Standard ECMA-335
2nd edition - December 2002

ECMA International

Standardizing Information and Communication Systems

Common Language Infrastructure (CLI) Partitions I to V

Standard ECMA-335
2nd edition - December 2002

ECMA International

Standardizing Information and Communication Systems

Common Language Infrastructure (CLI) Partitions I to V

Partition I: Concepts and Architecture

Partition II: Metadata Definition and Semantics

Partition III: CIL Instruction Set

Partition IV: Profiles and Libraries

Partition V: Annexes

Phone: +41 22 849.60.00 - **Fax:** +41 22 849.60.01 - **URL:** http://www.ecma.ch - **Internet:** helpdesk@ecma.ch
MBo ECMA-335.doc 15-01-03 11,25

Common Language Infrastructure (CLI)

Partition I:
Concepts and Architecture

Table of Contents

1 Scope

This International Standard defines the Common Language Infrastructure (CLI) in which applications written in multiple high-level languages may be executed in different system environments without the need to rewrite the applications to take into consideration the unique characteristics of those environments. This International Standard consists of the following parts:

- Partition I: Concepts and Architecture – Describes the overall architecture of the CLI, and provides the normative description of the Common Type System (CTS), the Virtual Execution System (VES), and the Common Language Specification (CLS). It also provides a non-normative description of the metadata and a comprehensive set of abbreviations, acronyms and definitions, which are included by reference into all other Partitions.
- Partition II: Metadata Definition and Semantics – 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).
- Partition III: CIL Instruction Set – Describes, in detail, the Common Intermediate Language (CIL) instruction set.
- Partition IV: Profiles and Libraries – Provides an overview of the CLI Libraries and a specification of their factoring into Profiles and Libraries. A companion document, considered to be part of this Partition but distributed in XML format, provides details of each class, value type, and interface in the CLI Libraries.
- Partition V: Annexes – Contains some sample programs written in CIL Assembly Language (ILAsm), information about a particular implementation of an assembler, a machine-readable description of the CIL instruction set which may be used to derive parts of the grammar used by this assembler as well as other tools that manipulate CIL, and a set of guidelines used in the design of the libraries of Partition IV.

2 Conformance

A system claiming conformance to this International Standard shall implement all the mandatory requirements of this standard, and shall specify the profile (see Partition IV) that it implements. The minimal implementation is the Kernel Profile (see Partition IV). A conforming implementation may also include additional functionality that does not prevent running code written to rely solely on the profile as specified in this standard. For example, it may provide additional classes, new methods on existing classes, or a new interface on a standardized class, but it shall not add methods or properties to interfaces specified in this standard.

A compiler that generates Common Intermediate Language (CIL, see Partition III) and claims conformance to this International Standard shall produce output files in the format specified in this standard and the CIL it generates shall be valid CIL as specified in this standard. Such a compiler may also claim that it generates *verifiable* code, in which case the CIL it generates shall be verifiable as specified in this standard.

3 References

The following normative documents contain provisions, which, through reference in this text, constitute provisions of this International Standard. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

[Note that many of these references are cited in the XML description of the class libraries.]

Extensible Markup Language (XML) 1.0 (Second Edition), 2000 October 6, http://www.w3.org/TR/2000/REC-xml-20001006

Federal Information Processing Standard (FIPS 180-1*), Secure Hash Standard (SHA-1),* 1995, April.

IEC 60559:1989, *Binary Floating-point Arithmetic for Microprocessor Systems* (previously designated IEC 559:1989).

ISO 639:1988, *Codes for the representation of names of languages*.

ISO 3166:1988, *Codes for the representation of names of countries*.

ISO/IEC 646:1991, *ISO 7-bit coded character set for information interchange*

ISO/IEC 9899:1990, *Programming languages — C.*

ISO/IEC 10646 (all parts), *Universal Multiple-Octet Coded Character Set (UCS).*

ISO/IEC 11578:1996 (E) *Open Systems Interconnection - Remote Procedure Call (RPC), Annex A: Universal Unique Identifier.*

ISO/IEC 14882:1998, *Programming languages — C++.*

ISO/IEC 23270:2002, *Programming languages — C#.*

RFC-768, *User Datagram Protocol*. J. Postel. 1980, August.

RFC-791, *Darpa Internet Program Protocol Specification*. 1981, September.

RFC-792, *Internet Control Message Protocol*. Network Working Group. J. Postel. 1981, September.

RFC-793, *Transmission Control Protocol*. J. Postel. 1981, September.

RFC-919, *Broadcasting Internet Datagrams*. Network Working Group. J. Mogul. 1984, October.

RFC-922, *Broadcasting Internet Datagrams in the presence of Subnets*. Network Working Group. J. Mogul. 1984, October.

RFC-1035, *Domain Names - Implementation and Specification*. Network Working Group. P. Mockapetris. 1987, November.

RFC-1036, *Standard for Interchange of USENET Messages*, Network Working Group. M. Horton and R. Adams. 1987, December.

RFC-1112. *Host Extensions for IP Multicasting*. Network Working Group. S. Deering 1989, August.

RFC-1222. *Advancing the NSFNET Routing Architecture*. Network Working Group. H-W Braun, Y. Rekhter. 1991 May. ftp://ftp.isi.edu/in-notes/rfc1222.txt

RFC-1510, *The Kerberos Network Authentication Service (V5)*. Network Working Group. J. Kohl and C. Neuman. 1993, September.

RFC-1741, *MIME Content Type for BinHex Encoded Files: Format*. Network Working Group. P. Faltstrom, D. Crocker, and E. Fair. 1994, December.

RFC-1764. *The PPP XNS IDP Control Protocol (XNSCP).* Network Working Group. S. Senum. 1995, March.

RFC-1766, *Tags for the Identification of Languages*. Network Working Group. H. Alvestrand. 1995, March.

RFC-1792. *TCP/IPX Connection Mib Specification*. Network Working Group. T. Sung. 1995, April.

