Common Language Infrastructure (CLI)
Partition IV:
Profiles and Libraries
(With Added Microsoft Specific Implementation Notes)

Table of contents

1	**Overview**	**1**
2	**Libraries and Profiles**	**2**
2.1	Libraries	2
2.2	Profiles	2
2.3	The relationship between Libraries and Profiles	3
3	**The Standard Profiles**	**4**
3.1	The Kernel Profile	4
3.2	The Compact Profile	4
4	**Kernel Profile feature requirements**	**5**
4.1	Features excluded from the Kernel Profile	5
4.1.1	Floating point	5
4.1.2	Non-vector arrays	5
4.1.3	Reflection	5
4.1.4	Application domains	6
4.1.5	Remoting	6
4.1.6	Vararg	6
4.1.7	Frame growth	6
4.1.8	Filtered exceptions	6
5	**The standard libraries**	**7**
5.1	General comments	7
5.2	Runtime infrastructure library	7
5.3	Base Class Library (BCL)	7
5.4	Network library	7
5.5	Reflection library	7
5.6	XML library	7
5.7	Extended numerics library	8
5.8	Extended array library	8
5.9	Vararg library	8
5.10	Parallel library	8
6	**Implementation-specific modifications to the system libraries**	**10**
7	**The XML specification**	**11**

7.1 Semantics 11
7.1.1 Value types as objects 19
7.1.2 Exceptions 19
7.2 XML signature notation issues 19
7.2.1 Serialization 19
7.2.2 Delegates 19
7.2.3 Properties 20
7.2.4 Nested types 20

1 Overview

[*Note:* While compiler writers are most concerned with issues of file format, instruction set design, and a common type system, application programmers are most interested in the programming library that is available to them in the language they are using. The Common Language Infrastructure (CLI) specifies a Common Language Specification (CLS, see Partition I) that shall be used to define the externally visible aspects (such as method signatures) when they are intended to be used from a wide range of programming languages. Since it is the goal of the CLI Libraries to be available from as many programming languages as possible, all of the library functionality is available through CLS-compliant types and type members.

The CLI Libraries were designed with the following goals in mind:

- To support for a wide variety of programming languages.
- To have consistent design patterns throughout.
- To have features on parity with the ISO/IEC C Standard library of 1990.
- To support more recent programming paradigms, notably networking, XML, runtime type inspection, instance creation, and dynamic method dispatch.
- To be factored into self-consistent libraries with minimal interdependence.

end note]

This partition provides an overview of the CLI Libraries, and a specification of their factoring into Profiles and Libraries. A companion file, considered to be part of this Partition but distributed in XML format, provides details of each type in the CLI Libraries. While the normative specification of the CLI Libraries is in XML form, it can be processed using an XSL transform to produce easily browsed information about the Class Libraries.

[*Note:* Partition VI contains an informative annex describing programming conventions used in defining the CLI Libraries. These conventions significantly simplify the use of libraries. Implementers are encouraged to follow them when creating additional (non-standard) Libraries. *end note*]

2 Libraries and Profiles

Libraries and Profiles, defined below, are constructs created for the purpose of standards conformance. They specify a set of features that shall be present in an implementation of the CLI, and a set of types that shall be available to programs run by that CLI.

[*Note:* There need not be any direct support for Libraries and Profiles in the Virtual Execution System (VES). They are not represented in the metadata and they have no impact on the structure or performance of an implementation of the CLI. Libraries and Profiles can span assemblies (the deployment unit), and the names of types in a single Library or Profile are not required to have a common prefix ("namespace"). *end note*]

In general, there is no way to test whether a feature is available at runtime, nor a way to enquire whether a particular Profile or Library is available. If present, however, the Reflection Library makes it possible to test, at runtime, for the existence of particular types and members.

2.1 Libraries

A Library specifies three things:

1. A set of types that shall be available, including their grouping into assemblies. (The standard library types are contained in three assemblies: mscorlib, System, and System.Xml. The specification for each type indicates the assembly in which it resides.)
2. A set of features of the CLI that shall be available.

   [*Note*: The set of features required for any particular Library is a subset of the complete set of CLI features. Each Library described in §5 has text that defines the CLI features that are required for implementations that support that Library. *end note*]
3. Modifications to types defined in *other* Libraries. These modifications typically involve the addition of methods and interfaces to types belonging to some other Library, and additional exception types that can be thrown by methods of that other Library's types. These modifications shall provide only additional functionality or specify behavior where it was previously unspecified; they shall not be used to alter previously specified behavior.

[*Example*: Consider the Extended Numerics Library. Since it provides a base data type, `Double`, it also specifies that the method `ToDouble` be added to the `System.Convert` class that is part of the Base Class Library. It also defines a new exception type, `System.NotFiniteNumberException`, and specifies existing methods in other Libraries methods that throw it (as it happens, there are no such methods). *end example*]

In the XML specification of the Libraries, each type specifies the Library to which it belongs. For those members (e.g., `Console.WriteLine(float)`) that are part of one Library (such as Extended Numerics (§5.7)), but whose type is in another Library (such as Base Class Library (§5.3)), the XML specifies the Library that defines the method. See §7.

