knowing nothing about the base class/interface definition. *end rationale*]

- Varargs methods cannot be members of generic types

  [*Rationale*: Implementing this feature would take considerable effort. Since varargs has very limited use among languages targetting the CLI, it was decided to exclude varargs methods from generic types. *end rationale*]

- When generic parameters are ignored, there shall be no cycles in the inheritance/interface hierarchy. To be precise, define a graph whose nodes are possibly-generic (but open) classes and interfaces, and whose edges are the following:
  - If a (possibly-generic) class or interface D extends or implements a class or interface B, then add an edge from D to B.
  - If a (possibly-generic) class or interface D extends or implements an instantiated class or interface B<*type-1, …, type-n*>, then add an edge from D to B.
  - The graph is valid if it contains no cycles.

  [*Note*: This algorithm is a natural generalization of the rules for non-generic types. See Partition I, §8.9.9 *end note*]

9.2 Generics and recursive inheritance graphs

[*Rationale*: Although inheritance graphs cannot be directly cyclic, instantiations given in parent classes or interfaces may introduce either direct or indirect cyclic dependencies, some of which are allowed (e.g., `C : IComparable<C>`), and some of which are disallowed (e.g., `class A<T> : B<A<A<T>>>` given `class B<U>`). *end rationale*]

Each type definition shall generate a finite *instantiation closure*. An instantiation closure is defined as follows:

1. Create a set containing a single generic type definition.
2. Form the closure of this set by adding all generic types referenced in the type signatures of base classes and implemented interfaces of all types in the set. Include nested instantiations in this set, so a referenced type `Stack<List<T>>` actually counts as both `List<T>` and `Stack<List<T>>`.
3. Construct a graph:
    - Whose nodes are the formal type parameters of types in the set. Use alpha-renaming as needed to avoid name clashes.
    - If T appears as the actual type argument to be substituted for U in some referenced type D<…, U, …> add a **non-expanding** (->) edge from T to U.
    - If T appears somewhere inside (but not as) the actual type argument to be substituted for U in referenced type D<…, U, …> add an **expanding** (=>) edge from T to U.

An expanding-cycle is a cycle in the instantiation closure that contains at least one expanding-edge (=>). The instantiation-closure of the system is finite if and only if the graph as constructed above contains no expanding-cycles.

[*Example*:

```
class B<U>
class A<T> : B<A<A<T>>>
```

generates the edges (using => for expanding-edges and -> for non-expanding-edges)

T -> T (generated by referenced type A<T>)
T => T (generated by referenced type A<A<T>>)
T => U (generated by referenced type B<A<A<T>>>)

This graph does contain an expanding-cycle, so the instantiation closure is infinite. *end example*]

[*Example*:

```
class B<U>
class A<T> : B<A<T>>
```

generates the edges

T -> T (generated by referenced type A<T>)
T => U (generated by referenced type B<A<T>>)

This graph does not contain an expanding-cycle, so the instantiation closure is finite. *end example*]

[*Example*:

```
class P<T>
class C<U,V> : P<D<V,U>>
class D<W,X> : P<C<W,X>>
```

generates the edges

U -> X V -> W U => T V => T (generated by referenced type D<V,U> and P<D<V,U>>)
W -> U X -> V W => T X => T (generated by referenced type C<W,X> and P<C<W,X>>)

This graph contains non-expanding-cycles (e.g. U -> X -> V -> W -> U), but no expanding-cycle, so the instantiation closure is finite. *end example*]

9.3 Generic method definitions

A generic method definition is one that includes a generic parameter list. A generic method can be defined within a non-generic type; or within a generic type, in which case the method's generic parameter(s) shall be additional to the generic parameter(s) of the owner. As with generic type definitions, each generic parameter on a generic method definition has a name and an optional set of constraints.

Generic methods can be static, instance, or virtual. Class or instance constructors (`.cctor`, or `.ctor`, respectively) shall not be generic.

The method generic parameters are in scope in the signature and body of the method, and in the generic parameter constraints. [*Note:* The signature includes the method return type. So, in the example:

```
.method … !!0 M`1<T>() { … }
```

the `!!0` is in scope—it's the generic parameter of `M`1<T>` even though it preceeds that parameter in the declaration.. *end note*]

Generic instance (virtual and non-virtual) methods can be defined as members of generic types, in which case the generic parameters of both the generic type and the generic method are in scope in the method signature and body, and in constraints on method generic parameters.

9.4 Instantiating generic types

GenArgs is used to represent a generic argument list:

GenArgs ::=
Type [',' *Type*]*

We say that a type is *closed* if it contains no generic parameters; otherwise, it is *open*.

A given generic type definition can be *instantiated* with *generic arguments* to produce an *instantiated type*.

[*Example*: Given suitable definitions for the generic class `MyList` and value type `Pair`, we could instantiate them as follows:

```
newobj instance void class MyList`1<int32>::.ctor()
initobj valuetype Pair`2<int32, valuetype Pair<string,int32>>
```

end example]

[*Example*:

```
ldtoken !0                    // !0 = generic parameter 0 in generic type definition
castclass class List`1<!1>    // !1 = generic parameter 1 in generic type definition
box !!1                       // !!1 = generic parameter 1 in generic method definition
```

end example]

The number of generic arguments in an instantiation shall match the number of generic parameters specified in the type or method definition.

The CLI does not support partial instantiation of generic types. And generic types shall not appear uninstantiated anywhere in metadata signature blobs.

The following kinds of type cannot be used as arguments in instantiations (of generic types or methods):

- Byref types (e.g., `System.Generic.Collection.List`1<string&>` is invalid)
- Byref-like types, i.e. value types that contain fields that can point into the CIL evaluation stack (e.g., `List<System.RuntimeArgumentHandle>` is invalid)
- Typed references (e.g. `List<System.TypedReference>` is invalid)
- Unmanaged pointers (e.g. `List<int32*>` is invalid)
- void (e.g., `List<System.Void>` is invalid)

[*Rationale*: Byrefs types cannot be used as generic arguments because some, indeed most, instantiations would be invalid. For example, since byrefs are not allowed as field types or as method return types, in the definition of `List`1<string&>`, one could not declare a field of type `!0`, nor a method that returned a type of `!0`. *end rationale*]

[*Rationale*: Unmanaged pointers are disallowed because as currently specified unmanaged pointers are not technically subclasses of System.Object. This restriction can be lifted, but currently the runtime enforces this restriction and this spec reflects that.]

Objects of instantiated types shall carry sufficient information to recover at runtime their exact type (including the types and number of their generic arguments). [*Rationale*: This is required to correctly implement casting and instance-of testing, as well as in reflection capabilities (`System.Object::GetType`). *end rationale*]

9.5 Generics variance

The CLI supports covariance and contravariance of generic parameters, but only in the signatures of interfaces and delegate classes.

The symbol "+" is used in the syntax of §10.1.7 to denote a covariant generic parameter, while "-" is used to denote a contravariant generic parameter

This block contains only informative text

Suppose we have a generic interface, which is covariant in its one generic parameter; e.g., `IA`1<+T>`. Then all instantiations satisfy `IA`1<GenArgB>` := `IA`1<GenArgA>`, so long as `GenArgB` := `GenArgA` using the notion from assignment compatibility. So, for example, an instance of type `IA`1<string>` can be assigned to a local of type type `IA`1<object>`.

Generic contravariance operates in the opposite sense: supposing that we have a contravariant interface `IB`1<-T>`, then `IB`1<GenArgB>` := `IB`1<GenArgA>`, so long as `GenArgA` := `GenArgB`.

[*Example:* (The syntax used is illustrative of a high-level language.)

```
// Covariant parameters can be used as result types
interface IEnumerator<+T> {
  T Current { get; }
  bool MoveNext();
}

// Covariant parameters can be used in covariant result types
interface IEnumerable<+T> {
  IEnumerator<T> GetEnumerator();
}

// Contravariant parameters can be used as argument types
interface IComparer<-T> {
  bool Compare(T x, T y);
}

// Contravariant parameters can be used in contravariant interface types
interface IKeyComparer<-T> : IComparer<T> {
  bool Equals(T x, T y);
  int GetHashCode(T obj);
}

// A contravariant delegate type
delegate void EventHandler<-T>(T arg);

// No annotation indicates non-variance.  Non-variant parameters can be used anywhere.
// The following type shall be non-variant because T appears in as a method argument as
// well as in a covariant interface type
interface ICollection<T> : IEnumerable<T> {
  void CopyTo(T[] array, int index);
  int Count { get; }
}
```

end example]

End informative text

9.6 Assignment compatibility of instantiated types

- Assignment compatibility is defined in Partition I.8.7.

[*Example*:

Assuming `Employee` := `Manager`,

```
IEnumerable<Manager> eManager = ...
IEnumerable<Employee> eEmployee = eManager;                 // Covariance
IComparer<object> objComp = ...
IComparer<string> strComp = objComp;                        // Contravariance
EventHandler<Employee> employeeHandler = ...
EventHandler<Manager> managerHandler = employeeHandler;     // Contravariance
```

end example]

[*Example*: Given the following:

```
interface IConverter<-T,+U> {
  U Convert(T x);
}
```

`IConverter<string, object>` := `IConverter<object, string>`

Given the following:

```
delegate U Function<-T,+U>(T arg);
```

`Function<string, object>` := `Function<object, string>`. *end example*]

[*Example*:

```
IComparer<object> objComp = ...
// Contravariance and interface inheritance
IKeyComparer<string> strKeyComp = objComp;

IEnumerable<string[]> strArrEnum = …
// Covariance on IEnumerable and covariance on arrays
IEnumerable<object[]> objArrEnum = strArrEnum;

IEnumerable<string>[] strEnumArr = ...
// Covariance on IEnumerable and covariance on arrays
IEnumerable<object>[] objEnumArr = strEnumArr;

IComparer<object[]> objArrComp = ...
// Contravariance on IComparer and covariance on arrays
IComparer<string[]> strArrComp = objArrComp;

IComparer<object>[] objCompArr = ...
// Contravariance on IComparer and covariance on arrays
IComparer<string>[] strCompArr = objCompArr;
```

end example]

9.7 Validity of member signatures

To achieve type safety, it is necessary to impose additional requirements on the well-formedness of signatures of members of covariant and contravariant generic types.

This block contains only informative text

- Covariant parameters can only appear in “producer,” “reader,” or “getter” positions in the type definition; i.e., in
  - result types of methods
  - inherited interfaces
- Contravariant parameters can only appear in “consumer,” “writer,” or “setter” positions in the type definition; i.e., in
  - argument types of methods
- NonVariant parameters can appear anywhere.

End informative text

We now define formally what it means for a co/contravariant generic type definition to be valid.

Generic type definition: A generic type definition G<*var_1 T_1, …, var_n T_n*> is *valid* if G is an interface or delegate type, and each of the following holds, given *S* = <*var_1 T_1, …, var_n T_n*>, where *var_n* is +, -, or nothing:

- Every instance method and virtual method declaration is valid with respect to S
- Every inherited interface declaration is valid with respect to S
- There are no restrictions on static members, instance constructors, or on the type's own generic parameter constraints.

Given the annotated generic parameters S = <*var_1 T_1, …, var_n T_n*>, we define what it means for various components of the type definition to be *valid with respect to S.* We define a negation operation on annotations, written –*S*, to mean "flip negatives to positives, and positives to negatives".

Think of

- "valid with respect to S" as "behaves covariantly"
- "valid with respect to ¬S" as "behaves contravariantly"
- "valid with respect to S and to ¬S" as "behaves non-variantly".

Note that the last of these has the effect of prohibiting covariant and contravariant parameters from a type; i.e., all generic parameters appearing shall be non-variant.

Methods. A method signature *t meth(t_1,…,t_n)* is valid with respect to *S* if

- its result type signature *t* is valid with respect to *S;* and
- each argument type signature *t_i* is valid with respect to ¬*S*.
- each method generic parameter constraint type *t_j* is valid with respect to ¬*S*.

[*Note*: In other words, the result behaves covariantly and the arguments behave contravariantly. Constraints on generic parameters also behave contravariantly. *end note*]

Type signatures. A type signature *t* is *valid with respect to S* if it is

- a non-generic type (e.g., an ordinary class or value type)
- a generic parameter *T_i* for which *var_i* is + or none (i.e., it is a generic parameter that is marked covariant or non-variant)
- an array type *u*[] and *u* is valid with respect to *S;* i.e., array types behave covariantly
- a closed generic type G<*t_1,…,t_n*> for which each
    - *t_i* is valid with respect to *S*, if the *i*'th parameter of G is declared covariant
    - *t_i* is valid with respect to ¬*S*, if the *i*'th parameter of G is declared contravariant
    - *t_i* is valid with respect to *S* and with respect to ¬*S*, if the *i*'th parameter of G is declared non-variant.

9.8 Signatures and binding

Members (fields and methods) of a generic type are referenced in CIL instructions using a metadata token, which specifies an entry in the *MemberRef* table (§22.25). Abstractly, the reference consists of two parts:

1. The type in which the member is declared, in this case, an instantiation of the generic type definition. For example: `IComparer`1<String>`.
2. The name and generic (uninstantiated) signature of the member. For example: `int32 Compare(!0,!0)`.

It is possible for distinct members to have identical types when instantiated, but which can be distinguished by *MemberRef*.

[*Example*:

```
.class public C`2<S,T> {
  .field string f
  .field !0 f
  .method instance void m(!0 x) {...}
  .method instance void m(!1 x) {...}
  .method instance void m(string x) {...}
}
```

The closed type `C`2<string,string>` is valid: it has three methods called `m`, all with the same parameter type; and two fields called `f` with the same type. They are all distinguished through the *MemberRef* encoding described above:

```
string C`2<string, string>::f
!0  C<string, string>::f
void C`2<string, string>::m(!0)
void C`2<string, string>::m(!1)
void C`2<string, string>::m(string)
```

The way in which a source language might resolve this kind of overloading is left to each individual language. For example, many might disallow such overloads.

end example]

9.9 Inheritance and overriding

Member inheritance is defined in Partition I, in "Member Inheritance". (Overriding and hiding are also defined in that partition, in "Hiding, overriding, and layout".) This definition is extended, in an obvious manner, in the presence of generics. Specifically, in order to determine whether a member hides (for static or instance members) or overrides (for virtual methods) a member from a base class or interface, simply substitute each generic parameter with its generic argument, and compare the resulting member signatures. [*Example:* The following illustrates this point:

Suppose the following definitions of a base class *B*, and a derived class *D*.

```
.class B
{ .method public virtual void V(int32 i) { … } }

.class D extends B
{ .method public virtual void V(int32 i) { … } }
```

In class `D`, `D.V` overrides the inherited method `B.V`, because their names and signatures match.

How does this simple example extend in the presence of generics, where class `D` derives from a generic instantiation? Consider this example:

```
.class B`1<T>
{ .method public virtual void V(!0) { … } }

.class D extends B`1<int32>
{ .method public virtual void V(int32) { … } }

.class E extends B`1<string>
{ .method public virtual void V(int32) { … } }
```

Class `D` derives from `B<int32>`. And class `B<int32>` defines the method:

```
public virtual void V(int32 t) { … }
```

where we have simply substituted `B`'s generic parameter `T`, with the specific generic argument `int32`. This matches the method `D.V` (same name and signature). Thus, for the same reasons as in the non-generic example above, it's clear that `D.V` overrides the inherited method `B.V`.

Contrast this with class `E`, which derives from `B<string>`. In this case, substituting `B`'s `T` with `string`, we see that `B.V` has this signature:

```
public virtual void V(string t) { … }
```

This signature differs from method `E.V`, which therefore does not override the base class's `B.V` method.

end example]

Type definitions are invalid if, after substituting base class generic arguments, two methods result in the same name and signature (including return type). The following illustrates this point:

[*Example:*

```
.class B`1<T>
{ .method public virtual void V(!0 t)      { … }
  .method public virtual void V(string x) { … }
}

.class D extends B`1<string> { } // Invalid
```

Class `D` is invalid, because it will inherit from `B<string>` two methods with identical signatures:

```
void V(string)
```

However, the following version of D is valid:

```
.class D extends B`1<string>
{ .method public virtual void  V(string t)   { … }
  .method public virtual void  W(string t)
  { …
    .override  method instance void class B`1<string>::V(!0)
    …
  }
}
```

end example]

When overriding generic methods (that is, methods with their own generic parameters) the number of generic parameters shall match exactly those of the overridden method. If an overridden generic method has one or more constraints on its generic arguments then:

- The overriding method can have constraints only on the same generic arguments;
- Any such constraint on a generic argument specified by the overriding method shall be no more restrictive than the constraint specified by the overridden method for the same generic argument;

[*Note*: Within the body of an overriding method, only constraints directly specified in its signature apply. When a method is invoked, it's the constraints associated with the metadata token in the call or callvirt instruction that are enforced. *end note*]

9.10 Explicit method overrides

A type, be it generic or non-generic, can implement particular virtual methods (whether the method was introduced in an interface or base class) using an explicit override. (See §10.3.2 and §15.1.4.)

The rules governing overrides are extended, in the presence of generics, as follows:

- If the implementing method is part of a non-generic type or a closed generic type, then the declaring method shall be part of a base class of that type or an interface implemented by that type. [*Example*:

```
.class interface I`1<T>
{ .method public abstract virtual void M(!0) {}
}

.class C implements class I`1<string>
{ .override method instance void class I`1<string>::M(!0) with
             method instance void class C::MInC(string)
  .method virtual void MInC(string s)
  { ldstr "I.M"
    call void [mscorlib]System.Console::WriteLine(string)
    ret
  }
}
```

end example]

- If the implementing method is generic, then the declared method shall also be generic and shall have the same number of method generic parameters.

  Neither the implementing method nor the declared method shall be an instantiated generic method. This means that an instantiated generic method cannot be used to implement an interface method, and that it is not possible to provide a special method for instantiating a generic method with specific generic parameters.
  [*Example:* Given the following

```
 .class interface I
{ .method public abstract virtual void M<T>(!!0) {}
  .method public abstract virtual void N() {}
}

neither of the following .override statements is allowed

.class C implements class I`1<string>
{ .override class I::M<string> with instance void class C::MInC(string)
  .override class I::N with instance void class C::MyFn<string>
  .method virtual void MInC(string s) { … }
  .method virtual void MyFn<T>() { … }
}
```

  end example]

9.11 Constraints on generic parameters

A generic parameter declared on a generic class or generic method can be *constrained* by one or more types (for encoding, see *GenericParamConstraint* table in §22.21) and by one or more special constraints (§10.1.7). Generic parameters can be instantiated only with generic arguments that are assignment compatible (when boxed) with each of the declared constraints and that satisfy all specified special constraints.

Generic parameter constraints shall have at least the same visibility as the generic type definition or generic method definition itself.

[*Note*: There are no other restrictions on generic parameter constraints. In particular, the following uses are valid: Constraints on generic parameters of generic classes can make recursive reference to the generic parameters, and even to the class itself.

```
.class public Set`1<(class IComparable<!0>) T> { … }

// can only be instantiated by a derived class!
.class public C`1<(class C<!0>) T> {}

.class public D extends C`1<class D> { … }
```

Constraints on generic parameters of generic methods can make recursive reference to the generic parameters of both the generic method and its enclosing class (if generic). The constraints can also reference the enclosing class itself.

```
.class public A`1<T> {
  .method public void M<(class IDictionary<!0,!!0>) U>() {}
}
```

Generic parameter constraints can be generic parameters or non-generic types such as arrays.

```
.class public List`1<T> {
  // The constraint on U is T itself
  .method public void AddRange<(!0) U>(class IEnumerable`1<!!0> items) { … }
}
```

end note]

Generic parameters can have multiple constraints: to inherit from at most one base class (if none is specified, the CLI defaults to inheriting from `System.Object`); and to implement zero or more interfaces. (The syntax for using constraints with a class or method is defined in §10.1.7.) [*Example*:

The following declaration shows a generic class *OrderedSet*<*T*>, in which the generic parameter *T* is constrained to inherit both from the class *Employee*, and to implement the interface *IComparable*<*T*>:

```
.class OrderedSet`1<(Employee, class [mscorlib]System.IComparable`1<!0>) T> { … }
```

end example]

[*Note*: Constraints on a generic parameter only restrict the types that the generic parameter may be instantiated with. Verification (see Partition III) requires that a field, property or method that a generic parameter is known to provide through meeting a constraint, cannot be directly accessed/called via the generic parameter unless it is first boxed (see Partition III) or the callvirt instruction is prefixed with the constrained. prefix instruction (see Partition III). *end note*]

This block contains only informative text

9.12 References to members of generic types

CIL instructions that reference type members are generalized to permit reference to members of instantiated types.

The number of generic arguments specified in the reference shall match the number specified in the definition of the type.

CIL instructions that reference methods are generalized to permit reference to instantiated generic methods.

End informative text

10 Defining types

Types (i.e., classes, value types, and interfaces) can be defined at the top-level of a module:

Decl ::=
.class *ClassHeader* '{' *ClassMember** '}'
| …

The logical metadata table created by this declaration is specified in §22.37.

[*Rationale:* For historical reasons, many of the syntactic categories used for defining types incorrectly use "class" instead of "type" in their name. All classes are types, but "types" is a broader term encompassing value types, and interfaces as well. *end rationale*]

10.1 Type header (*ClassHeader*)

A type header consists of

- any number of type attributes,
- optional generic parameters
- a name (an *Id*),
- a base type (or base class type), which defaults to `[mscorlib]System.Object`, and
- an optional list of interfaces whose contract this type and all its descendent types shall satisfy.

ClassHeader ::=
*ClassAttr** *Id* ['<' *GenPars* '>'] [extends *TypeSpec* [implements *TypeSpec*] [',' *TypeSpec*]*]

The optional generic parameters are used when defining a generic type (§10.1.7).

The `extends` keyword specifies the *base type* of a type. A type shall extend from exactly one other type. If no type is specified, *ilasm* will add an `extends` clause to make the type inherit from `System.Object`.

The `implements` keyword specifies the *interfaces* of a type. By listing an interface here, a type declares that all of its concrete implementations will support the contract of that interface, including providing implementations of any virtual methods the interface declares. See also §11 and §12.

[*Example:* This code declares the class `CounterTextBox`, which extends the class `System.Windows.Forms.TextBox` in the assembly `System.Windows.Forms`, and implements the interface `CountDisplay` in the module *Counter* of the current assembly. The attributes `private`, `auto` and `autochar` are described in the following subclauses.

```
.class private auto autochar CounterTextBox
   extends [System.Windows.Forms]System.Windows.Forms.TextBox
   implements [.module Counter]CountDisplay
{ // body of the class
}
```

end example]

A type can have any number of custom attributes attached. Custom attributes are attached as described in §21. The other (predefined) attributes of a type can be grouped into attributes that specify visibility, type layout information, type semantics information, inheritance rules, interoperation information, and information on special handling. The following subclauses provide additional information on each group of predefined attributes.

ClassAttr ::=	**Description**	**Clause**

ClassAttr ::=	**Description**	**Clause**
abstract	Type is abstract.	10.1.4
| ansi	Marshal strings to platform as ANSI.	10.1.5
| auto	Layout of fields is provided automatically.	10.1.2
| autochar	Marshal strings to platform as ANSI or Unicode (platform-specific).	10.1.5
| beforefieldinit	Need not initialize the type before a static method is called.	10.1.6
| explicit	Layout of fields is provided explicitly.	10.1.2
| interface	Declares an interface.	10.1.3
| nested assembly	Assembly accessibility for nested type.	10.1.1
| nested famandassem	Family and assembly accessibility for nested type.	10.1.1
| nested family	Family accessibility for nested type.	10.1.1
| nested famorassem	Family or assembly accessibility for nested type.	10.1.1
| nested private	Private accessibility for nested type.	10.1.1
| nested public	Public accessibility for nested type.	10.1.1
| private	Private visibility of top-level type.	10.1.1
| public	Public visibility of top-level type.	10.1.1
| rtspecialname	Special treatment by runtime.	10.1.6
| sealed	The type cannot be derived from.	10.1.4
| sequential	Layout of fields is sequential.	10.1.2
| serializable	Reserved (to indicate this type can be serialized).	10.1.6
| specialname	Might get special treatment by tools.	10.1.6
| unicode	Marshal strings to platform as Unicode.	10.1.5

10.1.1 Visibility and accessibility attributes

ClassAttr ::= …
| nested assembly
| nested famandassem
| nested family
| nested famorassem
| nested private
| nested public
| private
| public

See Partition I. A type that is not nested inside another type shall have exactly one visibility (`private` or `public`) and shall not have an accessiblity. Nested types shall have no visibility, but instead shall have exactly one of the accessibility attributes `nested assembly`, `nested famandassem`, `nested`

`family`, `nested famorassem`, `nested private`, or `nested public`. The default visibility for top-level types is `private`. The default accessibility for nested types is `nested private`.

10.1.2 Type layout attributes

ClassAttr `::= …`
`| auto`
`| explicit`
`| sequential`

The type layout specifies how the fields of an instance of a type are arranged. A given type shall have only one layout attribute specified. By convention, *ilasm* supplies `auto` if no layout attribute is specified. The layout attributes are:

`auto`: The layout shall be done by the CLI, with no user-supplied constraints.

`explicit`: The layout of the fields is explicitly provided (§10.7). However, a generic type shall not have explicit layout.

`sequential`: The CLI shall lay out the fields in sequential order, based on the order of the fields in the logical metadata table (§22.15).

[*Rationale:* The default `auto` layout should provide the best layout for the platform on which the code is executing. `sequential` layout is intended to instruct the CLI to match layout rules commonly followed by languages like C and C++ on an individual platform, where this is possible while still guaranteeing verifiable layout. `explicit` layout allows the CIL generator to specify the precise layout semantics. *end rationale*]

10.1.3 Type semantics attributes

ClassAttr `::= …`
`| interface`

The type semantic attributes specify whether an interface, class, or value type shall be defined. The `interface` attribute specifies an interface. If this attribute is not present and the definition extends (directly or indirectly) `System.ValueType`, and the definition is not for `System.Enum`, a value type shall be defined (§13). Otherwise, a class shall be defined (§11).

[*Example:*

```
.class interface public abstract auto ansi 'System.IComparable' { … }
```

`System.IComparable` is an interface because the interface attribute is present.

```
.class public sequential ansi serializable sealed beforefieldinit
    'System.Double' extends System.ValueType implements System.IComparable,
     … { … }
```

`System.Double` directly extends `System.ValueType`; `System.Double` is not the type `System.Enum`; so `System.Double` is a value type.

```
.class public abstract auto ansi serializable beforefieldinit 'System.Enum'
    extends System.ValueType implements System.IComparable, … { … }
```

Although `System.Enum` directly extends `System.ValueType`, `System.Enum` is not a value type, so it is a class.

```
.class public auto ansi serializable beforefieldinit 'System.Random'
    extends System.Object { … }
```

`System.Random` is a class because it is not an interface or a value type.

end example]

Note that the runtime size of a value type shall not exceed 1 MByte (0x100000 bytes)

10.1.4 Inheritance attributes

ClassAttr ::= …
| abstract
| sealed

Attributes that specify special semantics are `abstract` and `sealed`. These attributes can be used together.

`abstract` specifies that this type shall not be instantiated. If a type contains abstract methods, that type shall be declared as an abstract type.

`sealed` specifies that a type shall not have derived classes. All value types shall be sealed.

[*Rationale:* Virtual methods of sealed types are effectively instance methods, since they cannot be overridden. Framework authors should use sealed classes sparingly since they do not provide a convenient building block for user extensibility. Sealed classes can be necessary when the implementation of a set of virtual methods for a single class (typically multiple interfaces) becomes interdependent or depends critically on implementation details not visible to potential derived classes.

A type that is both `abstract` and `sealed` should have only static members, and serves as what some languages call a "namespace" or "static class". *end rationale*]

10.1.5 Interoperation attributes

ClassAttr ::= …
| ansi
| autochar
| unicode

These attributes are for interoperation with unmanaged code. They specify the default behavior to be used when calling a method (static, instance, or virtual) on the class, that has an argument or return type of `System.String` and does not itself specify marshalling behavior. Only one value shall be specified for any type, and the default value is `ansi`. The interoperation attributes are:

`ansi` specifies that marshalling shall be to and from ANSI strings.

`autochar` specifies marshalling behavior (either ANSI or Unicode), depending on the platform on which the CLI is running.

`unicode` specifies that marshalling shall be to and from Unicode strings.

In addition to these three attributes, §23.1.15 specifies an additional set of bit patterns (`CustomFormatClass` and `CustomStringFormatMask`), which have no standardized meaning. If these bits are set, but an implementation has no support for them, a `System.NotSupportedException` is thrown.

10.1.6 Special handling attributes

ClassAttr ::= …
| beforefieldinit
| rtspecialname
| serializable
| specialname

These attributes can be combined in any way.

`beforefieldinit` instructs the CLI that it need not initialize the type before a static method is called. See §10.5.3.

`rtspecialname` indicates that the name of this item has special significance to the CLI. There are no currently defined special type names; this is for future use. Any item marked `rtspecialname` shall also be marked `specialname`.

`serializable` Reserved for future use, to indicate that the fields of the type are to be serialized into a data stream (should such support be provided by the implementation).

`specialname` indicates that the name of this item can have special significance to tools other than the CLI. See, for example, Partition I .

[*Rationale:* If an item is treated specially by the CLI, then tools should also be made aware of that. The converse is not true. *end rationale*]

10.1.7 Generic parameters (*GenPars*)

Generic parameters are included when defining a generic type.

GenPars ::=
GenPar [`','` *GenPars*]

The *GenPar* non-terminal has the following production:

GenPar::=
[*GenParAttribs*]* [`'('` [*GenConstraints*] `')'`] *Id*

GenParAttribs::=
`'+'`
| `'-'`
| `class`
| `valuetype`
| `.ctor`

+ denotes a covariant generic parameter (§9.5).

– denotes a contravariant generic parameter (§9.5).

`class` is a special-purpose constraint that constrains *Id* to being a reference type. [*Note*: This includes type parameters which are themselves constrained to be reference types through a class or base type constraint. *end note*]

`valuetype` is a special-purpose constraint that constrains *Id* to being a value type, except that that type shall not be `System.Nullable<T>` or any concrete closed type of `System.Nullable<T>`. [*Note*: This includes type parameters which are themselves constrained to be value types. *end note*]

`.ctor` is a special-purpose constraint that constrains *Id* to being a concrete reference type (i.e., not abstract) that has a public constructor taking no arguments (the *default constructor*), or to being a value type. [*Note*: This includes type parameters which are, themselves, constrained either to be concrete reference types, or to being a value type. *end note*]

`class` and `valuetype` shall not both be specified for the same *Id*.

[*Example*:

```
.class C< + class .ctor (class System.IComparable<!0>) T > { … }
```

This declares a generic class `C<T>`, which has a covariant generic parameter named `T`. `T` is constrained such that it must implement `System.IComparable<T>`, and must be a concrete class with a public default constructor. *end example*]

Finally, the *GenConstraints* non-terminal has the following production:

GenConstraints ::=
Type [`',' GenConstraints`]

There shall be no duplicates of *Id* in the *GenPars* production.

[*Example*: Given appropriate definitions for interfaces `I1` and `I2`, and for class `Base`, the following code defines a class *Dict* that has two generic parameters, *K* and *V*, where *K* is constrained to implement both interfaces `I1` and `I2`, and *V* is constrained to derive from class `Base`:

```
.class Dict`2<(I1,I2)K, (Base)V> { … }
```

end example]

The following table shows the valid combinations of type and special constraints for a representative set of types. The first set of rows (Type Constraint `System.Object`) applies either when no base class constraint is specified or when the base class constraint is `System.Object`. The symbol ✓ means "set", the symbol **×** means "not set", and the symbol * means "either set or not set" or "don't care".

Type Constraint	Special Constraint			Meaning
	class	valuetype	.ctor	
(System.Object)	×	×	×	Any type
	✓	×	×	Any reference type
	✓	×	✓	Any reference type having a default constructor
	×	✓	*	Any value type **except** `System.Nullable<T>`
	×	×	✓	Any type with a public default constructor
	✓	✓	*	**Invalid**
System.ValueType	×	×	✓	Any value type **including** `System.Nullable<T>`
	×	✓	*	Any value type **except** `System.Nullable<T>`
	×	×	×	Any value type **and** `System.ValueType`, and `System.Enum`
	✓	×	×	`System.ValueType` and `System.Enum` **only**
	✓	×	✓	**Not meaningful**: Cannot be instantiated (no instantiable reference type can be derived from `System.ValueType`)
	✓	✓	*	**Invalid**

System.Enum	×	×	✓	Any enum type
	×	✓	*	
	×	×	×	Any enum type **and** `System.Enum`
	✓	×	×	System.Enum **only**
	✓	×	✓	**Not meaningful**: Cannot be instantiated (no instantiable reference type can be derived from `System.Enum`)
	✓	✓	*	**Invalid**
System.Exception (an example of any non-special reference Type)	×	×	×	`System.Exception`, or any class derived from `System.Exception`
	×	×	✓	Any `System.Exception` with a public default constructor
	✓	×	×	`System.Exception`, or any class derived from `System.Exception`. This is exactly the same result as if the class constraint was not specified
	✓	×	✓	Any Exception with a public default constructor.
	×	✓	*	**Not meaningful**: Cannot be instantiated (a value type cannot be derived from a reference type)
	✓	✓	*	**Invalid**
System.Delegate	×	×	×	`System.Delegate`, or any class derived from `System.Delegate`
	×	×	✓	**Not meaningful**: Cannot be instantiated (there is no default constructor)
	✓	×	×	`System.Delegate`, or any class derived from `System.Delegate`
	✓	×	✓	Any Delegate with a public .ctor. Invalid for known delegates (`System.Delegate`)
	×	✓	*	**Not meaningful**: Cannot be instantiated (a value type cannot be derived from a reference type)
	✓	✓	*	**Invalid**
System.Array	×	×	×	Any array
	*	×	✓	**Not meaningful**: Cannot be instantiated (no default constructor)
	✓	×	×	Any array
	×	✓	*	**Not meaningful**: Cannot be instantiated (a value type cannot be derived from a reference type)

	✓	✓	*	**Invalid**

[*Example*: The following instantiations are allowed or disallowed, based on the constraint. In all of these instances, the declaration itself is allowed. Items marked **Invalid** indicate where the attempt to instantiate the specified type fails verification, while those marked **Valid** do not.

```
.class public auto ansi beforefieldinit Bar`1<valuetype T>
```

Valid `ldtoken class Bar`1<int32>`

Invalid `ldtoken class Bar`1<class [mscorlib]System.Exception>`

Invalid `ldtoken class Bar`1<Nullable`1<int32>>`

Invalid `ldtoken class Bar`1<class [mscorlib]System.ValueType>`

```
.class public auto ansi beforefieldinit 'Bar`1'<class T>
```

Invalid `ldtoken class Bar`1<int32>`

Valid `ldtoken class Bar`1<class [mscorlib]System.Exception>`

Invalid `ldtoken class Bar`1<valuetype [mscorlib]System.Nullable`1<int32>>`

Valid `ldtoken class Bar`1<class [mscorlib]System.ValueType>`

```
.class public auto ansi beforefieldinit Bar`1<(class
       [mscorlib]System.ValueType) T>
```

Valid `ldtoken class Bar`1<int32>`

Invalid `ldtoken class Bar`1<class [mscorlib]System.Exception>`

Valid `ldtoken class Bar`1<valuetype [mscorlib]System.Nullable`1<int32>>`

Valid `ldtoken class Bar`1<class [mscorlib]System.ValueType>`

```
.class public auto ansi beforefieldinit Bar`1<class (int32)> T>
```

Invalid `ldtoken class Bar`1<int32>`

Invalid `ldtoken class Bar`1<class [mscorlib]System.Exception>`

Invalid `ldtoken class Bar`1<valuetype [mscorlib]System.Nullable`1<int32>>`

Invalid `ldtoken class Bar`1<class [mscorlib]System.ValueType>`

```
       Note: This type cannot be instantiated as no reference type can extend int32
.class public auto ansi beforefieldinit Bar`1<valuetype
              (class [mscorlib]System.Exception)> T>
```

Invalid `ldtoken class Bar`1<int32>`

Invalid `ldtoken class Bar`1<class [mscorlib]System.Exception>`

Invalid `ldtoken class Bar`1<valuetype [mscorlib]System.Nullable`1<int32>>`

Invalid `ldtoken class Bar`1<class [mscorlib]System.ValueType>`

```
       Note: This type cannot be instantiated as no value type can extend System.Exception
.class public auto ansi beforefieldinit Bar`1<.ctor (class Foo) T>
```

where `Foo` has no public .ctor, but `FooBar`, which derives from `Foo`, has a public .ctor:

Invalid `ldtoken class Bar`1<class Foo>`

Valid `ldtoken class Bar`1<class FooBar>`

end example]

10.2 Body of a type definition

A type can contain any number of further declarations. The directives `.event`, `.field`, `.method`, and `.property` are used to declare members of a type. The directive `.class` inside a type declaration is used to create a nested type, which is discussed in further detail in §10.6.

ClassMember `::=`	**Description**	**Clause**
`.class` *ClassHeader* `'{'` *ClassMember*`*` `'}'`	Defines a nested type.	10.6
`|` `.custom` *CustomDecl*	Custom attribute.	21
`|` `.data` *DataDecl*	Defines static data associated with the type.	16.3
`|` `.event` *EventHeader* `'{'` *EventMember*`*` `'}'`	Declares an event.	18
`|` `.field` *FieldDecl*	Declares a field belonging to the type.	16
`|` `.method` *MethodHeader* `'{'` *MethodBodyItem*`*` `'}'`	Declares a method of the type.	15
`|` `.override` *TypeSpec* `'::'` *MethodName* `with` *CallConv Type TypeSpec* `'::'` *MethodName* `'('` *Parameters* `')'`	Specifies that the first method is overridden by the definition of the second method.	10.3.2
`|` `.pack` *Int32*	Used for explicit layout of fields.	10.7
`|` `.param type` `'['` *Int32* `']'`	Specifies a type parameter for a generic type; for use in associating a custom attribute with that type parameter.	15.4.1.5
`|` `.property` *PropHeader* `'{'` *PropMember*`*` `'}'`	Declares a property of the type.	17
`|` `.size` *Int32*	Used for explicit layout of fields.	10.7
`|` *ExternSourceDecl*	Source line information.	5.7
`|` *SecurityDecl*	Declarative security permissions.	20

10.3 Introducing and overriding virtual methods

A virtual method of a base type is overridden by providing a direct implementation of the method (using a method definition, see §15.4) and not specifying it to be `newslot` (§15.4.2.3). An existing method body can also be used to implement a given virtual declaration using the `.override` directive (§10.3.2).

10.3.1 Introducing a virtual method

A virtual method is introduced in the inheritance hierarchy by defining a virtual method (§15.4). The definition can be marked `newslot` to always create a new virtual method for the defining class and any classes derived from it:

- If the definition is marked `newslot`, the definition always creates a new virtual method, even if a base class provides a matching virtual method. A reference to the virtual method via the class containing the method definition, or via a class derived from that class, refers to the new

definition (unless hidden by a `newslot` definition in a derived class). Any reference to the virtual method not via the class containing the method definition, nor via its derived classes, refers to the original definition.

- If the definition is not marked `newslot`, the definition creates a new virtual method only if there is not virtual method of the same name and signature inherited from a base class.

It follows that when a virtual method is marked `newslot`, its introduction will not affect any existing references to matching virtual methods in its base classes.

10.3.2 The .override directive

The `.override` directive specifies that a virtual method shall be implemented (overridden), in this type, by a virtual method with a different name, but with the same signature. This directive can be used to provide an implementation for a virtual method inherited from a base class, or a virtual method specified in an interface implemented by this type. The `.override` directive specifies a *Method Implementation* (MethodImpl) in the metadata (§15.1.4).

ClassMember `::=`	**Clause**
`.override` *TypeSpec* `'::'` *MethodName* `with` *CallConv Type TypeSpec* `'::'` *MethodName* `'('` *Parameters* `')'`	
`.override method` *CallConv Type TypeSpec* `'::'` *MethodName GenArity* `'('` *Parameters* `')'` `with method` *CallConv Type TypeSpec* `'::'` *MethodName GenArity* `'('` *Parameters* `')'`	
`|` …	10.2

GenArity `::= [ '<' '[' Int32 ']' '>' ]`

Int32 is the number of generic parameters.

The first *TypeSpec*`::`*MethodName* pair specifies the virtual method that is being overridden, and shall be either an inherited virtual method or a virtual method on an interface that the current type implements. The remaining information specifies the virtual method that provides the implementation.

While the syntax specified here (as well as the actual metadata format (§22.27)) allows any virtual method to be used to provide an implementation, a conforming program shall provide a virtual method actually implemented directly on the type containing the `.override` directive.

[*Rationale:* The metadata is designed to be more expressive than can be expected of all implementations of the VES. *end rationale*]

[*Example:* The following shows a typical use of the `.override` directive. A method implementation is provided for a method declared in an interface (see §12).

```
.class interface I
{ .method public virtual abstract void M() cil managed {}
}

.class C implements I
{ .method virtual public void M2()
  { // body of M2
  }
  .override I::M with instance void C::M2()
}
```

The `.override` directive specifies that the `C::M2` body shall provide the implementation of be used to implement `I::M` on objects of class `C`.

end example]

10.3.3 Accessibility and overriding

If the strict flag (§23.1.10) is specified then only accessible virtual methods can be overridden.

If a type overrides an inherited method through means other than a MethodImpl, it can *widen*, but it shall not *narrow*, the accessibility of that method. As a principle, if a client of a type is allowed to access a method of that type, then it should also be able to access that method (identified by name and signature) in any derived type. Table 7.1 specifies *narrow* and *widen* in this context—a "Yes" denotes that the derived class can apply that accessibility, a "No" denotes it is invalid.

If a type overrides an inherited method via a MethodImpl, it can *widen* or *narrow* the accessibility of that method.

Table 7.1: Valid Widening of Access to a Virtual Method

Derived class\Base type Accessibility	`Compiler-controlled`	`private`	`family`	`assembly`	`famandassem`	`famorassem`	`public`
`Compiler-controlled`	See note 3	No	No	No	No	No	No
`private`	See note 3	Yes	No	No	No	No	No
`family`	See note 3	Yes	Yes	No	Yes	See note 1	No
`assembly`	See note 3	Yes	No	See note 2	See note 2	No	No
`famandassem`	See note 3	Yes	No	No	See note 2	No	No
`famorassem`	See note 3	Yes	Yes	See note 2	Yes	Yes	No
`public`	See note 3	Yes	Yes	Yes	Yes	Yes	Yes

[1] Yes, provided both are in different assemblies; otherwise, No.

[2] Yes, provided both are in the same assembly; otherwise, No.

[3] Yes, provided both are in the same module; otherwise, No.

[*Note:* A method can be overridden even if it might not be accessed by the derived class.

If a method has `assembly` accessibility, then it shall have `public` accessibility if it is being overridden by a method in a different assembly. A similar rule applies to `famandassem`, where also `famorassem` is allowed outside the assembly. In both cases `assembly` or `famandassem`, respectively, can be used inside the same assembly. *end note*]

A special rule applies to `famorassem`, as shown in the table. This is the only case where the accessibility is apparently narrowed by the derived class. A `famorassem` method can be overridden with `family` accessibility by a type in another assembly.

[*Rationale:* Because there is no way to specify "family or specific other assembly" it is not possible to specify that the accessibility should be unchanged. To avoid narrowing access, it would be necessary to specify an accessibility of public, which would force widening of access even when it is not desired. As a compromise, the minor narrowing of "family" alone is permitted. *end rationale*]

10.4 Method implementation requirements

A type (concrete or abstract) can provide

- implementations for instance, static, and virtual methods that it introduces

- implementations for methods declared in interfaces that it has specified it will implement, or that its base type has specified it will implement
- alternative implementations for virtual methods inherited from its base class
- implementations for virtual methods inherited from an abstract base type that did not provide an implementation

A concrete (i.e., non-abstract) type shall provide, either directly or by inheritance, an implementation for

- all methods declared by the type itself
- all virtual methods of interfaces implemented by the type
- all virtual methods that the type inherits from its base type

10.5 Special members

There are three special members, all of which are methods that can be defined as part of a type: instance constructors, instance finalizers, and type initializers.