RFC-2236. *Internet Group Management Protocol, Version 2*. Network Working Group. W. Fenner. 1997, November.

RFC-2045, *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. Network Working Group. N. Freed. 1996, November.

RFC-2068, *Hypertext Transfer Protocol* -- *HTTP/1.1*, Network Working Group. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. 1997, January.

RFC-2396. *Uniform Resource Identifiers (URI): Generic Syntax*. Internet Engineering Task Force. T. Berners-Lee, R. Fielding, and L. Masinter. 1998 August. http://www.ietf.org/rfc/rfc2396.txt.

RFC-2616, *Hypertext Transfer Protocol* -- *HTTP/1.1*. Network Working Group. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. 1999 June. ftp://ftp.isi.edu/in-notes/rfc2616.txt

RFC-2617, *HTTP Authentication: Basic and Digest Access Authentication.* Network Working Group. J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. 1999 June, ftp://ftp.isi.edu/in-notes/rfc2617.txt

The Unicode Consortium. The Unicode Standard, Version 3.0, defined by: *The Unicode Standard*, *Version 3.0* (Reading, MA, Addison-Wesley, 2000. ISBN 0-201-61633-5), and Unicode Technical Report #15: *Unicode Normalization Forms*.

4 Conventions

The remainder of this section contains only informative text

4.1 Organization

The divisions of this International Standard are organized using a hierarchy. At the top level is the *Partition*. The next level is the *Chapter*, followed by *section* then *clause*. Divisions within a clause are also referred to as clauses rather than subclauses. Partitions are numbered using Roman numerals. All other divisions are numbered using Arabic digits with their place in the hierarchy indicated by nested numbers. For example, Partition II, 14.4.3.2 refers to clause 2 in clause 3 in section 4 in Chapter 14 in Partition II.

4.2 Informative Text

This International Standard is intended to be used by implementers, academics, and application programmers. As such, it contains explanatory material that, strictly speaking, is not necessary in a formal specification.

Examples are provided to illustrate possible forms of the constructions described. References are used to refer to related sections. Notes are provided to give advice or guidance to implementers or programmers. Annexes provide additional information.

Except for whole sections or clauses that are identified as being informative, informative text that is contained within normative sections and clauses is identified as follows:

The beginning and end of informative text is marked as shown in this chapter, using rectangular boxes.

As some informative passages span pages, informative text also contains a bold set of vertical black stripes in the right margin.

4.3 Hyperlinks

The text of this standard contains numerous hyperlinks that are intended to allow a person viewing it electronically to be able to transfer to the referenced sections. Such hyperlinks are displayed with a double underline, and, depending on the rendered format, may be colored; for example: “see Partition IV”, “see Section 8.7”, “see Chapter 22”, and “see clause 7.5.3”. Some tables also contain hyperlinked section number references, such as 5.10 and 6.7.

End informative text

5 Glossary

For the purpose of this International Standard, the following definitions apply. They are collected here for ease of reference, but the definition is presented in context elsewhere in the specification, as noted. Definitions enclosed in square brackets [ ] were not extracted from the body of the standard.

The remainder of this section and its subsections contain only informative text

Term	**Description**	**Pt**	**Ch**	**Section**
Abstract	Only an abstract object type is allowed to define method contracts for which the type or the VES does not also provide the implementation. Such method contracts are called abstract methods	I	8.9.6.2	Concreteness
Accessibility of members	A type scopes all of its members, and it also specifies the accessibility rules for its members. Except where noted, accessibility is decided based only on the statically visible type of the member being referenced and the type and assembly that is making the reference. The CTS supports seven different rules for accessibility: Compiler-Controlled; Private; Family; Assembly; Family-and-Assembly; Family-or-Assembly; Public.	I	8.5.3.2	Accessibility of Members
Aggregate data	Data items that have sub-components (arrays, structures, or object instances) but are passed by copying the value. The sub-components can include references to managed memory. Aggregate data is represented using a *value type…*	I	12.1.6	Aggregate Data
Application domain	A mechanism … to isolate applications running in the same operating system process from one another.	I	12.5	Proxies and Remoting
Array elements	The representation of a value (except for those of built-in types) can be subdivided into sub-values. These sub-values are either named, in which case they are called **fields**, or they are accessed by an indexing expression, in which case they are called **array elements**.	I	8.4.1	Fields, Array Elements, and Values
Argument	[Value of an operand to a method call]			
Array types	Types that describe values composed of array elements are **array types**.	I	8.4.1	Fields, Array Elements, and Values
Assembly	An assembly is a configured set of loadable code modules and other resources that together implement a unit of functionality.	I	8.5.2	Assemblies and Scoping
Assembly scope	Type names are scoped by the **assembly** that contains the implementation of the type….. The type name is said to be in the **assembly scope** of the assembly that implements the type.	I	8.5.2	Assemblies and Scoping
Assignment compatibility	Assignment compatibility of a value (described by a type signature) to a location (described by a location signature) is defined as follows: One of the types supported by the exact type of the value is the same as the type in the location signature.	I	8.7	Assignment Compatibility
Attributes	*Attributes* of types and their members attach descriptive	II	5.9	Attributes and