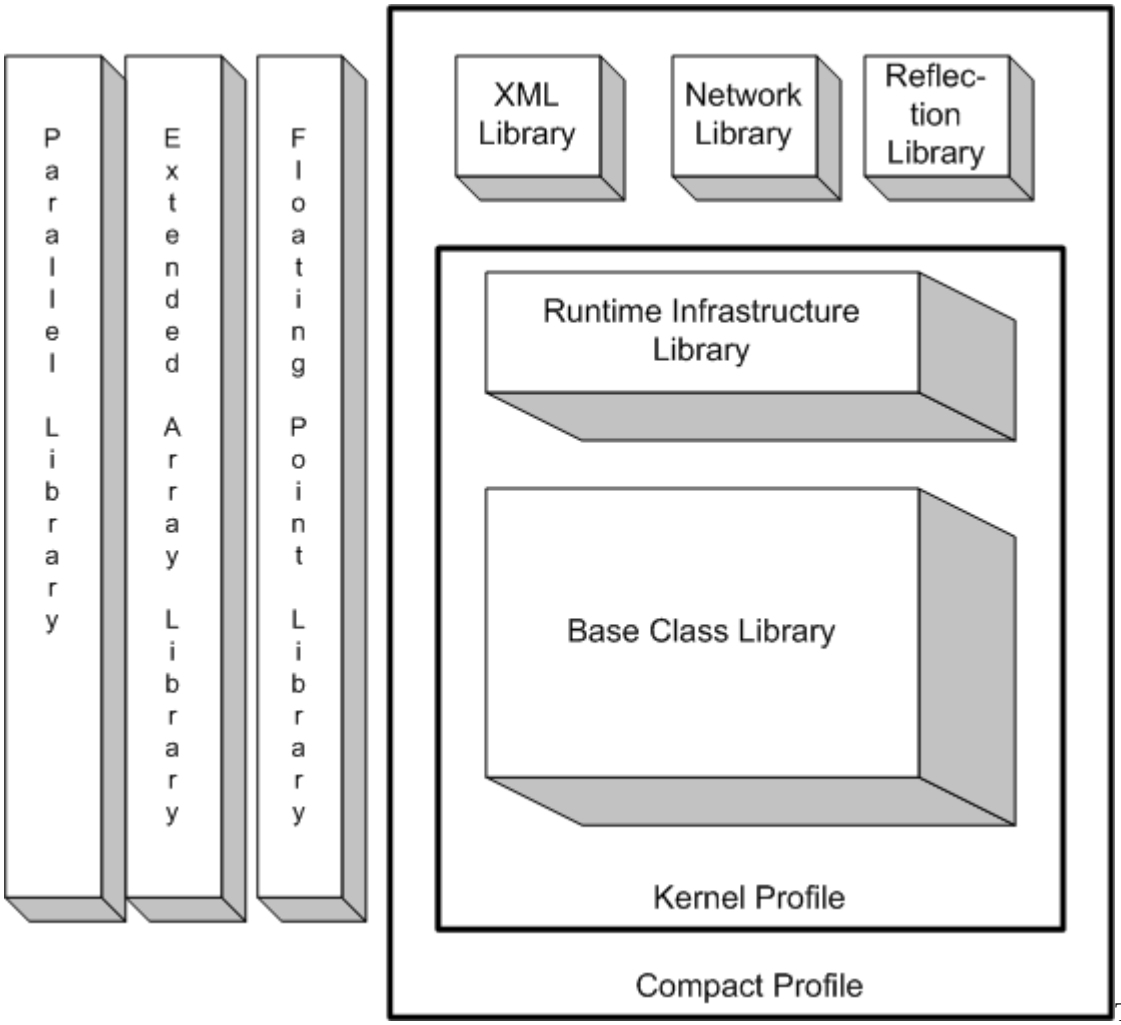
2.2 Profiles

A Profile is simply a set of Libraries, grouped together to form a consistent whole that provides a fixed level of functionality. A conforming implementation of the CLI shall specify the Profile it implements, as well as any additional Libraries that it provides. The Kernel Profile (§3.1) shall be included in all conforming implementations of the CLI. Thus, all Libraries and CLI features that are part of the Kernel Profile are available in all conforming implementations. This minimal feature set is described in §4.

[*Rationale:* The rules for combining Libraries together are complex, since each Library can add members to types defined in other libraries. By standardizing a small number of Profiles the interaction of the Libraries that are part of each Profile are specified completely. A Profile provides a consistent target for vendors of devices, compilers, tools, and applications. Each Profile specifies a trade-off of CLI feature and implementation complexity against resource constraints. By defining a very small number of Profiles, market for each Profile is increased, making each a desirable target for a class of applications across a wide range of implementations and tool sets. *end rationale*]

2.3 The relationship between Libraries and Profiles

This standard specifies two *Standard Profiles* (§3) and seven *Standard Libraries* (§5). The following diagram shows the relationship between the Libraries and the Profiles:



The Extended Array and Extended Numerics Libraries are not part of either Profile, but can be combined with either of them. Doing so adds the appropriate methods, exceptions, and interfaces to the types specified in the Profile.

3 The Standard Profiles

There are two Standard Profiles. The smallest conforming implementation of the CLI is the *Kernel Profile*, while the *Compact Profile* contains additional features useful for applications targeting a more resource-rich set of devices.

A conforming implementation of the CLI shall throw an appropriate exception (e.g., `System.NotImplementedException`, `System.MissingMethodException`, or `System.ExecutionEngineException`) when it encounters a feature specified in this Standard but not supported by the particular Profile (see Partition III).

[*Note:* Implementers should consider providing tools that statically detect features they do not support so users have an option of checking programs for the presence of such features before running them. *end note*]

[*Note:* Vendors of compliant CLI implementations should specify exactly which configurations of Standard Libraries and Standard Profiles they support. *end note*]

[*Note:* "Features" can be something like the use of a floating-point CIL instruction in the implementation of a method when the CLI upon which it is running does not support the Extended Numerics Library. Or, the "feature" might be a call to a method that this Standard specifies exists only when a particular Library is implemented and yet the code making the call is running on an implementation of the CLI that does not support that particular library. *end note*]

3.1 The Kernel Profile

This Profile is the minimal possible conforming implementation of the CLI. It contains the types commonly found in a modern programming language class library, plus the types needed by compilers targeting the CLI.

Contents: Base Class Library, Runtime Infrastructure Library

3.2 The Compact Profile

This Profile is designed to allow implementation on devices with only modest amounts of physical memory yet provides more functionality than the Kernel Profile alone. It also contains everything required to implement the proposed ECMAScript compact Profile.