10.5.1 Instance constructor

An *instance constructor* initializes an instance of a type, and is called when an instance of a type is created by the newobj instruction (see Partition III). An instance constructor shall be an instance (not static or virtual) method, it shall be named `.ctor`, and marked `instance`, `rtspecialname`, and `specialname` (§15.4.2.6). An instance constructor can have parameters, but shall not return a value. An instance constructor cannot take generic type parameters. An instance constructor can be overloaded (i.e., a type can have several instance constructors). Each instance constructor for a type shall have a unique signature. Unlike other methods, instance constructors can write into fields of the type that are marked with the `initonly` attribute (§16.1.2).

[*Example:* The following shows the definition of an instance constructor that does not take any parameters:

```
.class X {
  .method public rtspecialname specialname instance void .ctor() cil managed
    { .maxstack 1
    // call super constructor
    ldarg.0         // load this pointer
    call instance void [mscorlib]System.Object::.ctor()
    // do other initialization work
    ret
  }
}
```

end example]

10.5.2 Instance finalizer

The behavior of finalizers is specified in Partition I. The finalize method for a particular type is specified by overriding the virtual method `Finalize` in `System.Object`.

10.5.3 Type initializer

A type (class, interface, or value type) can contain a special method called a *type initializer*, which is used to initialize the type itself. This method shall be static, take no parameters, return no value, be marked with `rtspecialname` and `specialname` (§15.4.2.6), and be named `.cctor`.

Like instance constructors, type initializers can write into static fields of their type that are marked with the `initonly` attribute (§16.1.2).

[*Example:* The following shows the definition of a type initializer:

```
.class public EngineeringData extends [mscorlib]System.Object
{
.field private static initonly float64[] coefficient
.method private specialname rtspecialname static void .cctor() cil managed
  {
  .maxstack 1

  // allocate array of 4 Double
  ldc.i4.4
  newarr     [mscorlib]System.Double
  // point initonly field to new array
  stsfld     float64[] EngineeringData::coefficient
  // code to initialize array elements goes here
  ret
  }
}
```

end example]

[*Note:* Type initializers are often simple methods that initialize the type's static fields from stored constants or via simple computations. There are, however, no limitations on what code is permitted in a type initializer. *end note*]

10.5.3.1 Type initialization guarantees

The CLI shall provide the following guarantees regarding type initialization (but see also §10.5.3.2 and §10.5.3.3):

1. As to when type initializers are executed is specified in Partition I.
2. A type initializer shall be executed exactly once for any given type, unless explicitly called by user code.
3. No methods other than those called directly or indirectly from the type initializer are able to access members of a type before its initializer completes execution.

10.5.3.2 Relaxed guarantees

A type can be marked with the attribute `beforefieldinit` (§10.1.6) to indicate that the guarantees specified in §10.5.3.1 are not necessarily required. In particular, the final requirement above need not be provided: the type initializer need not be executed before a static method is called or referenced.

[*Rationale:* When code can be executed in multiple application domains it becomes particularly expensive to ensure this final guarantee. At the same time, examination of large bodies of managed code have shown that this final guarantee is rarely required, since type initializers are almost always simple methods for initializing static fields. Leaving it up to the CIL generator (and hence, possibly, to the programmer) to decide whether this guarantee is required therefore provides efficiency when it is desired at the cost of consistency guarantees. *end rationale*]

10.5.3.3 Races and deadlocks

In addition to the type initialization guarantees specified in §10.5.3.1, the CLI shall ensure two further guarantees for code that is called from a type initializer:

1. Static variables of a type are in a known state prior to any access whatsoever.
2. Type initialization alone shall not create a deadlock unless some code called from a type initializer (directly or indirectly) explicitly invokes blocking operations.

[*Rationale:* Consider the following two class definitions:

```
.class public A extends [mscorlib]System.Object
{ .field static public class A a
  .field static public class B b
  .method public static rtspecialname specialname void .cctor ()
  { ldnull                 // b=null
    stsfld class B A::b
    ldsfld class A B::a    // a=B.a
    stsfld class A A::a
    ret
  }
}

.class public B extends [mscorlib]System.Object
{ .field static public class A a
  .field static public class B b
  .method public static rtspecialname specialname void .cctor ()
  { ldnull                 // a=null
    stsfld class A B::a
    ldsfld class B A::b    // b=A.b
    stsfld class B B::b
    ret
  }
}
```

After loading these two classes, an attempt to reference any of the static fields causes a problem, since the type initializer for each of A and B requires that the type initializer of the other be invoked first. Requiring that no access to a type be permitted until its initializer has completed would create a deadlock situation. Instead, the CLI provides a weaker guarantee: the initializer will have started to run, but it need not have completed. But this alone would allow the full uninitialized state of a type to be visible, which would make it difficult to guarantee repeatable results.

There are similar, but more complex, problems when type initialization takes place in a multi-threaded system. In these cases, for example, two separate threads might start attempting to access static variables of separate types (A and B) and then each would have to wait for the other to complete initialization.

A rough outline of an algorithm to ensure points 1 and 2 above is as follows:

1. At class load-time (hence prior to initialization time) store zero or null into all static fields of the type.

2. If the type is initialized, you are done.

2.1. If the type is not yet initialized, try to take an initialization lock.

2.2. If successful, record this thread as responsible for initializing the type and proceed to step 2.3.

2.2.1. If not successful, see whether this thread or any thread waiting for this thread to complete already holds the lock.

2.2.2. If so, return since blocking would create a deadlock. This thread will now see an incompletely initialized state for the type, but no deadlock will arise.

2.2.3 If not, block until the type is initialized then return.

2.3 Initialize the base class type and then all interfaces implemented by this type.

2.4 Execute the type initialization code for this type.

2.5 Mark the type as initialized, release the initialization lock, awaken any threads waiting for this type to be initialized, and return.

end rationale]

10.6 Nested types

Nested types are specified in Partition I. For information about the logical tables associated with nested types, see §22.32.

[*Note:* A nested type is not associated with an instance of its enclosing type. The nested type has its own base type, and can be instantiated independently of its enclosing type. This means that the instance members of the enclosing type are not accessible using the `this` pointer of the nested type.

A nested type can access any members of its enclosing type, including private members, as long as those members are static or the nested type has a reference to an instance of the enclosing type. Thus, by using nested types, a type can give access to its private members to another type.

On the other hand, the enclosing type cannot access any private or family members of the nested type. Only members with `assembly`, `famorassem`, or `public` accessibility can be accessed by the enclosing type. *end note*]

[*Example:* The following shows a class declared inside another class. Each class declares a field. The nested class can access both fields, while the enclosing class does not have access to the enclosed class's field `b`.

```
.class public auto ansi X
{ .field static private int32 a
  .class auto ansi nested public Y
  { .field static private int32 b
    // ...
  }
}
```

end example]

10.7 Controlling instance layout

The CLI supports both sequential and explicit layout control, see § 10.1.2. For explicit layout it is also necessary to specify the precise layout of an instance; see also §22.18 and §22.16.

FieldDecl ::=
['[' *Int32* ']'] *FieldAttr** *Type* *Id*

The optional `int32` specified in brackets at the beginning of the declaration specifies the byte offset from the beginning of the instance of the type. (For a given type *t*, this beginning refers to the start of the set of members explicitly defined in type *t*, excluding all members defined in any types from which type *t* directly or indirectly inherits.) This form of explicit layout control shall not be used with global fields specified using the `at` notation §16.3.2).

Offset values shall be non-negative. It is possible to overlap fields in this way, though offsets occupied by an object reference shall not overlap with offsets occupied by a built-in value type or a part of another object reference. While one object reference can completely overlap another, this is unverifiable.

Fields can be accessed using pointer arithmetic and ldind to load the field indirectly or stind to store the field indirectly (see Partition III). See §22.16 and §22.18 for encoding of this information. For explicit layout, every field shall be assigned an offset.

The `.pack` directive specifies that fields should be placed within the runtime object at byte addresses which are a multiple of the specified number, or at natural alignment for that field type, whichever is *smaller*. For example, `.pack 2` would allow 32-bit-wide fields to be started on even addresses, whereas without any `.pack` directive, they would be naturally aligned; that is, placed on addresses that are a multiple of 4. The integer following `.pack` shall be one of the following: 0, 1, 2, 4, 8, 16, 32, 64, or 128. (A value of zero indicates that the pack size used should match the default for the current platform.) The `.pack` directive shall not be supplied for any type with explicit layout control.

The `.size` directive indicates a minimum size, and is intended to allow for padding. Therefore, the amount of memory allocated is the maximum of the size calculated from the layout and the `.size` directive. Note that if this directive applies to a value type, then the size shall be less than 1 MByte.

[*Note:* Metadata that controls instance layout is not a "hint," it is an integral part of the VES that shall be supported by all conforming implementations of the CLI. *end note*]

[*Example:* The following class uses sequential layout of its fields:

```
.class sequential public SequentialClass
{ .field public int32 a          // store at offset 0 bytes
  .field public int32 b          // store at offset 4 bytes
}
```

The following class uses explicit layout of its fields:

```
.class explicit public ExplicitClass
{ .field [0] public int32 a      // store at offset 0 bytes
  .field [6] public int32 b      // store at offset 6 bytes
}
```

The following value type uses `.pack` to pack its fields together:

```
.class value sealed public MyClass extends [mscorlib]System.ValueType
{ .pack 2
  .field  public int8  a   // store at offset 0 bytes
  .field  public int32 b   // store at offset 2 bytes (not 4)
}
```

The following class specifies a contiguous block of 16 bytes:

```
.class public BlobClass
{ .size 16
}
```

end example]

10.8 Global fields and methods

In addition to types with static members, many languages have the notion of data and methods that are not part of a type at all. These are referred to as *global* fields and methods.

The simplest way to understand global fields and methods in the CLI is to imagine that they are simply members of an invisible `abstract` public class. In fact, the CLI defines such a special class, named `<Module>`, that does not have a base type and does not implement any interfaces. (This class is a top-level class; i.e., it is not nested.)The only noticeable difference is in how definitions of this special class are treated when multiple modules are combined together, as is done by a class loader. This process is known as *metadata merging*.

For an ordinary type, if the metadata merges two definitions of the same type, it simply discards one definition on the assumption they are equivalent, and that any anomaly will be discovered when the type is used. For the special class that holds global members, however, members are unioned across all modules at merge time. If the same name appears to be defined for cross-module use in multiple modules then there is an error. In detail:

- If no member of the same kind (field or method), name, and signature exists, then add this member to the output class.
- If there are duplicates and no more than one has an accessibility other than `compilercontrolled`, then add them all to the output class.
- If there are duplicates and two or more have an accessibility other than `compilercontrolled`, an error has occurred.

[*Note*: Strictly speaking, the CLI does not support global statics, even though global fields and methods might be thought of as such. All global fields and methods in a module are owned by the manufactured class `"<Module>"`. However, each module has its own `"<Module>"` class. There's no way to even refer, early-bound, to such a global field or method in another module. (You can, however, "reach" them, late-bound, via Reflection.) *end note*]

11 Semantics of classes

Classes, as specified in Partition I, define types in an inheritance hierarchy. A class (except for the built-in class `System.Object` and the special class `<Module>`) shall declare exactly one base class. A class shall declare zero or more interfaces that it implements (§12). A concrete class can be instantiated to create an object, but an `abstract` class (§10.1.4) shall not be instantiated. A class can define fields (static or instance), methods (static, instance, or virtual), events, properties, and nested types (classes, value types, or interfaces).

Instances of a class (i.e., objects) are created only by explicitly using the newobj instruction (see Partition III). When a variable or field that has a class as its type is created (for example, by calling a method that has a local variable of a class type), the value shall initially be null, a special value that := with all class types even though it is not an instance of any particular class.

12 Semantics of interfaces

Interfaces, as specified in Partition I, each define a contract that other types can implement. Interfaces can have static fields and methods, but they shall not have instance fields or methods. Interfaces can define virtual methods, but only if those methods are `abstract` (see Partition I and §15.4.2.4).

[*Rationale:* Interfaces cannot define instance fields for the same reason that the CLI does not support multiple inheritance of base types: in the presence of dynamic loading of data types there is no known implementation technique that is both efficient when used and has no cost when not used. By contrast, providing static fields and methods need not affect the layout of instances and therefore does not raise these issues. *end rationale*]

Interfaces can be nested inside any type (interface, class, or value type).

12.1 Implementing interfaces

Classes and value types shall *implement* zero or more interfaces. Implementing an interface implies that all concrete instances of the class or value type shall provide an implementation for each `abstract` virtual method declared in the interface. In order to implement an interface, a class or value type shall either explicitly declare that it does so (using the `implements` attribute in its type definition, see §10.1) or shall be derived from a base class that implements the interface.

[*Note:* An `abstract` class (since it cannot be instantiated) need not provide implementations of the virtual methods of interfaces it implements, but any concrete class derived from it shall provide the implementation.

Merely providing implementations for all of the `abstract` methods of an interface is not sufficient to have a type implement that interface. Conceptually, this represents the fact that an interface represents a contract that can have more requirements than are captured in the set of `abstract` methods. From an implementation point of view, this allows the layout of types to be constrained only by those interfaces that are explicitly declared. *end note*]

Interfaces shall declare that they require the implementation of zero or more other interfaces. If one interface, A, declares that it requires the implementation of another interface, B, then A implicitly declares that it requires the implementation of all interfaces required by B. If a class or value type declares that it implements A, then all concrete instances shall provide implementations of the virtual methods declared in A and all of the interfaces A requires. [*Note*: The class need not explicitly declare that it implements the interfaces required by A. *end note*]

[*Example:* The following class implements the interface `IStartStopEventSource` defined in the module `Counter`.

```
.class private auto autochar StartStopButton
       extends [System.Windows.Forms]System.Windows.Forms.Button
       implements [.module Counter]IstartStopEventSource
{ // body of class
}
```

end example]

12.2 Implementing virtual methods on interfaces

Classes that implement an interface (§12.1) are required to provide implementations for the `abstract` virtual methods defined by that interface. There are three mechanisms for providing this implementation:

- Directly specifying an implementation, using the same name and signature as appears in the interface.
- Inheritance of an existing implementation from the base type.
- Use of an explicit `MethodImpl` (§15.1.4).

The VES shall use the following algorithm to determine the appropriate implementation of an interface's virtual abstract methods:

- If the base class implements the interface, start with the same virtual methods that it provides; otherwise, create an interface that has empty slots for all virtual functions.
- If this class explicitly specifies that it implements the interface (i.e., the interfaces that appear in this class's InterfaceImpl table, §22.23)
    - If the class defines any `public virtual newslot` methods whose name and signature match a virtual method on the interface, then use these new virtual methods to implement the corresponding interface method.
- If there are any virtual methods in the interface that still have empty slots, see if there are any `public virtual` methods, but not `public virtual newslot` methods, available on this class (directly or inherited) having the same name and signature, then use these to implement the corresponding methods on the interface.
- Apply all *MethodImpl*s that are specified for this class, thereby placing explicitly specified virtual methods into the interface in preference to those inherited or chosen by name matching.
- If the current class is not `abstract` and there are any interface methods that still have empty slots, then the program is invalid.

[*Rationale:* Interfaces can be thought of as specifying, primarily, a set of virtual methods that shall be implemented by any class that implements the interface. The class specifies a mapping from its own virtual methods to those of the interface. Thus it is virtual methods, not specific implementations of those methods that are associated with interfaces. Overriding a virtual method on a class with a specific implementation will thus affect not only the virtual method named in the class but also any interface virtual methods to which that same virtual method has been mapped. *end rationale*]

13 Semantics of value types

In contrast to reference types, value types (see Partition I) are not accessed by using a reference, but are stored directly in the location of that type.

[*Rationale:* Value types are used to describe the type of small data items. They can be compared to struct (as opposed to pointers to struct) types in C++. Compared to reference types, value types are accessed faster since there is no additional indirection involved. As elements of arrays they do not require allocating memory for the pointers as well as for the data itself. Typical value types are complex numbers, geometric points, and dates. *end rationale*]

Like other types, value types can have fields (static or instance), methods (static, instance, or virtual), properties, events, and nested types. A value of some value type can be converted into an instance of a corresponding reference type (its *boxed form*, a class automatically created for this purpose by the VES when a value type is defined) by a process called *boxing*. A boxed value type can be converted back into its value type representation, the *unboxed form*, by a process called *unboxing*. Value types shall be sealed, and they shall have a base type of either `System.ValueType` or `System.Enum` (see Partition IV). Value types shall implement zero or more interfaces, but this has meaning only in their boxed form (§13.3).

Unboxed value types are not considered subtypes of another type and it is not valid to use the isinst instruction (see Partition III) on unboxed value types. The isinst instruction can be used for boxed value types, however. Unboxed value types shall not be assigned the value *null* and they shall not be compared to *null*.

Value types support layout control in the same way as do reference types (§10.7). This is especially important when values are imported from native code.

Since ValueTypes represent direct layout of data, recursive struct definitions such as (in C#) `struct S {S x; S y;}` are not permitted. A struct shall have an acyclic finite **flattening graph**:

For a value type S, define the flattening graph G of S to be the smallest directed graph such that:

- S is in G.
- Whenever T is in G and T has an instance field of value type X then X is in G and there is an edge from T to X.
- Whenever T is in G and T has a static field of value type Y then Y is in G.

[*Example*:

```
class C<U> { }

struct S1<V> {
  S1<V> x;
}

struct S2<V> {
  static S2<V> x;
}

struct S3<V> {
  static S3<C<V>> x;
}

struct S4<V> {
  S4<C<V>>[] x;
}
```

Struct type `S1` has a finite but cyclic flattening graph and is invalid; `S2` has a finite acyclic flattening graph and is valid; `S3` has an infinite acyclic flattening graph and is invalid; `S4` has a finite acyclic flattening graph and is valid because field `S4<C<V>>.x` has reference type, not value type.

The `C<U>` type is not strictly necessary for the examples, but if it were not used, it might be unclear whether something like the following

```
struct S3<V> {
  static S3<S3<V>> x;
}
```

is problematic due to the inner or the outer occurrence of `S3<...>` in the field type. *end example*]

13.1 Referencing value types

The unboxed form of a value type shall be referred to by using the `valuetype` keyword followed by a type reference. The boxed form of a value type shall be referred to by using the `boxed` keyword followed by a type reference.

ValueTypeReference `::=`

`boxed` *TypeReference*

`|` `valuetype` *TypeReference*

13.2 Initializing value types

Like classes, value types can have both instance constructors (§10.5.1) and type initializers (§10.5.3). Unlike classes, whose fields are automatically initialized to null, the following rules constitute the only guarantee about the initilization of (unboxed) value types:

- Static variables shall be initialized to zero when a type is loaded (§10.5.3.3), hence statics whose type is a value type are zero-initialized when the type is loaded.
- Local variables shall be initialized to zero if the `localsinit` bit in the method header (§25.4.4) is set.
- Arrays shall be zero-initialized.
- Instances of classes (i.e., objects) shall be zero-initialized prior to calling their instance constructor.

[*Rationale:* Guaranteeing automatic initialization of unboxed value types is both difficult and expensive, especially on platforms that support thread-local storage and that allow threads to be created outside of the CLI and then passed to the CLI for management. *end rationale*]

[*Note:* Boxed value types are classes and follow the rules for classes. *end note*]

The instruction initobj (see Partition III) performs zero-initialization under program control. If a value type has a constructor, an instance of its unboxed type can be created as is done with classes. The newobj instruction (see Partition III) is used along with the initializer and its parameters to allocate and initialize the instance. The instance of the value type will be allocated on the stack. The Base Class Library provides the method `System.Array.Initialize` (see Partition IV) to zero all instances in an array of unboxed value types.

[*Example:* The following code declares and initializes three value type variables. The first variable is zero-initialized, the second is initialized by calling an instance constructor, and the third by creating the object on the stack and storing it into the local.

```
.assembly Test { }
.assembly extern System.Drawing {
  .ver 1:0:3102:0
  .publickeytoken = (b03f5f7f11d50a3a)
}

.method public static void Start()
{ .maxstack 3
  .entrypoint
  .locals init (valuetype [System.Drawing]System.Drawing.Size Zero,
          valuetype [System.Drawing]System.Drawing.Size Init,
          valuetype [System.Drawing]System.Drawing.Size Store)
```

```
    // Zero initialize the local named Zero
    ldloca Zero           // load address of local variable
    initobj valuetype [System.Drawing]System.Drawing.Size

    // Call the initializer on the local named Init
    ldloca Init           // load address of local variable
    ldc.i4 425            // load argument 1 (width)
    ldc.i4 300            // load argument 2 (height)
    call instance void [System.Drawing]System.Drawing.Size::.ctor(int32, int32)

    // Create a new instance on the stack and store into Store.  Note that
    // stobj is used here - but one could equally well  use stloc, stfld, etc.
    ldloca Store
    ldc.i4 425            // load argument 1 (width)
    ldc.i4 300            // load argument 2 (height)
    newobj instance void [System.Drawing]System.Drawing.Size::.ctor(int32, int32)
    stobj valuetype [System.Drawing]System.Drawing.Size
    ret
}
```

end example]

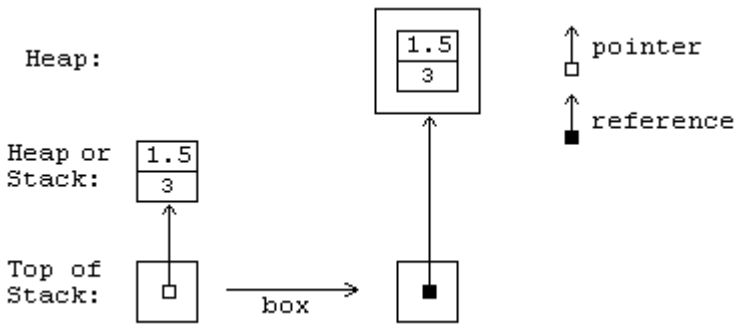
13.3 Methods of value types

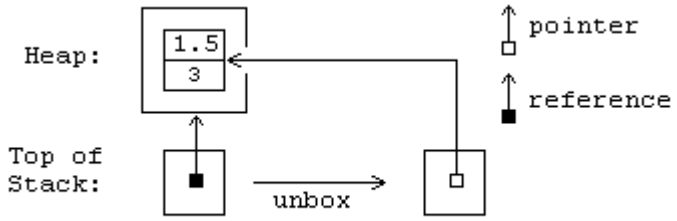
Value types can have static, instance and virtual methods. Static methods of value types are defined and called the same way as static methods of class types. As with classes, both instance and virtual methods of a boxed or unboxed value type can be called using the call instruction. The callvirt instruction shall not be used with unboxed value types (see Partition I), but it can be used on boxed value types.

Instance and virtual methods of classes shall be coded to expect a reference to an instance of the class as the *this* pointer. By contrast, instance and virtual methods of value types shall be coded to expect a managed pointer (see Partition I) to an unboxed instance of the value type. The CLI shall convert a boxed value type into a managed pointer to the unboxed value type when a boxed value type is passed as the *this* pointer to a virtual method whose implementation is provided by the unboxed value type.

[*Note:* This operation is the same as unboxing the instance, since the unbox instruction (see Partition III) is defined to return a managed pointer to the value type that shares memory with the original boxed instance.

The following diagrams are intended to help the reader understand the relationship between the boxed and unboxed representations of a value type.





end note]

[*Rationale:* An important use of instance methods on value types is to change internal state of the instance. This cannot be done if an instance of the unboxed value type is used for the this pointer, since it would be operating on a copy of the value, not the original value: unboxed value types are copied when they are passed as arguments.

Virtual methods are used to allow multiple types to share implementation code, and this requires that all classes that implement the virtual method share a common representation defined by the class that first introduces the method. Since value types can (and in the Base Class Library do) implement interfaces and virtual methods defined on `System.Object`, it is important that the virtual method be callable using a boxed value type so it can be manipulated as would any other type that implements the interface. This leads to the requirement that the EE automatically unbox value types on virtual calls. *end rationale*]

Table 1: Type of *this* given the CIL instruction and the declaring type of instance method.

	Value Type (Boxed or Unboxed)	**Interface**	**Object Type**
call	managed pointer to value type	invalid	object reference
callvirt	managed pointer to value type	object reference	object reference

[*Example:* The following converts an integer of the value type `int32` into a string. Recall that `int32` corresponds to the unboxed value type `System.Int32` defined in the Base Class Library. Suppose the integer is declared as:

```
.locals init (int32 x)
```

Then the call is made as shown below:

```
ldloca x            // load managed pointer to local variable
call instance string valuetype [mscorlib]System.Int32::ToString()
```

However, if `System.Object` (a class) is used as the type reference rather than `System.Int32` (a value type), the value of `x` shall be boxed before the call is made and the code becomes:

```
ldloc x
box valuetype [mscorlib]System.Int32
callvirt instance string [mscorlib]System.Object::ToString()
```

end example]

14 Semantics of special types

Special types are those that are referenced from CIL, but for which no definition is supplied: the VES supplies the definitions automatically based on information available from the reference.

14.1 Vectors

Type ::= …
| *Type* '[' ']'

Vectors are single-dimension arrays with a zero lower bound. They have direct support in CIL instructions (newarr, ldelem, stelem, and ldelema, see Partition III). The CIL Framework also provides methods that deal with multidimensional arrays and single-dimension arrays with a non-zero lower bound (§14.2). Two vectors have the same type if their element types are the same, regardless of their actual upper bounds.

Vectors have a fixed size and element type, determined when they are created. All CIL instructions shall respect these values. That is, they shall reliably detect attempts to do the following: index beyond the end of the vector, store the incorrect type of data into an element of a vector, and take the address of elements of a vector with an incorrect data type. See Partition III.

[*Example:* Declare a vector of Strings:

```
.field string[] errorStrings
```

Declare a vector of function pointers:

```
.field method instance void*(int32) [] myVec
```

Create a vector of 4 strings, and store it into the field `errorStrings`. The 4 strings lie at `errorStrings[0]` through `errorStrings[3]`:

```
ldc.i4.4
newarr string
stfld string[] CountDownForm::errorStrings
```

Store the string `"First"` into `errorStrings[0]`:

```
ldfld string[] CountDownForm::errorStrings
ldc.i4.0
ldstr "First"
stelem
```

end example]

Vectors are subtypes of `System.Array`, an abstract class pre-defined by the CLI. It provides several methods that can be applied to all vectors. See Partition IV.

14.2 Arrays

While vectors (§14.1) have direct support through CIL instructions, all other arrays are supported by the VES by creating subtypes of the abstract class `System.Array` (see Partition IV)

Type ::= …
| *Type* '[' [*Bound* [',' *Bound*]*] ']'

The *rank* of an array is the number of dimensions. The CLI does not support arrays with rank 0. The type of an array (other than a vector) shall be determined by the type of its elements and the number of dimensions.

Bound **::=**	**Description**
'...'	Lower and upper bounds unspecified. In the case of multi-dimensional arrays, the ellipsis can be omitted

| *Int32*	Zero lower bound, *Int32* upper bound
| *Int32* `'...'`	Lower bound only specified
| *Int32* `'...'` *Int32*	Both bounds specified

The class that the VES creates for arrays contains several methods whose implementation is supplied by the VES:

- A constructor that takes a sequence of `int32` arguments, one for each dimension of the array, that specify the number of elements in each dimension beginning with the first dimension. A lower bound of zero is assumed.
- A constructor that takes twice as many `int32` arguments as there are dimensions of the array. These arguments occur in pairs—one pair per dimension—with the first argument of each pair specifying the lower bound for that dimension, and the second argument specifying the total number of elements in that dimension. Note that vectors are not created with this constructor, since a zero lower bound is assumed for vectors.
- A `Get` method that takes a sequence of `int32` arguments, one for each dimension of the array, and returns a value whose type is the element type of the array. This method is used to access a specific element of the array where the arguments specify the index into each dimension, beginning with the first, of the element to be returned.
- A `Set` method that takes a sequence of `int32` arguments, one for each dimension of the array, followed by a value whose type is the element type of the array. The return type of `Set` is `void`. This method is used to set a specific element of the array where the arguments specify the index into each dimension, beginning with the first, of the element to be set and the final argument specifies the value to be stored into the target element.
- An `Address` method that takes a sequence of `int32` arguments, one for each dimension of the array, and has a return type that is a managed pointer to the array's element type. This method is used to return a managed pointer to a specific element of the array where the arguments specify the index into each dimension, beginning with the first, of the element whose address is to be returned.

[*Example:* The following creates an array, `MyArray`, of strings with two dimensions, with indexes 5…10 and 3…7. It then stores the string `"One"` into `MyArray[5, 3]`, retrieves it and prints it out. Then it computes the address of `MyArray[5, 4]`, stores `"Test"` into it, retrieves it, and prints it out.

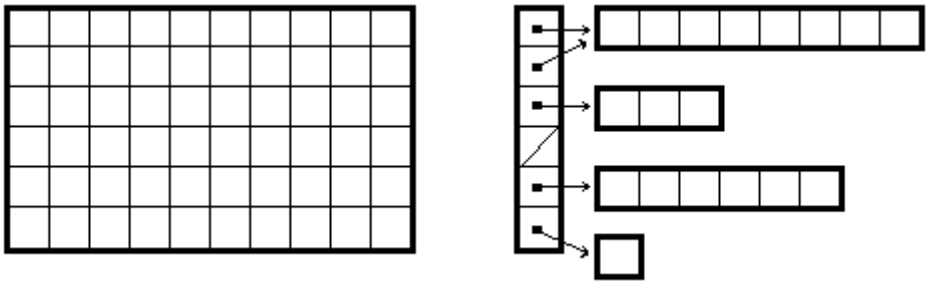
```
.assembly Test { }
.assembly extern mscorlib { }

.method public static void Start()
{ .maxstack 5
  .entrypoint
  .locals (class [mscorlib]System.String[,] myArray)

  ldc.i4.5   // load lower bound for dim 1
  ldc.i4.6   // load (upper bound - lower bound + 1) for dim 1
  ldc.i4.3   // load lower bound for dim 2
  ldc.i4.5   // load (upper bound - lower bound + 1) for dim 2
  newobj instance void string[,]::.ctor(int32, int32, int32, int32)
  stloc  myArray

  ldloc myArray
  ldc.i4.5
  ldc.i4.3
  ldstr "One"
  call instance void string[,]::Set(int32, int32, string)
```

```
    ldloc myArray
    ldc.i4.5
    ldc.i4.3
    call instance string string[,]::Get(int32, int32)
    call void [mscorlib]System.Console::WriteLine(string)

    ldloc myArray
    ldc.i4.5
    ldc.i4.4
    call instance string & string[,]::Address(int32, int32)
    ldstr "Test"
    stind.ref

    ldloc myArray
    ldc.i4.5
    ldc.i4.4
    call instance string string[,]::Get(int32, int32)
    call void [mscorlib]System.Console::WriteLine(string)
    ret
  }
```

end example]

The following text is informative

Whilst the elements of multi-dimensional arrays can be thought of as laid out in contiguous memory, arrays of arrays are different – each dimension (except the last) holds an array reference. The following picture illustrates the difference:



On the left is a [6, 10] rectangular array. On the right is not one, but a total of five arrays. The vertical array is an array of arrays, and references the four horizontal arrays. Note how the first and second elements of the vertical array both reference the same horizontal array.

Note that all dimensions of a multi-dimensional array shall have the same size. But in an array of arrays, it is possible to reference arrays of different sizes. For example, the figure on the right shows the vertical array referencing arrays of lengths 8, 8, 3, null (i.e., no array), 6 and 1, respectively.

There is no special support for these so-called *jagged arrays* in either the CIL instruction set or the VES. They are simply vectors whose elements reference other (recursively) jagged arrays.

End of informative text

14.3 Enums

An *enum* (short for *enumeration*) defines a set of symbols that all have the same type. A type shall be an enum if and only if it has an immediate base type of `System.Enum`. Since `System.Enum` itself has an immediate base type of `System.ValueType`, (see Partition IV) enums are value types (§13) The symbols of an enum are represented by an *underlying* integer type: one of { `bool`, `char`, `int8`, `unsigned int8`, `int16`, `unsigned int16`, `int32`, `unsigned int32`, `int64`, `unsigned int64`, `native int`, `unsigned native int` }

[*Note:* Unlike Pascal, the CLI does *not* provide a guarantee that values of the enum type are integers corresponding to one of the symbols. In fact, the CLS (see Partition I, CLS) defines a convention for using

enums to represent bit flags which can be combined to form integral value that are not named by the enum type itself. *end note*]

Enums obey additional restrictions beyond those on other value types. Enums shall contain only fields as members (they shall not even define type initializers or instance constructors); they shall not implement any interfaces; they shall have auto field layout (§10.1.2); they shall have exactly one instance field and it shall be of the underlying type of the enum; all other fields shall be static and literal (§16.1); and they shall not be initialized with the initobj instruction.

[*Rationale:* These restrictions allow a very efficient implementation of enums. *end rationale*]

The single, required, instance field stores the value of an instance of the enum. The static literal fields of an enum declare the mapping of the symbols of the enum to the underlying values. All of these fields shall have the type of the enum and shall have field init metadata that assigns them a value (§16.2).

For binding purposes (e.g., for locating a method definition from the method reference used to call it) enums shall be distinct from their underlying type. For all other purposes, including verification and execution of code, an unboxed enum freely interconverts with its underlying type. Enums can be boxed (§13) to a corresponding boxed instance type, but this type is *not* the same as the boxed type of the underlying type, so boxing does not lose the original type of the enum.

[*Example:* Declare an enum type and then create a local variable of that type. Store a constant of the underlying type into the enum (showing automatic coersion from the underlying type to the enum type). Load the enum back and print it as the underlying type (showing automatic coersion back). Finally, load the address of the enum and extract the contents of the instance field and print that out as well.

```
.assembly Test { }
.assembly extern mscorlib { }

.class sealed public ErrorCodes extends [mscorlib]System.Enum
{ .field public unsigned int8 MyValue
  .field public static literal valuetype ErrorCodes no_error = int8(0)
  .field public static literal valuetype ErrorCodes format_error = int8(1)
  .field public static literal valuetype ErrorCodes overflow_error = int8(2)
  .field public static literal valuetype ErrorCodes nonpositive_error = int8(3)
}

.method public static void Start()
{ .maxstack 5
  .entrypoint
  .locals init (valuetype ErrorCodes errorCode)

  ldc.i4.1           // load 1 (= format_error)
  stloc errorCode    // store in local, note conversion to enum
  ldloc errorCode
  call void [mscorlib]System.Console::WriteLine(int32)
  ldloca errorCode   // address of enum
  ldfld unsigned int8 valuetype ErrorCodes::MyValue
  call void [mscorlib]System.Console::WriteLine(int32)
  ret
}
```

end example]

14.4 Pointer types

Type ::= …	**Clause**
| *Type* '&'	14.4.2
| *Type* '*'	14.4.1

A *pointer type* shall be defined by specifying a signature that includes the type of the location at which it points. A *pointer* can be *managed* (reported to the CLI garbage collector, denoted by &, see §14.4.2) or *unmanaged* (not reported, denoted by *, see §14.4.1)

Pointers can contain the address of a field (of an object or value type) or of an element of an array. *Pointers* differ from object references in that they do not point to an entire type instance, but, rather, to the *interior* of an instance. The CLI provides two type-safe operations on pointers:

- *Loading* the value from the location referenced by the pointer.
- *Storing* an assignment-compatible value into the location referenced by the pointer.

For pointers into the same array or object (see Partition I) the following arithmetic operations are supported:

- Adding an integer value to a pointer (where that value is interpreted as a number of bytes), which results in a pointer of the same kind
- Subtracting an integer value from a pointer (where that value is interpreted as a number of bytes), which results in a pointer of the same kind. Note that subtracting a pointer from an integer value is not permitted.
- Two pointers, regardless of kind, can be subtracted from one another, producing an integer value that specifies the number of bytes between the addresses they reference.

The following is informative text

Pointers are compatible with `unsigned int32` on 32-bit architectures, and with `unsigned int64` on 64-bit architectures. They are best considered as `unsigned int`, whose size varies depending upon the runtime machine architecture.

The CIL instruction set (see Partition III) contains instructions to compute addresses of fields, local variables, arguments, and elements of vectors:

Instruction	Description
ldarga	Load address of argument
ldelema	Load address of vector element
ldflda	Load address of field
ldloca	Load address of local variable
ldsflda	Load address of static field

Once a pointer is loaded onto the stack, the ldind class of instructions can be used to load the data item to which it points. Similarly, the stind family of instructions can be used to store data into the location.

Note that the CLI will throw an `InvalidOperationException` for an ldflda instruction if the address is not within the current application domain. This situation arises typically only from the use of objects with a base type of `System.MarshalByRefObject` (see Partition IV).

14.4.1 Unmanaged pointers

Unmanaged pointers (`*`) are the traditional pointers used in languages like C and C++. There are no restrictions on their use, although, for the most part, they result in code that cannot be verified. While it is perfectly valid to mark locations that contain unmanaged pointers as though they were unsigned integers (and this is, in fact, how they are treated by the VES), it is often better to mark them as unmanaged pointers to a specific type of data. This is done by using `*`in a signature for a return value, local variable, or an argument, or by using a pointer type for a field or array element.

- Unmanaged pointers are not reported to the garbage collector and can be used in any way that an integer can be used.
- Verifiable code cannot dereference unmanaged pointers.
- Unverified code can pass an unmanaged pointer to a method that expects a managed pointer. This is safe only if one of the following is true:

a. The unmanaged pointer refers to memory that is not in memory used by the CLI for storing instances of objects ("garbage-collected memory" or "managed memory").

b. The unmanaged pointer contains the address of a field within an object.

c. The unmanaged pointer contains the address of an element within an array.

d. The unmanaged pointer contains the address where the element following the last element in an array would be located.

14.4.2 Managed pointers

Managed pointers (`&`) can point to an instance of a value type, a field of an object, a field of a value type, an element of an array, or the address where an element just past the end of an array would be stored (for pointer indexes into managed arrays). Managed pointers cannot be *null*, and they shall be reported to the garbage collector even if they do not point to managed memory.

Managed pointers are specified by using `&` in a signature for a return value, local variable or an argument, or by using a byref type for a field or array element.

- Managed pointers can be passed as arguments, stored in local variables, and returned as values.
- If a parameter is passed by reference, the corresponding argument is a managed pointer.
- Managed pointers cannot be stored in static variables, array elements, or fields of objects or value types.
- Managed pointers are *not* interchangeable with object references.
- A managed pointer cannot point to another managed pointer, but it can point to an object reference or a value type.
- A managed pointer can point to a local variable, or a method argument
- Managed pointers that do not point to managed memory can be converted (using conv.u or conv.ovf.u) into unmanaged pointers, but this is not verifiable.
- Unverified code that erroneously converts a managed pointer into an unmanaged pointer can seriously compromise the integrity of the CLI. See Partition III (Managed Pointers) for more details.

End informative text

14.5 Method pointers

Type `::=` …
`|` `method` *CallConv Type* `'*' '(' Parameters ')'`

Variables of type method pointer shall store the address of the entry point to a method with compatible signature. A pointer to a static or instance method is obtained with the ldftn instruction, while a pointer to a virtual method is obtained with the ldvirtftn instruction. A method can be called by using a method pointer with the calli instruction. See Partition III for the specification of these instructions.

[*Note:* Like other pointers, method pointers are compatible with `unsigned int64` on 64-bit architectures, and with `unsigned int32` and on 32-bit architectures. The preferred usage, however, is `unsigned native int`, which works on both 32- and 64-bit architectures. *end note*]

[*Example:* Call a method using a pointer. The method `MakeDecision::Decide` returns a method pointer to either `AddOne` or `Negate`, alternating on each call. The main program calls `MakeDecision::Decide` three times, and after each call uses a calli instruction to call the method specified. The output printed is `"-1 2 -1"` indicating successful alternating calls.

```
.assembly Test { }
.assembly extern mscorlib { }

.method public static int32 AddOne(int32 Input)
{ .maxstack 5
  ldarg Input
  ldc.i4.1
  add
  ret
}

.method public static int32 Negate(int32 Input)
{ .maxstack 5
  ldarg Input
  neg
  ret
}

.class value sealed public MakeDecision extends
       [mscorlib]System.ValueType
{ .field static bool Oscillate
  .method public static method int32 *(int32) Decide()
  { ldsfld bool valuetype MakeDecision::Oscillate
    dup
    not
    stsfld bool valuetype MakeDecision::Oscillate
    brfalse NegateIt
    ldftn int32 AddOne(int32)
    ret

NegateIt:
    ldftn int32 Negate(int32)
    ret
  }
}

.method public static void Start()
{ .maxstack 2
  .entrypoint

  ldc.i4.1
  call method int32 *(int32) valuetype MakeDecision::Decide()
  calli int32(int32)
  call  void [mscorlib]System.Console::WriteLine(int32)

  ldc.i4.1
  call method int32 *(int32) valuetype MakeDecision::Decide()
  calli int32(int32)
  call  void [mscorlib]System.Console::WriteLine(int32)

  ldc.i4.1
  call method int32 *(int32) valuetype MakeDecision::Decide()
  calli int32(int32)
  call  void [mscorlib]System.Console::WriteLine(int32)
  ret
}
```

end example]

14.6 Delegates

Delegates (see Partition I) are the object-oriented equivalent of function pointers. Unlike function pointers, delegates are object-oriented, type-safe, and secure. Delegates are reference types, and are declared in the form of classes. Delegates shall have a base type of `System.Delegate` (see Partition IV).

Delegates shall be declared sealed, and the only members a delegate shall have are either the first two or all four methods as specified here. These methods shall be declared `runtime` and `managed` (§15.4.3). They shall not have a body, since that body shall be created automatically by the VES. Other methods available on

delegates are inherited from the class `System.Delegate` in the Base Class Library (see Partition IV). The delegate methods are:

- The instance constructor (named `.ctor` and marked `specialname` and `rtspecialname`, see §10.5.1) shall take exactly two parameters, the first having type `System.Object`, and the second having type `System.IntPtr`. When actually called (via a newobj instruction, see Partition III), the first argument shall be an instance of the class (or one of its derived classes) that defines the target method, and the second argument shall be a method pointer to the method to be called.
- The `Invoke` method shall be `virtual` and its signature constrains the target method to which it can be bound; see §14.6.1. The verifier treats calls to the `Invoke` method on a delegate just like it treats calls to any other method.
- The `BeginInvoke` method (§14.6.3.1), if present, shall be `virtual` and have a signature related to, but not the same as, that of the `Invoke` method. There are two differences in the signature. First, the return type shall be `System.IAsyncResult` (see Partition IV). Second, there shall be two additional parameters that follow those of `Invoke`: the first of type `System.AsyncCallback` and the second of type `System.Object`.
- The `EndInvoke` method (§14.6.3) shall be `virtual` and have the same return type as the `Invoke` method. It shall take as parameters exactly those parameters of `Invoke` that are managed pointers, in the same order they occur in the signature for `Invoke`. In addition, there shall be an additional parameter of type `System.IAsyncResult`.

Unless stated otherwise, a standard delegate type shall provide the two optional asynchronous methods, `BeginInvoke` and `EndInvoke`.

[*Example:* The following declares a Delegate used to call functions that take a single integer and return nothing. It provides all four methods so it can be called either synchronously or asynchronously. Because no parameters are passed by reference (i.e., as managed pointers) there are no additional arguments to `EndInvoke`.

```
.assembly Test { }
.assembly extern mscorlib { }

.class private sealed StartStopEventHandler extends [mscorlib]System.Delegate
 { .method public specialname rtspecialname instance void .ctor(object Instance,
         native int Method) runtime managed {}
   .method public virtual void Invoke(int32 action) runtime managed {}
   .method public virtual class [mscorlib]System.IAsyncResult
        BeginInvoke(int32 action, class [mscorlib]System.AsyncCallback callback,
           object Instance) runtime managed {}
   .method public virtual void EndInvoke(class
        [mscorlib]System.IAsyncResult result) runtime managed {}
}
```

end example]

As with any class, an instance is created using the newobj instruction in conjunction with the instance constructor. The first argument to the constructor shall be the object on which the method is to be called, or it shall be null if the method is a static method. The second argument shall be a method pointer to a method on the corresponding class and with a signature that matches that of the delegate class being instantiated.

14.6.1 Delegate signature compatibility

Delegates can only be bound to target methods where the signatures of the delegate and the target method are compatible. Compatibility is determined by examining the parameter types, return type and calling convention. (Custom modifiers are not considered significant and do not impact compatibility.)

For a delegate and target method to be compatible, the calling conventions shall match exactly.

For a delegate and target method to be compatible, the parameter types shall be compatible per the following rules:

Use D and T to denote the types of parameters to a delegate and a target method (respectively), use D := T to indicate that the types of the parameters are compatible, use D != T to indicate the types of the parameters are incompatible, use D[] to indicate an array of type D, and for instantiation D of generic type G<V> use V^D to indicate the type parameter used for V.