	information to their definition.			Metadata
Base Class Library	This Library is part of the Kernel Profile. It is a simple runtime library for a modern programming language.	IV	5.1	Runtime Infrastructure Library
Binary operators	Binary operators take two arguments, perform some operation and return a value. They are represented as static methods on the class that defines the type of one of their two operands or the return type.	I	10.3.2	Binary Operators
Boolean Data Type	A CLI Boolean type occupies one byte in memory. A bit pattern of all zeroes denotes a value of false. A bit pattern with any bit set (analogous to a non-zero integer) denotes a value of true.	III	1.1.2	Boolean Data Type
Box	The **box** instruction is a widening (always typesafe) operation that converts a value type instance to **System.Object** by making a copy of the instance and embedding it in a newly allocated object.	I	12.1.6.2.5	Boxing and Unboxing
Boxed type	For every Value Type, the CTS defines a corresponding Reference Type called the **boxed type**.	I	8.2.4	Boxing and Unboxing of Values
Boxed value	The representation of a value of a boxed type (a **boxed value**) is a location where a value of the Value Type may be stored.	I	8.2.4	Boxing and Unboxing of Values
Built-in types	..Data types [that] are an integral part of the CTS and are supported directly by the Virtual Execution System (VES).	I	8.2.2	Built-In Types
By-ref parameters	The **address** of the data is passed from the caller to the callee, and the type of the parameter is therefore a managed or unmanaged pointer.	I	12.4.1.5	Parameter Passing
By-value parameters	The **value** of an object is passed from the caller to the callee	I	12.4.1.5	Parameter Passing
Calling Convention	A calling convention specifies how a method expects its arguments to be passed from the caller to the called method.	II	14.3	Calling Convention
Casting	Since a value can be of more than one type, a use of the value needs to clearly identify which of its types is being used. Since values are read from locations that are typed, the type of the value which is used is the type of the location from which the value was read. If a different type is to be used, the value is **cast** to one of its other types. .	I	8.3.3	Casting
CIL	[Common Intermediate Language]			
Class contract	A class contract specifies the representation of the values of the class type. Additionally, a class contract specifies the other contracts that the class type supports, e.g., which interfaces, methods, properties and events shall be implemented.	I	8.6	Contracts
Class type	A complete specification of the representation of the values of the class type and all of the contracts (class,	I	8.9.5	Class Type Definition

	interface, method, property, and event) that are supported by the class type.			
CLI	At the center of the Common Language Infrastructure (CLI) is a single type system, the Common Type System (CTS), that is shared by compilers, tools, and the CLI itself. It is the model that defines the rules the CLI follows when declaring, using, and managing types.	I	6	Overview of the Common Language Infrastructure
CLS	The Common Language Specification (CLS) is a set of conventions intended to promote language interoperability.	I	7	Common Language Specification (CLS)
CLS (consumer)	A CLS consumer is a language or tool that is designed to allow access to all of the features supplied by CLS-compliant frameworks (libraries), but not necessarily be able to produce them.	I	7	Common Language Specification (CLS)
CLS (extender)	A CLS extender is a language or tool that is designed to allow programmers to both use and extend CLS-compliant frameworks.	I	7	Common Language Specification (CLS)
CLS (framework)	A library consisting of CLS-compliant code is herein referred to as a “framework”.	I	7	Common Language Specification (CLS)
Code labels	Code labels are followed by a colon (“:”) and represent the address of an instruction to be executed	II	5.4	Labels and Lists of Labels
Coercion	Coercion takes a value of a particular type and a desired type and attempts to create a value of the desired type that has equivalent meaning to the original value.	I	8.3.2	Coercion
Common Language Specification (CLS)	The Common Language Specification (CLS) is a set of conventions intended to promote language interoperability.	I	7	Common Language Specification (CLS)
Common Type System (CTS)	The Common Type System (CTS) provides a rich type system that supports the types and operations found in many programming languages.	I	6	Overview of the Common Language Infrastructure
Compiler-controlled accessibility	Accessible only through use of a definition, not a reference, hence only accessible from within a single compilation unit and under the control of the compiler.	I	8.5.3.2	Accessibility of Members
Compound types	Types that describe values composed of fields are **compound types**.	I	8.4.1	Fields, Array Elements, and Values
Computed destinations	The destination of a method call may be either encoded directly in the CIL instruction stream (the **call** and **jmp** instructions) or computed (the **callvirt**, and **calli** instructions).	I	12.4.1.3	Computed Destinations
Concrete	An object type that is not marked **abstract** is by definition **concrete**.	I	8.9.6.2	Concreteness
Conformanc	A system claiming conformance to this International	I	2	Conformance

e	Standard shall implement all the mandatory requirements of this standard, and shall specify the profile that it implements.			
Contracts	**Contracts** are named. They are the shared assumptions on a set of **signatures** … between all implementers and all users of the contract.	I	8.6	Contracts
Conversion operators	Conversion operators are unary operations that allow conversion from one type to another. The operator method shall be defined as a static method on either the operand or return type.	I	10.3.3	Conversion Operators
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.	II	20	Custom Attributes
Custom modifiers	Custom modifiers, defined using modreq (“required modifier”) and modopt (“optional modifier”)**,** are similar to custom attributes …except that modifiers are part of a signature rather than attached to a declaration. Each modifer associates a type reference with an item in the signature.	II	7.1.1	modreq and modopt
Data labels	Data labels specify the location of a piece of data	II	5.4	Labels and Lists of Labels
Delegates	**Delegates** are the object-oriented equivalent of function pointers. . Delegates are created by defining a class that derives from the base type **System.Delegate**	I	8.9.3	Delegates
Derived Type	A derived type guarantees support for all of the type contracts of its base type. A type derives directly from its specified base type(s), and indirectly from their base type(s).	I	8.9.8	Type Inheritance
Enums	An *enum*, short for *enumeration*, defines a set of symbols that all have the same type.	II	13.3	Enums
Equality	For value types, the equality operator is part of the definition of the exact type. Definitions of equality should obey the following rules: • Equality should be an equivalence operator, as defined above. • Identity should imply equality, as stated earlier. • If either (or both) operand is a boxed value, equality should be computed by • first unboxing any boxed operand(s), and then • applying the usual rules for equality on the resulting values.	I	8.2.5.2	Equality
Equality of values	The values stored in the variables are **equal** if the sequences of characters are the same.	I	8.2.5	Identity and Equality of Values
Evaluation	Associated with each method state is an evaluation stack… The evaluation stack is made up of slots that can	I	12.3.2.1	The Evaluation