Contents: Kernel Profile, XML Library, Network Library, Reflection Library

4 Kernel Profile feature requirements

All conforming implementations of the CLI support at least the Kernel Profile. Consequently, all CLI features required by the Kernel Profile shall be implemented by all conforming implementations. This clause defines that minimal feature set, by enumerating the set of features that are not required; i.e., a minimal conforming implementation shall implement all CLI features except those specified in the remainder of this clause. The feature requirements of individual Libraries as specified in §5 are defined by reference to restricted items described in this clause. For ease of reference, each feature has a name indicated by the name of the clause or subclause heading. Where Libraries do not specify any additional feature requirement, it shall be assumed that only the features of the Kernel Profile as described in this clause are required.

4.1 Features excluded from the Kernel Profile

The following internal data types and constructs, specified elsewhere in this Standard, are *not* required of CLI implementations that conform only to the Kernel Profile. All other CLI features are required.

4.1.1 Floating point

The **floating point feature set** consists of the user-visible floating-point data types `float32` and `float64`, and support for an internal representation of floating-point numbers.

If omitted: The CIL instructions that deal specifically with these data types throw the `System.NotImplementedException` exception. These instructions are: ckfinite, conv.r.un, conv.r4, conv.r8, ldc.r4, ldc.r8, ldelem.r4, ldelem.r8, ldind.r4, ldind.r8, stelem.r4, stelem.r8, stind.r4, stind.r8. Any attempt to reference a signature including the floating-point data types shall throw the `System.NotImplementedException` exception. The precise timing of the exception is not specified.

[*Note:* These restrictions guarantee that the VES will not encounter any floating-point data. Hence the implementation of the arithmetic instructions (such as add) need not handle those types. *end note*]

Part of Library: `Extended Numerics` (§5.7)

4.1.2 Non-vector arrays

The **non-vector arrays feature set** includes support for arrays with more than one dimension or with lower bounds other than zero. This includes support for signatures referencing such arrays, runtime representations of such arrays, and marshalling of such arrays to and from native data types.

If omitted: Any attempt to reference a signature including a non-vector array shall throw the `System.NotImplementedException` exception. The precise timing of the exception is not specified.

[*Note:* The type `System.Array` is part of the Kernel Profile and is available in all conforming implementations of the CLI. An implementation that does not provide the non-vector array feature set can correctly assume that all instances of that type are vectors. *end note*]

Part of Library: `Extended Arrays (see` §5.8).

4.1.3 Reflection

The **reflection feature set** supports full reflection on data types. All of its functionality is exposed through methods in the Reflection Library.

If omitted: The Kernel Profile specifies an opaque type, `System.Type`, instances of which uniquely represent any type in the system and provide access to the name of the type.

[*Note:* With just the Kernel Profile there is no requirement, for example, to determine the members of the type, dynamically create instances of the type, or invoke methods of the type given an instance of `System.Type`. This can simplify the implementation of the CLI compared to that required when the Reflection Library is available. *end note*]

Part of Library: `Reflection (see` §5.5).

4.1.4 Application domains

The **application domain feature set** supports multiple application domains. The Kernel Profile requires that a single application domain exist.

If omitted: Methods for creating application domains (part of the Base Class Library, see §5.3) throw the `System.NotImplementedException` exception.

Part of Library: (none)

4.1.5 Remoting

The **remoting feature set** supports remote method invocation. It is provided primarily through special semantics of the class `System.MarshalByRefObject` as described in Partition I.

If omitted: The class `System.MarshalByRefObject` shall be treated as a simple class with no special meaning.

Part of Library: (none)

4.1.6 Vararg

The **vararg feature set** supports variable-length argument lists and runtime-typed pointers.

If omitted: Any attempt to reference a method with the `vararg` calling convention or the signature encodings associated with vararg methods (see Partition II) shall throw the `System.NotImplementedException` exception. Methods using the CIL instructions arglist, refanytype, mkrefany, and refanyval shall throw the `System.NotImplementedException` exception. The precise timing of the exception is not specified. The type `System.TypedReference` need not be defined.

Part of Library: Vararg (see §5.9).

4.1.7 Frame growth

The **frame growth feature set** supports dynamically extending a stack frame.

If omitted: Methods using the CIL localloc instruction shall throw the `System.NotImplementedException` exception. The precise timing of the exception is not specified.

Part of Library: (none)

4.1.8 Filtered exceptions

The **filtered exceptions feature set** supports user-supplied filters for exceptions.

If omitted: Methods using the CIL endfilter instruction or with an **`exceptionentry`** that contains a non-null **`filterstart`** (see Partition I) shall throw the `System.NotImplementedException` exception. The precise timing of the exception is not specified.

Part of Library: (none)

5 The standard libraries

The detailed content of each Library, in terms of the types it provides and the changes it makes to types in other Libraries, is provided in XML form. This clause provides a brief description of each Library's purpose as well as specifying the features of the CLI required by each Library beyond those required by the Kernel Profile.

5.1 General comments

Unless stated otherwise in the documentation of a method, all copy operations are shallow, not deep.

Some methods traffic in "default values". For a reference type, the default value is null; for a nullable value type, the default value is `HasValue` returns false; for a non-nullable value type, the default value is all-bits-zero (which for Boolean represents false, and for all arithmetic types represents zero).

5.2 Runtime infrastructure library

The Runtime Infrastructure Library is part of the Kernel Profile. It provides the services needed by a compiler to target the CLI and the facilities needed to dynamically load types from a stream in the file format specified in Partition II. For example, it provides `System.BadImageFormatException`, which is thrown when a stream that does not have the correct format is loaded.

Name used in XML: RuntimeInfrastructure

CLI Feature Requirement: None

5.3 Base Class Library (BCL)

The Base Class Library is part of the Kernel Profile. It is a simple runtime library for modern programming languages. It serves as the Standard for the runtime library for the language C# as well as one of the CLI Standard Libraries. It provides types to represent the built-in data types of the CLI, simple file access, custom attributes, security attributes, string manipulation, formatting, streams, collections, among other things.