1. [:= is reflexive] For all parameter types D, D := D.
2. [:= is transitive] For all parameter types D, T and U, if D := U and U := T then D := T.
3. D := T if T is the base class of D or an interface implemented by D and D is not a value type (includes primitives, pointers, function pointers)
4. D := T if D and T are both interfaces and the implementation of D requires the implementation of T.
5. D[] := T[] if D := T and the arrays are either both vectors (zero-based, rank one) or neither is a vector and both have the same rank.
6. If D and T are method pointers, then D := T if the signatures (parameter types, return types, calling convention, custom modifiers) are compatible per these rules.
7. D := T if D and T are instantiations of the generic type G<+V> and V^D is a subtype of V^T.
8. D := T if D and T are instantiations of the generic type G<-V> and V^T is a subtype of V^D.
9. D := T if D and T are instantiations of the generic type G<V> and $V^D == V^T$.
10. Otherwise, D != T.

For a delegate and target method to be compatible, the return type shall be compatible per the following rules: Use D and T to denote the return type of a delegate and a target method (respectively), use D := T to indicate that the return types are compatible, use D !:= T to indicate that the return types are incompatible, use D[] to indicate an array of type D, and for instantiation D of generic type G<V> use VD to indicate the type parameter used for V.

1. [:= is reflexive] For all return types D, D := D.
2. [:= is transitive] For all return types D, T and U, if D := U and U := T then D := U.
3. D := T if D is the base class of T or an interface implemented by T and T is not a value type (includes primitives, pointers, function pointers)
4. D := T if D and T are both interfaces and the implementation of T requires the implementation of D.
5. D[] := T[] if D := T and the arrays are either both vectors (zero-based, rank one) or neither is a vector and both have the same rank.
6. If D and T are method pointers, then D := T if the signatures (parameter types, return types, calling convention, custom modifiers) are compatible per these rules.
7. D := T if D and T are instantiations of the generic type G<+V> and V^T is a subtype of V^D.
8. D := T if D and T are instantiations of the generic type G<-V> and V^D is a subtype of V^T.
9. D := T if D and T are instantiations of the generic type G<V> and $V^D == V^T$.
10. Otherwise D != T.

14.6.2 Synchronous calls to delegates

The synchronous mode of calling delegates corresponds to regular method calls and is performed by calling the virtual method named `Invoke` on the delegate. The delegate itself is the first argument to this call (it serves as the *this* pointer), followed by the other arguments as specified in the signature. When this call is made, the caller shall block until the called method returns. The called method shall be executed on the same thread as the caller.

[*Example:* Continuing the previous example, define a class `Test` that declares a method, `onStartStop`, appropriate for use as the target for the delegate.

```
.class public Test
{ .field public int32 MyData
  .method public void onStartStop(int32 action)
  { ret        // put your code here
  }
  .method public specialname rtspecialname
          instance void .ctor(int32 Data)
  { ret        // call base class constructor, store state, etc.
  }
}
```

Then define a main program. This one constructs an instance of `Test` and then a delegate that targets the `onStartStop` method of that instance. Finally, call the delegate.

```
.method public static void Start()
{ .maxstack 3
  .entrypoint
  .locals (class StartStopEventHandler DelegateOne,
           class Test InstanceOne)
  // Create instance of Test class
  ldc.i4.1
  newobj instance void Test::.ctor(int32)
  stloc InstanceOne


  // Create delegate to onStartStop method of that class
  ldloc InstanceOne
  ldftn instance void Test::onStartStop(int32)
  newobj void StartStopEventHandler::.ctor(object, native int)
  stloc DelegateOne

  // Invoke the delegate, passing 100 as an argument
  ldloc DelegateOne
  ldc.i4 100
  callvirt instance void StartStopEventHandler::Invoke(int32)
  ret
}
```

Note that the example above creates a delegate to a non-virtual function. If `onStartStop` had been a virtual function, use the following code sequence instead:

```
ldloc InstanceOne
dup
ldvirtftn instance void Test::onStartStop(int32)
newobj void StartStopEventHandler::.ctor(object, native int)
stloc DelegateOne
// Invoke the delegate, passing 100 as an argument
ldloc DelegateOne
```

end example]

[*Note:* The code sequence above shall use dup – not ldloc `InstanceOne` twice. The dup code sequence is easily recognized as type-safe, whereas alternatives would require more complex analysis. Verifiability of code is discussed in Partition III *end note*]

14.6.3 Asynchronous calls to delegates

In the asynchronous mode, the call is dispatched, and the caller shall continue execution without waiting for the method to return. The called method shall be executed on a separate thread.

To call delegates asynchronously, the `BeginInvoke` and `EndInvoke` methods are used.

Note: if the caller thread terminates before the callee completes, the callee thread is unaffected. The callee thread continues execution and terminates silently

Note: the callee can throw exceptions. Any unhandled exception propagates to the caller via the `EndInvoke` method.

14.6.3.1 The BeginInvoke method

An asynchronous call to a delegate shall begin by making a virtual call to the `BeginInvoke` method. `BeginInvoke` is similar to the `Invoke` method (§14.6.1), but has two differences:

- It has two additional parameters, appended to the list, of type `System.AsyncCallback`, and `System.Object`.
- The return type of the method is `System.IAsyncResult`.

Although the `BeginInvoke` method therefore includes parameters that represent return values, these values are not updated by this method. The results instead are obtained from the `EndInvoke` method (see below).

Unlike a synchronous call, an asynchronous call shall provide a way for the caller to determine when the call has been completed. The CLI provides two such mechanisms. The first is through the result returned from the call. This object, an instance of the interface `System.IAsyncResult`, can be used to wait for the result to be computed, it can be queried for the current status of the method call, and it contains the `System.Object` value that was passed to the call to `BeginInvoke`. See Partition IV.

The second mechanism is through the `System.AsyncCallback` delegate passed to `BeginInvoke`. The VES shall call this delegate when the value is computed or an exception has been raised indicating that the result will not be available. The value passed to this callback is the same value passed to the call to `BeginInvoke`. A value of null can be passed for `System.AsyncCallback` to indicate that the VES need not provide the callback.

[*Rationale:* This model supports both a polling approach (by checking the status of the returned *`System.IAsyncResult`*) and an event-driven approach (by supplying a *`System.AsyncCallback`*) to asynchronous calls. *end rationale*]

A synchronous call returns information both through its return value and through output parameters. Output parameters are represented in the CLI as parameters with managed pointer type. Both the returned value and the values of the output parameters are not available until the VES signals that the asynchronous call has completed successfully. They are retrieved by calling the `EndInvoke` method on the delegate that began the asynchronous call.

14.6.3.2 The EndInvoke method

The `EndInvoke` method can be called at any time after `BeginInvoke`. It shall suspend the thread that calls it until the asynchronous call completes. If the call completes successfully, `EndInvoke` will return the value that would have been returned had the call been made synchronously, and its managed pointer arguments will point to values that would have been returned to the out parameters of the synchronous call.

`EndInvoke` requires as parameters the value returned by the originating call to `BeginInvoke` (so that different calls to the same delegate can be distinguished, since they can execute concurrently) as well as any managed pointers that were passed as arguments (so their return values can be provided).

15 Defining, referencing, and calling methods

Methods can be defined at the global level (outside of any type):

Decl ::= …
| .method *MethodHeader* '{' *MethodBodyItem** '}'

as well as inside a type:

ClassMember ::= …
| .method *MethodHeader* '{' *MethodBodyItem** '}'

15.1 Method descriptors

There are four constructs in ILAsm connected with methods. These correspond with different metadata constructs, as described in §23.

15.1.1 Method declarations

A *MethodDecl*, or method declaration, supplies the method name and signature (parameter and return types), but not its body. That is, a method declaration provides a *MethodHeader* but no *MethodBodyItem*s. These are used at call sites to specify the call target (call or callvirt instructions, see Partition III) or to declare an abstract method. A *MethodDecl* has no direct logical couterpart in the metadata; it can be either a *Method* or a *MethodRef*.

15.1.2 Method definitions

A *Method*, or method definition, supplies the method name, attributes, signature, and body. That is, a method definition provides a *MethodHeader* as well as one or more *MethodBodyItem*s. The body includes the method's CIL instructions, exception handlers, local variable information, and additional runtime or custom metadata about the method. See §10.

15.1.3 Method references

A *MethodRef*, or method reference, is a reference to a method. It is used when a method is called and that method's definition lies in another module or assembly. A *MethodRef* shall be resolved by the VES into a *Method* before the method is called at runtime. If a matching *Method* cannot be found, the VES shall throw a `System.MissingMethodException`. See §22.25.

15.1.4 Method implementations

A *MethodImpl*, or method implementation, supplies the executable body for an existing virtual method. It associates a *Method* (representing the body) with a *MethodDecl* or *Method* (representing the virtual method). A *MethodImpl* is used to provide an implementation for an inherited virtual method or a virtual method from an interface when the default mechanism (matching by name and signature) would not provide the correct result. See §22.27.

15.2 Static, instance, and virtual methods

Static methods are methods that are associated with a type, not with its instances.

Instance methods are associated with an instance of a type: within the body of an instance method it is possible to reference the particular instance on which the method is operating (via the *this pointer*). It follows that instance methods shall only be defined in classes or value types, but not in interfaces or outside of a type (i.e., globally). However, notice

1. Instance methods on classes (including boxed value types), have a *this* pointer that is by default an object reference to the class on which the method is defined.

2. Instance methods on (unboxed) value types, have a *this* pointer that is by default a managed pointer to an instance of the type on which the method is defined.

3. There is a special encoding (denoted by the syntactic item `explicit` in the calling convention, see §15.3) to specify the type of the *this* pointer, overriding the default values specified here.

4. The *this* pointer can be null.

Virtual methods are associated with an instance of a type in much the same way as for instance methods. However, unlike instance methods, it is possible to call a virtual method in such a way that the implementation of the method shall be chosen at runtime by the VES depending upon the type of object used for the *this* pointer. The particular *Method* that implements a virtual method is determined dynamically at runtime (a *virtual call*) when invoked via the callvirt instruction; whilst the binding is decided at compile time when invoked via the call instruction (see Partition III).

With virtual calls (only), the notion of inheritance becomes important. A derived class can *override* a virtual method inherited from its base classes, providing a new implementation of the method. The method attribute newslot specifies that the CLI shall not override the virtual method definition of the base type, but shall treat the new definition as an independent virtual method definition.

Abstract virtual methods (which shall only be defined in abstract classes or interfaces) shall be called only with a **callvirt** instruction. Similarly, the address of an abstract virtual method shall be computed with the ldvirtftn instruction, and the ldftn instruction shall not be used.

[*Rationale:* With a concrete virtual method there is always an implementation available from the class that contains the definition, thus there is no need at runtime to have an instance of a class available. Abstract virtual methods, however, receive their implementation only from a subtype or a class that implements the appropriate interface, hence an instance of a class that actually implements the method is required. *end rationale*]

15.3 Calling convention

CallConv ::= [`instance` [`explicit`]] [*CallKind*]

A calling convention specifies how a method expects its arguments to be passed from the caller to the called method. It consists of two parts: the first deals with the existence and type of the *this* pointer, while the second relates to the mechanism for transporting the arguments.

If the attribute `instance` is present, it indicates that a *this* pointer shall be passed to the method. This attribute shall be used for both instance and virtual methods.

Normally, a parameter list (which always follows the calling convention) does *not* provide information about the type of the *this* pointer, since this can be deduced from other information. When the combination `instance explicit` is specified, however, the first type in the subsequent parameter list specifies the type of the *this* pointer and subsequent entries specify the types of the parameters themselves.

CallKind ::=
`default`
| `unmanaged cdecl`
| `unmanaged fastcall`
| `unmanaged stdcall`
| `unmanaged thiscall`
| `vararg`

Managed code shall have only the `default` or `vararg` calling kind. `default` shall be used in all cases except when a method accepts an arbitrary number of arguments, in which case `vararg` shall be used.

When dealing with methods implemented outside the CLI it is important to be able to specify the calling convention required. For this reason there are 16 possible encodings of the calling kind. Two are used for the managed calling kinds. Four are reserved with defined meaning across many platforms, as follows:

- `unmanaged cdecl` is the calling convention used by Standard C
- `unmanaged stdcall` specifies a standard C++ call
- `unmanaged fastcall` is a special optimized C++ calling convention
- `unmanaged thiscall` is a C++ call that passes a this pointer to the method

Four more are reserved for existing calling conventions, but their use is not maximally portable. Four more are reserved for future standardization, and two are available for non-standard experimental use.

(In this context, "portable" means a feature that is available on all conforming implementations of the CLI.)

15.4 Defining methods

MethodHeader ::=
*MethAttr** [*CallConv*] *Type* [`marshal '('` [*NativeType*] `')'`] *MethodName* [`'<'` *GenPars* `'>'`] `'('` *Parameters* `')'` *ImplAttr**

The method head (see also §10) consists of

- the calling convention (*CallConv*, see §15.3)
- any number of predefined method attributes (*MethAttr*, see §15.4.1.5)
- a return type with optional attributes
- optional marshalling information (§7.4)
- a method name
- optional generic parameters (when defining generic methods, see §10.1.7)
- a signature
- and any number of implementation attributes (*ImplAttr*, see §15.4.3)

Methods that do not have a return value shall use `void` as the return type.

MethodName ::=
`.cctor`
| `.ctor`
| *DottedName*

Method names are either simple names or the special names used for instance constructors and type initializers.

Parameters ::= [*Param* [`','` *Param*]*]
Param ::=
...
| [*ParamAttr**] *Type* [`marshal '('` [*NativeType*] `')'`] [*Id*]

The *Id*, if present, is the name of the parameter. A parameter can be referenced either by using its name or the zero-based index of the parameter. In CIL instructions it is always encoded using the zero-based index (the name is for ease of use in ILAsm).

Note that, in contrast to calling a `vararg` method, the definition of a `vararg` method does *not* include any ellipsis ("**...**")

ParamAttr ::=
'[' in ']'
| '[' opt ']'
| '[' out ']'

The parameter attributes shall be attached to the parameters (§22.33) and hence are not part of a method signature.

[*Note:* Unlike parameter attributes, custom modifiers (`modopt` and `modreq`) *are* part of the signature. Thus, modifiers form part of the method's contract while parameter attributes do not. *end note*]

`in` and `out` shall only be attached to parameters of pointer (managed or unmanaged) type. They specify whether the parameter is intended to supply input to the method, return a value from the method, or both. If neither is specified `in` is assumed. The CLI itself does not enforce the semantics of these bits, although they can be used to optimize performance, especially in scenarios where the call site and the method are in different application domains, processes, or computers.

`opt` specifies that this parameter is intended to be optional from an end-user point of view. The value to be supplied is stored using the `.param` syntax (§15.4.1.4).

15.4.1 Method body

The method body shall contain the instructions of a program. However, it can also contain labels, additional syntactic forms and many directives that provide additional information to *ilasm* and are helpful in the compilation of methods of some languages.

MethodBodyItem ::=	**Description**	**Clause**
.custom *CustomDecl*	Definition of custom attributes.	21
| .data *DataDecl*	Emits data to the data section	16.3
| .emitbyte *Int32*	Emits an unsigned byte to the code section of the method.	15.4.1.1
| .entrypoint	Specifies that this method is the entry point to the application (only one such method is allowed).	15.4.1.2
| .locals [init] '(' *LocalsSignature* ')'	Defines a set of local variables for this method.	15.4.1.3
| .maxstack *Int32*	The int32 specifies the maximum number of elements on the evaluation stack during the execution of the method.	15.4.1
| .override *TypeSpec* '::' *MethodName*	Use current method as the implementation for the method specified.	10.3.2
| .override method *CallConv Type TypeSpec* '::' *MethodName GenArity* '(' *Parameters* ')'	Use current method as the implementation for the method specified.	10.3.2

MethodBodyItem ::=	**Description**	**Clause**
| `.param '[' Int32 ']' [ '=' FieldInit ]`	Store a constant *FieldInit* value for parameter *Int32*	15.4.1.4
| `.param type '[' Int32 ']'`	Specifies a type parameter for a generic method	15.4.1.5
| *ExternSourceDecl*	`.line` or `#line`	5.7
| *Instr*	An instruction	Partition VI
| *Id* `':'`	A label	5.4
| *ScopeBlock*	Lexical scope of local variables	15.4.4
| *SecurityDecl*	`.permission` or `.permissionset`	20
| *SEHBlock*	An exception block	19

15.4.1.1 The .emitbyte directive

MethodBodyItem ::= …
| `.emitbyte` *Int32*

This directive causes an unsigned 8-bit value to be emitted directly into the CIL stream of the method, at the point at which the directive appears.

[*Note:* The `.emitbyte` directive is used for generating tests. It is not required in generating regular programs. *end note*]

15.4.1.2 The .entrypoint directive

MethodBodyItem ::= …
| `.entrypoint`

The `.entrypoint` directive marks the current method, which shall be static, as the entry point to an application. The VES shall call this method to start the application. An executable shall have exactly one entry point method. This entry point method can be a global method or it can appear inside a type. (The effect of the directive is to place the metadata token for this method into the CLI header of the PE file)

The entry point method shall either accept no arguments or a vector of strings. If it accepts a vector of strings, the strings shall represent the arguments to the executable, with index 0 containing the first argument. The mechanism for specifying these arguments is platform-specific and is not specified here.

The return type of the entry point method shall be `void`, `int32`, or `unsigned int32`. If an `int32` or `unsigned int32` is returned, the executable can return an exit code to the host environment. A value of 0 shall indicate that the application terminated ordinarily.

The accessibility of the entry point method shall not prevent its use in starting execution. Once started the VES shall treat the entry point as it would any other method.

The entry point method cannot be defined in a generic class.

[*Example:* The following prints the first argument and returns successfully to the operating system:

```
.method public static int32 MyEntry(string[] s) cil managed
{ .entrypoint
  .maxstack 2
  ldarg.0                  // load and print the first argument
  ldc.i4.0
  ldelem.ref
  call void [mscorlib]System.Console::WriteLine(string)
  ldc.i4.0                 // return success
  ret
}
```

end example]

15.4.1.3 The .locals directive

The `.locals` statement declares one or more local variables (see Partition I) for the current method.

MethodBodyItem ::= …
| `.locals` [`init`] '(' *LocalsSignature* ')'
LocalsSignature ::= *Local* [',' *Local*]*
Local ::= *Type* [*Id*]

If present, the *Id* is the name of the corresponding local variable.

If `init` is specified, the variables are initialized to their default values according to their type: reference types are initialized to *null* and value types are zeroed out.

[*Note:* Verifiable methods shall include the `init` keyword. See Partition III. *end note*]

[*Example:* The following declares 4 local variables, each of which is to be initialized to its default value:

```
.locals init ( int32 i, int32 j, float32 f, int64[] vect)
```

end example]

15.4.1.4 The .param directive

MethodBodyItem ::= …
| `.param` '[' *Int32* ']' ['=' *FieldInit*]

This directive stores in the metadata a constant value associated with method parameter number *Int32*, see §22.9. While the CLI requires that a value be supplied for the parameter, some tools can use the presence of this attribute to indicate that the tool rather than the user is intended to supply the value of the parameter. Unlike CIL instructions, `.param` uses index 0 to specify the return value of the method, index 1 to specify the first parameter of the method, index 2 to specify the second parameter of the method, and so on.

[*Note:* The CLI attaches no semantic whatsoever to these values—it is entirely up to compilers to implement any semantic they wish (e.g., so-called default argument values). *end note*]

15.4.1.5 The .param type directive

MethodBodyItem ::= …
| `.param type` '[' *Int32* ']'

This directive allows type parameters for a generic type or method to be specified. *Int32* is the 1-based ordinal of the type or method parameter to which the directive applies. [*Note*: This directive is used in conjunction with a `.custom` directive to associate a custom attribute with a type parameter. *end note*]

When a `.param type` directive is used within class scope, it refers to a type parameter of that class. When the directive is used within method scope inside a class definition, it refers to a type parameter of that method. Otherwise, the program is ill-formed.

[*Example:*

```
.class public G<T,U> {
  .param type [1]          // refers to T
  .custom instance void TypeParamAttribute::.ctor() = (01 00 ... )
  .method public void Foo<M>(!!0 m) {
     .param type [1]       // refers to M
     .custom instance void AnotherTypeParamAttribute::.ctor() = (01 00 ... )
      …
  }
  …
}
```

end example]

15.4.2 Predefined attributes on methods

MethAttr ::=	**Description**	**Clause**
`abstract`	The method is `abstract` (shall also be virtual).	15.4.2.4
| `assembly`	Assembly accessibility	15.4.2.1
| `compilercontrolled`	Compiler-controlled accessibility.	15.4.2.1
| `famandassem`	Family and Assembly accessibility	15.4.2.1
| `family`	Family accessibility	15.4.2.1
| `famorassem`	Family or Assembly accessibility	15.4.2.1
| `final`	This virtual method cannot be overridden by derived classes.	15.4.2.2
| `hidebysig`	Hide by signature. Ignored by the runtime.	15.4.2.2
| `newslot`	Specifies that this method shall get a new slot in the virtual method table.	15.4.2.3
| `pinvokeimpl` `'('` *QSTRING* [`as` *QSTRING*] *PinvAttr** `')'`	Method is actually implemented in native code on the underlying platform	15.4.2.5
| `private`	Private accessibility	15.4.2.1
| `public`	Public accessibility.	15.4.2.1
| `rtspecialname`	The method name needs to be treated in a special way by the runtime.	15.4.2.6
| `specialname`	The method name needs to be treated in a special way by some tool.	15.4.2.6
| `static`	Method is static.	15.4.2.2
| `virtual`	Method is virtual.	15.4.2.2
| `strict`	Check accessibility on override	15.4.2.2

The following combinations of predefined attributes are invalid:

- `static` combined with any of `final`, `newslot`, or `virtual`

- `abstract` combined with any of `final` or `pinvokeimpl`
- `compilercontrolled` combined with any of `final`, `rtspecialname`, `specialname`, or `virtual`

15.4.2.1 Accessibility information

MethAttr `::=` …
`| assembly`
`| compilercontrolled`
`| famandassem`
`| family`
`| famorassem`
`| private`
`| public`

Only one of these attributes shall be applied to a given method. See Partition I.

15.4.2.2 Method contract attributes

MethAttr `::=` …
`| final`
`| hidebysig`
`| static`
`| virtual`
`| strict`

These attributes can be combined, except a method shall not be both `static` and `virtual`; only `virtual` methods shall be `final` or `strict`; and abstract methods shall not be `final`.

`final` methods shall not be overridden by derived classes of this type.

`hidebysig` is supplied for the use of tools and is ignored by the VES. It specifies that the declared method hides all methods of the base class types that have a matching method signature; when omitted, the method should hide all methods of the same name, regardless of the signature.

[*Rationale:* Some languages (such as C++) use a hide-by-name semantics while others (such as C#, Java™) use a hide-by-name-and-signature semantics. *end rationale*]

`static` and `virtual` are described in §15.2.

`strict virtual` methods can only be overridden if they are also accessible. See §23.1.10.

15.4.2.3 Overriding behavior

MethAttr `::=` …
`| newslot`

`newslot` shall only be used with `virtual` methods. See 10.3.

15.4.2.4 Method attributes

MethAttr ::= …
| abstract

`abstract` shall only be used with `virtual` methods that are not `final`. It specifies that an implementation of the method is not provided but shall be provided by a derived class. `abstract` methods shall only appear in `abstract` types (§10.1.4).

15.4.2.5 Interoperation attributes

MethAttr ::= …
| pinvokeimpl '(' *QSTRING* [as *QSTRING*] *PinvAttr** ')'

See §15.5.2and §22.20.

15.4.2.6 Special handling attributes

MethAttr ::= …
| rtspecialname
| specialname

The attribute `rtspecialname` specifies that the method name shall be treated in a special way by the runtime. Examples of special names are `.ctor` (object constructor) and `.cctor` (type initializer).

`specialname` indicates that the name of this method has special meaning to some tools.

15.4.3 Implementation attributes of methods

ImplAttr ::=	**Description**	**Clause**
cil	The method contains standard CIL code.	15.4.3.1
| forwardref	The body of this method is not specified with this declaration.	15.4.3.3
| internalcall	Denotes the method body is provided by the CLI itself	15.4.3.3
| managed	The method is a managed method.	15.4.3.2
| native	The method contains native code.	15.4.3.1
| noinlining	The runtime shall not expand the method inline.	15.4.3.3
| nooptimization	The runtime shall not optimize the method when generating native code.	15.4.3.3
| runtime	The body of the method is not defined, but is produced by the runtime.	15.4.3.1
| synchronized	The method shall be executed in a single threaded fashion.	15.4.3.3
| unmanaged	Specifies that the method is unmanaged.	15.4.3.2

15.4.3.1 Code implementation attributes

ImplAttr ::= …

| `cil`
| `native`
| `runtime`

These attributes are mutually exclusive; they specify the type of code the method contains.

`cil` specifies that the method body consists of cil code. Unless the method is declared `abstract`, the body of the method shall be provided if `cil` is used.

`native` specifies that a method was implemented using native code, tied to a specific processor for which it was generated. `native` methods shall not have a body but instead refer to a native method that declares the body. Typically, the PInvoke functionality (§15.5.2) of the CLI is used to refer to a native method.

`runtime` specifies that the implementation of the method is automatically provided by the runtime and is primarily used for the methods of delegates (§14.6).

15.4.3.2 Managed or unmanaged

ImplAttr `::= …`
| `managed`
| `unmanaged`

These shall not be combined. Methods implemented using CIL are `managed`. `unmanaged` is used primarily with PInvoke (§15.5.2).

15.4.3.3 Implementation information

ImplAttr `::= …`
| `forwardref`
| `internalcall`
| `noinlining`
| `nooptimization`
| `synchronized`

These attributes can be combined.

`forwardref` specifies that the body of the method is provided elsewhere. This attribute shall not be present when an assembly is loaded by the VES. It is used for tools (like a static linker) that will combine separately compiled modules and resolve the forward reference.

`internalcall` specifies that the method body is provided by this CLI (and is typically used by low-level methods in a system library). It shall not be applied to methods that are intended for use across implementations of the CLI.

`noinlining` specifies that the body of this method should not be included into the code of any caller methods, by a CIL-to-native-code compiler; it shall be kept as a separate routine.

`nooptimization` specifies that a CIL-to-native-code compiler should not perform code optimizations.

[*Rationale:* specifying that a method not be inlined ensures that it remains 'visible' for debugging (e.g., displaying stack traces) and profiling. It also provides a mechanism for the programmer to override the default heuristics a CIL-to-native-code compiler uses for inlining. *end rationale*]

`synchronized` specifies that the whole body of the method shall be single-threaded. If this method is an instance or virtual method, a lock on the object shall be obtained before the method is entered. If this method is a static method, a lock on the closed type shall be obtained before the method is entered. If a lock cannot be

obtained, the requesting thread shall not proceed until it is granted the lock. This can cause deadlocks. The lock is released when the method exits, either through a normal return or an exception. Exiting a synchronized method using a tail. call shall be implemented as though the tail. had not been specified. `noinlining` specifies that the runtime shall not inline this method. Inlining refers to the process of replacing the call instruction with the body of the called method. This can be done by the runtime for optimization purposes.

15.4.4 Scope blocks

ScopeBlock `::= '{'` *MethodBodyItem*`* '}'`

A *ScopeBlock* is used to group elements of a method body together. For example, it is used to designate the code sequence that constitutes the body of an exception handler.

15.4.5 vararg methods

`vararg` methods accept a variable number of arguments. They shall use the `vararg` calling convention (§15.3).

At each call site, a method reference shall be used to describe the types of the fixed and variable arguments that are passed. The fixed part of the argument list shall be separated from the additional arguments with an ellipsis (see Partition I). [*Note*: The method reference is represented by either a *MethodRef* (§22.25) or *MethodDef* (§22.26). A *MethodRef* might be needed even if the method is defined in the same assembly, because the *MethodDef* only describes the fixed part of the argument list. If the call site does not pass any additional arguments, then it can use the *MethodDef* for vararg methods defined in the same assembly. *end note*]

The `vararg` arguments shall be accessed by obtaining a handle to the argument list using the CIL instruction arglist (see Partition III). The handle can be used to create an instance of the value type `System.ArgIterator` which provides a type-safe mechanism for accessing the arguments (see Partition IV).

[*Example:* The following example shows how a `vararg` method is declared and how the first `vararg` argument is accessed, assuming that at least one additional argument was passed to the method:

```
.method public static vararg void MyMethod(int32 required) {
  .maxstack 3
  .locals init (valuetype [mscorlib]System.ArgIterator it, int32 x)

  ldloca     it                          // initialize the iterator
  initobj     valuetype [mscorlib]System.ArgIterator
  ldloca     it
  arglist                                // obtain the argument handle
  call instance void [mscorlib]System.ArgIterator::.ctor(valuetype
     [mscorlib]System.RuntimeArgumentHandle)   // call constructor of iterator

  /* argument value will be stored in x when retrieved, so load
   address of x */
  ldloca     x
  ldloca     it
  // retrieve the argument, the argument for required does not matter
  call instance typedref [mscorlib]System.ArgIterator::GetNextArg()

  call object [mscorlib]System.TypedReference::ToObject(typedref)  /* retrieve
the
     object */
  castclass [mscorlib]System.Int32            // cast and unbox
  unbox      int32
  cpobj      int32                       // copy the value into x
  // first vararg argument is stored in x
  ret
}
```

end example]

15.5 Unmanaged methods

In addition to supporting managed code and managed data, the CLI provides facilities for accessing pre-existing native code from the underlying platform, known as *unmanaged code*. These facilities are, by necessity, platform-specific and hence are only partially specified here.

This Standard specifies:

- A mechanism in the file format for providing function pointers to managed code that can be called from unmanaged code (§15.5.1).
- A mechanism for marking certain method definitions as being implemented in unmanaged code (called *platform invoke*, see §15.5.2).
- A mechanism for marking call sites used with method pointers to indicate that the call is to an unmanaged method (§15.5.3).
- A small set of pre-defined data types that can be passed (marshaled) using these mechanisms on all implementations of the CLI (§15.5.4). The set of types is extensible through the use of custom attributes and modifiers, but these extensions are platform-specific.

15.5.1 Method transition thunks

[*Note:* As this mechanism is not part of the Kernel Profile, it might not be present in all conforming implementations of the CLI. See Partition IV. *end note*]

In order to call managed code from unmanaged code, some platforms require a specific transition sequence to be performed. In addition, some platforms require that the representation of data types be converted (data marshaling). Both of these problems are solved by the `.vtfixup` directive. This directive can appear several times, but only at the top level of a CIL assembly file, as shown by the following grammar:

Decl ::=	**Clause**
`.vtfixup` *VTFixupDecl*	
| …	5.10

The `.vtfixup` directive declares that at a certain memory location there is a table that contains metadata tokens referring to methods that shall be converted into method pointers. The CLI will do this conversion automatically when the file containing the `.vtfixup` directive is loaded into memory for execution. The declaration specifies the number of entries in the table, the kind of method pointer that is required, the width of an entry in the table, and the location of the table:

VTFixupDecl ::=
[*Int32*] *VTFixupAttr*`*` `at` *DataLabel*

VTFixupAttr ::=
`fromunmanaged`
| `int32`
| `int64`

The attributes `int32` and `int64` are mutually exclusive, with `int32` being the default. These attributes specify the width of each slot in the table. Each slot contains a 32-bit metadata token (zero-padded if the table has 64-bit slots), and the CLI converts it into a method pointer of the same width as the slot.

If `fromunmanaged` is specified, the CLI will generate a thunk that will convert the unmanaged method call to a managed call, call the method, and return the result to the unmanaged environment. The thunk will also perform data marshalling in the platform-specific manner described for *platform invoke*.

The ILAsm syntax does not specify a mechanism for creating the table of tokens, but a compiler can simply emit the tokens as byte literals into a block specified using the `.data` directive.

15.5.2 Platform invoke

Methods defined in native code can be invoked using the *platform invoke* (also know as PInvoke or p/invoke) functionality of the CLI. Platform invoke will switch from managed to unmanaged state and back, and also handle necessary data marshalling. Methods that need to be called using PInvoke are marked as `pinvokeimpl`. In addition, the methods shall have the implementation attributes `native` and `unmanaged` (§15.4.2.4).

MethAttr ::=	**Description**	**Clause**
`pinvokeimpl '('` *QSTRING* [`as` *QSTRING*] *PinvAttr*`*` `')'`	Implemented in native code	
`|` …		15.4.1.5

The first quoted string is a platform-specific description indicating where the implementation of the method is located (for example, on Microsoft Windows™ this would be the name of the DLL that implements the method). The second (optional) string is the name of the method as it exists on that platform, since the platform can use name-mangling rules that force the name as it appears to a managed program to differ from the name as seen in the native implementation (this is common, for example, when the native code is generated by a C++ compiler).

Only static methods, defined at global scope (i.e., outside of any type), can be marked `pinvokeimpl`. A method declared with `pinvokeimpl` shall not have a body specified as part of the definition.

PinvAttr ::=	**Description (platform-specific, suggestion only)**
`ansi`	ANSI character set.
`| autochar`	Determine character set automatically.
`| cdecl`	Standard C style call
`| fastcall`	C style fastcall.
`| stdcall`	Standard C++ style call.
`| thiscall`	The method accepts an implicit *this* pointer.
`| unicode`	Unicode character set.
`| platformapi`	Use call convention appropriate to target platform.

The attributes `ansi`, `autochar`, and `unicode` are mutually exclusive. They govern how strings will be marshaled for calls to this method: `ansi` indicates that the native code will receive (and possibly return) a platform-specific representation that corresponds to a string encoded in the ANSI character set (typically this would match the representation of a C or C++ string constant); `autochar` indicates a platform-specific representation that is "natural" for the underlying platform; and `unicode` indicates a platform-specific representation that corresponds to a string encoded for use with Unicode methods on that platform.

The attributes `cdecl`, `fastcall`, `stdcall`, `thiscall`, and `platformapi` are mutually exclusive. They are platform-specific and specify the calling conventions for native code.

[*Example:* The following shows the declaration of the method `MessageBeep` located in the Microsoft Windows™ DLL user32.dll:

```
.method public static pinvokeimpl("user32.dll" stdcall) int8
       MessageBeep(unsigned int32) native unmanaged {}
```

end example]

15.5.3 Method calls via function pointers

Unmanaged methods can also be called via function pointers. There is no difference between calling managed or unmanaged methods with pointers. However, the unmanaged method needs to be declared with `pinvokeimpl` as described in §15.5.2. Calling managed methods with function pointers is described in §14.5.

15.5.4 Data type marshaling

While data type marshaling is necessarily platform-specific, this Standard specifies a minimum set of data types that shall be supported by all conforming implementations of the CLI. Additional data types can be supported in a platform-specific manner, using custom attributes and/or custom modifiers to specify any special handling required on the particular implementation.

The following data types shall be marshaled by all conforming implementations of the CLI; the native data type to which they conform is implementation-specific:

- All integer data types (`int8`, `int16`, `unsigned int8`, `bool`, `char`, etc.) including the `native` integer types.
- Enumerations, as their underlying data type.
- All floating-point data types (`float32` and `float64`), if they are supported by the CLI implementation for managed code.
- The type `string`.
- Unmanaged pointers to any of the above types.

In addition, the following types shall be supported for marshaling from managed code to unmanaged code, but need not be supported in the reverse direction (i.e., as return types when calling unmanaged methods or as parameters when calling from unmanaged methods into managed methods):

- One-dimensional zero-based arrays of any of the above
- Delegates (the mechanism for calling from unmanaged code into a delegate is platform-specific; it should not be assumed that marshaling a delegate will produce a function pointer that can be used directly from unmanaged code).

Finally, the type `System.Runtime.InteropServices.GCHandle` can be used to marshal an object to unmanaged code. The unmanaged code receives a platform-specific data type that can be used as an "opaque handle" to a specific object. See Partition IV.

16 Defining and referencing fields

Fields are typed memory locations that store the data of a program. The CLI allows the declaration of both instance and static fields. While static fields are associated with a type, and are shared across all instances of that type, instance fields are associated with a particular instance of that type. Once instantiated, an instance has its own copy of each instance field.

The CLI also supports global fields, which are fields declared outside of any type definition. Global fields shall be static.

A field is defined by the `.field` directive: (§22.15)

Field ::= `.field` *FieldDecl*

FieldDecl ::=
['[' *Int32* ']'] *FieldAttr** *Type* *Id* ['=' *FieldInit* | `at` *DataLabel*]

The *FieldDecl* has the following parts:

- An optional integer specifying the byte offset of the field within an instance (§10.7). If present, the type containing this field shall have the `explicit` layout attribute. An offset shall not be supplied for global or static fields.
- Any number of field attributes (§16.2).
- Type.
- Name.
- Optionally, either a *FieldInit* clause (§16.2) or a *DataLabel* (§5.4) clause.

Global fields shall have a data label associated with them. This specifies where, in the PE file, the data for that field is located. Static fields of a type can, but need not, be assigned a data label.

[*Example:*

```
.field private class [.module Counter.dll]Counter counter
.field public static initonly int32 pointCount
.field private int32 xOrigin
.field public static int32 count at D_0001B040
```

end example]

16.1 Attributes of fields

Attributes of a field specify information about accessibility, contract information, interoperation attributes, as well as information on special handling.

The following subclauses contain additional information on each group of predefined attributes of a field.

FieldAttr ::=	**Description**	**Clause**
`assembly`	Assembly accessibility.	16.1.1
| `famandassem`	Family and Assembly accessibility.	16.1.1
| `family`	Family accessibility.	16.1.1
| `famorassem`	Family or Assembly accessibility.	16.1.1
| `initonly`	Marks a constant field.	16.1.2
| `literal`	Specifies metadata field. No memory is allocated at runtime for this field.	16.1.2

FieldAttr ::=	**Description**	**Clause**
| `marshal '('` *NativeType* `')'`	Marshaling information.	16.1.3
| `notserialized`	Reserved (indicates this field is not to be serialized).	16.1.2
| `private`	Private accessibility.	16.1.1
| `compilercontrolled`	Compiler controlled accessibility.	16.1.1
| `public`	Public accessibility.	16.1.1
| `rtspecialname`	Special treatment by runtime.	16.1.4
| `specialname`	Special name for other tools.	16.1.4
| `static`	Static field.	16.1.2

16.1.1 Accessibility information

The accessibility attributes are `assembly`, `famandassem`, `family`, `famorassem`, `private`, `compilercontrolled`, and `public`. These attributes are mutually exclusive.

Accessibility attributes are described in §8.2.

16.1.2 Field contract attributes

Field contract attributes are `initonly`, `literal`, `static` and `notserialized`. These attributes can be combined; however, only `static` fields shall be `literal`. The default is an instance field that can be serialized.

`static` specifies that the field is associated with the type itself rather than with an instance of the type. Static fields can be accessed without having an instance of a type, e.g., by static methods. As a consequence, within an application domain, a static field is shared between all instances of a type, and any modification of this field will affect all instances. If `static` is not specified, an instance field is created.

`initonly` marks fields which are constant after they are initialized. These fields shall only be mutated inside a constructor. If the field is a static field, then it shall be mutated only inside the type initializer of the type in which it was declared. If it is an instance field, then it shall be mutated only in one of the instance constructors of the type in which it was defined. It shall not be mutated in any other method or in any other constructor, including constructors of derived classes.

[*Note:* The use of ldflda or ldsflda on an `initonly` field makes code unverifiable. In unverifiable code, the VES need not check whether `initonly` fields are mutated outside the constructors. The VES need not report any errors if a method changes the value of a constant. However, such code is not valid. *end note*]

`literal` specifies that this field represents a constant value; such fields shall be assigned a value. In contrast to `initonly` fields, `literal` fields do not exist at runtime. There is no memory allocated for them. `literal` fields become part of the metadata, but cannot be accessed by the code. `literal` fields are assigned a value by using the *FieldInit* syntax (§16.2).

[*Note:* It is the responsibility of tools generating CIL to replace source code references to the literal with its actual value. Hence changing the value of a literal requires recompilation of any code that references the literal. Literal values are, thus, not version-resilient. *end note*]

16.1.3 Interoperation attributes

There is one attribute for interoperation with pre-existing native applications; it is platform-specific and shall not be used in code intended to run on multiple implementations of the CLI. The attribute is `marshal` and specifies that the field's contents should be converted to and from a specified native data type when passed to unmanaged code. Every conforming implementation of the CLI will have default marshaling rules as well as restrictions on what automatic conversions can be specified using the `marshal` attribute. See also §15.5.4.

[*Note:* Marshaling of user-defined types is not required of all implementations of the CLI. It is specified in this standard so that implementations which choose to provide it will allow control over its behavior in a consistent manner. While this is not sufficient to guarantee portability of code that uses this feature, it does increase the likelihood that such code will be portable. *end note*]

16.1.4 Other attributes

The attribute `rtspecialname` indicates that the field name shall be treated in a special way by the runtime.

[*Rationale:* There are currently no field names that are required to be marked with `rtspecialname`. It is provided for extensions, future standardization, and to increase consistency between the declaration of fields and methods (instance and type initializer methods shall be marked with this attribute). By convention, the single instance field of an enumeration is named "value__" and marked with `rtspecialname`. *end rationale*]

The attribute `specialname` indicates that the field name has special meaning to tools other than the runtime, typically because it marks a name that has meaning for the CLS (see Partition I).

16.2 Field init metadata

The *FieldInit* metadata can optionally be added to a field declaration. The use of this feature shall not be combined with a data label.

The *FieldInit* information is stored in metadata and this information can be queried from metadata. But the CLI does not use this information to automatically initialize the corresponding fields. The field initializer is typically used with `literal` fields (§16.1.2) or parameters with default values. See §22.9.

The following table lists the options for a field initializer. Note that while both the type and the field initializer are stored in metadata there is no requirement that they match. (Any importing compiler is responsible for coercing the stored value to the target field type). The description column in the table below provides additional information.

FieldInit ::=	**Description**
`bool '(' true | false ')'`	Boolean value, encoded as **true** or **false**
`| bytearray '(' Bytes ')'`	String of bytes, stored without conversion. Can be padded with one zero byte to make the total byte-count an even number
`| char '(' Int32 ')'`	16-bit unsigned integer (Unicode character)
`| float32 '(' Float64 ')'`	32-bit floating-point number, with the floating-point number specified in parentheses.
`| float32 '(' Int32 ')'`	*Int32* is binary representation of float
`| float64 '(' Float64 ')'`	64-bit floating-point number, with the floating-point number specified in parentheses.
`| float64 '(' Int64 ')'`	*Int64* is binary representation of double
`| [unsigned] int8 '(' Int32 ')'`	8-bit integer with the value specified in parentheses.
`| [unsigned] int16 '(' Int32 ')'`	16-bit integer with the value specified in parentheses.
`| [unsigned] int32 '(' Int32 ')'`	32-bit integer with the value specified in parentheses.
`| [unsigned] int64 '(' Int64 ')'`	64-bit integer with the value specified in parentheses.
`| QSTRING`	String. *QSTRING* is stored as Unicode
`| nullref`	Null object reference

[*Example:* The following shows a typical use of this:

```
.field public static literal valuetype ErrorCodes no_error = int8(0)
```

The field named `no_error` is a literal of type `ErrorCodes` (a value type) for which no memory is allocated. Tools and compilers can look up the value and detect that it is intended to be an 8-bit signed integer whose value is 0. *end example*]

16.3 Embedding data in a PE file

There are several ways to declare a data field that is stored in a PE file. In all cases, the `.data` directive is used.

Data can be embedded in a PE file by using the `.data` directive at the top-level.

Decl ::=	**Clause**
`.data` *DataDecl*	
| …	6.6

Data can also be declared as part of a type:

ClassMember ::=	**Clause**
`.data` *DataDecl*	
| …	10.2

Yet another alternative is to declare data inside a method:

MethodBodyItem ::=	**Clause**
`.data` *DataDecl*	
| …	15.4.1

16.3.1 Data declaration

A `.data` directive contains an optional data label and the body which defines the actual data. A data label shall be used if the data is to be accessed by the code.

DataDecl ::= [*DataLabel* `'='`] *DdBody*

The body consists either of one data item or a list of data items in braces. A list of data items is similar to an array.