stack	hold any data type, including an unboxed instance of a value type.			Stack
Event contract	An event contract is specified with an event definition. There is an extensible set of operations for managing a named event, which includes three standard methods (register interest in an event, revoke interest in an event, fire the event). An event contract specifies method contracts for all of the operations that shall be implemented by any type that supports the event contract.	I	8.6	Contracts
Event definitions	The CTS supports events in precisely the same way that it supports properties… The conventional methods, however, are different and include means for subscribing and unsubscribing to events as well as for firing the event.	I	8.11.4	Event Definitions
Exception handling	Exception handling is supported in the CLI through exception objects and protected blocks of code	I	12.4.2	Exception Handling
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.	IV	5.7	Extended Array Library
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.Float, System.Double) and extended-precision (System.Decimal) data types.	IV	5.6	Extended Numerics Library
Family accessibility	accessible to referents that support the same type, i.e. an exact type and all of the types that inherit from it	I	8.5.3.2	Accessibility of Members
Family-and-assembly accessibilty	Accessible only to referents that qualify for both Family and Assembly access.	I	8.5.3.2	Accessibility of Members
Family-or-assembly accessibility	accessible only to referents that qualify for either Family or Assembly access.	I	8.5.3.2	Accessibility of Members
Field definitions	Field definitions name and a location signature.	I	8.11.2	Field Definitions
Field inheritance	A derived object type inherits all of the non-static fields of its base object type.	I	8.10.1	Field Inheritance
Fields	Fields are typed memory locations that store the data of a program.	II	15	Defining and Referencing Fields
File Names	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	II	5.8	File Names
Finalizers	A class definition that creates an object type may supply an instance method to be called when an instance of the class is no longer accessible.	I	8.9.6.7	Finalizers
Getter method	By convention, properties define a **getter** method (for accessing the current value of the property)…	I	8.11.3	Property Definitions

Global Fields	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.	II	9.8	Global Fields and Methods
Global 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.	II	9.8	Global Fields and Methods
Global state	The CLI manages multiple concurrent threads of control … multiple managed heaps, and a shared memory address space.	I	12.3.1	The Global State
GUID	[A unique identification string used with remote procedure calls.]			
hide-by-name	The introduction of a name in a given type hides all inherited members of the same kind (method or field) with the same name.	II	8.3	Hiding
hide-by-name-and-sig	The introduction of a name in a given type hides any inherited member of the same kind but with precisely the same type (for fields) or signature (for methods, properties, and events).	II	8.3	Hiding
Hiding	Hiding controls which method names inherited from a base type are available for compile-time name binding.	II	8	Visibility, Accessibility and Hiding
Homes	The **home** of a data value is where it is stored for possible reuse	I	12.1.6.1	Homes for Values
Identifiers	Identifiers are used to name entities	II	5.3	Identifiers
Identity	The identity operator is defined by the CTS as follows. • If the values have different exact types, then they are not identical. • Otherwise, if their exact type is a Value Type, then they are identical if and only if the bit sequences of the values are the same, bit by bit. Otherwise, if their exact type is a Reference Type, then they are identical if and only if the locations of the values are the same.	I	8.2.5.1	Identity
Identity of values	The values of the variables are **identical** if the locations of the sequences of characters are the same, i.e., there is in fact only one string in memory.	I	8.2.5	Identity and Equality of Values
Ilasm	An assembler language for CIL	II	2	Overview
Inheritance demand	When attached to a type ..[an inheritance demand] requires that any type that wishes to inherit from this type shall have the specified security permission. When attached to a non-final virtual method it requires that any type that wishes to override this method shall have the specified permission.	I	8.5.3.3	Security Permissions
Instance	Instance methods are associated with an instance of a type: within the body of an instance method it is possible	II	14.2	Static, Instance, and Virtual

Methods	to reference the particular instance on which the method is operating (via the *this pointer*).			Methods
Instruction pointer (IP)	An instruction pointer (**IP**) points to the next CIL instruction to be executed by the CLI in the present method.	I	12.3.2	Method State
Interface contract	Interface contracts specify which other contracts the interface supports, e.g. which interfaces, methods, properties and events shall be implemented.	I	8.6	Contracts
Interface type definition	An **interface definition** defines an interface type. An interface type is a named group of methods, locations and other contracts that shall be implemented by any object type that supports the interface contract of the same name.	I	8.9.4	Interface Type Definition
Interface type inheritance	Interface types may inherit from multiple interface types, i.e. an interface contract may list other interface contracts that shall also be supported.	I	8.9.11	Interface Type Inheritance
Interface types	Interface types describe a subset of the operations and none of the representation, and hence, cannot be an exact type of any value.	I	8.2.3	Classes, Interfaces and Objects
Interfaces	Interfaces…define a contract that other types may implement.	II	11	Semantics of Interfaces
Kernel Profile	This profile is the minimal possible conforming implementation of the CLI.	IV	3.1	The Kernel Profile
Labels	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.	II	5.4	Labels and Lists of Labels
Libraries	To a programmer a Library is a self-consistent set of types (classes, interfaces, and value types) that provide a useful set of functionality.	IV	2.1	Libraries
Local memory pool	The local memory pool is used to allocate objects whose type or size is not known at compile time and which the programmer does not wish to allocate in the managed heap.	I	12.3.2.4	Local Memory Pool
Local signatures	. A **local signature** specifies the contract on a local variable allocated during the running of a method.	I	8.6.1.3	Local Signatures
Location signatures	All locations are typed. This means that all locations have a **location signature**, which defines constraints on the location, its usage, and on the usage of the values stored in the location.	I	8.6.1.2	Location Signatures
Locations	Values are stored in **locations**. A location can hold a single value at a time. All locations are typed. The type of the location embodies the requirements that shall be met by values that are stored in the location.	I	8.3	Locations
Machine state	One of the design goals of the CLI is to hide the details of a method call frame from the CIL code generator.	I	12.3	Machine State