Name used in XML: BCL

CLI Feature Requirement: None

5.4 Network library

The Network Library is part of the Compact Profile. It provides simple networking services including direct access to network ports as well as HTTP support.

Name used in XML: Networking

CLI Feature Requirement: None

5.5 Reflection library

The Reflection Library is part of the Compact Profile. It provides the ability to examine the structure of types, create instances of types, and invoke methods on types, all based on a description of the type.

Name used in XML: Reflection

CLI Feature Requirement: Must support Reflection, see §5.1.

5.6 XML library

The XML Library is part of the Compact Profile. It provides a simple "pull-style" parser for XML. It is designed for resource-constrained devices, yet provides a simple user model. A conforming implementation of the CLI that includes the XML Library shall also implement the Network Library (see §5.4).

Name used in XML: XML

CLI Feature Requirement: None

5.7 Extended numerics library

The Extended Numerics Library is not part of any Profile, but can be supplied as part of any CLI implementation. It provides the support for floating-point (`System.Single`, `System.Double`) and extended-precision (`System.Decimal`) data types. Like the Base Class Library, this Library is directly referenced by the C# Standard.

[*Note:* Programmers who use this library will benefit if implementations specify which arithmetic operations on these data types are implemented primarily through hardware support. *end note*]

[*Rationale:* The Extended Numerics Library is kept separate because some commonly available processors do not provide direct support for the data types. While software emulation can be provided, the performance difference is often so large (1,000-fold or more) that it is unreasonable to build software using floating-point operations without being aware of whether the underlying implementation is hardware-based. *end rationale*]

Name used in XML: ExtendedNumerics

CLI Feature Requirement: Floating Point, see §4.1.1.

5.8 Extended array library

This Library is not part of any Profile, but can be supplied as part of any CLI implementation. It provides support for non-vector arrays. That is, arrays that have more than one dimension, or arrays that have non-zero lower bounds.

CLI Feature Requirement: Non-vector Arrays, see § 4.1.2.

5.9 Vararg library

The Vararg Library is not part of any Profile. It provides support for dealing with variable-length argument lists.

Name used in XML: Vararg

CLI Feature Requirement: None

5.10 Parallel library

This Library is not part of any Profile, but can be supplied as part of any CLI implementation. The purpose of the extended threading library is twofold:

1. Provide easy parallelism for non-expert programmers, so that multithreaded CPUs can be exploited. The Profile stresses simplicity over large scalability.
2. Not require changing the virtual machine or source languages. All features of the Profile can be implemented as a library on top of the existing CLI. The Profile can be used in conjunction with any CLI language that supports delegates.

The loop class hierarchy is summarized below:

```
ParallelLoop
   ParallelWhile
   ParallelForEach
   ParallelFor
```

The base class `ParallelLoop` factors out common functionality for parallel looping over a collection of values. The three derived classes distinguish three common kinds of parallel looping. If the collection might grow while being processed, then use `ParallelWhile`. Otherwise, if the collection implements `IEnumerable`, use `ParallelForEach`. If the collection or collections are indexible by `int32`, use `ParallelFor`.

To choose the kind of loop to use in a specific situation, consider how the loop could be written sequentially. If the loop could be written using "for (int *i*=0; *i*<*n*; ++*i*)", and *n* is known before the loop executes, use `ParallelFor`. If the loop could be written with a `foreach` statement, over collection that does not change

while the `foreach` is running, use ParallelForEach. If the loop could be written "while (collection is not yet empty) {remove item from collection and process it}", use `ParallelWhile`. When there is a choice, use `ParallelFor` if possible, because it is significantly more efficient.

Name used in XML: Parallel

CLI Feature Requirement: BCL

6 Implementation-specific modifications to the system libraries

Implementers are encouraged to extend or modify the types specified in this Standard to provide additional functionality. Implementers should notice, however, that type names beginning with "`System.`" and bearing the special Standard Public Key are intended for use by the Standard Libraries: such names not currently in use might be defined in a future version of this Standard.

To allow programs compiled against the Standard Libraries to work when run on implementations that have extended or modified the Standard Libraries, such extensions or modifications shall obey the following rules:

- The contract specified by virtual methods shall be maintained in new classes that override them.
- New exceptions can be thrown, but where possible these should be derived classes of the exceptions already specified as thrown rather than entirely new exception types. Exceptions initiated by methods of types defined in the Standard Libraries shall be derived from `System.Exception`.
- Interfaces and virtual methods shall not be added to an existing interface. Nor shall they be added to an abstract class unless that class provides an implementation.

  [*Rationale:* An interface or virtual method can be added only where it carries an implementation. This allows programs written when the interface or method was not present to continue to work. *end rationale*]
- Instance methods shall not be implemented as virtual methods.

  [*Rationale:* Methods specified as instance (non-static, non-virtual) in this standard are not permitted to be implemented as virtual methods in order to reduce the likelihood of creating non-portable files by using implementation-supplied libraries at compile time. Even though a compiler need not take a dependence on the distinction between virtual and instance methods, it is easy for a user to inadvertently override a virtual method and thus create non-portable code. The alternative of providing special files corresponding to this Standard for use at compile time is prone to user error. *end rationale*]
- The accessibility of fields and non-virtual methods can be widened from than specified in this Standard.

[*Note:* The following common extensions are permitted by these rules.

- Adding new members to existing types.
- Concrete (non-abstract) classes can implement interfaces not defined in this standard.
- Adding fields (values) to enumerations.
- An implementation can insert a new type into the hierarchy between a type specified in this standard and the type specified as its base type. That is, this standard specifies an inheritance relation between types but does not specify the immediate base type.
- Implementations can add overrides to existing virtual methods, provided the new overrides satisfy the existing contract.

end note]

[*Rationale:* An implementation might wish to split functionality across several types in order to provide non-standard extension mechanisms, or might wish to provide additional non-standard functionality through the new base type. As long as programs do not reference these non-standard types, they will remain portable across conforming implementations of the CLI. *end rationale*]

7 The XML specification

7.1 Semantics

The XML specification conforms to the Document Type Definition (DTD) in Figure 7-1. Only types that are included in a specified library are included in the XML.