DdBody ::=
DdItem
| `'{'` *DdItemList* `'}'`

A list of items consists of any number of items:

DdItemList ::= *DdItem* [`','` *DdItemList*]

The list can be used to declare multiple data items associated with one label. The items will be laid out in the order declared. The first data item is accessible directly through the label. To access the other items, pointer arithmetic is used, adding the size of each data item to get to the next one in the list. The use of pointer arithmetic will make the application non-verifiable. (Each data item shall have a *DataLabel* if it is to be referenced afterwards; missing a *DataLabel* is useful in order to insert alignment padding between data items)

A data item declares the type of the data and provides the data in parentheses. If a list of data items contains items of the same type and initial value, the grammar below can be used as a short cut for some of the types: the number of times the item shall be replicated is put in brackets after the declaration.

DdItem ::=	**Description**
'&' '(' *Id* ')'	Address of label
| bytearray '(' *Bytes* ')'	Array of bytes
| char '*' '(' *QSTRING* ')'	Array of (Unicode) characters
| float32 ['(' *Float64* ')']['[' *Int32* ']']	32-bit floating-point number, can be replicated
| float64 ['(' *Float64* ')']['[' *Int32* ']']	64-bit floating-point number, can be replicated
| int8 ['(' *Int32* ')']['[' *Int32* ']']	8-bit integer, can be replicated
| int16 ['(' *Int32* ')']['[' *Int32* ']']	16-bit integer, can be replicated
| int32 ['(' *Int32* ')']['[' *Int32* ']']	32-bit integer, can be replicated
| int64 ['(' *Int64* ')']['[' *Int32* ']']	64-bit integer, can be replicated

[*Example:*

The following declares a 32-bit signed integer with value 123:

```
.data theInt = int32(123)
```

The following declares 10 replications of an 8-bit unsigned integer with value 3:

```
.data theBytes = int8 (3) [10]
```

end example]

16.3.2 Accessing data from the PE file

The data stored in a PE File using the `.data` directive can be accessed through a `static` variable, either global or a member of a type, declared at a particular position of the data:

FieldDecl ::= *FieldAttr** *Type* *Id* `at` *DataLabel*

The data is then accessed by a program as it would access any other static variable, using instructions such as `ldsfld`, `ldsflda`, and so on (see Partition III).

The ability to access data from within the PE File can be subject to platform-specific rules, typically related to section access permissions within the PE File format itself.

[*Example:* The following accesses the data declared in the example of §16.3.1. First a static variable needs to be declared for the data, e.g., a global static variable:

```
.field public static int32 myInt at theInt
```

Then the static variable can be used to load the data:

```
ldsfld int32 myInt
// data on stack
```

end example]

16.4 Initialization of non-literal static data

This subclause and its subclauses contain only informative text.

Many languages that support static data provide for a means to initialize that data before the program begins execution. There are three common mechanisms for doing this, and each is supported in the CLI.

16.4.1 Data known at link time

When the correct value to be stored into the static data is known at the time the program is linked (or compiled for those languages with no linker step), the actual value can be stored directly into the PE file, typically into the data area (§16.3). References to the variable are made directly to the location where this data has been placed in memory, using the OS-supplied fix-up mechanism to adjust any references to this area if the file loads at an address other than the one assumed by the linker.

In the CLI, this technique can be used directly if the static variable has one of the primitive numeric types or is a value type with explicit type layout and no embedded references to managed objects. In this case the data is laid out in the data area as usual and the static variable is assigned a particular RVA (i.e., offset from the start of the PE file) by using a data label with the field declaration (using the `at` syntax).

This mechanism, however, does not interact well with the CLI notion of an application domain (see Partition I). An application domain is intended to isolate two applications running in the same OS process from one another by guaranteeing that they have no shared data. Since the PE file is shared across the entire process, any data accessed via this mechanism is visible to all application domains in the process, thus violating the application domain isolation boundary.

16.5 Data known at load time

When the correct value is not known until the PE file is loaded (for example, if it contains values computed based on the load addresses of several PE files) it can be possible to supply arbitrary code to run as the PE file is loaded, but this mechanism is platform-specific and might not be available in all conforming implementations of the CLI.

16.5.1 Data known at run time

When the correct value cannot be determined until type layout is computed, the user shall supply code as part of a type initializer to initialize the static data. The guarantees about type initialization are covered in §10.5.3.1. As will be explained below, global statics are modeled in the CLI as though they belonged to a type, so the same guarantees apply to both global and type statics.

Because the layout of managed types need not occur until a type is first referenced, it is not possible to statically initialize managed types by simply laying out the data in the PE file. Instead, there is a type initialization process that proceeds in the following steps:

1. All static variables are zeroed.
2. The user-supplied type initialization procedure, if any, is invoked as described in §10.5.3.

Within a type initialization procedure there are several techniques:

- *Generate explicit code* that stores constants into the appropriate fields of the static variables. For small data structures this can be efficient, but it requires that the initializer be converted to native code, which can prove to be both a code space and an execution time problem.
- *Box value types.* When the static variable is simply a boxed version of a primitive numeric type or a value type with explicit layout, introduce an additional static variable with known RVA that holds the unboxed instance and then simply use the `box` instruction to create the boxed copy.
- *Create a managed array from a static native array of data.* This can be done by marshaling the native array to a managed array. The specific marshaler to be used depends on the native array. e.g., it can be a safearray.
- *Default initialize a managed array of a value type.* The Base Class Library provides a method that zeroes the storage for every element of an array of unboxed value types (`System.Runtime.CompilerServices.InitializeArray`)

End informative text

17 Defining properties

A Property is declared by the using the `.property` directive. Properties shall only be declared inside of types (i.e., global properties are not supported).

ClassMember `::=`
`.property` *PropHeader* `'{'` *PropMember*`*` `'}'`

See §22.34 and §22.35 for how property information is stored in metadata.

PropHeader `::=`
[`specialname`][`rtspecialname`] *CallConv Type Id* `'('` *Parameters* `')'`

The `.property` directive specifies a calling convention (§15.3), type, name, and parameters in parentheses. `specialname` marks the property as *special* to other tools, while `rtspecialname` marks the property as *special* to the CLI. The signature for the property (i.e., the *PropHeader* production) shall match the signature of the property's `.get` method (see below)

[*Rationale:* There are currently no property names that are required to be marked with `rtspecialname`. It is provided for extensions, future standardization, and to increase consistency between the declaration of properties and methods (instance and type initializer methods shall be marked with this attribute). *end rationale*]

While the CLI places no constraints on the methods that make up a property, the CLS (see Partition I) specifies a set of consistency constraints.

A property can contain any number of methods in its body. The following table shows how these methods are identified, and provides short descriptions of each kind of item:

PropMember `::=`	**Description**	**Clause**
`|` `.custom` *CustomDecl*	Custom attribute.	21
`|` `.get` *CallConv Type* [*TypeSpec* `'::'`] *MethodName* `'('` *Parameters* `')'`	Specifies the getter for the property.	
`|` `.other` *CallConv Type* [*TypeSpec* `'::'`] *MethodName* `'('` *Parameters* `')'`	Specifies a method for the property other than the getter or setter.	
`|` `.set` *CallConv Type* [*TypeSpec* `'::'`] *MethodName* `'('` *Parameters* `')'`	Specifies the setter for the property.	
`|` *ExternSourceDecl*	`.line` or #line	5.7

`.get` specifies the *getter* for this property. The *TypeSpec* defaults to the current type. Only one *getter* can be specified for a property. To be CLS-compliant, the definition of *getter* shall be marked `specialname`.

`.set` specifies the *setter* for this property. The *TypeSpec* defaults to the current type. Only one *setter* can be specified for a property. To be CLS-compliant, the definition of *setter* shall be marked `specialname`.

`.other` is used to specify any other methods that this property comprises.

In addition, custom attributes (§21) or source line declarations can be specified.

[*Example:* This shows the declaration of the property called `count`.

```
.class public auto autochar MyCount extends [mscorlib]System.Object {
  .method virtual hidebysig public specialname instance int32 get_Count() {
  // body of getter
  }
```

```
    .method virtual hidebysig public specialname instance void set_Count(
        int32 newCount) {
    // body of setter
    }

    .method virtual hidebysig public instance void reset_Count() {
    // body of refresh method
    }

    // the declaration of the property
    .property int32 Count() {
      .get instance int32 MyCount::get_Count()
      .set instance void MyCount::set_Count(int32)
      .other instance void MyCount::reset_Count()
    }
  }
```

end example]

18 Defining events

Events are declared inside types, using the `.event` directive; there are no global events.

ClassMember ::=	**Clause**
`.event` *EventHeader* `'{'` *EventMember** `'}'`	
| …	9

See §22.13 and §22.11

EventHeader ::=
[`specialname`] [`rtspecialname`] [*TypeSpec*] *Id*

In typical usage, the *TypeSpec* (if present) identifies a delegate whose signature matches the arguments passed to the event's fire method.

The event head can contain the keywords `specialname` or `rtspecialname`. `specialname` marks the name of the property for other tools, while `rtspecialname` marks the name of the event as special for the runtime.

[*Rationale:* There are currently no event names that are required to be marked with `rtspecialname`. It is provided for extensions, future standardization, and to increase consistency between the declaration of events and methods (instance and type initializer methods shall be marked with this attribute). *end rationale*]

EventMember ::=	**Description**	**Clause**
`.addon` *CallConv Type* [*TypeSpec* `'::'`] *MethodName* `'('` *Parameters* `')'`	`Add` method for event.	
| `.custom` *CustomDecl*	Custom attribute.	21
| `.fire` *CallConv Type* [*TypeSpec* `'::'`] *MethodName* `'('` *Parameters* `')'`	Fire method for event.	
| `.other` *CallConv Type* [*TypeSpec* `'::'`] *MethodName* `'('` *Parameters* `')'`	Other method.	
| `.removeon` *CallConv Type* [*TypeSpec* `'::'`] *MethodName* `'('` *Parameters* `')'`	Remove method for event.	
| *ExternSourceDecl*	`.line` or `#line`	5.7

The `.addon` directive specifies the *add* method, and the *TypeSpec* defaults to the same type as the event. The CLS specifies naming conventions and consistency constraints for events, and requires that the definition of the *add* method be marked with `specialname`.

The `.removeon` directive specifies the *remove* method, and the *TypeSpec* defaults to the same type as the event. The CLS specifies naming conventions and consistency constraints for events, and requires that the definition of the *remove* method be marked with `specialname`.

The `.fire` directive specifies the *fire* method, and the *TypeSpec* defaults to the same type as the event. The CLS specifies naming conventions and consistency constraints for events, and requires that the definition of the *fire* method be marked with `specialname`.

An event can contain any number of other methods specified with the `.other` directive. From the point of view of the CLI, these methods are only associated with each other through the event. If they have special semantics, this needs to be documented by the implementer.

Events can also have custom attributes (§21) associated with them and they can declare source line information.

[*Example:* This shows the declaration of an event, its corresponding delegate, and typical implementations of the add, remove, and fire method of the event. The event and the methods are declared in a class called `Counter`.

```
  // the delegate
  .class private sealed auto autochar TimeUpEventHandler extends
       [mscorlib]System.Delegate {
    .method public hidebysig specialname rtspecialname instance void .ctor(object
        'object', native int 'method') runtime managed {}

    .method public hidebysig virtual instance void Invoke() runtime managed {}

    .method public hidebysig newslot virtual instance class
      [mscorlib]System.IAsyncResult BeginInvoke(class
      mscorlib]System.AsyncCallback callback, object 'object') runtime managed {}

    .method public hidebysig newslot virtual instance void EndInvoke(class
       [mscorlib]System.IAsyncResult result) runtime managed {}
  }


  // the class that declares the event
  .class public auto autochar Counter extends [mscorlib]System.Object {
    // field to store the handlers, initialized to null
    .field private class TimeUpEventHandler timeUpEventHandler
    // the event declaration
    .event TimeUpEventHandler startStopEvent {
      .addon instance void Counter::add_TimeUp(class TimeUpEventHandler 'handler')
      .removeon instance void Counter::remove_TimeUp(class TimeUpEventHandler
  'handler')
      .fire instance void Counter::fire_TimeUpEvent()
    }
    // the add method, combines the handler with existing delegates
    .method public hidebysig virtual specialname instance void add_TimeUp(class
        TimeUpEventHandler 'handler') {
      .maxstack 4
      ldarg.0
      dup

      ldfld    class TimeUpEventHandler Counter::TimeUpEventHandler
      ldarg    'handler'
      call     class[mscorlib]System.Delegate
        [mscorlib]System.Delegate::Combine(class [mscorlib]System.Delegate, class
        [mscorlib]System.Delegate)
      castclass TimeUpEventHandler
      stfld    class TimeUpEventHandler Counter::timeUpEventHandler
      ret
    }

    // the remove method, removes the handler from the delegate
    .method virtual public specialname void remove_TimeUp(class TimeUpEventHandler
          'handler') {
      .maxstack 4
      ldarg.0
      dup
      ldfld    class TimeUpEventHandler Counter::timeUpEventHandler

      ldarg    'handler'
      call     class[mscorlib]System.Delegate
         [mscorlib]System.Delegate::Remove(class
         [mscorlib]System.Delegate, class [mscorlib]System.Delegate)
      castclass TimeUpEventHandler
      stfld    class TimeUpEventHandler Counter::timeUpEventHandler
      ret
    }
```

```
    // the fire method
    .method virtual family specialname void fire_TimeUpEvent() {
      .maxstack 3
      ldarg.0
      ldfld    class TimeUpEventHandler Counter::timeUpEventHandler
      callvirt instance void TimeUpEventHandler::Invoke()
      ret
    }
  }      // end of class Counter
```

end example]

19 Exception handling

In the CLI, a method can define a range of CIL instructions that are said to be *protected*. This is called a *try block*. It can then associate one or more *handlers* with that try block. If an exception occurs during execution anywhere within the try block, an exception object is created that describes the problem. The CLI then takes over, transferring control from the point at which the exception was thrown, to the block of code that is willing to handle that exception. See Partition I.

No two handlers (fault, filter, catch, or finally) can have the same starting address. When an exception occurs it is necessary to convert the execution address to the correct most lexically nested try block in which the exception occurred.

SEHBlock ::=
TryBlock SEHClause [*SEHClause**]

The next few subclauses expand upon this simple description, by describing the five kinds of code block that take part in exception processing: `try`, `catch`, `filter`, `finally`, and `fault`. (Note that there are restrictions upon how many, and what kinds of *SEHClause* a given *TryBlock* can have; see Partition I for details.)

The remaining syntax items are described in detail below; they are collected here for reference.

TryBlock ::=	**Description**
`.try` *Label* `to` *Label*	Protect region from first label to prior to second
| `.try` *ScopeBlock*	*ScopeBlock* is protected

SEHClause ::=	**Description**
`catch` *TypeReference HandlerBlock*	Catch all objects of the specified type
| `fault` *HandlerBlock*	Handle all exceptions but not normal exit
| `filter` *Label HandlerBlock*	Enter handler only if filter succeeds
| `finally` *HandlerBlock*	Handle all exceptions and normal exit

HandlerBlock::=	**Description**
`handler` *Label* `to` *Label*	Handler range is from first label to prior to second
| *ScopeBlock*	*ScopeBlock* is the handler block

19.1 Protected blocks

A *try*, or *protected*, or *guarded*, block is declared with the `.try` directive.

TryBlock ::=	**Descriptions**
`.try` *Label* `to` *Label*	Protect region from first label to prior to second.
| `.try` *ScopeBlock*	*ScopeBlock* is protected

In the first case, the protected block is delimited by two labels. The first label is the first instruction to be protected, while the second label is the instruction just beyond the last one to be protected. Both labels shall be defined prior to this point.

The second case uses a scope block (§15.4.4) after the `.try` directive—the instructions within that scope are the ones to be protected.

19.2 Handler blocks

HandlerBlock `::=`	**Description**
`| handler` *Label* `to` *Label*	Handler range is from first label to prior to second
`|` *ScopeBlock*	*ScopeBlock* is the handler block

In the first case, the labels enclose the instructions of the handler block, the first label being the first instruction of the handler while the second is the instruction immediately after the handler. In the second case, the handler block is just a scope block.

19.3 Catch blocks

A catch block is declared using the `catch` keyword. This specifies the type of exception object the clause is designed to handle, and the handler code itself.

SEHClause `::=`
`catch` *TypeReference HandlerBlock*

[*Example:*

```
.try {
       …                          // protected instructions
       leave  exitSEH             // normal exit
} catch [mscorlib]System.FormatException {
       …                          // handle the exception
       pop                        // pop the exception object
       leave  exitSEH             // leave catch handler
}
exitSEH:                          // continue here
```

end example]

19.4 Filter blocks

A filter block is declared using the filter keyword.

SEHClause `::=` …
`| filter` *Label HandlerBlock*
`| filter` *Scope HandlerBlock*

The filter code begins at the specified label and ends at the first instruction of the handler block. (Note that the CLI demands that the filter block shall immediately precede, within the CIL stream, its corresponding handler block.)

[*Example:*

```
.method public static void m () {
    .try {
      …                    // protected instructions
      leave  exitSEH       // normal exit
    }

    filter {
      …                    // decide whether to handle
      pop                  // pop exception object
      ldc.i4.1             // EXCEPTION_EXECUTE_HANDLER
      endfilter            // return answer to CLI
    }
```

```
    {
      …                    // handle the exception
      pop                  // pop the exception object
      leave  exitSEH       // leave filter handler
    }
exitSEH:
    …
}
```

end example]

19.5 Finally blocks

A finally block is declared using the finally keyword. This specifies the handler code, with this grammar:

SEHClause `::=` …
`| finally` *HandlerBlock*

The last possible CIL instruction that can be executed in a finally handler shall be `endfinally`.

[*Example:*

```
.try {
       …                   // protected instructions
       leave exitTry       // shall use leave
} finally {
       …                   // finally handler
       endfinally
}
exitTry:                   // back to normal
```

19.6 Fault handlers

end example]

A fault block is declared using the fault keyword. This specifies the handler code, with this grammar:

SEHClause `::=` …
`| fault` *HandlerBlock*

The last possible CIL instruction that can be executed in a fault handler shall be `endfault`.

[*Example:*

```
.method public static void m() {
  startTry:
       …                   // protected instructions
       leave  exitSEH      // shall use leave
  endTry:

startFault:
       …                   // fault handler instructions
       endfault

endFault:
       .try startTry to endTry fault handler startFault to endFault

exitSEH:                   // back to normal
}
```

end example]

20 Declarative security

Many languages that target the CLI use attribute syntax to attach declarative security attributes to items in the metadata. This information is actually converted by the compiler into an XML-based representation that is stored in the metadata, see §22.11. By contrast, *ilasm* requires the conversion information to be represented in its input.

SecurityDecl ::=
.permissionset *SecAction* = '(' *Bytes* ')'
| .permission *SecAction TypeReference* '(' *NameValPairs* ')'

NameValPairs ::= *NameValPair* [',' *NameValPair*]*

NameValPair ::= *SQSTRING* '=' *SQSTRING*

In .permission, *TypeReference* specifies the permission class and *NameValPairs* specifies the settings. See §22.11

In .permissionset the bytes specify the encoded version of the security settings:

SecAction ::=	**Description**
assert	Assert permission so that callers do not need it.
| demand	Demand permission of all callers.
| deny	Deny permission so checks will fail.
| inheritcheck	Demand permission of a derived class.
| linkcheck	Demand permission of caller.
| permitonly	Reduce permissions so check will fail.
| reqopt	Request optional additional permissions.
| reqrefuse	Refuse to be granted these permissions.
| request	Hint that permission might be required.

21 Custom attributes

Custom attributes add user-defined annotations to the metadata. Custom attributes allow an instance of a type to be stored with any element of the metadata. This mechanism can be used to store application-specific information at compile time, and to access it either at runtime or when another tool reads the metadata. While any user-defined type can be used as an attribute, CLS compliance requires that attributes will be instances of types whose base class is `System.Attribute`. The CLI predefines some attribute types and uses them to control runtime behavior. Some languages predefine attribute types to represent language features not directly represented in the CTS. Users or other tools are welcome to define and use additional attribute types.

Custom attributes are declared using the directive `.custom`, followed by the method declaration for a type constructor, optionally followed by a *Bytes* in parentheses:

CustomDecl ::=
Ctor ['=' '(' *Bytes* ')']

The *Ctor* item represents a method declaration (§15.4), specific for the case where the method's name is `.ctor`. [*Example:*

```
.custom instance void myAttribute::.ctor(bool, bool) = ( 01 00 00 01 00
00 )
```

end example]

Custom attributes can be attached to *any* item in metadata, except a custom attribute itself. Commonly, custom attributes are attached to assemblies, modules, classes, interfaces, value types, methods, fields, properties, generic parameters, and events (the custom attribute is attached to the immediately preceding declaration)

The *Bytes* item is not required if the constructor takes no arguments. In such cases, all that matters is the presence of the custom attribute.

If the constructor takes parameters, their values shall be specified in the *Bytes* item. The format for this 'blob' is defined in §23.3.

[*Example:* The following shows a class that is marked with the attribute called `System.CLSCompliantAttribute` and a method that is marked with the attribute called `System.ObsoleteAttribute`.

```
.class public MyClass extends [mscorlib]System.Object
{ .custom instance void [mscorlib]System.CLSCompliantAttribute::.ctor(bool) =
    ( 01 00 01 00 00 )
  .method public static void CalculateTotals() cil managed
{ .custom instance void [mscorlib]System.ObsoleteAttribute::.ctor() =
    ( 01 00 00 00 )
  ret
}
```

end example]

21.1 CLS conventions: custom attribute usage

CLS imposes certain conventions upon the use of custom attributes in order to improve cross-language operation. See Partition I for details.

21.2 Attributes used by the CLI

There are two kinds of custom attributes, called *genuine custom attributes*, and *pseudo custom attributes*. Custom attributes and pseudo custom attributes are treated differently, at the time they are defined, as follows:

- A custom attribute is stored directly into the metadata; the'blob' which holds its defining data is stored as-is. That 'blob' can be retrieved later.

- A pseudo custom attribute is recognized because its name is one of a short list. Rather than store its 'blob' directly in metadata, that 'blob' is parsed, and the information it contains is used to set bits and/or fields within metadata tables. The 'blob' is then discarded; it cannot be retrieved later.

Pseudo custom attributes therefore serve to capture user directives, using the same familiar syntax the compiler provides for genuine custom attributes, but these user directives are then stored into the more space-efficient form of metadata tables. Tables are also faster to check at runtime than are genuine custom attributes.

Many custom attributes are invented by higher layers of software. They are stored and returned by the CLI, without its knowing or caring what they 'mean'. But all pseudo custom attributes, plus a collection of genuine custom attributes, are of special interest to compilers and to the CLI. An example of such custom attributes is `System.Reflection.DefaultMemberAttribute`. This is stored in metadata as a genuine custom attribute 'blob', but reflection uses this custom attribute when called to invoke the default member (property) for a type.

The following subclauses list all of the pseudo custom attributes and *distinguished* custom attributes, where *distinguished* means that the CLI and/or compilers pay direct attention to them, and their behavior is affected in some way.

In order to prevent name collisions into the future, all custom attributes in the `System` namespace are reserved for standardization.

21.2.1 Pseudo custom attributes

The following table lists the CLI pseudo custom attributes. (Not all of these attributes are specified in this Standard, but all of their names are reserved and shall not be used for other purposes. For details on these attributes, see the documentation for the corresponding class in Partition IV.) They are defined in the namespaces `System.Reflection`, `System.Runtime.CompilerServices`, and `System.Runtime.InteropServices` namespaces.

Attribute	Description
`AssemblyAlgorithmIDAttribute`	Records the ID of the hash algorithm used (reserved only)
`AssemblyFlagsAttribute`	Records the flags for this assembly (reserved only)
`DllImportAttribute`	Provides information about code implemented within an unmanaged library
`FieldOffsetAttribute`	Specifies the byte offset of fields within their enclosing class or value type
`InAttribute`	Indicates that a method parameter is an [in] argument
`MarshalAsAttribute`	Specifies how a data item should be marshalled between managed and unmanaged code (see §23.4).
`MethodImplAttribute`	Specifies details of how a method is implemented
`OutAttribute`	Indicates that a method parameter is an [out] argument
`StructLayoutAttribute`	Allows the caller to control how the fields of a class or value type are laid out in managed memory

These attributes affect bits and fields in metadata, as follows:

`AssemblyAlgorithmIDAttribute`: sets the *Assembly.HashAlgId* field.

`AssemblyFlagsAttribute`: sets the *Assembly.Flags* field.

`DllImportAttribute`: sets the *Method.Flags.PinvokeImpl* bit for the attributed method; also, adds a new row into the *ImplMap* table (setting *MappingFlags*, *MemberForwarded*, *ImportName* and *ImportScope* columns).

`FieldOffsetAttribute`: sets the *FieldLayout.OffSet* value for the attributed field.

`InAttribute`: sets the *Param.Flags.In* bit for the attributed parameter.

`MarshalAsAttribute`: sets the *Field.Flags.HasFieldMarshal* bit for the attributed field (or the *Param.Flags.HasFieldMarshal* bit for the attributed parameter); also enters a new row into the FieldMarshal table for both *Parent* and *NativeType* columns.

`MethodImplAttribute`: sets the *Method.ImplFlags* field of the attributed method.

`OutAttribute`: sets the *Param.Flags.Out* bit for the attributed parameter.

`StructLayoutAttribute`: sets the *TypeDef.Flags.LayoutMask* sub-field for the attributed type, and, optionally, the *TypeDef.Flags.StringFormatMask* sub-field, the *ClassLayout.PackingSize*,and *ClassLayout.ClassSize* fields for that type.

21.2.2 Custom attributes defined by the CLS

The CLS specifies certain Custom Attributes and requires that conformant languages support them. These attributes are located under `System`.

Attribute	Description
`AttributeUsageAttribute`	Used to specify how an attribute is intended to be used.
`ObsoleteAttribute`	Indicates that an element is not to be used.
`CLSCompliantAttribute`	Indicates whether or not an element is declared to be CLS compliant through an instance field on the attribute object.

21.2.3 Custom attributes for security

The following custom attributes are defined in the `System.Net` and `System.Security.Permissions` namespaces. Note that these are all base classes; the actual instances of security attributes found in assemblies will be sub-classes of these.

Attribute	Description
`CodeAccessSecurityAttribute`	This is the base attribute class for declarative security using custom attributes.
`DnsPermissionAttribute`	Custom attribute class for declarative security with DnsPermission
`EnvironmentPermissionAttribute`	Custom attribute class for declarative security with EnvironmentPermission.
`FileIOPermissionAttribute`	Custom attribute class for declarative security with FileIOPermission.
`ReflectionPermissionAttribute`	Custom attribute class for declarative security with ReflectionPermission.
`SecurityAttribute`	This is the base attribute class for declarative security from which CodeAccessSecurityAttribute is derived.
`SecurityPermissionAttribute`	Indicates whether the attributed method can affect security settings
`SocketPermissionAttribute`	Custom attribute class for declarative security with SocketPermission.
`WebPermissionAttribute`	Custom attribute class for declarative security with WebPermission.

Note that any other security-related custom attributes (i.e., any custom attributes that derive from `System.Security.Permissions.SecurityAttribute`) included into an assembly, can cause a conforming

implementaion of the CLI to reject such an assembly when it is loaded, or throw an exception at runtime if any attempt is made to access those security-related custom attributes. (This statement holds true for any custom attributes that cannot be resolved; security-related custom attributes are just one particular case)

21.2.4 Custom attributes for TLS

A custom attribute that denotes a TLS (thread-local storage, see §**Error! Reference source not found.**) field is defined in the `System` namespace.

Attribute	Description
`ThreadStaticAttribute`	Provides for type member fields that are relative for the thread.

21.2.5 Custom attributes, various

The following custom attributes control various aspects of the CLI:

Attribute	Namespace	Description
`ConditionalAttribute`	`System.Diagnostics`	Used to mark methods as callable, based on some compile-time condition. If the condition is false, the method will not be called
`DecimalConstantAttribute`	`System.Runtime.CompilerServices`	Stores the value of a decimal constant in metadata
`DefaultMemberAttribute`	`System.Reflection`	Defines the member of a type that is the default member used by reflection's InvokeMember.
`FaultModeAttribute`	`System.Runtime.CompilerServices`	Indicates whether exceptions from instruction checks are precise or imprecise.
`FlagsAttribute`	`System`	Custom attribute indicating an enumeration should be treated as a bitfield; that is, a set of flags
`IndexerNameAttribute`	`System.Runtime.CompilerServices`	Indicates the name by which a property having one or more parameters will be known in programming languages that do not support such a facility directly
`ParamArrayAttribute`	`System`	Indicates that the method will allow a variable number of arguments in its invocation

22 Metadata logical format: tables

This clause defines the structures that describe metadata, and how they are cross-indexed. This corresponds to how metadata is laid out, after being read into memory from a PE file. (For a description of metadata layout inside the PE file itself, see §24)

Metadata is stored in two kinds of structure: tables (arrays of records) and heaps. There are four heaps in any module: String, Blob, Userstring, and Guid. The first three are byte arrays (so valid indexes into these heaps might be 0, 23, 25, 39, etc). The Guid heap is an array of GUIDs, each 16 bytes wide. Its first element is numbered 1, its second 2, and so on.

Each entry in each column of each table is either a constant or an index.

Constants are either literal values (e.g., ALG_SID_SHA1 = 4, stored in the *HashAlgId* column of the *Assembly* table), or, more commonly, bitmasks. Most bitmasks (they are almost all called *Flags*) are 2 bytes wide (e.g., the *Flags* column in the *Field* table), but there are a few that are 4 bytes (e.g., the *Flags* column in the *TypeDef* table).

Each index is either 2 or 4 bytes wide. The index points into the same or another table, or into one of the four heaps. The size of each index column in a table is only made 4 bytes if it needs to be for that particular module. So, if a particular column indexes a table, or tables, whose highest row number fits in a 2-byte value, the indexer column need only be 2 bytes wide. Conversely, for tables containing 64K or more rows, an indexer of that table will be 4 bytes wide.

Indexes to tables begin at 1, so index 1 means the first row in any given metadata table. (An index value of zero denotes that it does not index a row at all; that is, it behaves like a null reference.)

There are two kinds of columns that index a metadata table. (For details of the physical representation of these tables, see §24.2.6):

- Simple – such a column indexes one, and only one, table. For example, the *FieldList* column in the *TypeDef* table always indexes the *Field* table. So all values in that column are simple integers, giving the row number in the target table
- Coded – such a column indexes any of several tables. For example, the *Extends* column in the *TypeDef* table can index into the *TypeDef* or *TypeRef* table. A few bits of that index value are reserved to define which table it targets. For the most part, this specification talks of index values after being decoded into row numbers within the target table. However, the specification includes a description of these coded indexes in the section that describes the physical layout of Metadata (§24).

Metadata preserves name strings, as created by a compiler or code generator, unchanged. Essentially, it treats each string as an opaque *blob*. In particular, it preserves case. The CLI imposes no limit on the length of names stored in metadata and subsequently processed by the CLI.

Matching AssemblyRefs and ModuleRefs to their corresponding Assembly and Module shall be performed case-blind (see Partition I). However, all other name matches (type, field, method, property, event) shall be exact – so that this level of resolution is the same across all platforms, whether their OS is case-sensitive or not.

Tables are given both a name (e.g., "Assembly") and a number (e.g., 0x20). The number for each table is listed immediately with its title in the following subclauses. The table numbers indicate the order in which their corresponding table shall appear in the PE file, and there is a set of bits (§24.2.6) saying whether a given table exists or not. The number of a table is the position within that set of bits.

A few of the tables represent extensions to regular CLI files. Specifically, ENCLog and ENCMap, which occur in temporary images, generated during "Edit and Continue" or "incremental compilation" scenarios, whilst debugging. Both table types are reserved for future use.

References to the methods or fields of a type are stored together in a metadata table called the *MemberRef* table. However, sometimes, for clearer explanation, this standard distinguishes between these two kinds of reference, calling them “MethodRef” and “FieldRef”.

Certain tables are required to be sorted by a primary key, as follows:

Table	Primary Key Column
ClassLayout	Parent
Constant	Parent
CustomAttribute	Parent
DeclSecurity	Parent
FieldLayout	Field
FieldMarshal	Parent
FieldRVA	Field
GenericParam	Owner
GenericParamConstraint	Owner
ImplMap	MemberForwarded
InterfaceImpl	Class
MethodImpl	Class
MethodSemantics	Association
NestedClass	NestedClass

Furthermore, the InterfaceImpl table is sorted using the Interface column as a secondary key, and the GenericParam table is sorted using the Number column as a secondary key.

Finally, the TypeDef table has a special ordering constraint: the definition of an enclosing class shall precede the definition of all classes it encloses.

Metadata items (records in the metadata tables) are addressed by metadata tokens. Uncoded metadata tokens are 4-byte unsigned integers, which contain the metadata table index in the most significant byte and a 1-based record index in the three least-significant bytes. Metadata tables and their respective indexes are described in §22.2 and later subclauses.

Coded metadata tokens also contain table and record indexes, but in a different format. For details on the encoding, see §24.2.6.

22.1 Metadata validation rules

This contains informative text only

The subclauses that follow describe the schema for each kind of metadata table, and explain the detailed rules that guarantee metadata emitted into any PE file is valid. Checking that metadata is valid ensures that later processing (such as checking the CIL instruction stream for type safety, building method tables, CIL-to-native-code compilation, and data marshalling) will not cause the CLI to crash or behave in an insecure fashion.

In addition, some of the rules are used to check compliance with the CLS requirements (see Partition I) even though these are not related to valid Metadata. These are marked with a trailing `[CLS]` tag.

The rules for valid metadata refer to an individual module. A module is any collection of metadata that *could* typically be saved to a disk file. This includes the output of compilers and linkers, or the output of script compilers (where the metadata is often held only in memory, but never actually saved to a file on disk).

The rules address intra-module validation only. As such, software that checks conformance with this standard need not resolve references or walk type hierarchies defined in other modules. However, even if two modules, A and B, analyzed separately, contain only valid metadata, they can still be in error when viewed together (e.g.,

a call from Module A, to a method defined in module B, might specify a call site signature that does not match the signatures defined for that method in B).

All checks are categorized as ERROR, WARNING, or CLS.

- An ERROR check reports something that might cause a CLI to crash or hang, it might run but produce wrong answers; or it might be entirely benign. Conforming implementations of the CLI can exist that will not accept metadata that violates an ERROR rule, and therefore such metadata is invalid and is not portable.
- A WARNING check reports something, not actually wrong, but possibly a slip on the part of the compiler. Normally, it indicates a case where a compiler could have encoded the same information in a more compact fashion or where the metadata represents a construct that can have no actual use at runtime. All conforming implementations shall support metadata that violate only WARNING rules; hence such metadata is both valid and portable.
- A CLS check reports lack of compliance with the Common Language Specification (see Partition I). Such metadata is both valid and portable, but programming languages might exist that cannot process it, even though all conforming implementations of the CLI support the constructs.

Validation rules fall into the following broad categories:

- **Number of Rows:** A few tables are allowed only one row (e.g., Module table). Most have no such restriction.
- **Unique Rows:** No table shall contain duplicate rows, where "duplicate" is defined in terms of its *key* column, or combination of columns.
- **Valid Indexes:** Columns which are indexes shall point somewhere sensible, as follows:
  - Every index into the String, Blob, or Userstring heaps shall point *into* that heap, neither before its start (offset 0), nor after its end.
  - Every index into the Guid heap shall lie between 1 and the maximum element number in this module, inclusive.
  - Every index (row number) into another metadata table shall lie between 0 and that table's row count + 1 (for some tables, the index can point just past the end of any target table, meaning it indexes nothing).
- **Valid Bitmasks:** Columns which are bitmasks shall have only valid permutations of bits set.
- **Valid RVAs:** There are restrictions upon fields and methods that are assigned RVAs (Relative Virtual Addresses, which are byte offsets, expressed from the address at which the corresponding PE file is loaded into memory).

Note that some of the rules listed below really don't say anything—for example, some rules state that a particular table is allowed zero or more rows—in which case, there is no way that the check can fail. This is done simply for completeness, to record that such details have indeed been addressed, rather than overlooked.

End informative text

The CLI imposes no limit on the length of names stored in metadata, and subsequently processed by a CLI implementation.

22.2 Assembly : 0x20

The *Assembly* table has the following columns:

- *HashAlgId* (a 4-byte constant of type AssemblyHashAlgorithm, §23.1.1)
- *MajorVersion*, *MinorVersion*, *BuildNumber*, *RevisionNumber* (each being 2-byte constants)
- *Flags* (a 4-byte bitmask of type AssemblyFlags, §23.1.2)

- *PublicKey* (an index into the Blob heap)
- *Name* (an index into the String heap)
- *Culture* (an index into the String heap)

The *Assembly* table is defined using the `.assembly` directive (§6.2); its columns are obtained from the respective `.hash algorithm`, `.ver`, `.publickey`, and `.culture` (§6.2.1). (For an example, see §6.2.)

This contains informative text only

1. The *Assembly* table shall contain zero or one row [ERROR]
2. *HashAlgId* shall be one of the specified values [ERROR]
3. *MajorVersion*, *MinorVersion*, *BuildNumber*, and *RevisionNumber* can each have any value
4. *Flags* shall have only those values set that are specified [ERROR]
5. *PublicKey* can be null or non-null
6. *Name* shall index a non-empty string in the String heap [ERROR]
7. The string indexed by *Name* can be of unlimited length
8. *Culture* can be null or non-null
9. If *Culture* is non-null, it shall index a single string from the list specified (§23.1.3) [ERROR]

[*Note: Name* is a simple name (e.g., "Foo", with no drive letter, no path, and no file extension); on POSIX-compliant systems, *Name* contains no colon, no forward-slash, no backslash, and no period. *end note*]

End informative text

22.3 AssemblyOS : 0x22

The *AssemblyOS* table has the following columns:

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

This record should not be emitted into any PE file. However, if present in a PE file, it shall be treated as if all its fields were zero. It shall be ignored by the CLI.

22.4 AssemblyProcessor : 0x21

The *AssemblyProcessor* table has the following column:

- *Processor* (a 4-byte constant)

This record should not be emitted into any PE file. However, if present in a PE file, it should be treated as if its field were zero. It should be ignored by the CLI.

22.5 AssemblyRef : 0x23

The *AssemblyRef* table has the following columns:

- *MajorVersion*, *MinorVersion*, *BuildNumber*, *RevisionNumber* (each being 2-byte constants)
- *Flags* (a 4-byte bitmask of type AssemblyFlags, §23.1.2)
- *PublicKeyOrToken* (an index into the Blob heap, indicating the public key or token that identifies the author of this Assembly)
- *Name* (an index into the String heap)

- *Culture* (an index into the String heap)
- *HashValue* (an index into the Blob heap)

The table is defined by the `.assembly extern` directive (§6.3). Its columns are filled using directives similar to those of the *Assembly* table except for the *PublicKeyOrToken* column, which is defined using the `.publickeytoken` directive. (For an example, see §6.3.)

This contains informative text only

1. *MajorVersion*, *MinorVersion*, *BuildNumber*, and *RevisionNumber* can each have any value
2. *Flags* shall have only one bit set, the `PublicKey` bit (§23.1.2). All other bits shall be zero. [ERROR]
3. *PublicKeyOrToken* can be null, or non-null (note that the *Flags*`.PublicKey` bit specifies whether the 'blob' is a full public key, or the short hashed token)
4. If non-null, then *PublicKeyOrToken* shall index a valid offset in the Blob heap [ERROR]
5. *Name* shall index a non-empty string, in the String heap (there is no limit to its length) [ERROR]
6. *Culture* can be null or non-null.
7. If non-null, it shall index a single string from the list specified (§23.1.3) [ERROR]
8. *HashValue* can be null or non-null
9. If non-null, then *HashValue* shall index a non-empty 'blob' in the Blob heap [ERROR]
10. The *AssemblyRef* table shall contain no duplicates (where duplicate rows are deemd to be those having the same *MajorVersion*, *MinorVersion*, *BuildNumber*, *RevisionNumber*, *PublicKeyOrToken*, *Name*, and *Culture*) [WARNING]

[*Note: Name* is a simple name (e.g., "Foo", with no drive letter, no path, and no file extension); on POSIX-compliant systems *Name* contains no colon, no forward-slash, no backslash, and no period. *end note*]

End informative text

22.6 AssemblyRefOS : 0x25

The *AssemblyRefOS* table has the following columns:

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

These records should not be emitted into any PE file. However, if present in a PE file, they should be treated as-if their fields were zero. They should be ignored by the CLI.

22.7 AssemblyRefProcessor : 0x24

The *AssemblyRefProcessor* table has the following columns:

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

These records should not be emitted into any PE file. However, if present in a PE file, they should be treated as-if their fields were zero. They should be ignored by the CLI.

22.8 ClassLayout : 0x0F

The *ClassLayout* table is used to define how the fields of a class or value type shall be laid out by the CLI. (Normally, the CLI is free to reorder and/or insert gaps between the fields defined for a class or value type.)

[*Rationale:* This feature is used to lay out a managed value type in exactly the same way as an unmanaged C struct, allowing a managed value type to be handed to unmanaged code, which then accesses the fields exactly as if that block of memory had been laid out by unmanaged code. *end rationale*]

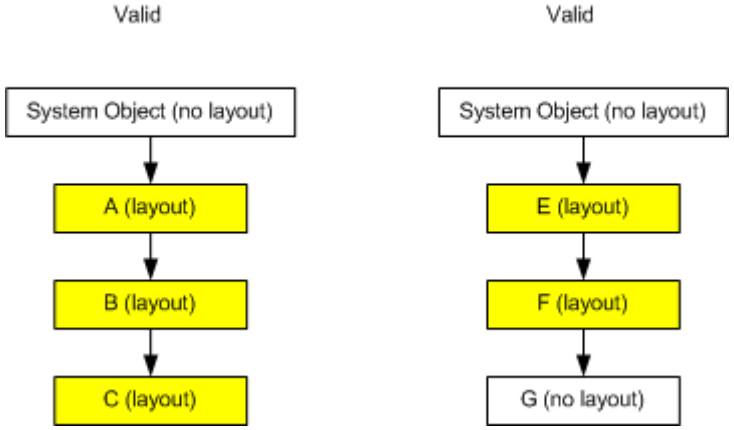
The information held in the *ClassLayout* table depends upon the *Flags* value for {*AutoLayout, SequentialLayout, ExplicitLayout*} in the owner class or value type.

A type *has layout* if it is marked *SequentialLayout* or *ExplicitLayout*. If any type within an inheritance chain has layout, then so shall all its base classes, up to the one that descends immediately from `System.ValueType` (if it exists in the type's hierarchy); otherwise, from `System.Object`.

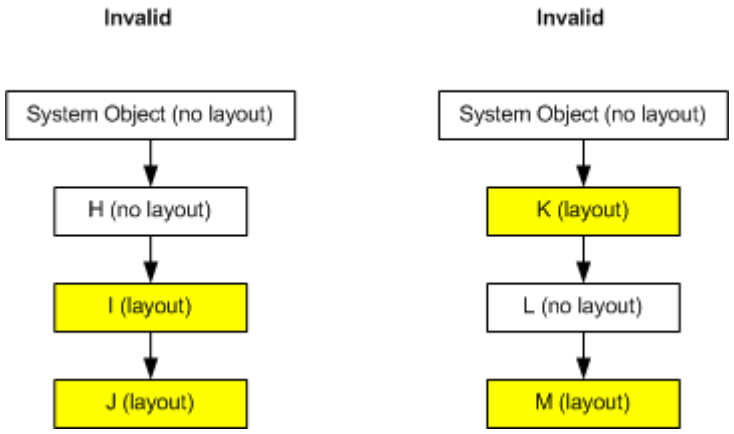
This contains informative text only

Layout cannot begin part way down the chain. But it *is* valid to *stop* "having layout" at any point down the chain.

For example, in the diagrams below, Class A derives from `System.Object`; class B derives from A; class C derives from B. `System.Object` has no layout. But A, B and C are all defined with layout, and that is valid.