	The machine state definitions … reflect these design choices, where machine state consists primarily of global state and method state.			
Managed code	Managed code is simply code that provides enough information to allow the CLI to provide a set of core services, including • Given an address inside the code for a method, locate the metadata describing the method • Walk the stack • Handle exceptions • Store and retrieve security information	I	6.2.1	Managed Code
Managed data	**Managed data** is data that is allocated and released automatically by the CLI, through a process called **garbage collection**. Only managed code can access managed data, but programs that are written in managed code can access both managed and unmanaged data.	I	6.2.2	Managed Data
Managed pointer types	[The O and &] datatype represents an object reference that is managed by the CLI	I	12.1.1.2	Managed Pointer Types: O and &
Managed Pointers	Managed pointers (&) may point to 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).	II	13.4.2	Managed Pointers
Manifest	An *assembly* is a set of one or more files deployed as a unit.	II	6	Assemblies, Manifests and Modules
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 coded via PInvoke dispatch or IJW ("It Just Works") thunking.	II	22.4	Marshalling Descriptors
Member	Fields, array elements, and methods are called **members** of the type. Properties and events are also members of the type.	I	8.4	Type Members
Member inheritance	Only object types may inherit implementations, hence only object types may inherit members	I	8.10	Member Inheritance
Memory store	By "memory store" we mean the regular process memory that the CLI operates within. Conceptually, this store is simply an array of bytes.	I	12.6.1	The Memory Store
Metadata	The CLI uses metadata to describe and reference the types defined by the Common Type System. Metadata is stored ("persisted") in a way that is independent of any particular programming language. Thus, metadata provides a common interchange mechanism for use	I	6	Overview of the Common Language Infrastructure

	between tools that manipulate programs (compilers, debuggers, etc.) as well as between these tools and the Virtual Execution System			
Metadata Token	This is a 4-byte value, that specifies a row in a metadata table, or a starting byte offset in the User String heap	III	1.9	Metadata Tokens
Method	A named **method** describes an operation that may be performed on values of an exact type.	I	8.2.3	Classes, Interfaces and Objects
Method contract	A method contract is specified with a method definition. A method contract is a named operation that specifies the contract between the implementation(s) of the method and the callers of the method.	I	8.6	Contracts
Method definitions	Method definitions are composed of a name, a method signature, and optionally an implementation of the method.	I	8.11.1	Method Definitions
Method inheritance	A derived object type inherits all of the instance and virtual methods of its base object type. It does not inherit constructors or static methods.	I	8.10.2	Method Inheritance
Method Pointers	Variables of type method pointer shall store the address of the entry point to a method with compatible signature.	II	13.5	Method Pointers
Method signatures	**Method signatures** are composed of • a calling convention, • a list of zero or more parameter signatures, one for each parameter of the method, • and a type signature for the result value if one is produced.	I	8.6.1.5	Method Signatures
Method state	Method state describes the environment within which a method executes. (In conventional compiler terminology, it corresponds to a superset of the information captured in the “invocation stack frame”).	I	12.3.2	Method State
methodInfo handle	This .. holds the signature of the method, the types of its local variables, and data about its exception handlers.	I	12.3.2	Method State
Module	A single file containing executable content	II	6	Assemblies, Manifests and Modules
Name Mangling	… the platform may 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).	II	14.5.2	Platform Invoke
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	II	7.4	Native Data Types

	data types.			
Native size types	The native-size, or generic, types (I, U, O, and &) are a mechanism in the CLI for deferring the choice of a value's size.	I	12.1.1	Native Size: native int, native unsigned int, O and &
Nested type definitions	A nested type definition is identical to a top-level type definition, with one exception: a top-level type has a visibility attribute, while the visibility of a nested type is the same as the visibility of the enclosing type.	I	8.11.5	Nested Type Definitions
Nested types	A type (called a nested type) can be a member of an enclosing type.	I	8.5.3.4	Nested Types
Network Library	This Library is part of the Compact Profile. It provides simple networking services including direct access to network ports as well as HTTP support.	IV	5.3	Network Library
OOP	[Object Oriented Programming]			
Object type	The object type describes the physical structure of the instance and the operations that are allowed on it.	I	8.9.6	Object Type Definitions
Object type inheritance	With the sole exception of **System.Object**, which does not inherit from any other object type, all object types shall either explicitly or implicitly declare support for (inherit from) exactly one other object type.	I	8.9.9	Object Type Inheritance
Objects	Each object is self-typing, that is, its type is explicitly stored in its representation. It has an identity that distinguishes it from all other objects, and it has slots that store other entities (which may be either objects or values). While the contents of its slots may be changed, the identity of an object never changes.	I	8	Common Type System
Opaque classes	Some languages provide multi-byte data structures whose contents are manipulated directly by address arithmetic and indirection operations. To support this feature, the CLI allows value types to be created with a specified size but no information about their data members.	I	12.1.6.3	Opaque Classes
Overloading	Within a single scope, a given name may refer to any number of methods provided they differ in any of the following: Number of parameters [and] Type of each argument	I	10.2	Overloading
Overriding	..Overriding deals with object layout and is applicable only to instance fields and virtual methods. The CTS provides two forms of member overriding, **new slot** and **expect existing slot**.	I	8.10.4	Hiding, Overriding, and Layout
Parameter	[Name used within the body of a method to refer to the corresponding argument of the method]			
Parameter passing	The CLI supports three kinds of parameter passing, all indicated in metadata as part of the signature of the method. Each parameter to a method has its own passing convention (e.g., the first parameter may be passed by-value while all others are passed by-ref).	I	12.4.1.5	Parameter Passing