There are three types of elements/attributes:

- Normative: An element or attribute is normative such that the XML specification would be incomplete without it.
- Informative: An element or attribute is informative if it specifies information that helps clarify the XML specification, but without it the specification still stands alone.
- Rendering/Formatting: An element or attribute is for rendering or formatting if it specifies information to help an XML rendering tool.

Unless explicitly stated otherwise, the text associated with an element or an attribute (e.g., #PCDATA, #CDATA) is normative or informative depending on the element or attribute with which it is associated, as described in the figure.

[*Note*: Many of the elements and attributes in the DTD are for rendering purposes. *end note*]

Figure 7-1: XML DTD

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!ELEMENT AssemblyCulture (#PCDATA)>
```

(Normative) Specifies the culture of the assembly that defines the current type. Currently this value is always “none”. It is reserved for future use.

```
<!ELEMENT AssemblyInfo (AssemblyName, AssemblyPublicKey,
AssemblyVersion, AssemblyCulture, Attributes)>
```

(Normative) Specifies information about the assembly of a given type. This information corresponds to sections of the metadata of an assembly as described in Partition II, and includes information from the AssemblyName, AssemblyPublicKey, AssemblyVersion, AssemblyCulture and Attributes elements.

```
<!ELEMENT AssemblyName (#PCDATA)>
```

(Normative) Specifies the name of the assembly to which a given type belongs. For example, all of the types in the BCL are members of the “mscorlib” assembly.

```
<!ELEMENT AssemblyPublicKey (#PCDATA)>
```

(Normative) Specifies the public key of the assembly. The public key is represented as a 128-bit value.

```
<!ELEMENT AssemblyVersion (#PCDATA)>
```

(Normative) Specifies the version of the assembly in the form 2.0.x.y, where x is a build number and y is a revision number.

```
<!ELEMENT Attribute (AttributeName, Excluded, ExcludedTypeName?,
ExcludedLibraryName?)>
```

(Normative) Specifies the text for a custom attribute on a type or a member of a type. This includes the attribute name and whether or not the attribute type itself is contained in another library.

```
<!ELEMENT AttributeName (#PCDATA)>
```

(Normative) Specifies the name of the custom attribute associated with a type or member of a type. Also contains the data needed to instantiate the attribute.

```
<!ELEMENT Attributes (Attribute*)>
```

(Normative) Specifies the list of the attributes on a given type or member of a type.

<!ELEMENT Base (BaseTypeName?, ExcludedBaseTypeName?, ExcludedLibraryName?)>

(Normative) Specifies the information related to the base type of the current type. Although the **ExcludedBaseTypeName** and **ExcludedLibraryName** elements are rarely found within this element, they are required when a type inherits from a type not found in the current library.

<!ELEMENT BaseTypeName (#PCDATA)>

(Normative) Specifies the fully qualified name of the class from which a type inherits (i.e., the type's base class).

<!ELEMENT Docs (summary?, altmember?, altcompliant?, param*, returns?, value?, exception*, threadsafe?, remarks?, example?, permission?)>

(Normative) Specifies the textual documentation of a given type or member of a type.

<!ELEMENT Excluded (#PCDATA)>

(Normative) Specifies, by a '0' or '1', whether a given member can be excluded from the current type in the absence of a given library. '0' specifies that it cannot be excluded.

<!ELEMENT ExcludedBaseTypeName (#PCDATA)>

(Normative) Specifies the fully qualified name of the type that the current type must inherit from if a given library were present in an implementation. The library name is specified in the **ExcludedLibraryName** element. An example is the System.Type class that inherits from System.Object, but if the Reflection library is present, it must inherit from System.Reflection.MemberInfo.

<!ELEMENT ExcludedLibrary (#PCDATA)>

(Normative) Specifies the library that must be present in order for a given member of a type to be required to be implemented. For example, System.Console.WriteLine(double) need only be implemented if the ExtendedNumerics library is available.

<!ELEMENT ExcludedLibraryName (#PCDATA)>

(Normative) This element appears only in the description of custom attributes. It specifies the name of the library that defines the described attribute. For example, the member that is invoked when no member name is specified for System.Text.StringBuilder (in C#, this is the indexer) is called "Chars". The attribute needed for this is System.Reflection.DefaultMemberAttribute. This is found in the RuntimeInfrastructure library. This element is used with the **ExcludedTypeName** element.

<!ELEMENT ExcludedTypeName (#PCDATA)>

(Normative) Specifies the fully qualified name of the attribute that is needed for a member to succesfully specify the given attribute. This element is related to the **ExcludedLibraryName** element and is used for attributes.

<!ELEMENT Interface (InterfaceName, Excluded)>

(Normative) Specifies information about an interface that a type implements. This element contains sub-elements specifying the interface name and whether another library is needed for the interface to be required in the current library.

<!ELEMENT InterfaceName (#PCDATA)>

(Normative) Represents the fully-qualified interface name that a type implements.

<!ELEMENT Interfaces (Interface*)>

(Normative) Specifies information on the interfaces, if any, a type implements. There is one **Interface** element for each interface implemented by the type.

<!ELEMENT Libraries (Types+)>

(Normative) This is the root element. Specifies all of the information necessary for all of the class libraries of the standard. This includes all of the types and all children elements underneath.

```
<!ELEMENT Member (MemberSignature+, MemberType, Attributes?,
ReturnValue, Parameters, MemberValue?, Docs, Excluded,
ExcludedLibrary*)>
```

(Normative) Specifies information about a member of a type. This information includes the signatures, type of the member, parameters, etc., all of which are elements in the XML specification.