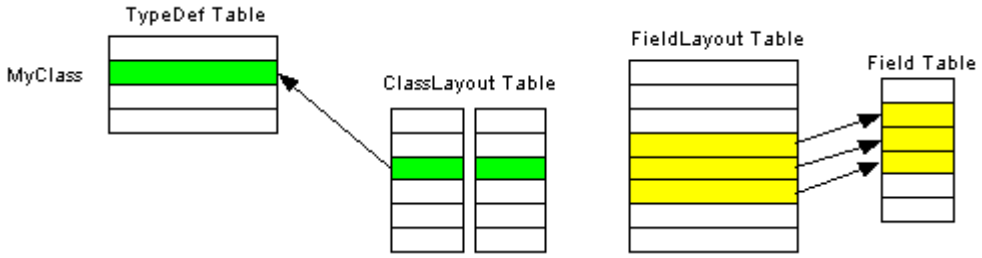


The situation with classes E, F, and G is similar. G has no layout, and this too is valid. The following picture shows two *invalid* setups:



On the left, the "chain with layout" does not start at the 'highest' class. And on the right, there is a 'hole' in the "chain with layout"

Layout information for a class or value type is held in two tables (*ClassLayout* and *FieldLayout*), as shown in the following diagram:



In this example, row 3 of the *ClassLayout* table points to row 2 in the *TypeDef* table (the definition for a Class, called "MyClass"). Rows 4–6 of the *FieldLayout* table point to corresponding rows in the *Field* table. This illustrates how the CLI stores the explicit offsets for the three fields that are defined in "MyClass" (there is always one row in the *FieldLayout* table for each field in the owning class or value type) So, the *ClassLayout* table acts as an extension to those rows of the *TypeDef* table that have layout info; since many classes do not have layout info, overall, this design saves space.

End informative text

The *ClassLayout* table has the following columns:

- *PackingSize* (a 2-byte constant)
- *ClassSize* (a 4-byte constant)
- *Parent* (an index into the *TypeDef* table)

The rows of the *ClassLayout* table are defined by placing `.pack` and `.size` directives on the body of the type declaration in which this type is declared (§10.2). When either of these directives is omitted, its corresponding value is zero. (See §10.7.)

ClassSize of zero does not mean the class has zero size. It means that no `.size` directive was specified at definition time, in which case, the actual size is calculated from the field types, taking account of packing size (default or specified) and natural alignment on the target, runtime platform.

This contains informative text only

1. A *ClassLayout* table can contain zero or more rows
2. *Parent* shall index a valid row in the *TypeDef* table, corresponding to a Class or ValueType (but not to an Interface) [ERROR]
3. The Class or ValueType indexed by *Parent* shall be *SequentialLayout* or *ExplicitLayout* (§23.1.15). (That is, *AutoLayout* types shall not own any rows in the *ClassLayout* table.) [ERROR]
4. If *Parent* indexes a *SequentialLayout* type, then:
   - *PackingSize* shall be one of {0, 1, 2, 4, 8, 16, 32, 64, 128}. (0 means use the default pack size for the platform on which the application is running.) [ERROR]
   - If *Parent* indexes a ValueType, then *ClassSize* shall be less than 1 MByte (0x100000 bytes). [ERROR]
5. If *Parent* indexes an *ExplicitLayout* type, then
   - if *Parent* indexes a ValueType, then *ClassSize* shall be less than 1 MByte (0x100000 bytes) [ERROR]
   - *PackingSize* shall be 0. (It makes no sense to provide explicit offsets for each field, as well as a packing size.) [ERROR]

6. Note that an *ExplicitLayout* type *might* result in a verifiable type, provided the layout does not create a type whose fields overlap.

7. Layout along the length of an inheritance chain shall follow the rules specified above (starting at 'highest' Type, with no 'holes', etc.) [ERROR]

End informative text

22.9 Constant : 0x0B

The *Constant* table is used to store compile-time, constant values for fields, parameters, and properties.

The *Constant* table has the following columns:

- *Type* (a 1-byte constant, followed by a 1-byte padding zero); see §23.1.16 . The encoding of *Type* for the **nullref** value for *FieldInit* in *ilasm* (§16.2) is `ELEMENT_TYPE_CLASS` with a *Value* of a 4-byte zero. Unlike uses of `ELEMENT_TYPE_CLASS` in signatures, this one is *not* followed by a type token.
- *Parent* (an index into the *Param*, *Field*, or *Property* table; more precisely, a *HasConstant* (§24.2.6) coded index)
- *Value* (an index into the Blob heap)

Note that *Constant* information does not directly influence runtime behavior, although it is visible via Reflection (and hence can be used to implement functionality such as that provided by `System.Enum.ToString`). Compilers inspect this information, at compile time, when importing metadata, but the value of the constant itself, if used, becomes embedded into the CIL stream the compiler emits. There are no CIL instructions to access the *Constant* table at runtime.

A row in the *Constant* table for a parent is created whenever a compile-time value is specified for that parent. (For an example, see §16.2.)

This contains informative text only

1. *Type* shall be exactly one of: `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`, or `ELEMENT_TYPE_STRING`; or `ELEMENT_TYPE_CLASS` with a *Value* of zero (§23.1.16) [ERROR]
2. *Type* shall not be any of: `ELEMENT_TYPE_I1`, `ELEMENT_TYPE_U2`, `ELEMENT_TYPE_U4`, or `ELEMENT_TYPE_U8` (§23.1.16) [CLS]
3. *Parent* shall index a valid row in the *Field*, *Property*, or *Param* table. [ERROR]
4. There shall be no duplicate rows, based upon *Parent* [ERROR]
5. *Type* shall match exactly the declared type of the *Param*, *Field*, or *Property* identified by *Parent* (in the case where the parent is an enum, it shall match exactly the underlying type of that enum). [CLS]

End informative text

22.10 CustomAttribute : 0x0C

The *CustomAttribute* table has the following columns:

- *Parent* (an index into *any* metadata table, except the *CustomAttribute* table itself; more precisely, a *HasCustomAttribute* (§24.2.6) coded index)
- *Type* (an index into the *MethodDef* or *MemberRef* table; more precisely, a *CustomAttributeType* (§24.2.6) coded index)
- *Value* (an index into the Blob heap)

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.

A row in the *CustomAttribute* table for a parent is created by the `.custom` attribute, which gives the value of the *Type* column and optionally that of the *Value* column (§21).

This contains informative text only

All binary values are stored in little-endian format (except for *PackedLen* items, which are used only as a count for the number of bytes to follow in a UTF8 string).

1. No *CustomAttribute* is required; that is, *Value* is permitted to be null.
2. *Parent* can be an index into *any* metadata table, *except* the *CustomAttribute* table itself [ERROR]
3. *Type* shall index a valid row in the *Method* or *MemberRef* table. That row shall be a constructor method (for the class of which this information forms an instance) [ERROR]
4. *Value* can be null or non-null.
5. If *Value* is non-null, it shall index a 'blob' in the Blob heap [ERROR]
6. The following rules apply to the overall structure of the *Value* 'blob' (§23.3):
   - *Prolog* shall be 0x0001 [ERROR]
   - There shall be as many occurrences of *FixedArg* as are declared in the Constructor method [ERROR]
   - *NumNamed* can be zero or more
   - There shall be exactly *NumNamed* occurrences of *NamedArg* [ERROR]
   - Each *NamedArg* shall be accessible by the caller [ERROR]
   - If *NumNamed* = 0 then there shall be no further items in the *CustomAttrib* [ERROR]
7. The following rules apply to the structure of *FixedArg* (§23.3):
   - If this item is not for a vector (a single-dimension array with lower bound of 0), then there shall be exactly one *Elem* [ERROR]
   - If this item is for a vector, then:
   - *NumElem* shall be 1 or more [ERROR]
   - This shall be followed by *NumElem* occurrences of *Elem* [ERROR]
8. The following rules apply to the structure of *Elem* (§23.3):
   - If this is a simple type or an enum (see §23.3 for how this is defined), then *Elem* consists simply of its value [ERROR]
   - If this is a string or a Type, then *Elem* consists of a *SerString* – *PackedLen* count of bytes, followed by the UTF8 characters [ERROR]
   - If this is a boxed simple value type (`bool`, `char`, `float32`, `float64`, `int8`, `int16`, `int32`, `int64`, `unsigned int8`, `unsigned int16`, `unsigned int32`, or `unsigned int64`), then *Elem* consists of the corresponding type denoter (`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`, or `ELEMENT_TYPE_R8`), followed by its value. [ERROR]
9. The following rules apply to the structure of *NamedArg* (§23.3):
   - The single byte `FIELD` (0x53) or `PROPERTY` (0x54) [ERROR]

- The type of the Field or Property is one of `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`, or the constant 0x50 (for an argument of type `System.Type`) [ERROR]
- The name of the Field or Property, respectively with the previous item, as a *SerString* – *PackedLen* count of bytes, followed by the UTF8 characters of the name [ERROR]
- A *FixedArg* (see above) [ERROR]

End informative text

22.11 DeclSecurity : 0x0E

Security attributes, which derive from `System.Security.Permissions.SecurityAttribute` (see Partition IV), can be attached to a *TypeDef*, a *Method*, or an *Assembly*. All constructors of this class shall take a `System.Security.Permissions.SecurityAction` value as their first parameter, describing what should be done with the permission on the type, method or assembly to which it is attached. Code access security attributes, which derive from `System.Security.Permissions. CodeAccessSecurityAttribute`, can have any of the security actions.

These different security actions are encoded in the *DeclSecurity* table as a 2-byte enum (see below). All security custom attributes for a given security action on a method, type, or assembly shall be gathered together, and one `System.Security.PermissionSet` instance shall be created, stored in the Blob heap, and referenced from the *DeclSecurity* table.

[*Note:* The general flow from a compiler's point of view is as follows. The user specifies a custom attribute through some language-specific syntax that encodes a call to the attribute's constructor. If the attribute's type is derived (directly or indirectly) from `System.Security.Permissions.SecurityAttribute` then it is a security custom attribute and requires special treatment, as follows (other custom attributes are handled by simply recording the constructor in the metadata as described in §22.10). The attribute object is constructed, and provides a method (*`CreatePermission`*) to convert it into a security permission object (an object derived from *`System.Security.Permission`*). All the permission objects attached to a given metadata item with the same security action are combined together into a `System.Security.PermissionSet`. This permission set is converted into a form that is ready to be stored in XML using its `ToXML` method to create a `System.Security.SecurityElement`. Finally, the XML that is required for the metadata is created using the `ToString` method on the security element. *end note*]

The *DeclSecurity* table has the following columns:

- *Action* (a 2-byte value)
- *Parent* (an index into the *TypeDef*, *MethodDef*, or *Assembly* table; more precisely, a *HasDeclSecurity* (§24.2.6) coded index)
- *PermissionSet* (an index into the Blob heap)

Action is a 2-byte representation of Security Actions (see `System.Security.SecurityAction` in Partition IV). The values 0–0xFF are reserved for future standards use. Values 0x20–0x7F and 0x100–0x07FF are for uses where the action can be ignored if it is not understood or supported. Values 0x80–0xFF and 0x0800–0xFFFF are for uses where the action shall be implemented for secure operation; in implementations where the action is not available, no access to the assembly, type, or method shall be permitted.

Security Action	Note	Explanation of behavior	Valid Scope
Assert	1	Without further checks, satisfy Demand for the specified permission.	Method, Type
Demand	1	Check that all callers in the call chain have been granted specified permission, throw `SecurityException` (see Partition IV) on failure.	Method, Type

Deny	1	Without further checks refuse Demand for the specified permission.	Method, Type
InheritanceDemand	1	The specified permission shall be granted in order to inherit from class or override virtual method.	Method, Type
LinkDemand	1	Check that the immediate caller has been granted the specified permission; throw `SecurityException` (see Partition IV) on failure.	Method, Type
NonCasDemand	2	Check that the current assembly has been granted the specified permission; throw `SecurityException` (see Partition IV) otherwise.	Method, Type
NonCasLinkDemand	2	Check that the immediate caller has been granted the specified permission; throw `SecurityException` (see Partition IV) otherwise.	Method, Type
PrejitGrant		Reserved for implementation-specific use.	Assembly
PermitOnly	1	Without further checks, refuse Demand for all permissions other than those specified.	Method, Type
RequestMinimum		Specify the minimum permissions required to run.	Assembly
RequestOptional		Specify the optional permissions to grant.	Assembly
RequestRefuse		Specify the permissions not to be granted.	Assembly

Note 1: The specified attribute shall derive from `System.Security.Permissions.CodeAccess-SecurityAttribute`

Note 2: The attribute shall derive from `System.Security.Permissions.SecurityAttribute`, but shall not derive from `System.Security.Permissions.CodeAccessSecurityAttribute`

Parent is a metadata token that identifies the *Method*, *Type*, or *Assembly* on which security custom attributes encoded in *PermissionSet* was defined.

PermissionSet is a 'blob' having the following format:

- A byte containing a period (.).
- A compressed int32 containing the number of attributes encoded in the blob.
- An array of attributes each containing the following:
    - A String, which is the fully-qualified type name of the attribute. (Strings are encoded as a compressed int to indicate the size followed by an array of UTF8 characters.)
    - A set of properties, encoded as the named arguments to a custom attribute would be (as in §23.3, beginning with NumNamed).

The permission set contains the permissions that were requested with an *Action* on a specific *Method*, *Type*, or *Assembly* (see *Parent*). In other words, the blob will contain an encoding of all the attributes on the *Parent* with that particular *Action*.

[*Note*: The first edition of this standard specified an XML encoding of a permission set. Implementations should continue supporting this encoding for backward compatibility. *end note*]

The rows of the *DeclSecurity* table are filled by attaching a `.permission` or `.permissionset` directive that specifies the *Action* and *PermissionSet* on a parent assembly (§6.6) or parent type or method (§10.2).

This contains informative text only

1. *Action* shall have only those values set that are specified [ERROR]

2. *Parent* shall be one of *TypeDef*, *MethodDef*, or *Assembly*. That is, it shall index a valid row in the *TypeDef* table, the *MethodDef* table, or the *Assembly* table. [ERROR]
3. If *Parent* indexes a row in the *TypeDef* table, that row should not define an Interface. The security system ignores any such parent; compilers should not emit such permissions sets. [WARNING]
4. If *Parent* indexes a TypeDef, then its *TypeDef.Flags.HasSecurity* bit shall be set [ERROR]
5. If *Parent* indexes a *MethodDef*, then its *MethodDef.Flags.HasSecurity* bit shall be set [ERROR]
6. *PermissionSet* shall index a 'blob' in the Blob heap [ERROR]
7. The format of the 'blob' indexed by *PermissionSet* shall represent a valid, encoded CLI object graph. (The encoded form of all standardized permissions is specified in Partition IV.) [ERROR]

End informative text

22.12 EventMap : 0x12

The *EventMap* table has the following columns:

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

Note that *EventMap* info does not directly influence runtime behavior; what counts is the information stored for each method that the event comprises.

The *EventMap* and *Event* tables result from putting the `.event` directive on a class (§18).

This contains informative text only

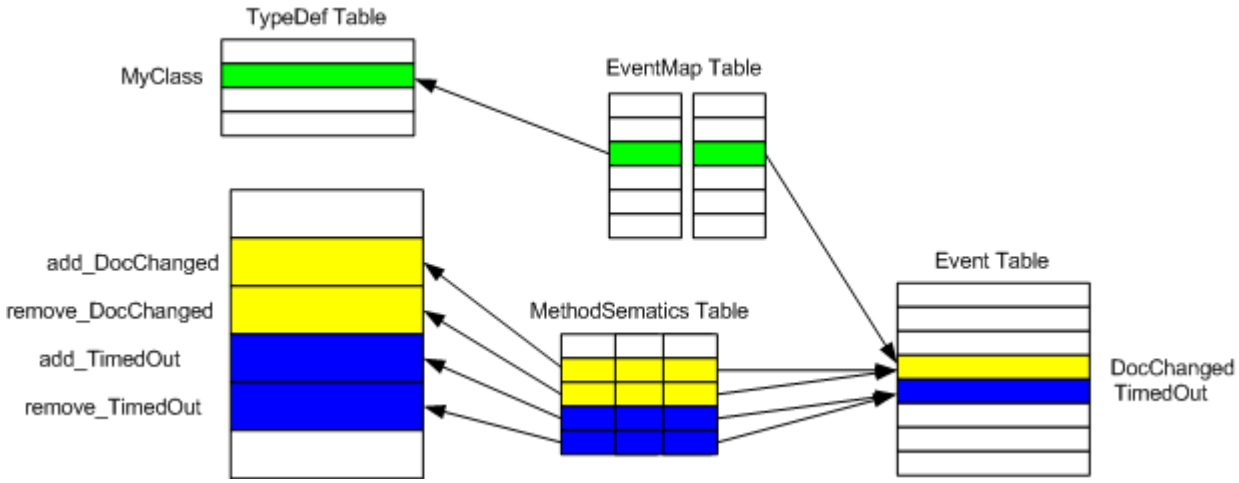
1. *EventMap* table can contain zero or more rows
2. There shall be no duplicate rows, based upon *Parent* (a given class has only one ‘pointer’ to the start of its event list) [ERROR]
3. There shall be no duplicate rows, based upon *EventList* (different classes cannot share rows in the *Event* table) [ERROR]

End informative text

22.13 Event : 0x14

Events are treated within metadata much like Properties; that is, as a way to associate a collection of methods defined on a given class. There are two required methods (*add_* and *remove_*) plus an optional one (*raise_*); others are permitted. All of the methods gathered together as an Event shall be defined on the class.

The association between a row in the *TypeDef* table and the collection of methods that make up a given Event is held in three separate tables (exactly analogous to the approach used for Properties), as follows:



Row 3 of the *EventMap* table indexes row 2 of the *TypeDef* table on the left (*MyClass*), whilst indexing row 4 of the *Event* table on the right (the row for an Event called *DocChanged*). This setup establishes that *MyClass* has an Event called *DocChanged*. But what methods in the *MethodDef* table are gathered together as 'belonging' to event *DocChanged*? That association is contained in the *MethodSemantics* table – its row 2 indexes event *DocChanged* to the right, and row 2 in the *MethodDef* table to the left (a method called add_DocChanged). Also, row 3 of the *MethodSemantics* table indexes *DocChanged* to the right, and row 3 in the *MethodDef* table to the left (a method called *remove_DocChanged*). As the shading suggests, *MyClass* has another event, called *TimedOut*, with two methods, *add_TimedOut* and *remove_TimedOut.*

Event tables do a little more than group together existing rows from other tables. The *Event* table has columns for *EventFlags*, *Name* (e.g., *DocChanged* and *TimedOut* in the example here), and *EventType.* In addition, the *MethodSemantics* table has a column to record whether the method it indexes is an *add_*, a *remove_*, a *raise_*, or *other* function.

The *Event* table has the following columns:

- *EventFlags* (a 2-byte bitmask of type *EventAttributes*, §23.1.4)
- *Name* (an index into the String heap)
- *EventType* (an index into a *TypeDef*, a *TypeRef*, or *TypeSpec* table; more precisely, a *TypeDefOrRef* (§24.2.6) coded index) (This corresponds to the Type of the Event; it is *not* the Type that owns this event.)

Note that *Event* information does not directly influence runtime behavior; what counts is the information stored for each method that the event comprises.

The *EventMap* and *Event* tables result from putting the `.event` directive on a class (§18).

This contains informative text only

1. The *Event* table can contain zero or more rows
2. Each row shall have one, and only one, owner row in the *EventMap* table [ERROR]
3. *EventFlags* shall have only those values set that are specified (all combinations valid) [ERROR]
4. *Name* shall index a non-empty string in the String heap [ERROR]
5. The *Name* string shall be a valid CLS identifier [CLS]
6. *EventType* can be null or non-null
7. If *EventType* is non-null, then it shall index a valid row in the *TypeDef* or *TypeRef* table [ERROR]

8. If *EventType* is non-null, then the row in the *TypeDef*, *TypeRef*, or *TypeSpec* table that it indexes shall be a Class (not an Interface or a ValueType) [ERROR]

9. For each row, there shall be one *add_* and one *remove_* row in the *MethodSemantics* table [ERROR]

10. For each row, there can be zero or one *raise_* row, as well as zero or more *other* rows in the *MethodSemantics* table [ERROR]

11. Within the rows owned by a given row in the *TypeDef* table, there shall be no duplicates based upon *Name* [ERROR]

12. There shall be no duplicate rows based upon *Name*, where *Name* fields are compared using CLS conflicting-identifier-rules [CLS]

End informative text

22.14 ExportedType : 0x27

The *ExportedType* table holds a row for each type:

a. 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.

   The actual target row in a *TypeDef* table is given by the combination of *TypeDefId* (in effect, row number) and *Implementation* (in effect, the module that holds the target *TypeDef* table). Note that this is the only occurrence in metadata of *foreign* tokens; that is, token values that have a meaning in *another* module. (A regular token value is an index into a table in the *current* module); OR

b. Originally defined in this Assembly but now moved to another Assembly. *Flags* must have *IsTypeForwarder* set and *Implementation* is an *AssemblyRef* indicating the Assembly the type may now be found in.

The full name of the type need not be stored directly. Instead, it can be split into two parts at any included "." (although typically this is done at the last "." in the full name). The part preceding the "." is stored as the *TypeNamespace* and that following the "." is stored as the *TypeName*. If there is no "." in the full name, then the *TypeNamespace* shall be the index of the empty string.

The *ExportedType* table has the following columns:

- *Flags* (a 4-byte bitmask of type *TypeAttributes*, §23.1.15)
- *TypeDefId* (a 4-byte index into a *TypeDef* table of another module in this Assembly). This column 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. Ignored and should be zero if *Flags* has *IsTypeForwarder* set.
- *TypeName* (an index into the String heap)
- *TypeNamespace* (an index into the String heap)
- *Implementation*. This is an index (more precisely, an *Implementation* (§24.2.6) coded index) into either of the following tables:
  - *File* table, where that entry says which module in the current assembly holds the *TypeDef*
  - *ExportedType* table, where that entry is the enclosing Type of the current nested Type
  - *AssemblyRef* table, where that entry says in which assembly the type may now be found (*Flags* must have the *IsTypeForwarder* flag set).

The rows in the *ExportedType* table are the result of the `.class extern` directive (§6.7).

This contains informative text only

The term "*FullName*" refers to the string created as follows: if the *TypeNamespace* is null, then use the *TypeName*, otherwise use the concatenation of *Typenamespace*, ".", and *TypeName*.

1. The *ExportedType* table can contain zero or more rows
2. There shall be no entries in the *ExportedType* table for Types that are defined in the current module—just for Types defined in other modules within the Assembly [ERROR]
3. *Flags* shall have only those values set that are specified [ERROR]
4. If *Implementation* indexes the *File* table, then *Flags.VisibilityMask* shall be `public` (§23.1.15) [ERROR]
5. If *Implementation* indexes the *ExportedType* table, then *Flags.VisibilityMask* shall be `NestedPublic` (§23.1.15) [ERROR]
6. If non-null, *TypeDefId* should index a valid row in a *TypeDef* table in a module somewhere within this Assembly (but not this module), and the row so indexed should have its *Flags.Public* = 1 (§23.1.15) [WARNING]
7. *TypeName* shall index a non-empty string in the String heap [ERROR]
8. *TypeNamespace* can be null, or non-null
9. If *TypeNamespace* is non-null, then it shall index a non-empty string in the String heap [ERROR]
10. *FullName* shall be a valid CLS identifier [CLS]
11. If this is a nested Type, then *TypeNamespace* should be null, and *TypeName* should represent the unmangled, simple name of the nested Type [ERROR]
12. *Implementation* shall be a valid index into either of the following: [ERROR]
    - the *File* table; that file shall hold a definition of the target Type in its *TypeDef* table
    - a different row in the current *ExportedType* table—this identifies the enclosing Type of the current, nested Type
13. *FullName* shall match exactly the corresponding *FullName* for the row in the *TypeDef* table indexed by *TypeDefId* [ERROR]
14. Ignoring nested Types, there shall be no duplicate rows, based upon *FullName* [ERROR]
15. For nested Types, there shall be no duplicate rows, based upon *TypeName* and enclosing Type [ERROR]
16. The complete list of Types exported from the current Assembly is given as the catenation of the *ExportedType* table with all public Types in the current *TypeDef* table, where "public" means a *Flags.VisibilityMask* of either *Public* or *NestedPublic*. There shall be no duplicate rows, in this concatenated table, based upon *FullName* (add Enclosing Type into the duplicates check if this is a nested Type) [ERROR]

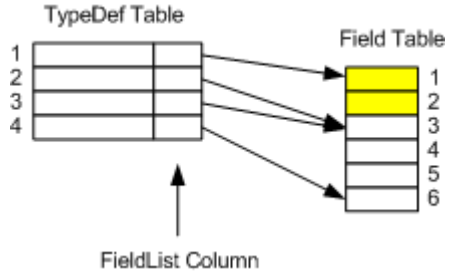
End informative text

22.15 Field : 0x04

The *Field* table has the following columns:

- *Flags* (a 2-byte bitmask of type *FieldAttributes*, §23.1.5)
- *Name* (an index into the String heap)
- *Signature* (an index into the Blob heap)

Conceptually, each row in the *Field* table is owned by one, and only one, row in the *TypeDef* table. However, the owner of any row in the *Field* table is not stored anywhere in the *Field* table itself. There is merely a 'forward-pointer' from each row in the *TypeDef* table (the *FieldList* column), as shown in the following illustration.



The *TypeDef* table has rows 1–4. The first row in the *TypeDef* table corresponds to a pseudo type, inserted automatically by the CLI. It is used to denote those rows in the *Field* table corresponding to global variables. The *Field* table has rows 1–6. Type 1 (pseudo type for 'module') owns rows 1 and 2 in the *Field* table. Type 2 owns no rows in the *Field* table, even though its *FieldList* indexes row 3 in the *Field* table. Type 3 owns rows 3–5 in the *Field* table. Type 4 owns row 6 in the *Field* table. So, in the *Field* table, rows 1 and 2 belong to Type 1 (global variables); rows 3–5 belong to Type 3; row 6 belongs to Type 4.

Each row in the *Field* table results from a top-level `.field` directive (§5.10), or a `.field` directive inside a Type (§10.2). (For an example, see §14.5.)

This contains informative text only

1. The *Field* table can contain zero or more rows
2. Each row shall have one, and only one, owner row in the *TypeDef* table [ERROR]
3. The owner row in the *TypeDef* table shall not be an Interface [CLS]
4. *Flags* shall have only those values set that are specified [ERROR]
5. The *FieldAccessMask* subfield of Flags shall contain precisely one of `CompilerControlled`, `Private`, `FamANDAssem`, `Assembly`, `Family`, `FamORAssem`, or `Public` (§23.1.5) [ERROR]
6. *Flags* can set either or neither of `Literal` or `InitOnly`, but not both (§23.1.5) [ERROR]
7. If *Flags.Literal* = 1 then *Flags.Static* shall also be 1 (§23.1.5) [ERROR]
8. If *Flags.RTSpecialName* = 1, then *Flags.SpecialName* shall also be 1 (§23.1.5) [ERROR]
9. If *Flags.HasFieldMarshal* = 1, then this row shall 'own' exactly one row in the *FieldMarshal* table (§23.1.5) [ERROR]
10. If *Flags.HasDefault* = 1, then this row shall 'own' exactly one row in the *Constant* table (§23.1.5) [ERROR]
11. If *Flags.HasFieldRVA* = 1, then this row shall 'own' exactly one row in the *Field's RVA* table (§23.1.5) [ERROR]
12. *Name* shall index a non-empty string in the String heap [ERROR]
13. The *Name* string shall be a valid CLS identifier [CLS]
14. *Signature* shall index a valid field signature in the Blob heap [ERROR]
15. If *Flags.CompilerControlled* = 1 (§23.1.5), then this row is ignored completely in duplicate checking.
16. If the owner of this field is the internally-generated type called `<Module>`, it denotes that this field is defined at module scope (commonly called a global variable). In this case:
    - *Flags.Static* shall be 1 [ERROR]

- *Flags.MemberAccessMask* subfield shall be one of `Public`, `CompilerControlled`, or `Private` (§23.1.5) [ERROR]
- module-scope fields are not allowed [CLS]

17. There shall be no duplicate rows in the *Field* table, based upon owner+*Name*+*Signature* (where owner is the owning row in the *TypeDef* table, as described above) (Note however that if *Flags.CompilerControlled* = 1, then this row is completely excluded from duplicate checking) [ERROR]
18. There shall be no duplicate rows in the *Field* table, based upon owner+*Name*, where *Name* fields are compared using CLS conflicting-identifier-rules. So, for example,`"int i"` and `"float i"` would be considered CLS duplicates. (Note however that if *Flags.CompilerControlled* = 1, then this row is completely excluded from duplicate checking, as noted above) [CLS]
19. If this is a field of an Enum then:
    a. owner row in *TypeDef* table shall derive directly from `System.Enum` [ERROR]
    b. the owner row in *TypeDef* table shall have no other instance fields [CLS]
    c. its *Signature* shall be one of `ELEMENT_TYPE_U1`, `ELEMENT_TYPE_I2`, `ELEMENT_TYPE_I4`, or `ELEMENT_TYPE_I8` (§23.1.16): [CLS]
20. its *Signature* shall be an integral type. [ERROR]

End informative text

22.16 FieldLayout : 0x10

The *FieldLayout* table has the following columns:

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

Note that each Field in any Type is defined by its Signature. When a Type instance (i.e., an object) is laid out by the CLI, each Field is one of four kinds:

- Scalar: for any member of built-in type, such as `int32`. The size of the field is given by the size of that intrinsic, which varies between 1 and 8 bytes
- ObjectRef: for `ELEMENT_TYPE_CLASS`, `ELEMENT_TYPE_STRING`, `ELEMENT_TYPE_OBJECT`, `ELEMENT_TYPE_ARRAY`, `ELEMENT_TYPE_SZARRAY`
- Pointer: for `ELEMENT_TYPE_PTR`, `ELEMENT_TYPE_FNPTR`
- ValueType: for `ELEMENT_TYPE_VALUETYPE`. The instance of that ValueType is actually laid out in this object, so the size of the field is the size of that ValueType

Note that metadata specifying explicit structure layout can be valid for use on one platform but not on another, since some of the rules specified here are dependent on platform-specific alignment rules.

A row in the *FieldLayout* table is created if the `.field` directive for the parent field has specified a field offset (§16).

This contains informative text only

1. A *FieldLayout* table can contain zero or more or rows
2. The Type whose Fields are described by each row of the *FieldLayout* table shall have *Flags.ExplicitLayout* (§23.1.15) set [ERROR]
3. *Offset* shall be zero or more [ERROR]

4. *Field* shall index a valid row in the *Field* table [ERROR]
5. *Flags.Static* for the row in the *Field* table indexed by *Field* shall be non-static (i.e., zero 0) [ERROR]
6. Among the rows owned by a given Type there shall be no duplicates, based upon *Field*. That is, a given Field of a Type cannot be given two offsets. [ERROR]
7. Each Field of kind *ObjectRef* shall be naturally aligned within the Type [ERROR]
8. Among the rows owned by a given Type it is perfectly valid for several rows to have the same value of *Offset*. ObjectRef and a valuetype cannot have the same offset [ERROR]
9. Every Field of an ExplicitLayout Type shall be given an offset; that is, it shall have a row in the *FieldLayout* table [ERROR]

End informative text

22.17 FieldMarshal : 0x0D

The *FieldMarshal* table has two columns. It 'links' an existing row in the *Field* or *Param* table, to information in the Blob heap that defines how that field or parameter (which, as usual, covers the method return, as parameter number 0) shall be marshalled when calling to or from unmanaged code via PInvoke dispatch.

Note that *FieldMarshal* information is used only by code paths that arbitrate operation with unmanaged code. In order to execute such paths, the caller, on most platforms, would be installed with elevated security permission. Once it invokes unmanaged code, it lies outside the regime that the CLI can check—it is simply trusted not to violate the type system.

The *FieldMarshal* table has the following columns:

- *Parent* (an index into *Field* or *Param* table; more precisely, a *HasFieldMarshal* (§24.2.6) coded index)
- *NativeType* (an index into the Blob heap)

For the detailed format of the 'blob', see §23.4

A row in the *FieldMarshal* table is created if the `.field` directive for the parent field has specified a `marshal` attribute (§16.1).

This contains informative text only

1. A *FieldMarshal* table can contain zero or more rows
2. *Parent* shall index a valid row in the Field or Param table (*Parent* values are encoded to say which of these two tables each refers to) [ERROR]
3. *NativeType* shall index a non-null 'blob' in the Blob heap [ERROR]
4. No two rows shall point to the same parent. In other words, after the *Parent* values have been decoded to determine whether they refer to the *Field* or the *Param* table, no two rows can point to the same row in the Field table or in the Param table [ERROR]
5. The following checks apply to the *MarshalSpec* 'blob' (§23.4):
   a. *NativeIntrinsic* shall be exactly one of the constant values in its production (§23.4) [ERROR]
   b. If `ARRAY`, then *ArrayElemType* shall be exactly one of the constant values in its production [ERROR]
   c. If `ARRAY`, then *ParamNum* can be zero
   d. If `ARRAY`, then *ParamNum* cannot be < 0 [ERROR]

e. If ARRAY, and *ParamNum* > 0, then *Parent* shall point to a row in the Param table, not in the Field table [ERROR]

f. If ARRAY, and *ParamNum* > 0, then *ParamNum* cannot exceed the number of parameters supplied to the *MethodDef* (or *MethodRef* if a VARARG call) of which the parent Param is a member [ERROR]

g. If ARRAY, then *ElemMult* shall be >= 1 [ERROR]

h. If ARRAY and *ElemMult* != 1 issue a warning, because it is probably a mistake [WARNING]

i. If ARRAY and *ParamNum* = 0, then *NumElem* shall be >= 1 [ERROR]

j. If ARRAY and *ParamNum* != 0 and NumElem != 0 then issue a warning, because it is probably a mistake [WARNING]

End informative text

22.18 FieldRVA : 0x1D

The *FieldRVA* table has the following columns:

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

Conceptually, each row in the *FieldRVA* table is an extension to exactly one row in the *Field table*, and records the RVA (Relative Virtual Address) within the image file at which this field's initial value is stored.

A row in the *FieldRVA* table is created for each static parent field that has specified the optional `data` label §16). The RVA column is the relative virtual address of the data in the PE file (§16.3).

This contains informative text only

1. *RVA* shall be non-zero [ERROR]
2. *RVA* shall point into the current module's data area (not its metadata area) [ERROR]
3. *Field* shall index a valid row in the *Field* table [ERROR]
4. Any field with an RVA shall be a ValueType (not a Class or an Interface). Moreover, it shall not have any private fields (and likewise for any of its fields that are themselves ValueTypes). (If any of these conditions were breached, code could overlay that global static and access its private fields.) Moreover, no fields of that ValueType can be Object References (into the GC heap) [ERROR]
5. So long as two RVA-based fields comply with the previous conditions, the ranges of memory spanned by the two ValueTypes can overlap, with no further constraints. This is not actually an additional rule; it simply clarifies the position with regard to overlapped RVA-based fields

End informative text

22.19 File : 0x26

The *File* table has the following columns:

- *Flags* (a 4-byte bitmask of type FileAttributes, §23.1.6)
- *Name* (an index into the String heap)
- *HashValue* (an index into the Blob heap)

The rows of the *File* table result from `.file` directives in an Assembly (§6.2.3)

This contains informative text only

1. *Flags* shall have only those values set that are specified (all combinations valid) [ERROR]
2. *Name* shall index a non-empty string in the String heap. It shall be in the format <filename>.<extension> (e.g., "foo.dll", but *not* "c:\utils\foo.dll") [ERROR]
3. *HashValue* shall index a non-empty 'blob' in the Blob heap [ERROR]
4. There shall be no duplicate rows; that is, rows with the same *Name* value [ERROR]
5. If this module contains a row in the *Assembly* table (that is, if this module "holds the manifest") then there shall not be any row in the *File* table for this module; i.e., no self-reference [ERROR]
6. If the *File* table is empty, then this, by definition, is a single-file assembly. In this case, the *ExportedType* table should be empty [WARNING]

End informative text

22.20 GenericParam : 0x2A

The *GenericParam* table has the following columns:

- *Number* (the 2-byte index of the generic parameter, numbered left-to-right, from zero)
- *Flags* (a 2-byte bitmask of type *GenericParamAttributes*, §23.1.7)
- *Owner* (an index into the *TypeDef* or *MethodDef* table, specifying the Type or Method to which this generic parameter applies; more precisely, a *TypeOrMethodDef* (§24.2.6) 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)

The *GenericParam* table stores the generic parameters used in generic type definitions and generic method definitions. These generic parameters can be constrained (i.e., generic arguments shall extend some class and/or implement certain interfaces) or unconstrained. (Such constraints are stored in the *GenericParamConstraint* table.)

Conceptually, each row in the *GenericParam* table is owned by one, and only one, row in either the *TypeDef* or *MethodDef* tables.

[*Example*:

```
.class Dict`2<([mscorlib]System.IComparable) K, V>
```

The generic parameter *K* of class *Dict* is constrained to implement `System.IComparable`.

```
.method static void ReverseArray<T>(!!0[] 'array')
```

There is no constraint on the generic parameter *T* of the generic method `ReverseArray`.

end example]

This contains informative text only

1. *GenericParam* table can contain zero or more rows
2. Each row shall have one, and only one, owner row in the *TypeDef* or *MethodDef* table (i.e., no row sharing) [ERROR]
3. Every generic type shall own one row in the *GenericParam* table for each of its generic parameters [ERROR]
4. Every generic method shall own one row in the *GenericParam* table for each of its generic parameters [ERROR]

Flags:

- Can hold the value *Covariant* or *Contravariant*, but only if the owner row corresponds to a generic interface, or a generic delegate class. [ERROR]
- Otherwise, shall hold the value *None* indicating nonvariant (i.e., where the parameter is nonvariant or the owner is a non delegate class, a value-type, or a generic method) [ERROR]

If *Flags* == Covariant then the corresponding generic parameter can appear in a type definition only as [ERROR]:

- The result type of a method
- A generic parameter to an inherited interface

If *Flags* == Contravariant then the corresponding generic parameter can appear in a type definition only as the argument to a method [ERROR]

Number shall have a value >= 0 and < the number of generic parameters in owner type or method. [ERROR]

Successive rows of the *GenericParam* table that are owned by the same method shall be ordered by increasing *Number* value; there shall be no gaps in the *Number* sequence [ERROR]

Name shall be non-null and index a string in the String heap [ERROR]

[*Rationale*: Otherwise, Reflection output is not fully usable. *end rationale*]

There shall be no duplicate rows based upon *Owner+Name* [ERROR] [*Rationale*: Otherwise, code using Reflection cannot disambiguate the different generic parameters. *end rationale*]

There shall be no duplicate rows based upon *Owner+Number* [ERROR]

End informative text

22.21 GenericParamConstraint : 0x2C

The *GenericParamConstraint* table has the following 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* (§24.2.6) coded index)

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.

Note that if *Constraint* is a *TypeRef* to `System.ValueType`, then it means the constraint type shall be `System.ValueType`, or one of its sub types. However, since `System.ValueType` itself is a reference type, this particular mechanism does not guarantee that the type is a non-reference type.

This contains informative text only

1. The *GenericParamConstraint* table can contain zero or more rows
2. Each row shall have one, and only one, owner row in the *GenericParam* table (i.e., no row sharing) [ERROR]
3. Each row in the *GenericParam* table shall 'own' a separate row in the *GenericParamConstraint* table for each constraint that generic parameter has [ERROR]

4. All of the rows in the *GenericParamConstraint* table that are owned by a given row in the *GenericParam* table shall form a contiguous range (of rows) [ERROR]
5. Any generic parameter (corresponding to a row in the *GenericParam table*) shall own zero or one row in the *GenericParamConstraint table* corresponding to a class constraint. [ERROR]
6. Any generic parameter (corresponding to a row in the *GenericParam table*) shall own zero or more rows in the *GenericParamConstraint* table corresponding to an interface constraint. [ERROR]
7. There shall be no duplicate rows based upon *Owner+Constraint* [ERROR]
8. *Constraint* shall not reference `System.Void`. [ERROR]

End informative text

22.22 ImplMap : 0x1C

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*).

[*Note:* A typical example would be: associate the managed Method stored in row N of the *Method* table (so *MemberForwarded* would have the value N) with the routine called "GetEnvironmentVariable" (the string indexed by *ImportName)* in the DLL called "kernel32" (the string in the *ModuleRef* table indexed by *ImportScope*). The CLI intercepts calls to managed Method number N, and instead forwards them as calls to the unmanaged routine called "GetEnvironmentVariable" in "kernel32.dll" (including marshalling any arguments, as required)

The CLI does not support this mechanism to access *fields* that are exported from a DLL, only methods. *end note*]

The *ImplMap* table has the following columns:

- *MappingFlags* (a 2-byte bitmask of type *PInvokeAttributes*, §23.1.7)
- *MemberForwarded* (an index into the *Field* or *MethodDef* table; more precisely, a *MemberForwarded* (§24.2.6) coded index). However, it only ever indexes the *MethodDef* table, since *Field* export is not supported.
- *ImportName* (an index into the String heap)
- *ImportScope* (an index into the *ModuleRef* table)

A row is entered in the *ImplMap* table for each parent Method (§15.5) that is defined with a `.pinvokeimpl` interoperation attribute specifying the *MappingFlags*, *ImportName*, and *ImportScope*.

This contains informative text only

1. *ImplMap* can contain zero or more rows
2. *MappingFlags* shall have only those values set that are specified [ERROR]
3. *MemberForwarded* shall index a valid row in the *MethodDef* table [ERROR]
4. The *MappingFlags.CharSetMask* (§23.1.7) in the row of the *MethodDef* table indexed by *MemberForwarded* shall have at most one of the following bits set: `CharSetAnsi`, `CharSetUnicode`, or `CharSetAuto` (if none is set, the default is `CharSetNotSpec`) [ERROR]
5. *ImportName* shall index a non-empty string in the String heap [ERROR]
6. *ImportScope* shall index a valid row in the *ModuleRef* table [ERROR]

7. The row indexed in the *MethodDef* table by *MemberForwarded* shall have its *Flags.PinvokeImpl* = 1, and *Flags.Static* = 1 [ERROR]

End informative text

22.23 InterfaceImpl : 0x09

The *InterfaceImpl* table has the following columns:

- *Class* (an index into the *TypeDef* table)
- *Interface* (an index into the *TypeDef*, *TypeRef*, or *TypeSpec* table; more precisely, a *TypeDefOrRef* (§24.2.6) coded index)

The *InterfaceImpl* table records the interfaces a type implements explicitly. Conceptually, each row in the *InterfaceImpl* table indicates that *Class* implements *Interface.*

This contains informative text only

1. The *InterfaceImpl* table can contain zero or more rows
2. *Class* shall be non-null [ERROR]
3. If *Class* is non-null, then:
   a. *Class* shall index a valid row in the *TypeDef* table [ERROR]
   b. *Interface* shall index a valid row in the *TypeDef* or *TypeRef* table [ERROR]
   c. The row in the *TypeDef*, *TypeRef*, or *TypeSpec* table indexed by *Interface* shall be an interface (*Flags.Interface* = 1), not a Class or ValueType [ERROR]
4. There should be no duplicates in the *InterfaceImpl* table, based upon non-null *Class* and *Interface* values [WARNING]
5. There can be many rows with the same value for *Class* (since a class can implement many interfaces)
6. There can be many rows with the same value for *Interface* (since many classes can implement the same interface)

End informative text

22.24 ManifestResource : 0x28

The *ManifestResource* table has the following columns:

- *Offset* (a 4-byte constant)
- *Flags* (a 4-byte bitmask of type *ManifestResourceAttributes*, §23.1.9)
- *Name* (an index into the String heap)
- *Implementation* (an index into a *File* table, a *AssemblyRef* table, or null; more precisely, an *Implementation* (§24.2.6) coded index)

The *Offset* specifies the byte offset within the referenced file at which this resource record begins. The *Implementation* specifies which file holds this resource. The rows in the table result from `.mresource` directives on the Assembly (§6.2.2).

This contains informative text only

1. The *ManifestResource* table can contain zero or more rows

2. *Offset* shall be a valid offset into the target file, starting from the Resource entry in the CLI header [ERROR]
3. *Flags* shall have only those values set that are specified [ERROR]
4. The *VisibilityMask* (§23.1.9) subfield of *Flags* shall be one of `Public` or `Private` [ERROR]
5. *Name* shall index a non-empty string in the String heap [ERROR]
6. *Implementation* can be null or non-null (if null, it means the resource is stored in the current file)
7. If *Implementation* is null, then *Offset* shall be a valid offset in the current file, starting from the Resource entry in the CLI header [ERROR]
8. If *Implementation* is non-null, then it shall index a valid row in the *File* or *AssemblyRef* table [ERROR]
9. There shall be no duplicate rows, based upon *Name* [ERROR]
10. If the resource is an index into the *File* table, *Offset* shall be zero [ERROR]

End informative text

22.25 MemberRef : 0x0A

The *MemberRef* table combines two sorts of references, to Methods and to Fields of a class, known as 'MethodRef' and 'FieldRef', respectively. The *MemberRef* table has the following columns:

- *Class* (an index into the *MethodDef*, *ModuleRef*,*TypeDef*, *TypeRef*, or *TypeSpec* tables; more precisely, a MemberRefParent (§24.2.6) coded index)
- *Name* (an index into the String heap)
- *Signature* (an index into the Blob heap)

An entry is made into the *MemberRef* table whenever a reference is made in the CIL code to a method or field which is defined in another module or assembly. (Also, an entry is made for a call to a method with a `VARARG` signature, even when it is defined in the same module as the call site.)