Parameter Signatures	**Parameter signatures** define constraints on how an individual value is passed as part of a method invocation.	I	8.6.1.4	Parameter Signatures
Pinned	While a method with a pinned local variable is executing the VES shall not relocate the object to which the local refers.	II	7.1.2	Pinned
PInvoke	Methods defined in native code may be invoked using the **platform invoke** (also know as PInvoke or p/invoke) functionality of the CLI.	II	14.5.2	Platform Invoke
Pointer type	A **pointer type** is a compile time description of a value whose representation is a machine address of a location.	I	8.2.1	Value Types and Reference Types
Pointers	Pointers may contain the address of a field (of an object or value type) or an element of an array.	II	13.4	Pointer Types
Private accessibility	Accessible only to referents in the implementation of the exact type that defines the member.	I	8.5.3.2	Accessibility of Members
Profiles	A Profile is simply a set of Libraries, grouped together to form a consistent whole that provides a fixed level of functionality.	IV	2.2	Profiles
Properties	. Propert[ies] define named groups of accessor method definitions that implement the named event or property behavior.	I	8.11	Member Definitions
Property contract	A property contract is specified with a property definition. There is an extensible set of operations for handling a named value, which includes a standard pair for reading the value and changing the value. A property contract specifies method contracts for the subset of these operations that shall be implemented by any type that supports the property contract.	I	8.6	Contracts
Property definitions	A property definition defines a named value and the methods that access the value. A property definition defines the accessing contracts on that value.	I	8.11.3	Property Definitions
Public accessibility	Accessible to all referents	I	8.5.3.2	Accessibility of Members
Qualified name	…Consider a compound type `Point` that has a field named `x`. The name “field `x`” by itself does not uniquely identify the named field, but the **qualified name** “field `x` in type `Point`” does.	I	8.5.2	Assemblies and Scoping
Rank	The *rank* of an array is the number of dimensions.	II	13.2	Arrays
Reference demand	Any attempt to resolve a reference to the marked item shall have specified security permission.	I	8.5.3.3	Security Permissions
Reference types	Reference Types describe values that are represented as the location of a sequence of bits. There are three kinds of Reference Types:	I	8.2.1	Value Types and Reference Types
Reflection Library	This 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	IV	5.4	Reflection Library

	based on a description of the type.			
Remoting boundary	A **remoting boundary** exists if it is not possible to share the identity of an object directly across the boundary. For example, if two objects exist on physically separate machines that do not share a common address space, then a remoting boundary will exist between them.	I	12.5	Proxies and Remoting
Return state handle	This handle is used to restore the method state on return from the current method.	I	12.3.2	Method State
Runtime Infrastructure Library	This 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.	IV	5.1	Runtime Infrastructure Library
Scopes	Names are collected into groupings called **scopes**.	I	8.5.2	Assemblies and Scoping
Sealed	Specifies that a type shall not have subclasses	II	9.1.4	Inheritance Attributes
Sealed type	An object type declares it shall not be used as a base type (be inherited from) by declaring that it is a **sealed** type.	I	8.9.9	Object Type Inheritance
Security descriptor	This descriptor is not directly accessible to managed code but is used by the CLI security system to record security overrides (**assert**, **permit-only**, and **deny**).	I	12.3.2	Method State
Security permissions	Access to members is also controlled by security demands that may be attached to an assembly, type, method, property, or event.	I	8.5.3.3	Security Permissions
Serializable fields	A field that is marked **serializable** is to be serialized as part of the persistent state of a value of the type.	I	8.11.2	Field Definitions
Setter method	By convention, properties define …optionally a **setter** method (for modifying the current value of the property).	I	8.11.3	Property Definitions
Signatures	Signatures are the part of a contract that can be checked and automatically enforced. Signatures are formed by adding constraints to types and other signatures.	I	8.6.1	Signatures
Simple labels	A simple label is a special name that represents an address	II	5.4	Labels and Lists of Labels
Special members	There are three special members, all methods, that can be defined as part of a type: instance constructors, instance finalizers, and type initializers.	II	9.5	Special Members
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.	II	13	Semantics of Special TYpes
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.	IV	3	The Standard Profiles

Static fields	Types may declare locations that are associated with the type rather than any particular value of the type. Such locations are **static fields** of the type.	I	8.4.3	Static Fields and Static Methods
Static methods	…Types may also declare methods that are associated with the type rather than with values of the type. Such methods are **static methods** of the type.	I	8.4.3	Static Fields and Static Methods
Super Calls	In some cases, it may be desirable to re-use code defined in the base type. E.g., an overriding virtual method may want to call its previous version. This kind of re-use is called a *super call*, since the overridden method of the base type is called.			
This	When they are invoked, instance and virtual methods are passed the value on which this invocation is to operate (known as **this** or a **this pointer**).	I	8.4.2	Methods
Thunk	A (typically) small piece of code used to provide a transition between two pieces of code where special handling is required			
Try block	In the CLI, a method may define a range of CIL instructions that are said to be *protected.* This is called the try block.	II	18	Exception Handling
Type definers	Type definers construct a new type from existing types.	I	8.9	Type Definers
Type definition	The **type definition**: • Defines a name for the type being defined, i.e. the **type name**, and specifies a scope in which that name will be found • Defines a **member scope** in which the names of the different kinds of members (fields, methods, events, and properties) are bound. The tuple of (member name, member kind, and member signature) is unique within a member scope of a type. • Implicitly assigns the type to the assembly scope of the assembly that contains the type definition.	I	8.5.2	Assemblies and Scoping
Type inheritance	Inheritance of types is another way of saying that the derived type guarantees support for all of the type contracts of the base type. In addition, the derived type usually provides additional functionality or specialized behavior.	I	8.9.8	Type Inheritance
Type members	Object type definitions include member definitions for all of the members of the type. Briefly, members of a type include fields into which values are stored, methods that may be invoked, properties that are available, and events that may be raised.	I	8.4	Type Members
Type safety	An implementation that lives up to the enforceable part of the contract (the named signatures) is said to be **typesafe**.	I	8.8	Type Safety and Verification