```
<!ATTLIST Member
  MemberName NMTOKEN #REQUIRED
```

(Normative) **MemberName** specifies the name of the current member.

```
>
<!ELEMENT MemberOfLibrary (#PCDATA)>
```

(Normative) **PCDATA** is the name of the library containing the type.

```
<!ELEMENT MemberSignature EMPTY>
```

(Normative) Specifies the text (in source code format) for the signature of a given member of a type.

```
<!ATTLIST MemberSignature
  Language CDATA #REQUIRED
```

(Normative) **CDATA** is the programming language in which the signature is written. All members are described in both ILAsm and C#.

```
  Value CDATA #REQUIRED
```

(Normative) **CDATA** is the text of the member signature in a given language.

```
>
<!ELEMENT MemberType (#PCDATA)>
```

(Normative) Specifies the kind of the current member. The member kinds are: method, property, constructor, field, and event.

```
<!ELEMENT MemberValue (#PCDATA)>
```

(Normative) Specifies the value of a static literal field.

```
<!ELEMENT Members (Member*)>
```

(Normative) Specifies information about all of the members of a given type.

```
<!ELEMENT PRE EMPTY>
```

(Rendering/Formatting) This element exists for rendering purposes only to specify, for example, that future text should be separated from the previous text

```
<!ELEMENT Parameter (Attributes?)>
```

(Normative) Specifies the information about a specific parameter of a method or property.

```
  <!ATTLIST Parameter
        Name NMTOKEN #REQUIRED
```

(Normative) Specifies the name of the parameter.

```
        Type CDATA #REQUIRED
```

(Normative) Specifies the fully-qualified name of the type of the parameter.

```
>
<!ELEMENT Parameters (Parameter*)>
```

(Normative) Specifies information for the parameters of a given method or property. The information specified is included in each **Parameter** element of this element. This element will contain one **Parameter** for each parameter of the method or property.

```
<!ELEMENT ReturnType (#PCDATA)>
```

(Normative) Specifies the fully-qualified name of the type that the current member returns.

```
<!ELEMENT ReturnValue (ReturnType?)>
```

(Normative) Specifies the return type of a member. **ReturnType** shall be present for all kinds of members except constructors.

```
<!ELEMENT SPAN (#PCDATA | para | paramref | SPAN | see | block)*>
```

(Rendering/Formatting) This element specifies that the text should be segmented from other text (e.g., with a carriage return). References to parameters, other types, and even blocks of text can be included within a **SPAN** element.

```
<!ELEMENT ThreadingSafetyStatement (#PCDATA)>
```

(Normative) Specifies a thread safety statement for a given type.

```
<!ELEMENT Type (TypeSignature+, MemberOfLibrary, AssemblyInfo,
ThreadingSafetyStatement?, Docs, Base, Interfaces, Attributes?,
Members, TypeExcluded)>
```

(Normative) Specifies all of the information for a given type.

```
<!ATTLIST Type
```

```
  Name CDATA #REQUIRED
```

(Informative) Specifies the simple name (e.g., “String” rather than “System.String”) of a given type.

```
  FullName CDATA #REQUIRED
```

(Normative) Specifies the fully-qualified name of a given type. For generic types, this includes the spelling of generic parameter names.

```
  FullNameSP CDATA #REQUIRED
```

(Informative) Specifies the fully-qualified name with each ‘.’ of the fully qualified name replaced by an ‘_’.

```
>
```

```
<!ELEMENT TypeExcluded (#PCDATA)>
```

(Normative) **PCDATA** shall be ‘0’.

```
<!ELEMENT TypeSignature EMPTY>
```

(Normative) Specifies the text for the signature (in code representation) of a given type.

```
<!ATTLIST TypeSignature
```

```
  Language CDATA #REQUIRED
```

(Normative) Specifies the language the specified type signature is written in. All type signatures are specified in both ILAsm and C#.

```
  Value CDATA #REQUIRED
```

(Normative) **CDATA** is the type signature in the specified language.

```
>
```

```
<!ELEMENT Types (Type+)>
```

(Normative) Specifies information about all of the types of a library.

```
<!ATTLIST Types
```

Library NMTOKEN #REQUIRED

(Normative) Specifies the library in which all of the types are defined. An example of such a library is “BCL”.

>

<!ELEMENT altcompliant EMPTY>

(Informative) Specifies that an alternative, CLS compliant method call exists for the current non-CLS compliant method. For example, this element exists in the System.IO.TextWriter.WriteLine(ulong) method to show that System.IO.TextWriter.WriteLine(long) is an alternative, CLS compliant method.

<!ATTLIST altcompliant

cref CDATA #REQUIRED

(Informative) Specifies the link to the actual documentation for the alternative CLS compliant method. [**Note:** In this specification, **CDATA** matches the documentation comment format specified in Appendix E of the C# language standard.]

>

<!ELEMENT altmember EMPTY>

(Informative) Specifies that an alternative, equivalent member call exists for the current method. This element is used for operator overloads.

<!ATTLIST altmember

cref CDATA #REQUIRED

(Informative) Specifies the link to the actual documentation for the alternative member call. [**Note:** In this specification, **CDATA** matches the documentation comment format specified in Appendix E of the C# language standard.]

>

<!ELEMENT block (#PCDATA | see | para | paramref | list | block | c | subscript | code | sup | pi)*>

(Rendering/Formatting) Specifies that the children should be formatted according to the **type** specified as an attribute.

<!ATTLIST block

subset CDATA #REQUIRED

(Rendering/Formatting) This attribute is reserved for future use and currently only has the value of ‘none’.

type NMTOKEN #REQUIRED

(Rendering/Formatting) Specifies the type of block that follows, one of: usage, overrides, note, example, default, behaviors.

>

<!ELEMENT c (#PCDATA | para | paramref | code | see)*>

(Rendering/Formatting) Specifies that the text is the output of a code sample.

<!ELEMENT code (#PCDATA)>

(Informative) Specifies the text is a code sample.

<!ATTLIST code

lang CDATA #IMPLIED

(Informative) Specifies the programming language of the code sample. This specification uses C# as the language for the samples.