This contains informative text only

1. *Class* shall be one of the following: [ERROR]
   a. a *TypeRef* token, if the class that defines the member is defined in another module. (Note that it is unusual, but valid, to use a *TypeRef* token when the member is defined in this same module, in which case, its *TypeDef* token can be used instead.)
   b. a *ModuleRef* token, if the member is defined, in another module of the same assembly, as a global function or variable.
   c. a *MethodDef* token, when used to supply a call-site signature for a vararg method that is defined in this module. The *Name* shall match the *Name* in the corresponding *MethodDef* row. The *Signature* shall match the *Signature* in the target method definition [ERROR]
   d. a *TypeSpec* token, if the member is a member of a generic type
2. *Class* shall not be null (as this would indicate an unresolved reference to a global function or variable) [ERROR]
3. *Name* shall index a non-empty string in the String heap [ERROR]
4. The *Name* string shall be a valid CLS identifier [CLS]
5. *Signature* shall index a valid field or method signature in the Blob heap. In particular, it shall embed exactly one of the following 'calling conventions': [ERROR]
   a. `DEFAULT` (0x0)

b. `VARARG` (0x5)

c. `FIELD` (0x6)

d. `GENERIC (0x10)`

6. The *MemberRef* table shall contain no duplicates, where duplicate rows have the same *Class*, *Name*, and *Signature* [WARNING]

7. *Signature* shall not have the `VARARG` (0x5) calling convention [CLS]

8. There shall be no duplicate rows, where *Name* fields are compared using CLS conflicting-identifier-rules. (In particular, note that the return type and whether parameters are marked `ELEMENT_TYPE_BYREF` (§23.1.16) are ignored in the CLS. For example, `.method int32 M()` and `.method float64 M()` result in duplicate rows by CLS rules. Similarly, `.method void N(int32 i)` and `.method void N(int32& i)` also result in duplicate rows by CLS rules.) [CLS]

9. If *Class* and *Name* resolve to a field, then that field shall not have a value of `CompilerControlled` (§23.1.5) in its *Flags.FieldAccessMask* subfield [ERROR]

10. If *Class* and *Name* resolve to a method, then that method shall not have a value of `CompilerControlled` in its *Flags.MemberAccessMask* (§23.1.10) subfield [ERROR]

11. The type containing the definition of a *MemberRef* shall be a *TypeSpec* representing an instantiated type.

End informative text

22.26 MethodDef : 0x06

The *MethodDef* table has the following columns:

- *RVA* (a 4-byte constant)
- *ImplFlags* (a 2-byte bitmask of type *MethodImplAttributes*, §23.1.10)
- *Flags* (a 2-byte bitmask of type *MethodAttributes*, §23.1.10)
- *Name* (an index into the String heap)
- *Signature* (an index into the Blob heap)
- *ParamList* (an index into the *Param* table). It marks the first of a contiguous run of Parameters owned by this method. The run continues to the smaller of:
  - the last row of the *Param* table
  - the next run of Parameters, found by inspecting the *ParamList* of the next row in the *MethodDef* table

Conceptually, every row in the *MethodDef* table is owned by one, and only one, row in the *TypeDef* table.

The rows in the *MethodDef* table result from `.method` directives (§15). The RVA column is computed when the image for the PE file is emitted and points to the `COR_ILMETHOD` structure for the body of the method (§25.4)

[*Note*: If *Signature* is `GENERIC` (0x10), the generic arguments are described in the *GenericParam* table (§22.20). *end note*]

This contains informative text only

1. The *MethodDef* table can contain zero or more rows

2. Each row shall have one, and only one, owner row in the *TypeDef* table [ERROR]

3. *ImplFlags* shall have only those values set that are specified [ERROR]

4. *Flags* shall have only those values set that are specified [ERROR]
5. If Name is .ctor and the method is marked `SpecialName`, there shall not be a row in the *GenericParam* table which has this *MethodDef* as its owner. [ERROR]
6. The *MemberAccessMask* (§23.1.10) subfield of *Flags* shall contain precisely one of `CompilerControlled`, `Private`, `FamANDAssem`, `Assem`, `Family`, `FamORAssem`, or `Public` [ERROR]
7. The following combined bit settings in *Flags* are invalid [ERROR]
   a. `Static | Final`
   b. `Static | Virtual`
   c. `Static | NewSlot`
   d. `Final | Abstract`
   e. `Abstract | PinvokeImpl`
   f. `CompilerControlled | SpecialName`
   g. `CompilerControlled | RTSpecialName`
8. An abstract method shall be virtual. So, if *Flags.Abstract* = 1 then *Flags.Virtual* shall also be 1 [ERROR]
9. If *Flags.RTSpecialName* = 1 then *Flags.SpecialName* shall also be 1 [ERROR]
10. If *Flags.HasSecurity* = 1, then at least one of the following conditions shall be true: [ERROR]
    - this Method owns at least row in the *DeclSecurity* table
    - this Method has a custom attribute called *SuppressUnmanagedCodeSecurityAttribute*
11. If this Method owns one (or more) rows in the *DeclSecurity* table then *Flags.HasSecurity* shall be 1 [ERROR]
12. If this Method has a custom attribute called *SuppressUnmanagedCodeSecurityAttribute* then *Flags.HasSecurity* shall be 1 [ERROR]
13. A Method can have a custom attribute called *DynamicSecurityMethodAttribute*, but this has no effect whatsoever upon the value of its *Flags.HasSecurity*
14. *Name* shall index a non-empty string in the String heap [ERROR]
15. Interfaces cannot have instance constructors. So, if this Method is owned by an Interface, then its *Name* cannot be `.ctor` [ERROR]
16. The *Name* string shall be a valid CLS identifier (unless *Flags.RTSpecialName* is set - for example, `.cctor` is valid) [CLS]
17. *Signature* shall index a valid method signature in the Blob heap [ERROR]
18. If *Flags.CompilerControlled* = 1, then this row is ignored completely in duplicate checking
19. If the owner of this method is the internally-generated type called `<Module>`, it denotes that this method is defined at module scope. [*Note*: In C++, the method is called *global* and can be referenced only within its compiland, from its point of declaration forwards. *end note*] In this case:
    a. *Flags.Static* shall be 1 [ERROR]
    b. *Flags.Abstract* shall be 0 [ERROR]
    c. *Flags.Virtual* shall be 0 [ERROR]
    d. *Flags.MemberAccessMask* subfield shall be one of `CompilerControlled`, `Public`, or `Private` [ERROR]

e. module-scope methods are not allowed [CLS]

20. It makes no sense for ValueTypes, which have no *identity*, to have synchronized methods (unless they are boxed). So, if the owner of this method is a ValueType then the method cannot be synchronized. That is, *ImplFlags.Synchronized* shall be 0 [ERROR]

21. There shall be no duplicate rows in the *MethodDef* table, based upon owner + *Name* + *Signature* (where owner is the owning row in the *TypeDef* table). (Note that the *Signature* encodes whether or not the method is generic, and for generic methods, it encodes the number of generic parameters.) (Note, however, that if *Flags.CompilerControlled* = 1, then this row is excluded from duplicate checking) [ERROR]

22. There shall be no duplicate rows in the *MethodDef* table, based upon owner + *Name* + *Signature*, where *Name* fields are compared using CLS conflicting-identifier-rules; also, the Type defined in the signatures shall be different. So, for example, `"int i"` and `"float i"` would be considered CLS duplicates; also, the return type of the method is ignored (Note, however, that if *Flags.CompilerControlled* = 1, this row is excluded from duplicate checking as explained above.) [CLS]

23. If `Final`, `NewSlot`, or `Strict` are set in *Flags*, then *Flags.Virtual* shall also be set [ERROR]

24. If *Flags.PInvokeImpl* is set, then *Flags.Virtual* shall be 0 [ERROR]

25. If *Flags.Abstract != 1* then exactly one of the following shall also be true: [ERROR]
    - RVA != 0
    - Flags.PInvokeImpl = 1
    - ImplFlags.Runtime = 1

26. If the method is `CompilerControlled`, then the RVA shall be non-zero or marked with `PinvokeImpl` = 1 [ERROR]

27. *Signature* shall have exactly one of the following managed calling conventions [ERROR]
    a. `DEFAULT` (0x0)
    b. `VARARG` (0x5)
    c. `GENERIC` (0x10)

28. *Signature* shall have the calling convention `DEFAULT` (0x0) or `GENERIC` (0x10). [CLS]

29. *Signature:* If and only if the method is not `Static` then the calling convention byte in *Signature* has its `HASTHIS` (0x20) bit set [ERROR]

30. *Signature:* If the method is `static`, then the `HASTHIS` (0x20) bit in the calling convention shall be 0 [ERROR]

31. If `EXPLICITTHIS` (0x40) in the signature is set, then `HASTHIS` (0x20) shall also be set (note that if `EXPLICITTHIS` is set, then the code is not verifiable) [ERROR]

32. The `EXPLICITTHIS` (0x40) bit can be set only in signatures for function pointers: signatures whose MethodDefSig is preceded by `FNPTR` (0x1B) [ERROR]

33. If *RVA* = 0, then either: [ERROR]
    - Flags.Abstract = 1, or
    - ImplFlags.Runtime = 1, or
    - Flags.PinvokeImpl = 1, or

34. If *RVA* != 0, then: [ERROR]
    a. *Flags.Abstract* shall be 0, and

b. *ImplFlags.CodeTypeMask* shall have exactly one of the following values: `Native`, `CIL`, or `Runtime`, and

c. *RVA* shall point into the CIL code stream in this file

35. If *Flags.PinvokeImpl* = 1 then [ERROR]

    - *RVA* = 0 *and* the method owns a row in the *ImplMap* table

36. If *Flags.RTSpecialName* = 1 then *Name* shall be one of: [ERROR]

    a. `.ctor` (an object constructor method)

    b. `.cctor` (a class constructor method)

37. Conversely, if *Name* is any of the above special names then *Flags.RTSpecialName* shall be set [ERROR]

38. If *Name* = `.ctor` (an object constructor method) then:

    a. return type in *Signature* shall be `ELEMENT_TYPE_VOID` (§23.1.16) [ERROR]

    b. *Flags.Static* shall be 0 [ERROR]

    c. *Flags.Abstract* shall be 0 [ERROR]

    d. *Flags.Virtual* shall be 0 [ERROR]

    e. 'Owner' type shall be a valid Class or ValueType (not `<Module>` and not an Interface) in the TypeDef table [ERROR]

    f. there can be zero or more `.ctors` for any given 'owner'

39. If *Name* = `.cctor` (a class constructor method) then:

    a. the return type in *Signature* shall be `ELEMENT_TYPE_VOID` (§23.1.16) [ERROR]

    b. *Signature* shall have `DEFAULT` (0x0) for its calling convention [ERROR]

    c. there shall be no parameters supplied in *Signature* [ERROR]

    d. *Flags.Static* shall be set [ERROR]

    e. *Flags.Virtual* shall be clear [ERROR]

    f. *Flags.Abstract* shall be clear [ERROR]

40. Among the set of methods owned by any given row in the *TypeDef* table there can only be 0 or 1 methods named `.cctor` [ERROR]

End informative text

22.27 MethodImpl : 0x19

MethodImpl tables let a compiler override the default inheritance rules provided by the CLI. Their original use was to allow a class `C`, that inherited method `M` from both interfaces `I` and `J`, to provide implementations for both methods (rather than have only one slot for `M` in its vtable). However, *MethodImpl*s can be used for other reasons too, limited only by the compiler writer's ingenuity within the constraints defined in the Validation rules below.

In the example above, *Class* specifies `C`, *MethodDeclaration* specifies `I::M`, *MethodBody* specifies the method which provides the implementation for `I::M` (either a method body within `C`, or a method body implemented by a base class of `C`).

The *MethodImpl* table has the following columns:

- *Class* (an index into the *TypeDef* table)

- *MethodBody* (an index into the *MethodDef* or *MemberRef* table; more precisely, a *MethodDefOrRef* (§24.2.6) coded index)
- *MethodDeclaration* (an index into the *MethodDef* or *MemberRef* table; more precisely, a *MethodDefOrRef* (§24.2.6) coded index)

ILAsm uses the `.override` directive to specify the rows of the *MethodImpl* table (§10.3.2 and §15.4.1).

This contains informative text only

1. The *MethodImpl* table can contain zero or more rows
2. *Class* shall index a valid row in the *TypeDef* table [ERROR]
3. *MethodBody* shall index a valid row in the *MethodDef* or *MemberRef* table [ERROR]
4. The method indexed by *MethodDeclaration* shall have *Flags.Virtual* set [ERROR]
5. The owner Type of the method indexed by *MethodDeclaration* shall not have *Flags.Sealed* = 0 [ERROR]
6. The method indexed by *MethodBody* shall be a member of *Class* or some base class of *Class* (*MethodImpl*s do not allow compilers to 'hook' arbitrary method bodies) [ERROR]
7. The method indexed by *MethodBody* shall be virtual [ERROR]
8. The method indexed by *MethodBody* shall have its *Method.RVA* != 0 (cannot be an unmanaged method reached via PInvoke, for example) [ERROR]
9. *MethodDeclaration* shall index a method in the ancestor chain of *Class* (reached via its *Extends* chain) or in the interface tree of *Class* (reached via its *InterfaceImpl* entries) [ERROR]
10. The method indexed by *MethodDeclaration* shall not be final (its *Flags.Final* shall be 0) [ERROR]
11. If MethodDeclaration has the Strict flag set, the method indexed by *MethodDeclaration* shall be accessible to *Class*. [ERROR]
12. The method signature defined by *MethodBody* shall match those defined by *MethodDeclaration* [ERROR]
13. There shall be no duplicate rows, based upon *Class+MethodDeclaration* [ERROR]

End informative text

22.28 MethodSemantics : 0x18

The *MethodSemantics* table has the following columns:

- *Semantics* (a 2-byte bitmask of type *MethodSemanticsAttributes*, §23.1.12)
- *Method* (an index into the *MethodDef* table)
- *Association* (an index into the *Event* or *Property* table; more precisely, a *HasSemantics* (§24.2.6) coded index)

The rows of the *MethodSemantics* table are filled by `.property` (§17) and `.event` directives (§18). (See §22.13 for more information.)

This contains informative text only

1. *MethodSemantics* table can contain zero or more rows
2. *Semantics* shall have only those values set that are specified [ERROR]
3. *Method* shall index a valid row in the *MethodDef* table, and that row shall be for a method defined on the same class as the Property or Event this row describes [ERROR]

4. All methods for a given Property or Event shall have the same accessibility (ie the *MemberAccessMask* subfield of their *Flags* row) and cannot be `CompilerControlled` [CLS]
5. *Semantics:* constrained as follows:
    - If this row is for a Property, then exactly one of `Setter`, `Getter`, or `Other` shall be set [ERROR]
    - If this row is for an Event, then exactly one of `AddOn`, `RemoveOn`, `Fire`, or `Other` shall be set [ERROR]
6. If this row is for an Event, and its *Semantics* is `Addon` or `RemoveOn`, then the row in the *MethodDef* table indexed by *Method* shall take a Delegate as a parameter, and return void [ERROR]
7. If this row is for an Event, and its *Semantics* is `Fire`, then the row indexed in the *MethodDef* table by *Method* can return any type
8. For each property, there shall be a setter, or a getter, or both [CLS]
9. Any getter method for a property whose *Name* is **xxx** shall be called **get_xxx** [CLS]
10. Any setter method for a property whose *Name* is **xxx** shall be called **set_xxx** [CLS]
11. If a property provides both getter and setter methods, then these methods shall have the same value in the *Flags.MemberAccessMask* subfield [CLS]
12. If a property provides both getter and setter methods, then these methods shall have the same value for their *Method.Flags.Virtual* [CLS]
13. Any getter and setter methods shall have *Method.Flags.SpecialName* = 1 [CLS]
14. Any getter method shall have a return type which matches the signature indexed by the *Property.Type* field [CLS]
15. The last parameter for any setter method shall have a type which matches the signature indexed by the *Property.Type* field [CLS]
16. Any setter method shall have return type `ELEMENT_TYPE_VOID` (§23.1.16) in *Method.Signature* [CLS]
17. If the property is indexed, the indexes for getter and setter shall agree in number and type [CLS]
18. Any *AddOn* method for an event whose *Name* is **xxx** shall have the signature: `void add_xxx (<DelegateType> handler)` [CLS]
19. Any *RemoveOn* method for an event whose *Name* is **xxx** shall have the signature: `void remove_xxx(<DelegateType> handler)` [CLS]
20. Any *Fire* method for an event whose *Name* is **xxx** shall have the signature: `void raise_xxx(Event e)` [CLS]

End informative text

22.29 MethodSpec : 0x2B

The *MethodSpec* table has the following columns:

- *Method* (an index into the *MethodDef* or *MemberRef* table, specifying to which generic method this row refers; that is, which generic method this row is an instantiation of; more precisely, a *MethodDefOrRef* (§24.2.6) coded index)
- *Instantiation* (an index into the *Blob* heap (§23.2.15), holding the signature of this instantiation)

The *MethodSpec* table records the signature of an instantiated generic method.

Each unique instantiation of a generic method (i.e., a combination of Method and Instantiation) shall be represented by a single row in the table.

This contains informative text only

1. The *MethodSpec* table can contain zero or more rows
2. One or more rows can refer to the same row in the *MethodDef* or *MemberRef* table. (There can be multiple instantiations of the same generic method.)
3. The signature stored at *Instantiation* shall be a valid instantiation of the signature of the generic method stored at *Method* [ERROR]
4. There shall be no duplicate rows based upon *Method+Instantiation* [ERROR]

End informative text

22.30 Module : 0x00

The *Module* table has the following columns:

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

The *Mvid* column shall index a unique GUID in the GUID heap (§24.2.5) that identifies this instance of the module. The *Mvid* can be ignored on read by conforming implementations of the CLI. The *Mvid* should be newly generated for every module, using the algorithm specified in ISO/IEC 11578:1996 (Annex A) or another compatible algorithm.

[*Note:* The term GUID stands for Globally Unique IDentifier, a 16-byte long number typically displayed using its hexadecimal encoding. A GUID can be generated by several well-known algorithms including those used for UUIDs (Universally Unique IDentifiers) in RPC and CORBA, as well as CLSIDs, GUIDs, and IIDs in COM. *end note*]

[*Rationale:* While the VES itself makes no use of the Mvid, other tools (such as debuggers, which are outside the scope of this standard) rely on the fact that the Mvid almost always differs from one module to another. *end rationale*]

The *Generation*, *EncId*, and *EncBaseId* columns can be written as zero, and can be ignored by conforming implementations of the CLI. The rows in the *Module* table result from `.module` directives in the Assembly (§6.4).

This contains informative text only

1. The *Module* table shall contain one and only one row [ERROR]
2. *Name* shall index a non-empty string. This string should match exactly any corresponding *ModuleRef.Name* string that resolves to this module. [ERROR]
3. *Mvid* shall index a non-null GUID in the Guid heap [ERROR]

End informative text

22.31 ModuleRef : 0x1A

The *ModuleRef* table has the following column:

- *Name* (an index into the String heap)

The rows in the *ModuleRef* table result from `.module extern` directives in the Assembly (§6.5).

This contains informative text only

1. *Name* shall index a non-empty string in the String heap. This string shall enable the CLI to locate the target module (typically, it might name the file used to hold the module) [ERROR]
2. There should be no duplicate rows [WARNING]
3. *Name* should match an entry in the *Name* column of the *File* table. Moreover, that entry shall enable the CLI to locate the target module (typically it might name the file used to hold the module) [ERROR]

End informative text

22.32 NestedClass : 0x29

The *NestedClass* table has the following columns:

- *NestedClass* (an index into the *TypeDef* table)
- *EnclosingClass* (an index into the *TypeDef* table)

NestedClass is defined as lexically 'inside' the text of its enclosing Type.

This contains informative text only

The *NestedClass* table records which Type definitions are nested within which other Type definition. In a typical high-level language, the nested class is defined as lexically 'inside' the text of its enclosing Type

1. The *NestedClass* table can contain zero or more rows
2. *NestedClass* shall index a valid row in the *TypeDef* table [ERROR]
3. *EnclosingClass* shall index a valid row in the *TypeDef* table (note particularly, it is not allowed to index the *TypeRef* table) [ERROR]
4. There should be no duplicate rows (ie same values for *NestedClass* and *EnclosingClass*) [WARNING]
5. A given Type can only be nested by one encloser. So, there cannot be two rows with the same value for *NestedClass*, but different value for *EnclosingClass* [ERROR]
6. A given Type can 'own' several different nested Types, so it is perfectly valid to have two or more rows with the same value for *EnclosingClass* but different values for *NestedClass*

End informative text

22.33 Param : 0x08

The *Param* table has the following columns:

- *Flags* (a 2-byte bitmask of type ParamAttributes, §23.1.13)
- *Sequence* (a 2-byte constant)
- *Name* (an index into the String heap)

Conceptually, every row in the *Param* table is owned by one, and only one, row in the *MethodDef* table

The rows in the *Param* table result from the parameters in a method declaration (§15.4), or from a `.param` attribute attached to a method (§15.4.1).

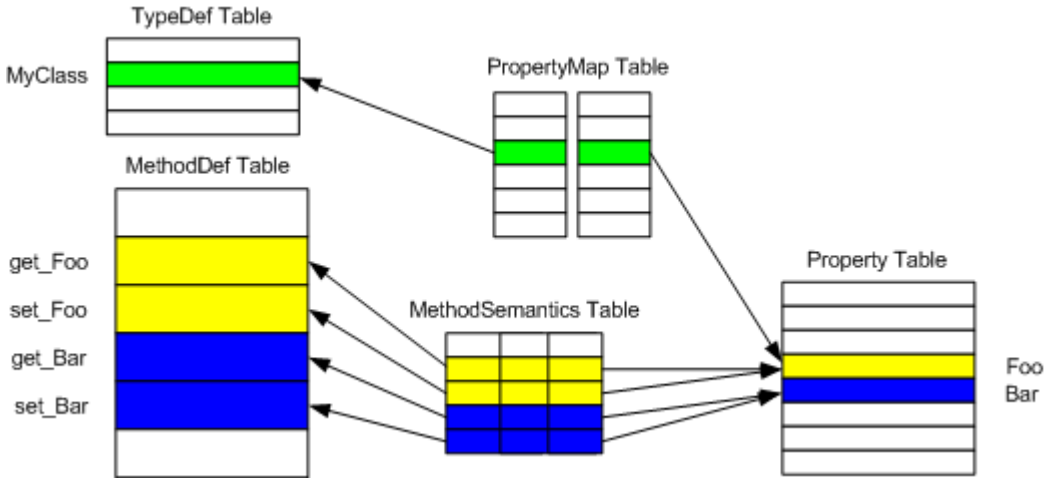
This contains informative text only

1. *Param* table can contain zero or more rows
2. Each row shall have one, and only one, owner row in the *MethodDef* table [ERROR]
3. *Flags* shall have only those values set that are specified (all combinations valid) [ERROR]
4. *Sequence* shall have a value >= 0 and <= number of parameters in owner method. A *Sequence* value of 0 refers to the owner method's return type; its parameters are then numbered from 1 onwards [ERROR]
5. Successive rows of the *Param* table that are owned by the same method shall be ordered by increasing *Sequence* value - although gaps in the sequence are allowed [WARNING]
6. If *Flags.HasDefault* = 1 then this row shall own exactly one row in the *Constant* table [ERROR]
7. If *Flags.HasDefault* = 0, then there shall be no rows in the *Constant* table owned by this row [ERROR]
8. parameters cannot be given default values, so *Flags.HasDefault* shall be 0 [CLS]
9. if *Flags.FieldMarshal* = 1 then this row shall own exactly one row in the *FieldMarshal* table [ERROR]
10. *Name* can be null or non-null
11. If *Name* is non-null, then it shall index a non-empty string in the String heap [WARNING]

End informative text

22.34 Property : 0x17

Properties within metadata are best viewed as a means to gather together collections of methods defined on a class, give them a name, and not much else. The methods are typically *get_* and *set_* methods, already defined on the class, and inserted like any other methods into the *MethodDef* table. The association is held together by three separate tables, as shown below:



Row 3 of the *PropertyMap* table indexes row 2 of the *TypeDef* table on the left (*MyClass*), whilst indexing row 4 of the *Property* table on the right – the row for a property called *Foo*. This setup establishes that *MyClass* has a property called *Foo*. But what methods in the *MethodDef* table are gathered together as 'belonging' to property *Foo*? That association is contained in the *MethodSemantics* table – its row 2 indexes property *Foo* to the right, and row 2 in the *MethodDef* table to the left (a method called *get_Foo*). Also, row 3 of the *MethodSemantics* table indexes *Foo* to the right, and row 3 in the *MethodDef* table to the left (a method called *set_Foo*). As the shading suggests, *MyClass* has another property, called *Bar*, with two methods, *get_Bar* and *set_Bar*.

Property tables do a little more than group together existing rows from other tables. The *Property* table has columns for *Flags*, *Name* (eg *Foo* and *Bar* in the example here) and *Type*. In addition, the *MethodSemantics* table has a column to record whether the method it points at is a *set_*, a *get_* or *other*.

[*Note:* The CLS (see Partition I) refers to instance, virtual, and static properties. The signature of a property (from the *Type* column) can be used to distinguish a static property, since instance and virtual properties will have the "HASTHIS" bit set in the signature (§23.2.1) while a static property will not. The distinction between an instance and a virtual property depends on the signature of the getter and setter methods, which the CLS requires to be either both virtual or both instance. *end note*]

The *Property* (0x17) table has the following columns:

- *Flags* (a 2-byte bitmask of type PropertyAttributes, §23.1.14)
- *Name* (an index into the String heap)
- *Type* (an index into the 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)

This contains informative text only

1. *Property* table can contain zero or more rows
2. Each row shall have one, and only one, owner row in the *PropertyMap* table (as described above) [ERROR]
3. *PropFlags* shall have only those values set that are specified (all combinations valid) [ERROR]
4. *Name* shall index a non-empty string in the String heap [ERROR]
5. The *Name* string shall be a valid CLS identifier [CLS]
6. *Type* shall index a non-null signature in the Blob heap [ERROR]
7. The signature indexed by *Type* shall be a valid signature for a property (ie, low nibble of leading byte is 0x8). Apart from this leading byte, the signature is the same as the property's *get_* method [ERROR]
8. Within the rows owned by a given row in the *TypeDef* table, there shall be no duplicates based upon *Name*+*Type* [ERROR]
9. There shall be no duplicate rows based upon *Name*, where *Name* fields are compared using CLS conflicting-identifier-rules (in particular, properties cannot be overloaded by their *Type* – a class cannot have two properties, `"int Foo"` and `"String Foo"`, for example) [CLS]

End informative text

22.35 PropertyMap : 0x15

The *PropertyMap* table has the following columns:

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

The *PropertyMap* and *Property* tables result from putting the `.property` directive on a class (§17).

This contains informative text only

1. *PropertyMap* table can contain zero or more rows

2. There shall be no duplicate rows, based upon *Parent* (a given class has only one 'pointer' to the start of its property list) [ERROR]

3. There shall be no duplicate rows, based upon *PropertyList* (different classes cannot share rows in the *Property* table) [ERROR]

End informative text

22.36 StandAloneSig : 0x11

Signatures are stored in the metadata Blob heap. In most cases, they are indexed by a column in some table—*Field.Signature*, *Method.Signature*, *MemberRef.Signature*, etc. However, there are two cases that require a metadata token for a signature that is not indexed by any metadata table. The *StandAloneSig* table fulfils this need. It has just one column, which points to a Signature in the Blob heap.

The signature shall describe either:

- a method – code generators create a row in the *StandAloneSig* table for each occurrence of a calli CIL instruction. That row indexes the call-site signature for the function pointer operand of the calli instruction
- local variables – code generators create one row in the *StandAloneSig* table for each method, to describe all of its local variables. The `.locals` directive (§15.4.1) in ILAsm generates a row in the *StandAloneSig* table.

The*StandAloneSig* table has the following column:

- *Signature* (an index into the Blob heap)

[*Example:*

```
// On encountering the calli instruction, ilasm generates a signature
// in the blob heap (DEFAULT, ParamCount = 1, RetType = int32, Param1 = int32),
// indexed by the StandAloneSig table:
.assembly Test {}
.method static int32 AddTen(int32)
{ ldarg.0
  ldc.i4  10
  add
  ret
}

.class Test
{ .method static void main()
  { .entrypoint
    ldc.i4.1
    ldftn int32 AddTen(int32)
    calli int32(int32)
    pop
    ret
  }
}
```

end example]

This contains informative text only

1. The *StandAloneSig* table can contain zero or more rows
2. *Signature* shall index a valid signature in the Blob heap [ERROR]
3. The signature 'blob' indexed by *Signature* shall be a valid METHOD or LOCALS signature [ERROR]
4. Duplicate rows are allowed

End informative text

22.37 TypeDef : 0x02

The *TypeDef* table has the following columns:

- *Flags* (a 4-byte bitmask of type *TypeAttributes*, §23.1.15)
- *TypeName* (an index into the String heap)
- *TypeNamespace* (an index into the String heap)
- *Extends* (an index into the *TypeDef*, *TypeRef*, or *TypeSpec* table; more precisely, a *TypeDefOrRef* (§24.2.6) coded index)
- *FieldList* (an index into the *Field* table; it marks the first of a contiguous run of Fields owned by this Type). The run continues to the smaller of:
  - the last row of the *Field* table
  - the next run of Fields, found by inspecting the *FieldList* of the next row in this *TypeDef* table
- *MethodList* (an index into the *MethodDef* table; it marks the first of a continguous run of Methods owned by this Type). The run continues to the smaller of:
  - the last row of the *MethodDef* table
  - the next run of Methods, found by inspecting the *MethodList* of the next row in this *TypeDef* table

The first row of the *TypeDef* table represents the pseudo class that acts as *parent* for functions and variables defined at module scope.

Note that any *type* shall be one, and only one, of

- Class (*Flags.Interface* = 0, and derives ultimately from `System.Object`)
- Interface (*Flags.Interface* = 1)
- Value type, derived ultimately from `System.ValueType`

For any given type, there are two separate and distinct chains of pointers to other types (the pointers are actually implemented as indexes into metadata tables). The two chains are:

- Extension chain – defined via the *Extends* column of the *TypeDef* table. Typically, a *derived Class extends* a *base Class* (always one, and only one, base *Class*)
- Interface chains – defined via the *InterfaceImpl* table. Typically, a *Class* implements zero, one or more *Interface*s

These two chains (extension and interface) are always kept separate in metadata. The *Extends* chain represents one-to-one relations—that is, one *Class extends* (or 'derives from') exactly one other *Class* (called its immediate base class). The *Interface* chains can represent one-to-many relations—that is, one *Class* might well implement two or more *Interface*s.

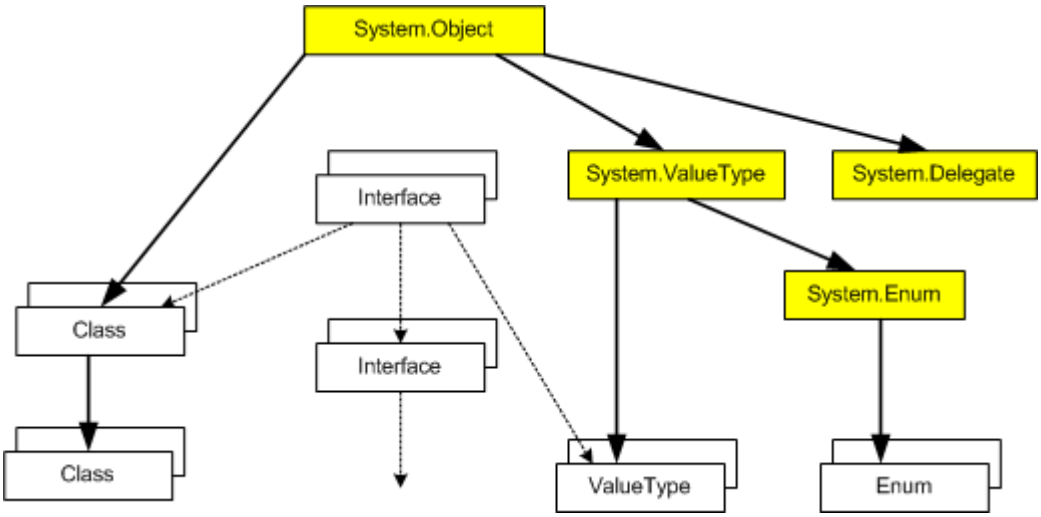
An interface can also implement one or more other interfaces—metadata stores those links via the *InterfaceImpl* table (the nomenclature is a little inappropriate here—there is no "implementation" involved; perhaps a clearer name might have been *Interface* table, or *InterfaceInherit* table)

Another slightly specialized type is a *nested* type which is declared in ILAsm as lexically nested within an enclosing type declaration. Whether a type is nested can be determined by the value of its *Flags.Visibility* sub-field – it shall be one of the set {*NestedPublic*, *NestedPrivate*, *NestedFamily*, *NestedAssembly*, *NestedFamANDAssem*, *NestedFamORAssem*}.

If a type is generic, its parameters are defined in the *GenericParam* table (§22.20). Entries in the *GenericParam* table reference entries in the *TypeDef* table; there is no reference from the *TypeDef* table to the *GenericParam* table.

This contains informative text only

The roots of the inheritance hierarchies look like this:



There is one system-defined root, `System.Object`. All Classes and ValueTypes shall derive, ultimately, from `System.Object`; Classes can derive from other Classes (through a single, non-looping chain) to any depth required. This *Extends* inheritance chain is shown with heavy arrows.

(See below for details of the `System.Delegate` Class)

Interfaces do not inherit from one another; however, they can have zero or more required interfaces, which shall be implemented. The *Interface* requirement chain is shown as light, dashed arrows. This includes links between Interfaces and Classes/ValueTypes – where the latter are said to *implement* that interface or interfaces.

Regular ValueTypes (i.e., excluding Enums – see later) are defined as deriving directly from System.ValueType. Regular ValueTypes cannot be derived to a depth of more than one. (Another way to state this is that user-defined ValueTypes shall be *sealed*.) User-defined Enums shall derive directly from System.Enum. Enums cannot be derived to a depth of more than one below System.Enum. (Another way to state this is that user-defined Enums shall be *sealed*.) System.Enum derives directly from System.ValueType.

User-defined delegates derive from `System.Delegate`. Delegates cannot be derived to a depth of more than one.

For the directives to declare types see §9.

1. A *TypeDef* table can contain one or more rows.
2. Flags:
   a. *Flags* shall have only those values set that are specified [ERROR]
   b. can set 0 or 1 of `SequentialLayout` and `ExplicitLayout` (if none set, then defaults to `AutoLayout`) [ERROR]
   c. can set 0 or 1 of `UnicodeClass` and `AutoClass` (if none set, then defaults to `AnsiClass`) [ERROR]
   d. If *Flags.HasSecurity* = 1, then at least one of the following conditions shall be true: [ERROR]
      - this Type owns at least one row in the *DeclSecurity* table

- this Type has a custom attribute called `SuppressUnmanagedCodeSecurityAttribute`

e. If this Type owns one (or more) rows in the *DeclSecurity* table then *Flags.HasSecurity* shall be 1 [ERROR]

f. If this Type has a custom attribute called `SuppressUnmanagedCodeSecurityAttribute` then *Flags.HasSecurity* shall be 1 [ERROR]

g. Note that it is valid for an Interface to have `HasSecurity` set. However, the security system ignores any permission requests attached to that Interface

3. *Name* shall index a non-empty string in the String heap [ERROR]
4. The *TypeName* string shall be a valid CLS identifier [CLS]
5. *TypeNamespace* can be null or non-null
6. If non-null, then *TypeNamespace* shall index a non-empty string in the String heap [ERROR]
7. If non-null, *TypeNamespace*'s string shall be a valid CLS Identifier [CLS]
8. Every Class (with the exception of `System.Object` and the special class `<Module>`) shall extend one, and only one, other Class - so *Extends* for a Class shall be non-null [ERROR]
9. `System.Object` shall have an *Extends* value of null [ERROR]
10. `System.ValueType` shall have an *Extends* value of `System.Object` [ERROR]
11. With the exception of `System.Object` and the special class `<Module>`, for any Class, *Extends* shall index a valid row in the *TypeDef*, *TypeRef*, or *TypeSpec* table, where valid means 1 <= row <= rowcount. In addition, that row itself shall be a Class (not an Interface or ValueType) In addition, that base Class shall not be sealed (its *Flags.Sealed* shall be 0) [ERROR]
12. A Class cannot extend itself, or any of its children (i.e., its derived Classes), since this would introduce loops in the hierarchy tree [ERROR] (For generic types, see §9.1 and §9.2.)
13. An Interface never *extends* another Type - so *Extends* shall be null (Interfaces *do* implement other Interfaces, but recall that this relationship is captured via the *InterfaceImpl* table, rather than the *Extends* column) [ERROR]
14. *FieldList* can be null or non-null
15. A Class or Interface can 'own' zero or more fields
16. A ValueType shall have a non-zero size - either by defining at least one field, or by providing a non-zero *ClassSize* [ERROR]
17. If *FieldList* is non-null, it shall index a valid row in the *Field* table, where valid means 1 <= row <= rowcount+1 [ERROR]
18. *MethodList* can be null or non-null
19. A Type can 'own' zero or more methods
20. The runtime size of a ValueType shall not exceed 1 MByte (0x100000 bytes) [ERROR]
21. If *MethodList* is non-null, it shall index a valid row in the *MethodDef* table, where valid means 1 <= row <= rowcount+1 [ERROR]
22. A Class which `has` one or more abstract methods cannot be instantiated, and shall have *Flags.Abstract* = 1. Note that the methods *owned* by the class include all of those inherited from its base class and interfaces it implements, plus those defined via its *MethodList*. (The CLI shall analyze class definitions at runtime; if it finds a class to have one or more abstract methods, but has *Flags.Abstract* = 0, it will throw an exception) [ERROR]
23. An Interface shall have *Flags.Abstract* = 1 [ERROR]
24. It is valid for an abstract Type to have a constructor method (ie, a method named `.ctor`)

25. Any non-abstract Type (ie *Flags.Abstract* = 0) shall provide an implementation (body) for every method its contract requires. Its methods can be inherited from its base class, from the interfaces it implements, or defined by itself. The implementations can be inherited from its base class, or defined by itself [ERROR]

26. An Interface (*Flags.Interface* = 1) can own static fields (*Field.Static* = 1) but cannot own instance fields (*Field.Static* = 0) [ERROR]

27. An Interface cannot be sealed (if *Flags.Interface = 1,* then *Flags.Sealed* shall be 0) [ERROR]

28. All of the methods owned by an Interface (*Flags.Interface* = 1) shall be abstract (*Flags.Abstract* = 1) [ERROR]

29. There shall be no duplicate rows in the *TypeDef* table, based on *TypeNamespace+TypeName* (unless this is a nested type - see below) [ERROR]

30. If this is a nested type, there shall be no duplicate row in the *TypeDef* table, based upon *TypeNamespace+TypeName+OwnerRowInNestedClassTable* [ERROR]

31. There shall be no duplicate rows, where *TypeNamespace+TypeName* fields are compared using CLS conflicting-identifier-rules (unless this is a nested type - see below) [CLS]

32. If this is a nested type, there shall be no duplicate rows, based upon *TypeNamespace+TypeName+OwnerRowInNestedClassTable* and where *TypeNamespace+TypeName* fields are compared using CLS conflicting-identifier-rules [CLS]

33. If *Extends* = `System.Enum` (i.e., type is a user-defined Enum) then:

    a. shall be sealed (`Sealed` = 1) [ERROR]

    b. shall not have any methods of its own (*MethodList* chain shall be zero length) [ERROR]

    c. shall not implement any interfaces (no entries in *InterfaceImpl* table for this type) [ERROR]

    d. shall not have any properties [ERROR]

    e. shall not have any events [ERROR]

    f. any static fields shall be literal (have *Flags.Literal* = 1) [ERROR]

    g. shall have one or more static, literal fields, each of which has the type of the Enum [CLS]

    h. shall be exactly one instance field, of built-in integer type [ERROR]

    i. the *Name* string of the instance field shall be "value__", the field shall be marked `RTSpecialName`, and that field shall have one of the CLS integer types [CLS]

    j. shall not have any static fields unless they are literal [ERROR]

34. A Nested type (defined above) shall own exactly one row in the *NestedClass* table, where 'owns' means a row in that *NestedClass* table whose *NestedClass* column holds the TypeDef token for this type definition [ERROR]

35. A ValueType shall be sealed [ERROR]

End informative text

22.38 TypeRef : 0x01

The *TypeRef* table has the following columns:

- *ResolutionScope* (an index into a *Module*, *ModuleRef*, *AssemblyRef* or *TypeRef* table, or null; more precisely, a *ResolutionScope* (§24.2.6) coded index)
- *TypeName* (an index into the String heap)
- *TypeNamespace* (an index into the String heap)

This contains informative text only

1. *ResolutionScope* shall be exactly one of:
   a. null - in this case, there shall be a row in the *ExportedType* table for this Type - its *Implementation* field shall contain a *File* token or an *AssemblyRef* token that says where the type is defined [ERROR]
   b. a *TypeRef* token, if this is a nested type (which can be determined by, for example, inspecting the *Flags* column in its *TypeDef* table - the accessibility subfield is one of the `tdNestedXXX` set) [ERROR]
   c. a *ModuleRef* token, if the target type is defined in another module within the same Assembly as this one [ERROR]
   d. a *Module* token, if the target type is defined in the current module - this should not occur in a CLI ("compressed metadata") module [WARNING]
   e. an *AssemblyRef* token, if the target type is defined in a different Assembly from the current module [ERROR]
2. *TypeName* shall index a non-empty string in the String heap [ERROR]
3. *TypeNamespace* can be null, or non-null
4. If non-null, *TypeNamespace* shall index a non-empty string in the String heap [ERROR]
5. The *TypeName* string shall be a valid CLS identifier [CLS]
6. There shall be no duplicate rows, where a duplicate has the same *ResolutionScope*, *TypeName* and *TypeNamespace* [ERROR]
7. There shall be no duplicate rows, where *TypeName* and *TypeNamespace* fields are compared using CLS conflicting-identifier-rules [CLS]

End informative text

22.39 TypeSpec : 0x1B

The *TypeSpec* table has just one column, which indexes the specification of a Type, stored in the Blob heap. This provides a metadata token for that Type (rather than simply an index into the Blob heap). This is required, typically, for array operations, such as creating, or calling methods on the array class.

The *TypeSpec* table has the following column:

- *Signature* (index into the Blob heap, where the blob is formatted as specified in §23.2.14)

Note that TypeSpec tokens can be used with any of the CIL instructions that take a *TypeDef* or *TypeRef* token; specifically, castclass, cpobj, initobj, isinst, ldelema, ldobj, mkrefany, newarr, refanyval, sizeof, stobj, box, and unbox.

This contains informative text only

1. The *TypeSpec* table can contain zero or more rows
2. *Signature* shall index a valid Type specification in the Blob heap [ERROR]
3. There shall be no duplicate rows, based upon *Signature* [ERROR]

End informative text

23 Metadata logical format: other structures

23.1 Bitmasks and flags

This subclause explains the flags and bitmasks used in the metadata tables. When a conforming implementation encounters a metadata structure (such as a flag) that is not specified in this standard, the behavior of the implementation is unspecified.

23.1.1 Values for AssemblyHashAlgorithm

Algorithm	Value
None	0x0000
Reserved (MD5)	0x8003
SHA1	0x8004

23.1.2 Values for AssemblyFlags

Flag	Value	Description
PublicKey	0x0001	The assembly reference holds the full (unhashed) public key.
SideBySideCompatible	0x0000	The assembly is side-by-side compatible
<reserved>	0x0030	Reserved: both bits shall be zero
Retargetable	0x0100	The implementation of this assembly used at runtime is not expected to match the version seen at compile time. (See the text following this table.)
EnableJITcompileTracking	0x8000	Reserved (a conforming implementation of the CLI can ignore this setting on read; some implementations might use this bit to indicate that a CIL-to-native-code compiler should generate CIL-to-native code map)
DisableJITcompileOptimizer	0x4000	Reserved (a conforming implementation of the CLI can ignore this setting on read; some implementations might use this bit to indicate that a CIL-to-native-code compiler should not generate optimized code)

In portable programs, the Retargetable (0x100) bit shall be set on all references to assemblies specified in this Standard.