Type signatures	Type signatures define the constraints on a value and its usage.	I	8.6.1.1	Type Signatures
Typed reference parameters	A runtime representation of the data type is passed along with the address of the data, and the type of the parameter is therefore one specially supplied for this purpose.	I	12.4.1.5	Parameter Passing
Types	Types describe values. All places where values are stored, passed, or operated upon have a type, e.g. all variables, parameters, evaluation stack locations, and method results. The type defines the allowable values and the allowable operations supported by the values of the type. All operators and functions have expected types for each of the values accessed or used.	I	8.2	Values and Types
Unary operators	Unary operators take one argument, perform some operation on it, and return the result. They are represented as static methods on the class that defines the type of their one operand or their return type.	I	10.3.1	Unary Operators
Unbox	**Unbox** is a narrowing (runtime exception may be generated) operation that converts a **System.Object** (whose runtime type is a value type) to a value type instance.	I	12.1.6.2.5	Boxing and Unboxing
Unmanaged Code	[Code that does not require the runtime for execution. This code may not use the common type system or other features of the runtime. Traditional native code (before the CLI) is considered unmanaged]			
Unmanaged pointer types	An **unmanaged pointer type** (also known simply as a "pointer type") is defined by specifying a location signature for the location the pointer references. Any signature of a pointer type includes this location signature.	I	8.9.2	Unmanaged Pointer Types
Validation	Validation refers to a set of tests that can be performed on any file to check that the file format, metadata, and CIL are self-consistent.	II	3	Validation and Verification
Value type inheritance	Value Types, in their unboxed form, do not inherit from any type.	I	8.9.10	Value Type inheritance
Value types	In contrast to classes, value types (see Partition I) are not accessed by using a reference but are stored directly in the location of that type.	II	12	Semantics of Value Types
Values	The representation of a value (except for those of built-in types) can be subdivided into sub-values. These sub-values are either named, in which case they are called **fields**, or they are accessed by an indexing expression, in which case they are called **array elements**.	I	8.4.1	Fields, Array Elements, and Values
Vararg Methods	vararg methods accept a variable number of arguments.	II	14.4.5	Vararg methods
Variable argument lists	The CLI works in conjunction with the class library to implement methods that accept argument lists of unknown length and type ("varargs methods").	I	12.3.2.3	Variable Argument Lists

Vectors	Vectors are single-dimension arrays with a zero lower bound.	II	13.1	Vectors
Verifiability	Memory safety is a property that ensures programs running in the same address space are correctly isolated from one another …Thus, it is desirable to test whether programs are memory safe prior to running them. Unfortunately, it is provably impossible to do this with 100% accuracy. Instead, the CLI can test a stronger restriction, called *verifiability*.	III	1.8	Verifiability
Verification	*Verification* refers to a check 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.	II	3	Validation and Verification
Version Number	The version number of the assembly, specified as four 32-bit integers	II	6.2.1.4	Version Numbers
Virtual call	..A virtual method may be invoked by a special mechanism (a **virtual call**) that chooses the implementation based on the dynamically detected type of the instance used to make the virtual call rather than the type statically known at compile time.	I	8.4.4	Virtual Methods
Virtual calling convention	The CIL provides a "virtual calling convention" that is converted by an interpreter or JIT compiler into a native calling convention.	I	12.4.1.4	Virtual Calling Convention
Virtual execution system	The Virtual Execution System (VES) provides an environment for executing managed code. It provides direct support for a set of built-in data types, defines a hypothetical machine with an associated machine model and state, a set of control flow constructs, and an exception handling model.	I	6	Overview of the Common Language Infrastructure
Virtual methods	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 depends upon the type of object used for the *this* pointer.	II	14.2	Static, Instance, and Virtual Methods
Visibility	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.	II	8.1	Visibility of Top-Level Types and Accessibility of Nested Types
Widen	If a type overrides an inherited method, it may *widen*, but it shall not *narrow*, the accessibility of that method.	II	9.3.3	Accessibility and Overriding
XML Library	This 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.	IV	5.5	XML Library

End informative text

6 Overview of the Common Language Infrastructure

The Common Language Infrastructure (CLI) provides a specification for executable code and the execution environment (the Virtual Execution System, or VES) in which it runs. Executable code is presented to the VES as **modules**. A module is a single file containing executable content in the format specified in Partition II.

The remainder of this section and its subsections contain only informative text

At the center of the Common Language Infrastructure (CLI) is a unified type system, the Common Type System (CTS), that is shared by compilers, tools, and the CLI itself. It is the model that defines the rules the CLI follows when declaring, using, and managing types. The CTS establishes a framework that enables cross-language integration, type safety, and high performance code execution. This section describes the architecture of CLI by describing the CTS.

The following four areas are covered in this section:

- **The Common Type System**. See Chapter 8. The Common Type System (CTS) provides a rich type system that supports the types and operations found in many programming languages. The Common Type System is intended to support the complete implementation of a wide range of programming languages.
- **Metadata.** See Chapter 9. The CLI uses metadata to describe and reference the types defined by the Common Type System. Metadata is stored ("persisted") in a way that is independent of any particular programming language. Thus, metadata provides a common interchange mechanism for use between tools that manipulate programs (compilers, debuggers, etc.) as well as between these tools and the Virtual Execution System.
- **The Common Language Specification.** See Chapter 10. The Common Language Specification is an agreement between language designers and framework (class library) designers. It specifies a subset of the CTS Type System and a set of usage conventions. Languages provide their users the greatest ability to access frameworks by implementing at least those parts of the CTS that are part of the CLS. Similarly, frameworks will be most widely used if their publicly exposed aspects (classes, interfaces, methods, fields, etc.) use only types that are part of the CLS and adhere to the CLS conventions.
- **The Virtual Execution System**. See Chapter 12. The Virtual Execution System (VES) implements and enforces the CTS model. The VES is responsible for loading and running programs written for the CLI. It provides the services needed to execute managed code and data, using the metadata to connect separately generated modules together at runtime (late binding).