```
>
<!ELEMENT codelink EMPTY>
```

(Informative) Specifies a piece of code to which a link might be made from another sample. **[Note:** the XML format specified here does not provide a means of creating such a link.**]**

```
<!ATTLIST codelink
  SampleID CDATA #REQUIRED
```

(Informative) SampleID is the unique id assigned to this code sample.

```
  SnippetID CDATA #REQUIRED
```

(Informative) SnippetID is the unique id assigned to a section of text within the sample code.

```
>
<!ELEMENT description (#PCDATA | SPAN | paramref | para | see | c |
permille | block | sub)*>
```

(Normative) Specifies the text for a description for a given term element in a list or table. This element also specifies the text for a column header in a table.

```
<!ELEMENT example (#PCDATA | para | code | c | codelink | see)*>
```

(Informative) Specifies that the text will be an example on the usage of a type or a member of a given type.

```
<!ELEMENT exception (#PCDATA | paramref | see | para | SPAN | block)*>
```

(Normative) Specifies text that provides the information for an exception that shall be thrown by a member of a type, unless specified otherwise. This element can contain just text or other rendering options such as blocks, etc.

```
<!ATTLIST exception
  cref CDATA #REQUIRED
```

(Rendering/Formatting) Specifies a link to the documentation of the exception. [**Note:** In this specification, **CDATA** matches the documentation comment format specified in Appendix E of the C# language standard.]

```
>
<!ELEMENT i (#PCDATA)>
```

(Rendering/Formatting) Specifies that the text should be italicized.

```
<!ELEMENT item (term, description*)>
```

(Rendering/Formatting) Specifies a specific item of a list or a table.

```
<!ELEMENT list (listheader?, item*)>
```

(Rendering/Formatting) Specifies that the text should be displayed in a list format.

```
<!ATTLIST list
  type NMTOKEN #REQUIRED
```

(Rendering/Formatting) Specifies the type of list in which the following text will be represented. Values in the specification are: bullet, number and table.

```
>
<!ELEMENT listheader (term, description+)>
```

(Rendering/Formatting) Specifies the header of all columns in a given list or table.

```
<!ELEMENT onequarter EMPTY>
```

(Rendering/Formatting) Specifies that text, in the form of ¼, is to be displayed.

```
<!ELEMENT para (#PCDATA | see | block | paramref | c | onequarter |
superscript | sup | permille | SPAN | list | pi | theta | sub)*>
```

(Rendering/Formatting) Specifies that the text is part of what can be considered a paragraph of its own.

```
<!ELEMENT param (#PCDATA | c | paramref | see | block | para | SPAN)*>
```

(Normative) Specifies the information on the meaning or purpose of a parameter. The name of the parameter and a textual description will be associated with this element.

```
<!ATTLIST param
  name CDATA #REQUIRED
```

(Nomrative) Specifies the name of the parameter being described.

```
>
<!ELEMENT paramref EMPTY>
```

(Rendering/Formatting) Specifies a reference to a parameter of a member of a type.

```
<!ATTLIST paramref
  name CDATA #REQUIRED
```

(Rendering/Formatting) Specifies the name of the parameter to which the **paramref** element is referring.

```
>
<!ELEMENT permille EMPTY>
```

(Rendering/Formatting) Represents the current text is to be displayed as the '‰' symbol.

```
<!ELEMENT permission (#PCDATA | see | paramref | para | block)*>
```

(Normative) Specifies the permission, given as a fully-qualified type name and supportive text, needed to call a member of a type.

```
<!ATTLIST permission
  cref CDATA #REQUIRED
```

(Rendering/Formatting) Specifies a link to the documentation of the permission. [**Note:** In this specification, **CDATA** matches the documentation comment format specified in Appendix E of the C# language standard.]

```
>
<!ELEMENT pi EMPTY>
```

(Rendering/Fomatting) Represents the current text is to be displayed as the 'π' symbol

```
<!ELEMENT pre EMPTY>
```

(Rendering/Formatting) Specifies a break between the preceding and following text.

```
<!ELEMENT remarks (#PCDATA | para | block | list | c | paramref | see |
pre | SPAN | code | PRE)*>
```

(Normative) Specifies additional information, beyond that supplied by the **summary**, on a type or member of a type.

```
<!ELEMENT returns (#PCDATA | para | list | paramref | see)*>
```

(Normative) Specifies text that describes the return value of a given type member.

```
<!ELEMENT see EMPTY>
```

(Informative) Specifies a link to another type or member.

```
<!ATTLIST see
```

```
  cref CDATA #IMPLIED
```

(Informative) **cref** specifies the fully-qualified name of the type or member to link to. [**Note:** In this specification, **CDATA** matches the documentation comment format specified in Appendix E of the C# language standard.]

```
  langword CDATA #IMPLIED
```

(Informative) **langword** specifies that the link is to a language agnostic keyword such as "null".

```
  qualify CDATA #IMPLIED
```

(Informative) Qualify indicates that the type or member specified in the link must be displayed as fully-qualified. Value of this attribute is 'true' or 'false', with a default value of 'false'

```
>
```

```
<!ELEMENT sub (#PCDATA | paramref)*>
```

(Rendering/Formatting) Specifies that current piece of text is to be displayed in subscript notation.

```
<!ELEMENT subscript EMPTY>
```

(Rendering/Formatting) Specifies that current piece of text is to be displayed in subscript notation.

```
<!ATTLIST subscript
```

```
  term CDATA #REQUIRED
```

(Rendering/Formatting) Specifies the value to be rendered as a subscript.

```
>
```

```
<!ELEMENT summary (#PCDATA | para | see | block | list)*>
```

(Normative) Specifies a summary description of a given type or member of a type.