23.1.3 Values for Culture

ar-SA	ar-IQ	ar-EG	ar-LY
ar-DZ	ar-MA	ar-TN	ar-OM
ar-YE	ar-SY	ar-JO	ar-LB
ar-KW	ar-AE	ar-BH	ar-QA
bg-BG	ca-ES	zh-TW	zh-CN
zh-HK	zh-SG	zh-MO	cs-CZ
da-DK	de-DE	de-CH	de-AT
de-LU	de-LI	el-GR	en-US
en-GB	en-AU	en-CA	en-NZ

en-IE	en-ZA	en-JM	en-CB
en-BZ	en-TT	en-ZW	en-PH
es-ES-Ts	es-MX	es-ES-Is	es-GT
es-CR	es-PA	es-DO	es-VE
es-CO	es-PE	es-AR	es-EC
es-CL	es-UY	es-PY	es-BO
es-SV	es-HN	es-NI	es-PR
fi-FI	fr-FR	fr-BE	fr-CA
fr-CH	fr-LU	fr-MC	he-IL
hu-HU	is-IS	it-IT	it-CH
ja-JP	ko-KR	nl-NL	nl-BE
nb-NO	nn-NO	pl-PL	pt-BR
pt-PT	ro-RO	ru-RU	hr-HR
lt-sr-SP	cy-sr-SP	sk-SK	sq-AL
sv-SE	sv-FI	th-TH	tr-TR
ur-PK	id-ID	uk-UA	be-BY
sl-SI	et-EE	lv-LV	lt-LT
fa-IR	vi-VN	hy-AM	lt-az-AZ
cy-az-AZ	eu-ES	mk-MK	af-ZA
ka-GE	fo-FO	hi-IN	ms-MY
ms-BN	kk-KZ	ky-KZ	sw-KE
lt-uz-UZ	cy-uz-UZ	tt-TA	pa-IN
gu-IN	ta-IN	te-IN	kn-IN
mr-IN	sa-IN	mn-MN	gl-ES
kok-IN	syr-SY	div-MV	

Note on RFC 1766, Locale names: a typical string would be “en-US”. The first part (“en” in the example) uses ISO 639 characters (“Latin-alphabet characters in lowercase. No diacritical marks of modified characters are used”). The second part (“US” in the example) uses ISO 3166 characters (similar to ISO 639, but uppercase); that is, the familiar ASCII characters a–z and A–Z, respectively. However, whilst RFC 1766 recommends the first part be lowercase and the second part be uppercase, it allows mixed case. Therefore, the validation rule checks only that *Culture* is one of the strings in the list above—but the check is totally case-blind—where case-blind is the familiar fold on values less than U+0080

23.1.4 Flags for events [EventAttributes]

Flag	Value	Description
SpecialName	0x0200	Event is special.
RTSpecialName	0x0400	CLI provides 'special' behavior, depending upon the name of the event

23.1.5 Flags for fields [FieldAttributes]

Flag	Value	Description
FieldAccessMask	0x0007	These 3 bits contain one of the following values:
CompilerControlled	0x0000	Member not referenceable

Private	0x0001	Accessible only by the parent type
FamANDAssem	0x0002	Accessible by sub-types only in this Assembly
Assembly	0x0003	Accessibly by anyone in the Assembly
Family	0x0004	Accessible only by type and sub-types
FamORAssem	0x0005	Accessibly by sub-types anywhere, plus anyone in assembly
Public	0x0006	Accessibly by anyone who has visibility to this scope field contract attributes
Static	0x0010	Defined on type, else per instance
InitOnly	0x0020	Field can only be initialized, not written to after init
Literal	0x0040	Value is compile time constant
NotSerialized	0x0080	Reserved (to indicate this field should not be serialized when type is remoted)
SpecialName	0x0200	Field is special
Interop Attributes		
PInvokeImpl	0x2000	Implementation is forwarded through PInvoke.
Additional flags		
RTSpecialName	0x0400	CLI provides 'special' behavior, depending upon the name of the field
HasFieldMarshal	0x1000	Field has marshalling information
HasDefault	0x8000	Field has default
HasFieldRVA	0x0100	Field has RVA

23.1.6 Flags for files [FileAttributes]

Flag	Value	Description
ContainsMetaData	0x0000	This is not a resource file
ContainsNoMetaData	0x0001	This is a resource file or other non-metadata-containing file

23.1.7 Flags for Generic Parameters [GenericParamAttributes]

Flag	Value	Description
VarianceMask	0x0003	These 2 bits contain one of the following values:
None	0x0000	The generic parameter is non-variant and has no special constraints
Covariant	0x0001	The generic parameter is covariant
Contravariant	0x0002	The generic parameter is contravariant
SpecialConstraintMask	0x001C	These 3 bits contain one of the following values:
ReferenceTypeConstraint	0x0004	The generic parameter has the `class` special constraint
NotNullableValueTypeConstraint	0x0008	The generic parameter has the `valuetype` special constraint
DefaultConstructorConstraint	0x0010	The generic parameter has the `.ctor` special constraint

23.1.8 Flags for ImplMap [PInvokeAttributes]

Flag	Value	Description
NoMangle	0x0001	PInvoke is to use the member name as specified
Character set		
CharSetMask	0x0006	This is a resource file or other non-metadata-containing file. These 2 bits contain one of the following values:
CharSetNotSpec	0x0000	
CharSetAnsi	0x0002	
CharSetUnicode	0x0004	
CharSetAuto	0x0006	
SupportsLastError	0x0040	Information about target function. Not relevant for fields
Calling convention		
CallConvMask	0x0700	These 3 bits contain one of the following values:
CallConvWinapi	0x0100	
CallConvCdecl	0x0200	
CallConvStdcall	0x0300	
CallConvThiscall	0x0400	
CallConvFastcall	0x0500	

23.1.9 Flags for ManifestResource [ManifestResourceAttributes]

Flag	Value	Description
VisibilityMask	0x0007	These 3 bits contain one of the following values:
Public	0x0001	The Resource is exported from the Assembly
Private	0x0002	The Resource is private to the Assembly

23.1.10 Flags for methods [MethodAttributes]

Flag	Value	Description
MemberAccessMask	0x0007	These 3 bits contain one of the following values:
CompilerControlled	0x0000	Member not referenceable
Private	0x0001	Accessible only by the parent type
FamANDAssem	0x0002	Accessible by sub-types only in this Assembly
Assem	0x0003	Accessibly by anyone in the Assembly
Family	0x0004	Accessible only by type and sub-types
FamORAssem	0x0005	Accessibly by sub-types anywhere, plus anyone in assembly
Public	0x0006	Accessibly by anyone who has visibility to this scope
Static	0x0010	Defined on type, else per instance
Final	0x0020	Method cannot be overridden

Virtual	0x0040	Method is virtual
HideBySig	0x0080	Method hides by name+sig, else just by name
VtableLayoutMask	0x0100	Use this mask to retrieve vtable attributes. This bit contains one of the following values:
ReuseSlot	0x0000	Method reuses existing slot in vtable
NewSlot	0x0100	Method always gets a new slot in the vtable
Strict	0x0200	Method can only be overriden if also accessible
Abstract	0x0400	Method does not provide an implementation
SpecialName	0x0800	Method is special
Interop attributes		
PInvokeImpl	0x2000	Implementation is forwarded through PInvoke
UnmanagedExport	0x0008	Reserved: shall be zero for conforming implementations
Additional flags		
RTSpecialName	0x1000	CLI provides 'special' behavior, depending upon the name of the method
HasSecurity	0x4000	Method has security associate with it
RequireSecObject	0x8000	Method calls another method containing security code.

23.1.11 Flags for methods [MethodImplAttributes]

Flag	Value	Description
CodeTypeMask	0x0003	These 2 bits contain one of the following values:
IL	0x0000	Method impl is CIL
Native	0x0001	Method impl is native
OPTIL	0x0002	Reserved: shall be zero in conforming implementations
Runtime	0x0003	Method impl is provided by the runtime
ManagedMask	0x0004	Flags specifying whether the code is managed or unmanaged. This bit contains one of the following values:
Unmanaged	0x0004	Method impl is unmanaged, otherwise managed
Managed	0x0000	Method impl is managed
Implementation info and interop		
ForwardRef	0x0010	Indicates method is defined; used primarily in merge scenarios
PreserveSig	0x0080	Reserved: conforming implementations can ignore
InternalCall	0x1000	Reserved: shall be zero in conforming implementations
Synchronized	0x0020	Method is single threaded through the body
NoInlining	0x0008	Method cannot be inlined
MaxMethodImplVal	0xffff	Range check value
NoOptimization	0x0040	Method will not be optimized when generating native code

23.1.12 Flags for MethodSemantics [MethodSemanticsAttributes]

Flag	Value	Description
Setter	0x0001	Setter for property
Getter	0x0002	Getter for property
Other	0x0004	Other method for property or event
AddOn	0x0008	AddOn method for event
RemoveOn	0x0010	RemoveOn method for event
Fire	0x0020	Fire method for event

23.1.13 Flags for params [ParamAttributes]

Flag	Value	Description
In	0x0001	Param is [In]
Out	0x0002	Param is [out]
Optional	0x0010	Param is optional
HasDefault	0x1000	Param has default value
HasFieldMarshal	0x2000	Param has FieldMarshal
Unused	0xcfe0	Reserved: shall be zero in a conforming implementation

23.1.14 Flags for properties [PropertyAttributes]

Flag	Value	Description
SpecialName	0x0200	Property is special
RTSpecialName	0x0400	Runtime(metadata internal APIs) should check name encoding
HasDefault	0x1000	Property has default
Unused	0xe9ff	Reserved: shall be zero in a conforming implementation

23.1.15 Flags for types [TypeAttributes]

Flag	Value	Description
Visibility attributes		
VisibilityMask	0x00000007	Use this mask to retrieve visibility information. These 3 bits contain one of the following values:
NotPublic	0x00000000	Class has no public scope
Public	0x00000001	Class has public scope
NestedPublic	0x00000002	Class is nested with public visibility
NestedPrivate	0x00000003	Class is nested with private visibility
NestedFamily	0x00000004	Class is nested with family visibility
NestedAssembly	0x00000005	Class is nested with assembly visibility

NestedFamANDAssem	0x00000006	Class is nested with family and assembly visibility
NestedFamORAssem	0x00000007	Class is nested with family or assembly visibility
Class layout attributes		
LayoutMask	0x00000018	Use this mask to retrieve class layout information. These 2 bits contain one of the following values:
AutoLayout	0x00000000	Class fields are auto-laid out
SequentialLayout	0x00000008	Class fields are laid out sequentially
ExplicitLayout	0x00000010	Layout is supplied explicitly
Class semantics attributes		
ClassSemanticsMask	0x00000020	Use this mask to retrive class semantics information. This bit contains one of the following values:
Class	0x00000000	Type is a class
Interface	0x00000020	Type is an interface
Special semantics in addition to class semantics		
Abstract	0x00000080	Class is abstract
Sealed	0x00000100	Class cannot be extended
SpecialName	0x00000400	Class name is special
Implementation Attributes		
Import	0x00001000	Class/Interface is imported
Serializable	0x00002000	Reserved (Class is serializable)
String formatting Attributes		
StringFormatMask	0x00030000	Use this mask to retrieve string information for native interop. These 2 bits contain one of the following values:
AnsiClass	0x00000000	LPSTR is interpreted as ANSI
UnicodeClass	0x00010000	LPSTR is interpreted as Unicode
AutoClass	0x00020000	LPSTR is interpreted automatically
CustomFormatClass	0x00030000	A non-standard encoding specified by CustomStringFormatMask
CustomStringFormatMask	0x00C00000	Use this mask to retrieve non-standard encoding information for native interop. The meaning of the values of these 2 bits is unspecified.
Class Initialization Attributes		
BeforeFieldInit	0x00100000	Initialize the class before first static field access
Additional Flags		

RTSpecialName	0x00000800	CLI provides 'special' behavior, depending upon the name of the Type
HasSecurity	0x00040000	Type has security associate with it
IsTypeForwarder	0x00200000	This *ExportedType* entry is a type forwarder

23.1.16 Element types used in signatures

The following table lists the values for ELEMENT_TYPE constants. These are used extensively in metadata signature *blobs* – see §23.2

Name	Value	Remarks
ELEMENT_TYPE_END	0x00	Marks end of a list
ELEMENT_TYPE_VOID	0x01	
ELEMENT_TYPE_BOOLEAN	0x02	
ELEMENT_TYPE_CHAR	0x03	
ELEMENT_TYPE_I1	0x04	
ELEMENT_TYPE_U1	0x05	
ELEMENT_TYPE_I2	0x06	
ELEMENT_TYPE_U2	0x07	
ELEMENT_TYPE_I4	0x08	
ELEMENT_TYPE_U4	0x09	
ELEMENT_TYPE_I8	0x0a	
ELEMENT_TYPE_U8	0x0b	
ELEMENT_TYPE_R4	0x0c	
ELEMENT_TYPE_R8	0x0d	
ELEMENT_TYPE_STRING	0x0e	
ELEMENT_TYPE_PTR	0x0f	Followed by *type*
ELEMENT_TYPE_BYREF	0x10	Followed by *type*
ELEMENT_TYPE_VALUETYPE	0x11	Followed by TypeDef or TypeRef token
ELEMENT_TYPE_CLASS	0x12	Followed by TypeDef or TypeRef token
ELEMENT_TYPE_VAR	0x13	Generic parameter in a generic type definition, represented as *number* (compressed unsigned integer)
ELEMENT_TYPE_ARRAY	0x14	*type rank boundsCount bound1 … loCount lo1 …*
ELEMENT_TYPE_GENERICINST	0x15	Generic type instantiation. Followed by *type type-arg-count type-1 ... type-n*
ELEMENT_TYPE_TYPEDBYREF	0x16	
ELEMENT_TYPE_I	0x18	System.IntPtr
ELEMENT_TYPE_U	0x19	System.UIntPtr

ELEMENT_TYPE_FNPTR	0x1b	Followed by full method signature
ELEMENT_TYPE_OBJECT	0x1c	System.Object
ELEMENT_TYPE_SZARRAY	0x1d	Single-dim array with 0 lower bound
ELEMENT_TYPE_MVAR	0x1e	Generic parameter in a generic method definition, represented as *number* (compressed unsigned integer)
ELEMENT_TYPE_CMOD_REQD	0x1f	Required modifier : followed by a TypeDef or TypeRef token
ELEMENT_TYPE_CMOD_OPT	0x20	Optional modifier : followed by a TypeDef or TypeRef token
ELEMENT_TYPE_INTERNAL	0x21	Implemented within the CLI
ELEMENT_TYPE_MODIFIER	0x40	Or'd with following element types
ELEMENT_TYPE_SENTINEL	0x41	Sentinel for vararg method signature
ELEMENT_TYPE_PINNED	0x45	Denotes a local variable that points at a pinned object
	0x50	Indicates an argument of type System.Type.
	0x51	Used in custom attributes to specify a boxed object (§23.3).
	0x52	Reserved
	0x53	Used in custom attributes to indicate a FIELD (§22.10, 23.3).
	0x54	Used in custom attributes to indicate a PROPERTY (§22.10, 23.3).
	0x55	Used in custom attributes to specify an enum (§23.3).

23.2 Blobs and signatures

The word *signature* is conventionally used to describe the type info for a function or method; that is, the type of each of its parameters, and the type of its return value. Within metadata, the word *signature* is also used to 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 several kinds of Signature, as follows:

- MethodRefSig (differs from a MethodDefSig only for VARARG calls)
- MethodDefSig
- FieldSig
- PropertySig
- LocalVarSig
- TypeSpec
- MethodSpec

The value of the first byte of a Signature 'blob' indicates what kind of Signature it is. Its lowest 4 bits hold one of the following: C, DEFAULT, FASTCALL, STDCALL, THISCALL, or VARARG (whose values are defined in §23.2.3), which qualify method signatures; FIELD, which denotes a field signature (whose value is defined in §23.2.4); or PROPERTY, which denotes a property signature (whose value is defined in §23.2.5). This subclause defines the

binary 'blob' format for each kind of Signature. In the syntax diagrams that accompany many of the definitions, shading is used to combine into a single diagram what would otherwise be multiple diagrams; the accompanying text describes the use of shading.

Signatures are compressed before being stored into the Blob heap (described below) by compressing the integers embedded in the signature. The maximum encodable unsigned integer is 29 bits long, 0x1FFFFFFF. For signed integers, as occur in ArrayShape (§23.2.13), the range is -2^{28} (0xF0000000) to $2^{28}-1$ (0x0FFFFFFF). The compression algorithm used is as follows (bit 0 is the least significant bit):

- For unsigned integers:
    - If the value lies between 0 (0x00) and 127 (0x7F), inclusive, encode as a one-byte integer (bit 7 is clear, value held in bits 6 through 0)
    - If the value lies between 2^8 (0x80) and $2^{14} - 1$ (0x3FFF), inclusive, encode as a 2-byte integer with bit 15 set, bit 14 clear (value held in bits 13 through 0)
    - Otherwise, encode as a 4-byte integer, with bit 31 set, bit 30 set, bit 29 clear (value held in bits 28 through 0)
- For signed integers:
    - If the value lies between -64 (0xFFFFFFC0) and 63 (0x3F), inclusive, encode as a one-byte integer: bit 7 clear, value bits 5 through 0 held in bits 6 through 1, sign bit (value bit 31) in bit 0.
    - If the value lies between -8192 (0xFFFFE000) and 8191 (0x1FFF), inclusive, encode as a two-byte integer: 15 set, bit 14 clear, value bits 12 through 0 held in bits 13 through 1, sign bit (value bit 31) in bit 0.
    - If the value lies between -268435456 (0xF000000) and 268435455 (0x0FFFFFFF), inclusive, encode as a four-byte integer: 31 set, 30 set, bit 29 clear, value bits 27 through 0 held in bits 28 through 1, sign bit (value bit 31) in bit 0.

  [*Note:* When uncompressing the sign bit is used to fill all the missing bits. *end note*]
- A null string should be represented with the reserved single byte 0xFF, and no following data

[*Note:* The tables below show several examples. The first column gives a value, expressed in familiar (C-like) hex notation. The second column shows the corresponding, compressed result, as it would appear in a PE file, with successive bytes of the result lying at successively higher byte offsets within the file. (This is the opposite order from how regular binary integers are laid out in a PE file.)

Unsigned examples:

Original Value	**Compressed Representation**
0x03	03
0x7F	7F (7 bits set)
0x80	8080
0x2E57	AE57
0x3FFF	BFFF
0x4000	C000 4000
0x1FFF FFFF	DFFF FFFF

Signed examples:

Original Value	**Compressed Representation**
3	06

-3	7B
64	8080
-64	01
8192	C000 4000
-8192	8001
268435455	DFFF FFFE
-268435456	C000 0001

end note]

The most significant bits (the first ones encountered in a PE file) of a "compressed" field, can reveal whether it occupies 1, 2, or 4 bytes, as well as its value. For this to work, the "compressed" value, as explained above, is stored in big-endian order; i.e., with the most significant byte at the smallest offset within the file.

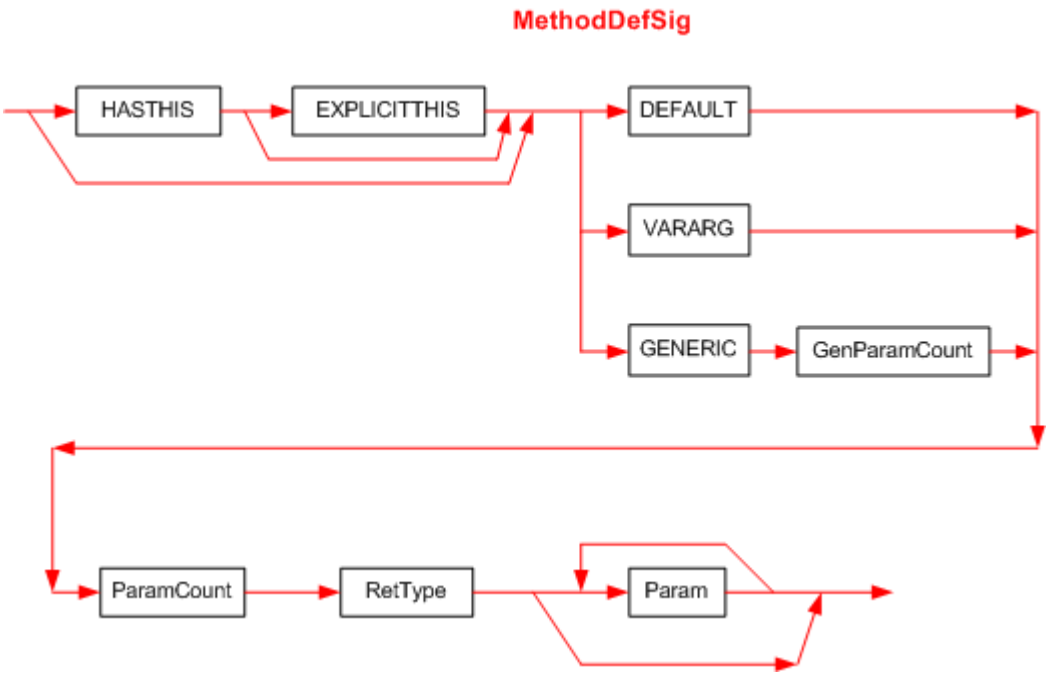
Signatures make extensive use of constant values called `ELEMENT_TYPE_xxx` – see §23.1.16. In particular, signatures include two modifiers called:

`ELEMENT_TYPE_BYREF` – this element is a managed pointer (see Partition I). This modifier can only occur in the definition of *LocalVarSig* (§23.2.6), *Param* (§23.2.10) or *RetType* (§23.2.11). It shall not occur within the definition of a Field (§23.2.4)

`ELEMENT_TYPE_PTR` – this element is an unmanaged pointer (see Partition I). This modifier can occur in the definition of *LocalVarSig* (§23.2.6), *Param* (§23.2.10), *RetType* (§23.2.11) or *Field* (§23.2.4)

23.2.1 MethodDefSig

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



This diagram uses the following abbreviations:

HASTHIS = 0x20, used to encode the keyword instance in the calling convention, see §15.3

EXPLICITTHIS = 0x40, used to encode the keyword explicit in the calling convention, see §15.3

DEFAULT = 0x0, used to encode the keyword default in the calling convention, see §15.3

VARARG = 0x5, used to encode the keyword vararg in the calling convention, see §15.3

GENERIC = 0x10, used to indicate that the method has one or more generic parameters.

The first byte of the Signature holds bits for HASTHIS, EXPLICITTHIS and calling convention (DEFAULT, VARARG, or GENERIC). These are ORed together.

GenParamCount is the number of generic parameters for the method. This is a compressed unsigned integer. [*Note*: For generic methods, both *MethodDef* and *MemberRef* shall include the GENERIC calling convention, together with *GenParamCount*; these are significant for binding—they enable the CLI to overload on generic methods by the number of generic parameters they include. *end note*]

ParamCount is an unsigned integer that holds the number of parameters (0 or more). It can be any number between 0 and 0x1FFFFFFF. The compiler compresses it too (see Partition II Metadata Validation) – before storing into the 'blob' (*ParamCount* counts just the method parameters – it does not include the method's return type)

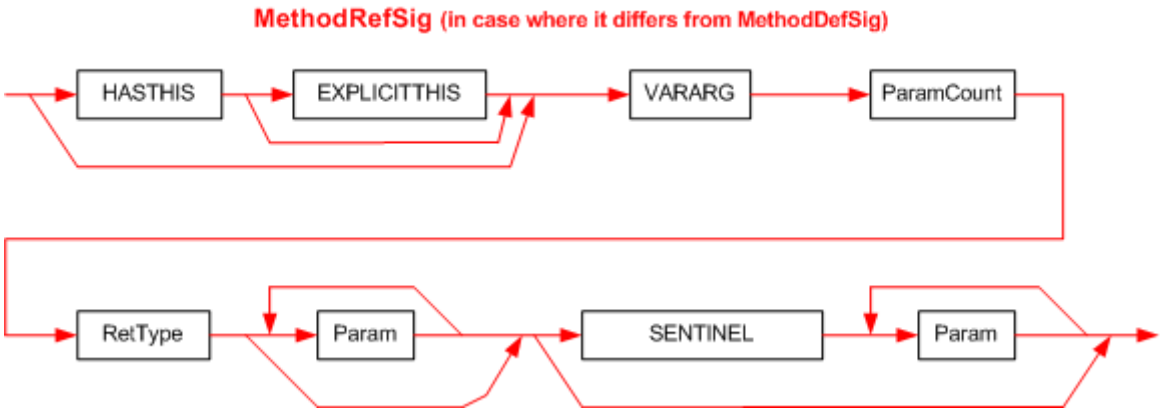
The *RetType* item describes the type of the method's return value (§23.2.11)

The *Param* item describes the type of each of the method's parameters. There shall be *ParamCount* instances of the *Param* item (§23.2.10).

23.2.2 MethodRefSig

A MethodRefSig is indexed by the MemberRef.Signature column. This provides the *call site* Signature for a method. Normally, this call site Signature shall match exactly the Signature specified in the definition of the target method. For example, if a method Foo is defined that takes two unsigned int32s and returns void; then any call site shall index a signature that takes exactly two unsigned int32s and returns void. In this case, the syntax diagram for a MethodRefSig is identical with that for a MethodDefSig – see §23.2.1

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



This diagram uses the following abbreviations:

HASTHIS = 0x20, used to encode the keyword instance in the calling convention, see §15.3

EXPLICITTHIS = 0x40, used to encode the keyword explicit in the calling convention, see §15.3

VARARG = 0x5, used to encode the keyword vararg in the calling convention, see 15.3

SENTINEL = 0x41 (§23.1.16), used to encode "..." in the parameter list, see §15.3

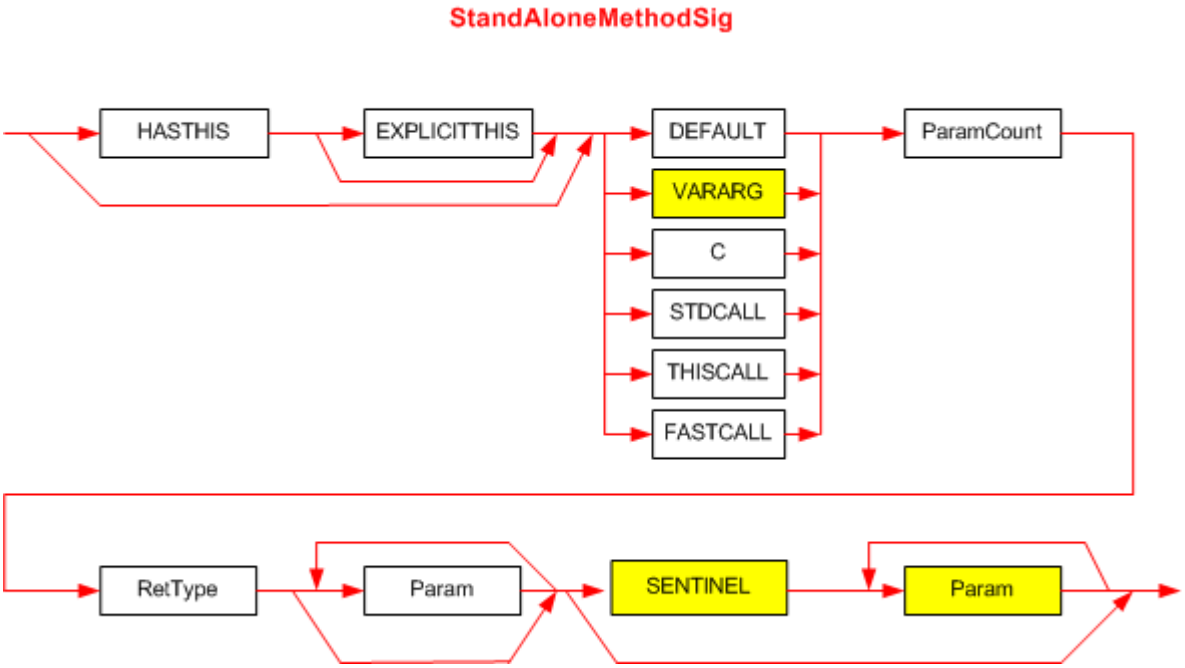
- The first byte of the Signature holds bits for HASTHIS, EXPLICITTHIS, and the calling convention VARARG. These are ORed together.
- *ParamCount* is an unsigned integer that holds the number of parameters (0 or more). It can be any number between 0 and 0x1FFFFFFF The compiler compresses it (see Partition II Metadata Validation) – 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 (§23.2.11)
- The *Param* item describes the type of each of the method's parameters. There shall be *ParamCount* instances of the *Param* item (§23.2.10).

The *Param* item describes the type of each of the method's parameters. There shall be *ParamCount* instances of the *Param* item.This starts just like the MethodDefSig for a VARARG method (§23.2.1). But then a SENTINEL token is appended, followed by extra *Param* items to describe the extra VARARG arguments. Note that the *ParamCount* item shall indicate the total number of *Param* items in the Signature – before and after the SENTINEL byte (0x41).

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

23.2.3 StandAloneMethodSig

A StandAloneMethodSig is indexed by the StandAloneSig.Signature column. It is typically created as preparation for executing a calli instruction. It is similar to a MethodRefSig, in that it represents a call site signature, but its calling convention can specify an unmanaged target (the calli instruction invokes either managed, or unmanaged code). Its syntax diagram is:



This diagram uses the following abbreviations (§15.3):

HASTHIS for 0x20

EXPLICITTHIS for 0x40

DEFAULT for 0x0

VARARG for 0x5

C for 0x1

STDCALL for 0x2

THISCALL for 0x3

FASTCALL for 0x4

SENTINEL for 0x41 (§23.1.16 and §15.3)

- The first byte of the Signature holds bits for HASTHIS, EXPLICITTHIS and calling convention – DEFAULT, VARARG, C, STDCALL, THISCALL, or FASTCALL. These are OR'd together.
- *ParamCount* is an unsigned integer that holds the number of non-vararg and vararg parameters, combined. It can be any number between 0 and 0x1FFFFFFF The compiler compresses it (see Partition II Metadata Validation) – 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 (§23.2.11)
- The first *Param* item describes the type of each of the method's non-vararg parameters. The (optional) second *Param* item describes the type of each of the method's vararg parameters. There shall be *ParamCount* instances of *Param* (§23.2.10).

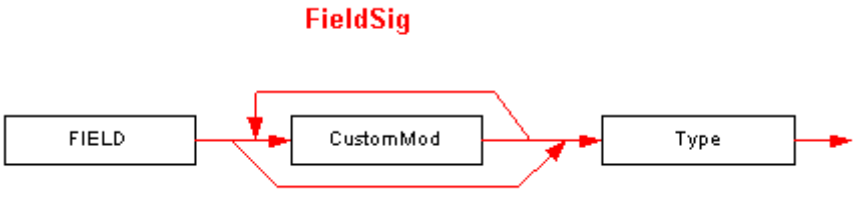
This is the most complex of the various method signatures. Two separate diagrams have been combined into one in this diagram, using shading to distinguish between them. Thus, for the following calling conventions: DEFAULT (managed), STDCALL, THISCALL and FASTCALL (unmanaged), the signature ends just before the SENTINEL item (these are all non vararg signatures). However, for the managed and unmanaged vararg calling conventions:

VARARG (managed) and C (unmanaged), the signature can include the SENTINEL and final Param items (they are not required, however). These options are indicated by the shading of boxes in the syntax diagram.

In the unusual case that a call site supplies no extra arguments, the signature shall not include a SENTINEL (this is the route shown by the lower arrow that bypasses SENTINEL and goes to the end of the StandAloneMethodSig definition).

23.2.4 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 can be a static or instance field in a class, or it can be a global variable. The syntax diagram for a FieldSig looks like this:



This diagram uses the following abbreviations:

FIELD for 0x6

CustomMod is defined in §23.2.7. *Type* is defined in §23.2.12

23.2.5 PropertySig

A PropertySig is indexed by the Property.Type column. It captures the type information for a Property – essentially, the signature of its *getter* method:

the number of parameters supplied to its *getter* method

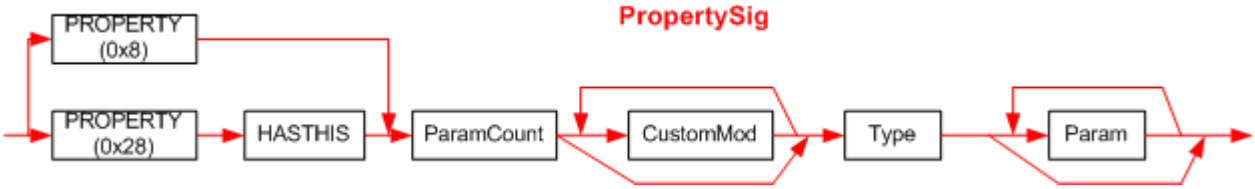
the base type of the Property (the type returned by its *getter* method)

type information for each parameter in the g*etter* method (that is, the index parameters)

Note that the signatures of getter and setter are related precisely as follows:

- The types of a *getter's paramCount* parameters are exactly the same as the first *paramCount* parameters of the *setter*
- The return type of a *getter* is exactly the same as the type of the last parameter supplied to the *setter*

The syntax diagram for a PropertySig looks like this:



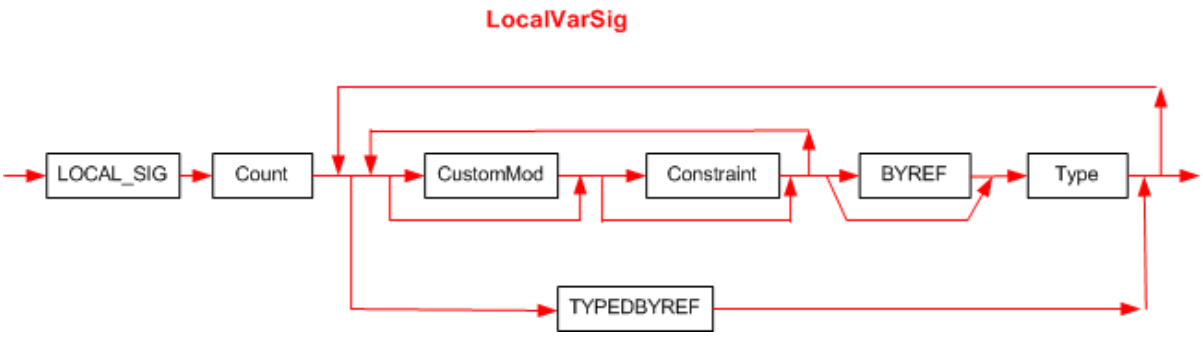
The first byte of the Signature holds bits for `HASTHIS` and `PROPERTY`. These are OR'd together.

Type specifies the type returned by the *Getter* method for this property. *Type* is defined in §23.2.12. *Param* is defined in §23.2.10.

ParamCount is a compressed unsigned integer that holds the number of index parameters in the *getter* methods (0 or more). (§23.2.1) (*ParamCount* counts just the method parameters – it does not include the method's base type of the Property)

23.2.6 LocalVarSig

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



This diagram uses the following abbreviations:

`LOCAL_SIG` for 0x7, used for the `.locals` directive, see§15.4.1.3

`BYREF` for `ELEMENT_TYPE_BYREF` (§23.1.16)

Constraint is defined in §23.2.9.

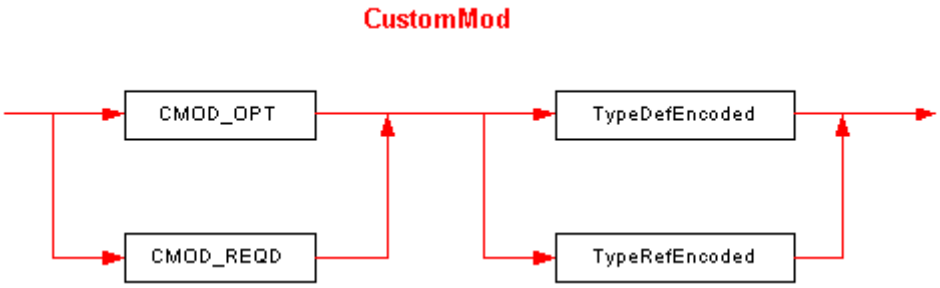
Type is defined in §23.2.12

Count is a compressed unsigned integer that holds the number of local variables. It can be any number between 1 and 0xFFFE.

There shall be *Count* instances of the *Type* in the LocalVarSig

23.2.7 CustomMod

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



This diagram uses the following abbreviations:

CMOD_OPT for ELEMENT_TYPE_CMOD_OPT (§23.1.16)

CMOD_REQD for ELEMENT_TYPE_CMOD_REQD (§23.1.16)

The CMOD_OPT or CMOD_REQD value is compressed, see §23.2.

The CMOD_OPT or CMOD_REQD is followed by a metadata token that indexes a row in the *TypeDef* table or the *TypeRef* table. However, these tokens are encoded and compressed – see §23.2.8 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 shall 'understand' the semantic implied by this CustomModifier in order to reference the surrounding Signature.

23.2.8 TypeDefOrRefEncoded

These items are compact ways to store a *TypeDef*, *TypeRef*, or *TypeSpec* token in a Signature (§23.2.12).

Consider a regular *TypeRef* token, such as 0x01000012. The top byte of 0x01 indicates that this is a *TypeRef* token (see Partition VI for a list of the supported metadata token types). 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:

1. encode the table that this token indexes as the least significant 2 bits. The bit values to use are 0, 1 and 2, specifying the target table is the *TypeDef*, *TypeRef* or *TypeSpec* table, respectively
2. shift the 3-byte row index (0x000012 in this example) left by 2 bits and OR into the 2-bit encoding from step 1
3. compress the resulting value (§23.2). This example yields the following encoded value:

```
a)  encoded = value for TypeRef table = 0x01 (from 1. above)

b)  encoded = ( 0x000012 << 2 ) |  0x01

            = 0x48 | 0x01

            = 0x49

c)  encoded = Compress (0x49)

            = 0x49
```

So, instead of the original, regular *TypeRef* token value of 0x01000012, requiring 4 bytes of space in the Signature 'blob', this TypeRef token is encoded as a single byte.

23.2.9 Constraint

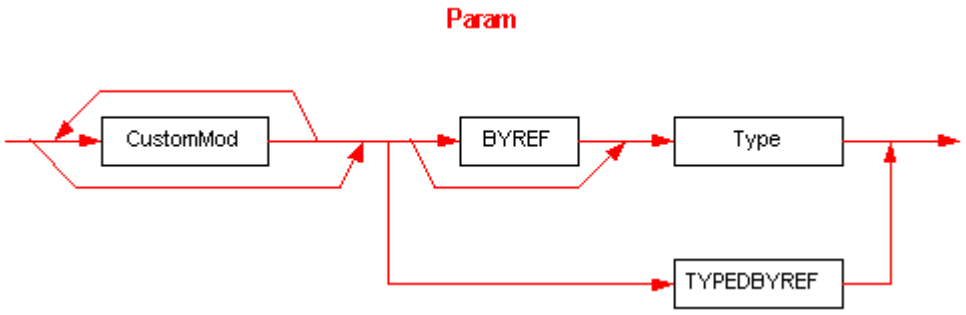
The *Constraint* item in Signatures currently has only one possible value, ELEMENT_TYPE_PINNED (§23.1.16), which specifies that the target type is pinned in the runtime heap, and will not be moved by the actions of garbage collection.

A *Constraint* can only be applied within a LocalVarSig (not a FieldSig). The Type of the local variable shall either be a reference type (in other words, it *points* to the actual variable – for example, an Object, or a String); or it shall 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.

23.2.10 Param

The *Param* (parameter) item in *Signatures* has this syntax diagram:



This diagram uses the following abbreviations:

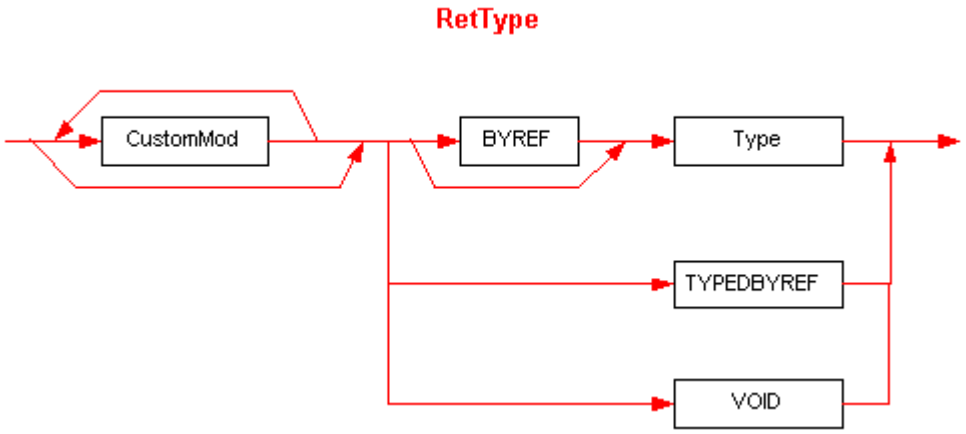
`BYREF` for 0x10 (§23.1.16)

`TYPEDBYREF` for 0x16 (§23.1.16)

CustomMod is defined in §23.2.7. *Type* is defined in §23.2.12

23.2.11 RetType

The *RetType* (return type) item in Signatures has this syntax diagram:



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

`BYREF` for `ELEMENT_TYPE_BYREF` (§23.1.16)

`TYPEDBYREF` for `ELEMENT_TYPE_TYPEDBYREF` (§23.1.16)

`VOID` for `ELEMENT_TYPE_VOID` (§23.1.16)

23.2.12 Type

Type is encoded in signatures as follows (`I1` is an abbreviation for `ELEMENT_TYPE_I1`, `U1` is an abbreviation for `ELEMENT_TYPE_U1`, and so on; see 23.1.16):

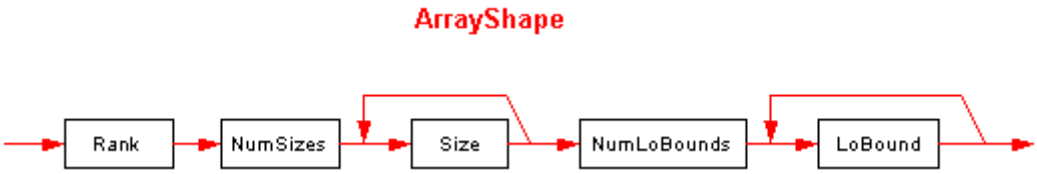
```
Type ::=
BOOLEAN | CHAR | I1 | U1 | I2 | U2 | I4 | U4 | I8 | U8 | R4 | R8 | I | U
| ARRAY Type ArrayShape (general array, see §23.2.13)
```

```
| CLASS TypeDefOrRefEncoded
| FNPTR MethodDefSig
| FNPTR MethodRefSig
| GENERICINST (CLASS | VALUETYPE) TypeDefOrRefEncoded GenArgCount Type *
| MVAR number
| OBJECT
| PTR CustomMod* Type
| PTR CustomMod* VOID
| STRING
| SZARRAY CustomMod* Type (single dimensional, zero-based array i.e., vector)
| VALUETYPE TypeDefOrRefEncoded
| VAR number
```

The *GenArgCount* non-terminal is an int32 value (compressed) specifying the number of generic arguments in this signature. The *number* non-terminal following MVAR or VAR is an unsigned integer value (compressed).

23.2.13 ArrayShape

An ArrayShape has the following syntax diagram:



Rank is an unsigned integer (stored in compressed form, see §23.2) that specifies the number of dimensions in the array (shall be 1 or more). *NumSizes* is a compressed unsigned integer that says how many dimensions have specified sizes (it shall be 0 or more). *Size* is a compressed unsigned 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 unsigned integer that says how many dimensions have specified lower bounds (it shall be 0 or more). And *LoBound* is a compressed signed integer specifying the lower bound of that dimension – the sequence starts at the first dimension, and goes on for a total of *NumLoBounds* items. None of the dimensions in these two sequences can be skipped, but the number of specified dimensions can be less than *Rank*.