```
<!ELEMENT sup (#PCDATA | i | paramref)*>
```

(Rendering/Formatting) Specifies that the current piece of text is to be displayed in superscript notation.

```
<!ELEMENT superscript EMPTY>
```

(Rendering/Formatting) Specifies that current piece of text is to be displayed in superscript notation.

```
<!ATTLIST superscript
```

```
  term CDATA #REQUIRED
```

(Rendering/Formatting) Specifies the value to be rendered as a superscript.

```
>
```

```
<!ELEMENT term (#PCDATA | block | see | paramref | para | c | sup | pi
| theta)*>
```

(Rendering/Formatting) Specifies the text is a list item or an item in the primary column of a table.

```
<!ELEMENT theta EMPTY>
```

(Rendering/Formatting) Specifies that text, in the form of 'θ', is to be displayed.

```
<!ELEMENT threadsafe (para+)>
```

(Normative) Specifies that the text describes additional detail, beyond that specified by **ThreadingSafetyStatement**, the thread safety implications of the current type. For example, the text will describe what an implementation must do in terms of synchronization.

```
<!ELEMENT value (#PCDATA | para | list | see)*>
```

(Normative) Specifies description information on the “value” passed into the set method of a property.

7.1.1 Value types as objects

Throughout the textual descriptions of methods in the XML, there are places where a parameter of type `object` or an interface type is expected, but the description refers to passing a value type for that parameter. In these cases, the caller shall box the value type before making the call.

7.1.2 Exceptions

Many members of types defined in the XML have associated exception conditions. Unless it is stated otherwise in a member’s definition, the exceptions listed for any given member shall be thrown when the stated conditions occur.

7.2 XML signature notation issues

For each type and member described in the XML, there is an ILAsm and C# signature pair. These are intended to be equivalent and to provide sufficient information to allow these types and members to be implemented correctly. Each signature pair shows both the low-level and one high-level view of these signatures. However, as written in the XML, the members of a given pair of signatures are not always written in an equivalent manner, even though they are intended to produce identical behavior. The differences in signature notation are described in this subclause.

7.2.1 Serialization

As shown in the ILAsm signatures, many of the types in the standard library have the predefined attribute `serializable` attached. A type that is marked with this attribute is to be serialized as part of the persistent state of a value of the type. This standard does not require that a conforming implementation provide support for serialization (or its counterpart, deserialization), nor does it specify the mechanism by which these operations might be accomplished.

Consider the ILAsm and C# signatures in the XML for `System.String`:

```
[ILAsm]
.class public sealed serializable String …

[C#]
public sealed class String …
```

Although the C# standard does not address the issue of serialization, if this library type is written in C#, when the C# declaration above is compiled, the intent is that the code generated for the class contains the `serializable` attribute as shown. [*Note*: Some implementations provide an attribute type, `System.SerializableAttribute`, for this purpose. *end note*]

7.2.2 Delegates

The standard library contains a number of delegate types. However, as recorded in the XML, their ILAsm signatures are incomplete. Consider `System.EventHandler` as an example; its ILAsm signature is defined in the XML as follows:

```
.class public sealed serializable EventHandler extends System.Delegate {
.method public hidebysig newslot virtual instance void Invoke(object
   sender, class System.EventArgs e) }
```

However, this type also has a constructor and two optional asynchronous methods, `BeginInvoke` and `EndInvoke`, all of which are described in Partition II, “Delegates”. The signatures for these three members for `System.EventHandler` are as follows:

```
.method public hidebysig specialname rtspecialname void .ctor(object
   'object', native int 'method')
.method public hidebysig newslot virtual class System.IAsyncResult
```

```
    BeginInvoke(object sender, class System.EventArgs e, class
    System.AsyncCallback callback, object 'object')
  .method public hidebysig newslot virtual void EndInvoke(
    class System.IAsyncResult result)
```

The other standard delegate types have a corresponding constructor and method pair whose signatures can be deduced from the ILAsm in the XML and the information in Partition II, "Delegates".

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

7.2.3 Properties

The standard library contains many types that have properties. However, as recorded in the XML, their ILAsm signatures are incomplete. Consider the read-write instance property `System.Collections.ArrayList.Capacity`. Its ILAsm signature is defined in the XML as follows:

```
.property int32 Capacity {
  public hidebysig virtual specialname int32 get_Capacity()
  public hidebysig virtual specialname void set_Capacity(int32 value)
}
```

However, this is an abbreviation of the ILAsm syntax. The complete (and correct) signature for this property is as follows:

```
.property instance int32 Capacity() {
  .get instance int32 ArrayList::get_Capacity()
  .set instance void ArrayList::set_Capacity(int32)
}
.method public hidebysig newslot specialname virtual instance int32
   get_Capacity() { … }
.method public hidebysig newslot specialname virtual instance void
   set_Capacity(int32 'value') { … }
```

As a second example, consider the readonly static property `System.DateTime.Now`; its ILAsm signature is defined in the XML as follows:

```
.property valuetype System.DateTime Now {
  public hidebysig static specialname valuetype System.DateTime
    get_Now()
}
```

However, the complete (and correct) signature for this property is:

```
.property valuetype System.DateTime Now() {
  .get valuetype System.DateTime DateTime::get_Now()
}
.method public hidebysig specialname static valuetype System.DateTime
   get_Now() { ... }
```

All other properties (including those that are indexed) are formatted in the XML in a similar abbreviated manner.

7.2.4 Nested types

With one exception, the definitions of all members of any given type are contained in the XML for that type. The exception is for nested types. Nested types have their own definition in the XML, where their names are qualified by the name of the type in which they are nested. [*Example*: The type `System.Collections.Generic.List<T>` contains the nested type `Enumerator`. These types are described in the BCL library of the XML under the names `List<T>` and `List<T>.Enumerator`, respectively. *end example*]