Here are a few examples, all for element type `int32`:

	Type	Rank	NumSizes	Size	NumLoBounds	LoBound
`[0...2]`	I4	1	1	3	0	
`[,,,,,,]`	I4	7	0		0	
`[0...3, 0...2,,,,]`	I4	6	2	4 3	2	0 0
`[1...2, 6...8]`	I4	2	2	2 3	2	1 6
`[5, 3...5, , ]`	I4	4	2	5 3	2	0 3

[*Note:* definitions can nest, since the Type can itself be an array. *end note*]

23.2.14 TypeSpec

The signature in the Blob heap indexed by a *TypeSpec* token has the following format –

```
TypeSpecBlob ::=
```

```
  PTR      CustomMod*  VOID
| PTR      CustomMod*  Type
| FNPTR    MethodDefSig
| FNPTR    MethodRefSig
| ARRAY    Type  ArrayShape
| SZARRAY  CustomMod*  Type
| GENERICINST (CLASS | VALUETYPE) TypeDefOrRefEncoded GenArgCount Type Type*
```

For compactness, the ELEMENT_TYPE_ prefixes have been omitted from this list. So, for example, "PTR" is shorthand for ELEMENT_TYPE_PTR. (§23.1.16) Note that a TypeSpecBlob does *not* begin with a calling-convention byte, so it differs from the various other signatures that are stored into Metadata.

23.2.15 MethodSpec

The signature in the Blob heap indexed by a *MethodSpec* token has the following format –

```
MethodSpecBlob ::=
  GENRICINST GenArgCount Type Type*
```

GENRICINST has the value 0x0A. [*Note*: This value is known as IMAGE_CEE_CS_CALLCONV_GENERICINST in the Microsoft CLR implementation. *end note*] The *GenArgCount* is a compressed unsigned integer indicating the number of generic arguments in the method. The blob then specifies the instantiated type, repeating a total of *GenArgCount* times.

23.2.16 Short form signatures

The general specification for signatures leaves some leeway in how to encode certain items. For example, it appears valid 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. The following table shows which short-forms should be used in place of each long-form item. (As usual, for compactness, the ELEMENT_TYPE_ prefix have been omitted here – so VALUETYPE is short for ELEMENT_TYPE_VALUETYPE)

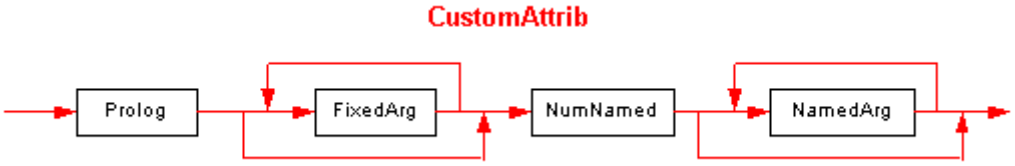
Long Form		**Short Form**
Prefix	**TypeRef to:**	
CLASS	System.String	STRING
CLASS	System.Object	OBJECT
VALUETYPE	System.Void	VOID
VALUETYPE	System.Boolean	BOOLEAN
VALUETYPE	System.Char	CHAR
VALUETYPE	System.Byte	U1
VALUETYPE	System.Sbyte	I1
VALUETYPE	System.Int16	I2
VALUETYPE	System.UInt16	U2
VALUETYPE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

VALUETYPE	System.IntPtr	I
VALUETYPE	System.UIntPtr	U
VALUETYPE	System.TypedReference	TYPEDBYREF

[*Note:* arrays shall be encoded in signatures using one of ELEMENT_TYPE_ARRAY or ELEMENT_TYPE_SZARRAY. There is no long form involving a TypeRef to System.Array. *end note*]

23.3 Custom attributes

A Custom Attribute has the following syntax diagram:

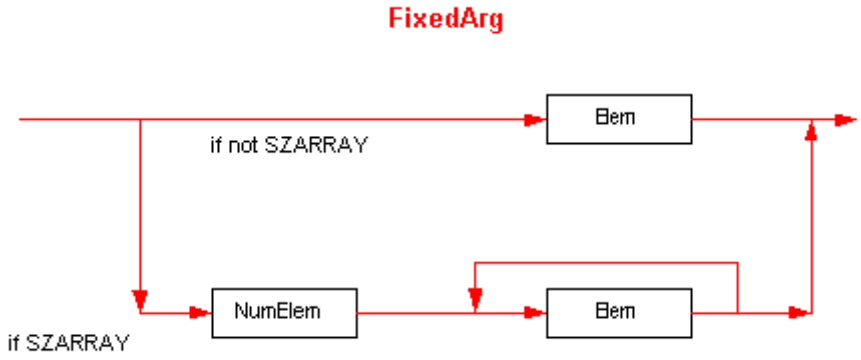


All binary values are stored in little-endian format (except *PackedLen* items, which are used only as counts for the number of bytes to follow in a UTF8 string). If there are no fields, parameters, or properties specified the entire attribute is represented as an empty blob.

CustomAttrib starts with a *Prolog* – an unsigned *int16*, 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's row in the *MethodDef* table; this information is *not* repeated in the *CustomAttrib* itself. As the syntax diagram shows, there can be zero or more *FixedArg*s. (Note that VARARG constructor methods are not allowed in the definition of Custom Attributes.)

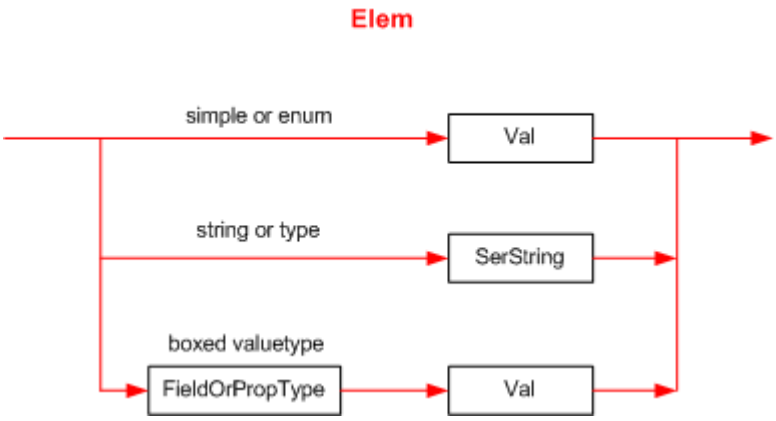
Next is a description of the optional "named" fields and properties. This starts with *NumNamed* – an unsigned *int16* giving the number of "named" properties or fields that follow. Note that *NumNamed* shall always be present. A value of zero indicates that there are no "named" properties or fields to follow (and of course, in this case, the *CustomAttrib* shall end immediately after *NumNamed*). In the case where *NumNamed* is non-zero, it is followed by *NumNamed* repeats of *NamedArg*s.



The format for each *FixedArg* depends upon whether that argument is an SZARRAY or not – this is shown in the lower and upper paths, respectively, of the syntax diagram. So each *FixedArg* is either a single *Elem*, or *NumElem* repeats of *Elem*.

(SZARRAY is the single byte 0x1D, and denotes a vector – a single-dimension array with a lower bound of zero.)

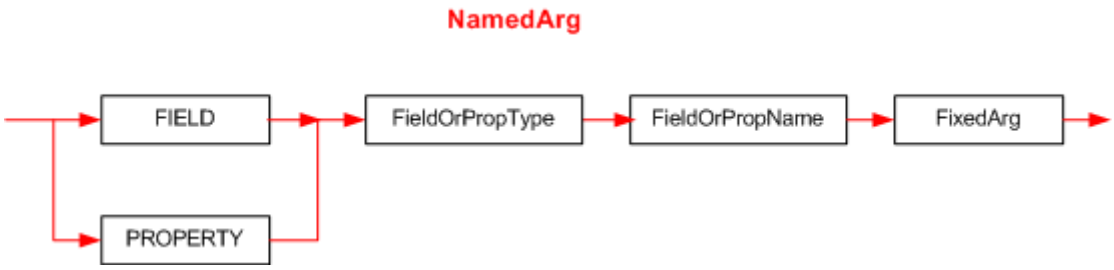
NumElem is an unsigned *int32* specifying the number of elements in the SZARRAY, or 0xFFFFFFFF to indicate that the value is null.



An *Elem* takes one of the forms in this diagram, as follows:

- If the parameter kind is simple (first line in the above diagram) (`bool`, `char`, `float32`, `float64`, `int8`, `int16`, `int32`, `int64`, `unsigned int8`, `unsigned int16`, `unsigned int32` or `unsigned int64`) then the 'blob' contains its binary value (*Val*). (A *bool* is a single byte with value 0 (false) or 1 (true); *char* is a two-byte Unicode character; and the others have their obvious meaning.) This pattern is also used if the parameter kind is an *enum* -- simply store the value of the enum's underlying integer type.
- If the parameter kind is *string*, (middle line in above diagram) then the blob contains a *SerString* – a *PackedLen* count of bytes, followed by the UTF8 characters. If the string is null, its *PackedLen* has the value 0xFF (with no following characters). If the string is empty (“”), then *PackedLen* has the value 0x00 (with no following characters).
- If the parameter kind is *System.Type*, (also, the middle line in above diagram) its value is stored as a *SerString* (as defined in the previous paragraph), representing its canonical name. The canonical name is its full type name, followed optionally by the assembly where it is defined, its version, culture and public-key-token. If the assembly name is omitted, the CLI looks first in the current assembly, and then in the system library (mscorlib); in these two special cases, it is permitted to omit the assembly-name, version, culture and public-key-token.
- If the parameter kind is *System.Object*, (third line in the above diagram) the value stored represents the “boxed” instance of that value-type. In this case, the blob contains the actual type's *FieldOrPropType* (see below), followed by the argument’s unboxed value. [*Note*: it is not possible to pass a value of **null** in this case. *end note*]
- If the type is a boxed simple value type (`bool`, `char`, `float32`, `float64`, `int8`, `int16`, `int32`, `int64`, `unsigned int8`, `unsigned int16`, `unsigned int32` or `unsigned int64`) then *FieldOrPropType* is immediately preceded by a byte containing the value 0x51 .

  The *FieldOrPropType* shall be exactly one of: `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`. A single-dimensional, zero-based array is specified as a single byte 0x1D followed by the *FieldOrPropType* of the element type. (See §23.1.16) An enum is specified as a single byte 0x55 followed by a *SerString*.



A *NamedArg* is simply a *FixedArg* (discussed above) preceded by information to identify which field or property it represents. [*Note*: Recall that the CLI allows fields and properties to have the same name; so we require a means to disambiguate such situations. *end note*]

`FIELD` is the single byte 0x53.

`PROPERTY` is the single byte 0x54.

The *FieldOrPropName* is the name of the field or property, stored as a *SerString* (defined above).

A number of examples involving custom attributes are contained in Annex B of Partition VI.

23.4 Marshalling descriptors

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 code via PInvoke dispatch. The ILAsm syntax `marshal` can be used to create a marshalling descriptor, as can the pseudo custom attribute *MarshalAsAttribute* – see §21.2.1)

Note that a conforming implementation of the CLI need only support marshalling of the types specified earlier – see §15.5.4.

Marshalling descriptors make use of constants named NATIVE_TYPE_xxx. Their names and values are listed in the following table:

Name	Value
NATIVE_TYPE_BOOLEAN	0x02
NATIVE_TYPE_I1	0x03
NATIVE_TYPE_U1	0x04
NATIVE_TYPE_I2	0x05
NATIVE_TYPE_U2	0x06
NATIVE_TYPE_I4	0x07
NATIVE_TYPE_U4	0x08
NATIVE_TYPE_I8	0x09
NATIVE_TYPE_U8	0x0a
NATIVE_TYPE_R4	0x0b
NATIVE_TYPE_R8	0x0c
NATIVE_TYPE_LPSTR	0x14
NATIVE_TYPE_LPWSTR	0x15
NATIVE_TYPE_INT	0x1f
NATIVE_TYPE_UINT	0x20
NATIVE_TYPE_FUNC	0x26
NATIVE_TYPE_ARRAY	0x2a

The 'blob' has the following format –

```
MarshalSpec ::=
  NativeIntrinsic
| ARRAY ArrayElemType
| ARRAY ArrayElemType ParamNum
| ARRAY ArrayElemType ParamNum NumElem

NativeIntrinsic ::=
  BOOLEAN | I1 | U1 | I2 | U2 | I4 | U4 | I8 | U8 | R4 | R8
| LPSTR | LPSTR | INT | UINT | FUNC
```

For compactness, the `NATIVE_TYPE_` prefixes have been omitted in the above lists; for example, "`ARRAY`" is shorthand for `NATIVE_TYPE_ARRAY`.

```
ArrayElemType ::=
   NativeIntrinsic
```

ParamNum is an unsigned integer (compressed as described in §23.2) specifying the parameter in the method call that provides the number of elements in the array – see below.

NumElem is an unsigned integer compressed as described in §23.2 (specifying the number of elements or additional elements – see below).

[*Note:* For example, in the method declaration:

```
.method void M(int32[] ar1, int32 size1, unsigned int8[] ar2, int32 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

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 indicates that parameter number 2 in the method (the one called `size1`) will specify the number of elements in the actual array – let's suppose its value on a particular call is 42. The value of *NumElem* is 0. The calculated total size, in bytes, of the array is given by the formula:

```
if ParamNum = 0
   SizeInBytes = NumElem * sizeof (elem)
else
   SizeInBytes = ( @ParamNum +  NumElem ) * sizeof (elem)
endif
```

The syntax "*@ParamNum*" is used here 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` (§23.1.16) of size 4 bytes. *end note*]

24 Metadata physical layout

The physical on-disk representation of metadata is a direct reflection of the logical representation described in §22 and §23. That is, data is stored in streams representating the metadata tables and heaps. The main complication is that, where the logical representation is abstracted from the number of bytes needed for indexing into tables and columns, the physical representation has to take care of that explicitly by defining how to map logical metadata heaps and tables into their physical representations.

Unless stated otherwise, all binary values are stored in little-endian format.

24.1 Fixed fields

Complete CLI components (metadata and CIL instructions) are stored in a subset of the current Portable Executable (PE) File Format (§25). Because of this heritage, some of the fields in the physical representation of metadata have fixed values. When writing these fields it is best that they be set to the value indicated, on reading they should be ignored.

24.2 File headers

24.2.1 Metadata root

The root of the physical metadata starts with a magic signature, several bytes of version and other miscellaneous information, followed by a count and an array of stream headers, one for each stream that is present. The actual encoded tables and heaps are stored in the streams, which immediately follow this array of headers.

Offset	Size	Field	Description
0	4	**Signature**	Magic signature for physical metadata : 0x424A5342.
4	2	**MajorVersion**	Major version, 1 (ignore on read)
6	2	**MinorVersion**	Minor version, 1 (ignore on read)
8	4	**Reserved**	Reserved, always 0 (§24.1).
12	4	**Length**	Number of bytes allocated to hold version string (including null terminator), call this *x*. Call the length of the string (including the terminator) *m (*we require *m* <= 255); the length *x* is *m* rounded up to a multiple of four.
16	*m*	**Version**	UTF8-encoded null-terminated version string of length *m* (see below)
16+*m*	*x*−*m*		Padding to next 4 byte boundary.
16+*x*	2	**Flags**	Reserved, always 0 (§24.1).
16+*x*+2	2	**Streams**	Number of streams, say *n*.
16+*x*+4		**StreamHeaders**	Array of *n* StreamHdr structures.

The Version string shall be “Standard CLI 2005” for any file that is intended to be executed on any conforming implementation of the CLI, and all conforming implementations of the CLI shall accept files that use this version string. Other strings shall be used when the file is restricted to a vendor-specific implementation of the CLI. Future versions of this standard shall specify different strings, but they shall begin “Standard CLI”. Other standards that specify additional functionality shall specify their own specific version strings beginning with “Standard□”, where “□” represents a single space. Vendors that provide implementation-specific extensions shall provide a version string that does *not* begin with “Standard□”. (For the first version of this Standard, the Version string was “Standard CLI 2002”.)

24.2.2 Stream header

A stream header gives the names, and the position and length of a particular table or heap. Note that the length of a Stream header structure is not fixed, but depends on the length of its name field (a variable length null-terminated string).

Offset	Size	Field	Description
0	4	**Offset**	Memory offset to start of this stream from start of the metadata root (§24.2.1)
4	4	**Size**	Size of this stream in bytes, shall be a multiple of 4.
8		**Name**	Name of the stream as null-terminated variable length array of ASCII characters, padded to the next 4-byte boundary with \0 characters. The name is limited to 32 characters.

Both logical tables and heaps are stored in streams. There are five possible kinds of streams. A stream header with name “#Strings” that points to the physical representation of the string heap where identifier strings are stored; a stream header with name “#US” that points to the physical representation of the user string heap; a stream header with name “#Blob” that points to the physical representation of the blob heap, a stream header with name “#GUID” that points to the physical representation of the GUID heap; and a stream header with name “#~” that points to the physical representation of a set of tables.

Each kind of stream shall occur at most once, that is, a meta-data file shall not contain two “#US” streams, or five “#Blob” streams. Streams need not be there if they are empty.

The next subclauses describe the structure of each kind of stream in more detail.

24.2.3 #Strings heap

The stream of bytes pointed to by a “#Strings” header is the physical representation of the logical string heap. The physical heap can contain garbage, that is, it can contain parts that are unreachable from any of the tables, but parts that are reachable from a table shall contain a valid null-terminated UTF8 string. When the #String heap is present, the first entry is always the empty string (i.e., \0).

24.2.4 #US and #Blob heaps

The stream of bytes pointed to by a “#US” or “#Blob” header are the physical representation of logical Userstring and 'blob' heaps respectively. Both these heaps can contain garbage, as long as any part that is reachable from any of the tables contains a valid 'blob'. Individual blobs are stored with their length encoded in the first few bytes:

- If the first one byte of the 'blob' is $0bbbbbbb_2$, then the rest of the 'blob' contains the $bbbbbbb_2$ bytes of actual data.
- If the first two bytes of the 'blob' are $10bbbbbb_2$ and x, then the rest of the 'blob' contains the ($bbbbbb_2$ << 8 + x) bytes of actual data.
- If the first four bytes of the 'blob' are $110bbbbb_2$, x, y, and z, then the rest of the 'blob' contains the ($bbbbb_2$ << 24 + x << 16 + y << 8 + z) bytes of actual data.

The first entry in both these heaps is the empty 'blob' that consists of the single byte 0x00.

Strings in the #US (user string) heap are encoded using 16-bit Unicode encodings. The count on each string is the number of bytes (not characters) in the string. Furthermore, there is an additional terminal byte (so all byte counts are odd, not even). This final byte holds the value 1 if and only if any UTF16 character within the string has any bit set in its top byte, or its low byte is any of the following: 0x01–0x08, 0x0E–0x1F, 0x27, 0x2D, 0x7F. Otherwise, it holds 0. The 1 signifies Unicode characters that require handling beyond that normally provided for 8-bit encoding sets.

24.2.5 #GUID heap

The “#GUID” header points to a sequence of 128-bit GUIDs. There might be unreachable GUIDs stored in the stream.

24.2.6 #~ stream

The “#~” streams contain the actual physical representations of the logical metadata tables (§22). A “#~” stream has the following top-level structure:

Offset	Size	Field	Description
0	4	**Reserved**	Reserved, always 0 (§24.1).
4	1	**MajorVersion**	Major version of table schemata; shall be 2 (§24.1).
5	1	**MinorVersion**	Minor version of table schemata; shall be 0 (§24.1).
6	1	**HeapSizes**	Bit vector for heap sizes.
7	1	**Reserved**	Reserved, always 1 (§24.1).
8	8	**Valid**	Bit vector of present tables, let *n* be the number of bits that are 1.
16	8	**Sorted**	Bit vector of sorted tables.
24	4*n	**Rows**	Array of *n* 4-byte unsigned integers indicating the number of rows for each present table.
24+4*n		**Tables**	The sequence of physical tables.

The HeapSizes field is a bitvector that encodes the width of indexes into the various heaps. 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.

Heap size flag	Description
0x01	Size of “#String” stream >= 2^{16}.
0x02	Size of “#GUID” stream >= 2^{16}.
0x04	Size of “#Blob” stream >= 2^{16}.

The Valid field is a 64-bit bitvector that has a specific bit set for each table that is stored in the stream; the mapping of tables to indexes is given at the start of §22. For example when the DeclSecurity table is present in the logical metadata, bit 0x0e should be set in the Valid vector. It is invalid to include non-existent tables in Valid, so all bits above 0x2c shall be zero.

The Rows array contains the number of rows for each of the tables that are present. When decoding physical metadata to logical metadata, the number of 1’s in Valid indicates the number of elements in the Rows array.

A crucial aspect in the encoding of a logical table is its *schema*. The schema for each table is given in §22. For example, the table with assigned index 0x02 is a *TypeDef* table, which, according to its specification in §22.37, has the following columns: a 4-byte-wide flags, an index into the String heap, another index into the String heap, an index into *TypeDef* , *TypeRef* , or *TypeSpec* table, an index into *Field* table, and an index into *MethodDef* table.

The physical representation of a table with *n* columns and *m* rows with schema ($C_0,\ldots,C_{n-1}$) consists of the concatenation of the physical representation of each of its rows. The physical representation of a row with schema ($C_0,\ldots,_{n-1}$) is the concatenation of the physical representation of each of its elements. The physical representation of a row cell *e* at a column with type *C* is defined as follows:

- If e is a constant, it is stored using the number of bytes as specified for its column type *C* (i.e., a 2-bit mask of type PropertyAttributes)
- If *e* is an index into the GUID heap, 'blob', or String heap, it is stored using the number of bytes as defined in the HeapSizes field.
- If *e* is a simple index into a table with index *i*, it is stored using 2 bytes if table *i* has less than 2^{16} rows, otherwise it is stored using 4 bytes.
- If *e* is a *coded index* that points into table t_i out of *n* possible tables $t_0, \ldots t_{n-1}$, then it is stored as e << (log n) | tag{ $t_0, \ldots t_{n-1}$}[t_i] using 2 bytes if the maximum number of rows of tables $t_0, \ldots t_{n-1}$, is less than $2^{(16 - (\log n))}$, and using 4 bytes otherwise. The family of finite maps tag{ $t_0, \ldots t_{n-1}$} is defined below. Note that decoding a physical row requires the inverse of this mapping. [For example, the *Parent* column of the *Constant* table indexes a row in the *Field*, *Param*, or *Property* tables. The actual table is encoded into the low 2 bits of the number, using the values: 0 => *Field*, 1 => *Param*, 2 => *Property*.The remaining bits hold the actual row number being indexed. For example, a value of 0x321, indexes row number 0xC8 in the *Param* table.]

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

HasConstant: 2 bits to encode tag	**Tag**
Field	0
Param	1
Property	2

HasCustomAttribute: 5 bits to encode tag	**Tag**
MethodDef	0
Field	1
TypeRef	2
TypeDef	3
Param	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

[*Note*: HasCustomAttributes only has values for tables that are "externally visible"; that is, that correspond to items in a user source program. For example, an attribute can be attached to a TypeDef table and a Field table, but not a ClassLayout table. As a result, some table types are missing from the enum above. *end note*]

HasFieldMarshall: 1 bit to encode tag	**Tag**
Field	0
Param	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**
Field	0
MethodDef	1

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

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

TypeOrMethodDef: 1 bit to encode tag	**Tag**
TypeDef	0
MethodDef	1

25 File format extensions to PE

This contains informative text only

The file format for CLI components is a strict extension of the current Portable Executable (PE) File Format. This extended PE format enables the operating system to recognize runtime images, accommodates code emitted as CIL or native code, and accommodates runtime metadata as an integral part of the emitted code. There are also specifications for a subset of the full Windows PE/COFF file format, in sufficient detail that a tool or compiler can use the specifications to emit valid CLI images.

The PE format frequently uses the term RVA (Relative Virtual Address). An RVA is the address of an item *once loaded into memory*, with the base address of the image file subtracted from it (i.e., the offset from the base address where the file is loaded). The RVA of an item will almost always differ from its position within the file on disk. To compute the file position of an item with RVA *r*, search all the sections in the PE file to find the section with RVA *s*, length *l* and file position *p* in which the RVA lies, ie $s \leq r < s+l$. The file position of the item is then given by $p+(r-s)$.

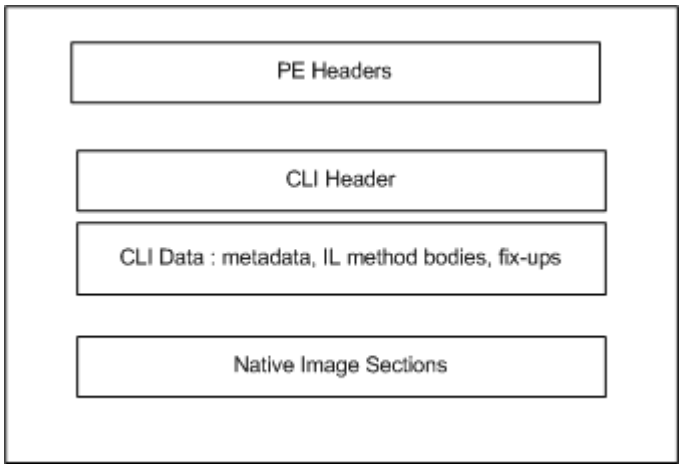
Unless stated otherwise, all binary values are stored in little-endian format.

End informative text

25.1 Structure of the runtime file format

The figure below provides a high-level view of the CLI file format. All runtime images contain the following:

- PE headers, with specific guidelines on how field values should be set in a runtime file.
- A CLI header that contains all of the runtime specific data entries. The runtime header is read-only and shall be placed in any read-only section.
- The sections that contain the actual data as described by the headers, including imports/exports, data, and code.



The CLI header (§25.3.3) is found using CLI Header directory entry in the PE header. The CLI header in turn contains the address and sizes of the runtime data (for metadata, see §24; for CIL see § 25.4) in the rest of the image. Note that the runtime data can be merged into other areas of the PE format with the other data based on the attributes of the sections (such as read only versus execute, etc.).

25.2 PE headers

A PE image starts with an MS-DOS header followed by a PE signature, followed by the PE file header, and then the PE optional header followed by PE section headers.

25.2.1 MS-DOS header

The PE format starts with an MS-DOS stub of exactly the following 128 bytes to be placed at the front of the module. At offset 0x3c in the DOS header is a 4-byte unsigned integer offset, lfanew, to the PE signature (shall be “PE\0\0”), immediately followed by the PE file header.

0x4d	0x5a	0x90	0x00	0x03	0x00	0x00	0x00
0x04	0x00	0x00	0x00	0xFF	0xFF	0x00	0x00
0xb8	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x40	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00	*lfanew*			
0x0e	0x1f	0xba	0x0e	0x00	0xb4	0x09	0xcd
0x21	0xb8	0x01	0x4c	0xcd	0x21	0x54	0x68
0x69	0x73	0x20	0x70	0x72	0x6f	0x67	0x72
0x61	0x6d	0x20	0x63	0x61	0x6e	0x6e	0x6f
0x74	0x20	0x62	0x65	0x20	0x72	0x75	0x6e
0x20	0x69	0x6e	0x20	0x44	0x4f	0x53	0x20
0x6d	0x6f	0x64	0x65	0x2e	0x0d	0x0d	0x0a
0x24	0x00	0x00	0x00	0x00	0x00	0x00	0x00

25.2.2 PE file header

Immediately after the PE signature is the PE File header consisting of the following:

Offset	Size	Field	Description
0	2	Machine	Always 0x14c.
2	2	Number of Sections	Number of sections; indicates size of the Section Table, which immediately follows the headers.
4	4	Time/Date Stamp	Time and date the file was created in seconds since January 1st 1970 00:00:00 or 0.
8	4	Pointer to Symbol Table	Always 0 (§24.1).
12	4	Number of Symbols	Always 0 (§24.1).
16	2	Optional Header Size	Size of the optional header, the format is described below.
18	2	Characteristics	Flags indicating attributes of the file, see §25.2.2.1.

25.2.2.1 Characteristics

A CIL-only DLL sets flag 0x2000 to 1, while a CIL-only .exe has flag 0x2000 set to zero:

Flag	Value	Description
IMAGE_FILE_DLL	0x2000	The image file is a dynamic-link library (DLL).

Except for the IMAGE_FILE_DLL flag (0x2000), flags 0x0002, 0x0004, 0x008, and 0x0100 shall all be set, while all others shall always be zero (§24.1).

25.2.3 PE optional header

Immediately after the PE Header is the PE Optional Header. This header contains the following information:

Offset	Size	Header part	Description
0	28	Standard fields	These define general properties of the PE file, see §25.2.3.1.
28	68	NT-specific fields	These include additional fields to support specific features of Windows, see 25.2.3.2.
96	128	Data directories	These fields are address/size pairs for special tables, found in the image file (for example, Import Table and Export Table).

25.2.3.1 PE header standard fields

These fields are required for all PE files and contain the following information:

Offset	Size	Field	Description
0	2	Magic	Always 0x10B (§24.1).
2	1	LMajor	Always 6 (§24.1).
3	1	LMinor	Always 0 (§24.1).
4	4	Code Size	Size of the code (text) section, or the sum of all code sections if there are multiple sections.
8	4	Initialized Data Size	Size of the initialized data section, or the sum of all such sections if there are multiple data sections.
12	4	Uninitialized Data Size	Size of the uninitialized data section, or the sum of all such sections if there are multiple unitinitalized data sections.
16	4	Entry Point RVA	RVA of entry point , needs to point to bytes 0xFF 0x25 followed by the RVA in a section marked execute/read for EXEs or 0 for DLLs
20	4	Base Of Code	RVA of the code section. (This is a hint to the loader.)
24	4	Base Of Data	RVA of the data section. (This is a hint to the loader.)

This contains informative text only

The entry point RVA shall always be either the x86 entry point stub or be 0. On non-CLI aware platforms, this stub will call the entry point API of mscoree (_CorExeMain or _CorDllMain). The mscoree entry point will use the module handle to load the metadata from the image, and invoke the entry point specified in vthe CLI header.

End informative text

25.2.3.2 PE header Windows NT-specific fields

These fields are Windows NT specific:

Offset	Size	Field	Description
28	4	Image Base	Always 0x400000 (§24.1).
32	4	Section Alignment	Always 0x2000 (§24.1).
36	4	File Alignment	Either 0x200 or 0x1000.
40	2	OS Major	Always 4 (§24.1).

42	2	OS Minor	Always 0 (§24.1).
44	2	User Major	Always 0 (§24.1).
46	2	User Minor	Always 0 (§24.1).
48	2	SubSys Major	Always 4 (§24.1).
50	2	SubSys Minor	Always 0 (§24.1).
52	4	Reserved	Always 0 (§24.1).
56	4	Image Size	Size, in bytes, of image, including all headers and padding; shall be a multiple of Section Alignment.
60	4	Header Size	Combined size of MS-DOS Header, PE Header, PE Optional Header and padding; shall be a multiple of the file alignment.
64	4	File Checksum	Always 0 (§24.1).
68	2	SubSystem	Subsystem required to run this image. Shall be either IMAGE_SUBSYSTEM_WINDOWS_CE_GUI (0x3) or IMAGE_SUBSYSTEM_WINDOWS_GUI (0x2).
70	2	DLL Flags	Always 0 (§24.1).
72	4	Stack Reserve Size	Always 0x100000 (1Mb) (§24.1).
76	4	Stack Commit Size	Always 0x1000 (4Kb) (§24.1).
80	4	Heap Reserve Size	Always 0x100000 (1Mb) (§24.1).
84	4	Heap Commit Size	Always 0x1000 (4Kb) (§24.1).
88	4	Loader Flags	Always 0 (§24.1)
92	4	Number of Data Directories	Always 0x10 (§24.1).

25.2.3.3 PE header data directories

The optional header data directories give the address and size of several tables that appear in the sections of the PE file. Each data directory entry contains the RVA and Size of the structure it describes, in that order.

Offset	Size	Field	Description
96	8	Export Table	Always 0 (§24.1).
104	8	Import Table	RVA and Size of Import Table, (§25.3.1).
112	8	Resource Table	Always 0 (§24.1).
120	8	Exception Table	Always 0 (§24.1).
128	8	Certificate Table	Always 0 (§24.1).
136	8	Base Relocation Table	Relocation Table; set to 0 if unused (§25.3.2).
144	8	Debug	Always 0 (§24.1).
152	8	Copyright	Always 0 (§24.1).
160	8	Global Ptr	Always 0 (§24.1).
168	8	TLS Table	Always 0 (§24.1).
176	8	Load Config Table	Always 0 (§24.1).

184	8	Bound Import	Always 0 (§24.1).
192	8	IAT	RVA and Size of Import Address Table, (§25.3.1).
200	8	Delay Import Descriptor	Always 0 (§24.1).
208	8	CLI Header	CLI Header with directories for runtime data, (§25.3.1).
216	8	Reserved	Always 0 (§24.1).

The tables pointed to by the directory entries are stored in one of the PE file's sections; these sections themselves are described by section headers.

25.3 Section headers

Immediately following the optional header is the Section Table, which contains a number of section headers. This positioning is required because the file header does not contain a direct pointer to the section table; the location of the section table is determined by calculating the location of the first byte after the headers.

Each section header has the following format, for a total of 40 bytes per entry:

Offset	Size	Field	Description
0	8	Name	An 8-byte, null-padded ASCII string. There is no terminating null if the string is exactly eight characters long.
8	4	VirtualSize	Total size of the section in bytes. If this value is greater than SizeOfRawData, the section is zero-padded.
12	4	VirtualAddress	For executable images this is the address of the first byte of the section, when loaded into memory, relative to the image base.
16	4	SizeOfRawData	Size of the initialized data on disk in bytes, shall be a multiple of FileAlignment from the PE header. If this is less than VirtualSize the remainder of the section is zero filled. Because this field is rounded while the VirtualSize field is not it is possible for this to be greater than VirtualSize as well. When a section contains only uninitialized data, this field should be 0.
20	4	PointerToRawData	Offset of section's first page within the PE file. This shall be a multiple of FileAlignment from the optional header. When a section contains only uninitialized data, this field should be 0.
24	4	PointerToRelocations	RVA of Relocation section.
28	4	PointerToLinenumbers	Always 0 (§24.1).
32	2	NumberOfRelocations	Number of relocations, set to 0 if unused.
34	2	NumberOfLinenumbers	Always 0 (§24.1).
36	4	Characteristics	Flags describing section's characteristics, see below.

The following table defines the possible characteristics of the section.

Flag	Value	Description
IMAGE_SCN_CNT_CODE	0x00000020	Section contains executable code.
IMAGE_SCN_CNT_INITIALIZED_DATA	0x00000040	Section contains initialized data.
IMAGE_SCN_CNT_UNINITIALIZED_DATA	0x00000080	Section contains uninitialized data.
IMAGE_SCN_MEM_EXECUTE	0x20000000	Section can be executed as code.

IMAGE_SCN_MEM_READ	0x40000000	Section can be read.
IMAGE_SCN_MEM_WRITE	0x80000000	Section can be written to.

25.3.1 Import Table and Import Address Table (IAT)

The Import Table and the Import Address Table (IAT) are used to import the `_CorExeMain` (for a .exe) or `_CorDllMain` (for a .dll) entries of the runtime engine (mscoree.dll). The Import Table directory entry points to a one element zero terminated array of Import Directory entries (in a general PE file there is one entry for each imported DLL):

Offset	Size	Field	Description
0	4	ImportLookupTable	RVA of the Import Lookup Table
4	4	DateTimeStamp	Always 0 (§24.1).
8	4	ForwarderChain	Always 0 (§24.1).
12	4	Name	RVA of null-terminated ASCII string “mscoree.dll”.
16	4	ImportAddressTable	RVA of Import Address Table (this is the same as the RVA of the IAT descriptor in the optional header).
20	20		End of Import Table. Shall be filled with zeros.

The Import Lookup Table and the Import Address Table (IAT) are both one element, zero terminated arrays of RVAs into the Hint/Name table. Bit 31 of the RVA shall be set to 0. In a general PE file there is one entry in this table for every imported symbol.

Offset	Size	Field	Description
0	4	Hint/Name Table RVA	A 31-bit RVA into the Hint/Name Table. Bit 31 shall be set to 0 indicating import by name.
4	4		End of table, shall be filled with zeros.

The IAT should be in an executable and writable section as the loader will replace the pointers into the Hint/Name table by the actual entry points of the imported symbols.

The Hint/Name table contains the name of the dll-entry that is imported.

Offset	Size	Field	Description
0	2	Hint	Shall be 0.
2	variable	Name	Case sensitive, null-terminated ASCII string containing name to import. Shall be “_CorExeMain” for a .exe file and “_CorDllMain” for a .dll file.

25.3.2 Relocations

In a pure CIL image, a single fixup of type IMAGE_REL_BASED_HIGHLOW (0x3) is required for the x86 startup stub which access the IAT to load the runtime engine on down level loaders. When building a mixed CIL/native image or when the image contains embedded RVAs in user data, the relocation section contains relocations for these as well.

The relocations shall be in their own section, named “.reloc”, which shall be the final section in the PE file. The relocation section contains a Fix-Up Table. The fixup table is broken into blocks of fixups. Each block represents the fixups for a 4K page, and each block shall start on a 32-bit boundary.

Each fixup block starts with the following structure:

Offset	Size	Field	Description

0	4	PageRVA	The RVA of the block in which the fixup needs to be applied. The low 12 bits shall be zero.
4	4	Block Size	Total number of bytes in the fixup block, including the Page RVA and Block Size fields, as well as the Type/Offset fields that follow, rounded up to the next multiple of 4.

The Block Size field is then followed by (BlockSize –8)/2 Type/Offset. Each entry is a word (2 bytes) and has the following structure (if necessary, insert 2 bytes of 0 to pad to a multiple of 4 bytes in length):

Offset	Size	Field	Description
0	4 bits	Type	Stored in high 4 bits of word. Value indicating which type of fixup is to be applied (described above)
0	12 bits	Offset	Stored in remaining 12 bits of word. Offset from starting address specified in the Page RVA field for the block. This offset specifies where the fixup is to be applied.

25.3.3 CLI header

The CLI header contains all of the runtime-specific data entries and other information. The header should be placed in a read-only, sharable section of the image. This header is defined as follows:

Offset	Size	Field	Description
0	4	Cb	Size of the header in bytes
4	2	MajorRuntimeVersion	The minimum version of the runtime required to run this program, currently 2.
6	2	MinorRuntimeVersion	The minor portion of the version, currently 0.
8	8	MetaData	RVA and size of the physical metadata (§24).
16	4	Flags	Flags describing this runtime image. (§25.3.3.1).
20	4	EntryPointToken	Token for the *MethodDef* or File of the entry point for the image
24	8	Resources	RVA and size of implementation-specific resources.
32	8	StrongNameSignature	RVA of the hash data for this PE file used by the CLI loader for binding and versioning
40	8	CodeManagerTable	Always 0 (§24.1).
48	8	VTableFixups	RVA of an array of locations in the file that contain an array of function pointers (e.g., vtable slots), see below.
56	8	ExportAddressTableJumps	Always 0 (§24.1).
64	8	ManagedNativeHeader	Always 0 (§24.1).

25.3.3.1 Runtime flags

The following flags describe this runtime image and are used by the loader.

Flag	Value	Description
COMIMAGE_FLAGS_ILONLY	0x00000001	Always 1 (§24.1).

COMIMAGE_FLAGS_32BITREQUIRED	0x00000002	Image can only be loaded into a 32-bit process, for instance if there are 32-bit vtablefixups, or casts from native integers to int32. CLI implementations that have 64-bit native integers shall refuse loading binaries with this flag set.
COMIMAGE_FLAGS_STRONGNAMESIGNED	0x00000008	Image has a strong name signature.
COMIMAGE_FLAGS_TRACKDEBUGDATA	0x00010000	Always 0 (§24.1).

25.3.3.2 Entry point metadata token

- The entry point token (§15.4.1.2) is always a *MethodDef* token (§22.26) or File token (§22.19) when the entry point for a multi-module assembly is not in the manifest assembly. The signature and implementation flags in metadata for the method indicate how the entry is run

25.3.3.3 Vtable fixup

Certain languages, which choose not to follow the common type system runtime model, can 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 (§15.5.1). 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.

Offset	Size	Field	Description
0	4	**VirtualAddress**	RVA of Vtable
4	2	**Size**	Number of entries in Vtable
6	2	**Type**	Type of the entries, as defined in table below

Constant	Value	Description
COR_VTABLE_32BIT	0x01	Vtable slots are 32 bits.
COR_VTABLE_64BIT	0x02	Vtable slots are 64 bits.
COR_VTABLE_FROM_UNMANAGED	0x04	Transition from unmanaged to managed code.
COR_VTABLE_CALL_MOST_DERIVED	0x10	Call most derived method described by the token (only valid for virtual methods).

25.3.3.4 Strong name signature

This header entry points to the strong name hash for an image that can be used to deterministically identify a module from a referencing point (§6.2.1.3).

25.4 Common Intermediate Language physical layout

This section contains the layout of the data structures used to describe a CIL method and its exceptions. Method bodies can be stored in any read-only section of a PE file. The *MethodDef* (§22.26) records in metadata carry each method's RVA.

A method consists of a method header immediately followed by the method body, possibly followed by extra method data sections (§25.4.5), typically exception handling data. If exception-handling data is present, then

CorILMethod_MoreSects flag (§25.4.4) shall be specified in the method header and for each chained item after that.

There are two flavors of method headers - tiny (§25.4.2) and fat (§25.4.3). The two least significant bits in a method header indicate which type is present (§25.4.1). The tiny header is 1 byte long and stores only the method's code size. A method is given a tiny header if it has no local variables, maxstack is 8 or less, the method has no exceptions, the method size is less than 64 bytes, and the method has no flags above 0x7. Fat headers carry full information - local vars signature token, maxstack, code size, flag. Tiny method headers can start on any byte boundary. Fat method headers shall start on a 4-byte boundary.

25.4.1 Method header type values

The two least significant bits of the first byte of the method header indicate what type of header is present. These 2 bits will be one and only one of the following:

Value	Value	Description
`CorILMethod_TinyFormat`	`0x2`	The method header is tiny (§25.4.2) .
`CorILMethod_FatFormat`	`0x3`	The method header is fat (§25.4.3).

25.4.2 Tiny format

Tiny headers use a 6-bit length encoding. The following is true for all tiny headers:

- No local variables are allowed
- No exceptions
- No extra data sections
- The operand stack shall be no bigger than 8 entries

A Tiny Format header is encoded as follows:

Start Bit	Count of Bits	Description
`0`	`2`	Flags (CorILMethod_TinyFormat shall be set, see §25.4.4).
`2`	`6`	Size, in bytes, of the method body immediately following this header.

25.4.3 Fat format

The fat format is used whenever the tiny format is not sufficient. This can be true for one or more of the following reasons:

- The method is too large to encode the size (i.e., at least 64 bytes)
- There are exceptions
- There are extra data sections
- There are local variables
- The operand stack needs more than 8 entries

A fat header has the following structure

Offset	Size	Field	Description
`0`	`12 (bits)`	**Flags**	Flags (CorILMethod_FatFormat shall be set in bits 0:1, see §25.4.4)
`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

25.4.4 Flags for method headers

The first byte of a method header can also contain the following flags, valid only for the Fat format, that indicate how the method is to be executed:

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 (§25.4.5).
CorILMethod_InitLocals	0x10	Call default constructor on all local variables.

25.4.5 Method data section

At the next 4-byte boundary following the method body can be extra method data sections. These method data sections start with a two byte header (1 byte for flags, 1 byte for the length of the actual data) or a 4-byte header (1 byte for flags, and 3 bytes for length of the actual data). The first byte determines the kind of the header, and what data is in the actual section:

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 least-significant byte first format. If not set, the header is small with a 1-byte length
CorILMethod_Sect_MoreSects	0x80	Another data section occurs after this current section

Currently, the method data sections are only used for exception tables (§19). The layout of a small exception header structure as is a follows:

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 (§25.4.6).

The layout of a fat exception header structure is as follows:

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 (§25.4.6).

25.4.6 Exception handling clauses

Exception handling clauses also come in small and fat versions.

The small form of the exception clause should be used whenever the code sizes for the try block and the handler code are both smaller than 256 bytes and both their offsets are smaller than 65536. The format for a small exception clause is as follows:

Offset	Size	Field	Description
0	2	**Flags**	Flags, see below.
2	2	**TryOffset**	Offset in bytes of try block from start of method body.
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

The layout of the fat form of exception handling clauses is as follows:

Offset	Size	Field	Description
0	4	**Flags**	Flags, see below.
4	4	**TryOffset**	Offset in bytes of try block from start of method body.
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

The following flag values are used for each exception-handling clause:

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)

26 Index