Together, these aspects of the CLI form a unifying framework for designing, developing, deploying, and executing distributed components and applications. The appropriate subset of the Common Type System is available from each programming language that targets the CLI. Language-based tools communicate with each other and with the Virtual Execution System using metadata to define and reference the types used to construct the application. The Virtual Execution System uses the metadata to create instances of the types as needed and to provide data type information to other parts of the infrastructure (such as remoting services, assembly downloading, security, etc.).

6.1 Relationship to Type Safety

Type safety is usually discussed in terms of what it does, e.g. guaranteeing encapsulation between different objects, or in terms of what it prevents, e.g. memory corruption by writing where one shouldn't. However, from the point of view of the Common Type System, type safety guarantees that:

- **References are what they say they are** - Every reference is typed and the object or value referenced also has a type, and they are assignment compatible (see Section 8.7).
- **Identities are who they say they are** - There is no way to corrupt or spoof an object, and by implication a user or security domain. The access to an object is through accessible functions and fields. An object may still be designed in such a way that security is compromised. However, a

local analysis of the class, its methods, and the things it uses, as opposed to a global analysis of all uses of a class, is sufficient to assess the vulnerabilities.

- **Only appropriate operations can be invoked** – The reference type defines the accessible functions and fields. This includes limiting visibility based on where the reference is, e.g. protected fields only visible in subclasses.

The Common Type System promotes type safety e.g. everything is typed. Type safety can be optionally enforced. The hard problem is determining if an implementation conforms to a typesafe declaration. Since the declarations are carried along as metadata with the compiled form of the program, a compiler from the Common Intermediate Language (CIL) to native code (see Section 8.8) can type-check the implementations.

6.2 Relationship to Managed Metadata-driven Execution

Metadata describes code by describing the types that the code defines and the types that it references externally. The compiler produces the metadata when the code is produced. Enough information is stored in the metadata to:

- **Manage code execution** – not just load and execute, but also memory management and execution state inspection.
- **Administer the code** – Installation, resolution, and other services
- **Reference types in the code** – Importing into other languages and tools as well as scripting and automation support.

The Common Type System assumes that the execution environment is metadata-driven. Using metadata allows the CLI to support:

- **Multiple execution models** - The metadata also allows the execution environment to deal with a mixture of interpreted, JITted, native and legacy code and still present uniform services to tools like debuggers or profilers, consistent exception handling and unwinding, reliable code access security, and efficient memory management.
- **Auto support for services** - Since the metadata is available at execution time, the execution environment and the base libraries can automatically supply support for reflection, automation, serialization, remote objects, and inter-operability with existing unmanaged native code with little or no effort on the part of the programmer.
- **Better optimization** – Using metadata references instead of physical offsets, layouts, and sizes allows the CLI to optimize the physical layouts of members and dispatch tables. In addition, this allows the generated code to be optimized to match the particular CPU or environment.
- **Reduced binding brittleness** – Using metadata references reduces version-to-version brittleness by replacing compile-time object layout with load-time layout and binding by name.
- **Flexible deployment resolution** - Since we can have metadata for both the reference and the definition of a type, more robust and flexible deployment and resolution mechanisms are possible. Resolution means that by looking in the appropriate set of places it is possible to find the implementation that best satisfies these requirements for use in this context. There are five elements of information in the foregoing: two items are made available via metadata (requirements and context); the others come from application packaging and deployment (where to look, how to find an implementation, and how to decide the best match).

6.2.1 Managed Code

Managed code is simply code that provides enough information to allow the CLI to provide a set of core services, including

- Given an address inside the code for a method, locate the metadata describing the method
- Walk the stack
- Handle exceptions

- Store and retrieve security information

This standard specifies a particular instruction set, the Common Intermediate Language (CIL, see Partition III), and a file format (see Partition II) for storing and transmitting managed code.

6.2.2 Managed Data

Managed data is data that is allocated and released automatically by the CLI, through a process called **garbage collection**.

6.2.3 Summary

The Common Type System is about integration between languages: using another language's objects as if they were one's own.

The objective of the CLI is to make it easier to write components and applications from any language. It does this by defining a standard set of types, making all components fully self-describing, and providing a high performance common execution environment. This ensures that all CLI-compliant system services and components will be accessible to all CLI-aware languages and tools. In addition, this simplifies deployment of components and applications that use them, all in a way that allows compilers and other tools to leverage the high performance execution environment. The Common Type System covers, at a high level, the concepts and interactions that make all of this possible.

The discussion is broken down into four areas:

- Type System – What types are and how to define them.
- Metadata – How types are described and how those descriptions are stored.
- Common Language Specification – Restrictions required for language interoperability.
- Virtual Execution System – How code is executed and types are instantiated, interact, and die.

End informative text

7 Common Language Specification (CLS)

7.1 Introduction

The Common Language Specification (CLS) is a set of rules intended to promote language interoperability. These rules shall be followed in order to conform to the CLS. They are described in greater detail in subsequent chapters and are summarized in Chapter 11. CLS conformance is a characteristic of types that are generated for execution on a CLI implementation. Such types must conform to the CLI specification, in addition to the CLS rules. These additional rules apply only to types that are visible in assemblies other than those in which they are defined, and to the members (fields, methods, properties, events, and nested types) that are accessible outside the assembly (i.e. those that have an accessibility of **public, family**, or **family-or-assembly**).

Note: A library consisting of CLS-compliant code is herein referred to as a "framework". Compilers that generate code for the CLI may be designed to make use of such libraries, but not to be able to produce or extend such library code. These compilers are referred to as "consumers". Compilers that are designed to both produce and extend frameworks are referred to as "extenders". In the description of each CLS rule, additional informative text is provided to assist the reader in understanding the rule's implication for each of these situations.

7.2 Views of CLS Compliance

This section and its subsections contain only informative text

The CLS is a set of rules that apply to generated assemblies. Because the CLS is designed to support interoperability for libraries and the high-level programming languages used to write them, it is often useful to think of the CLS rules from the perspective of the high-level source code and tools, such as compilers, that are used in the process of generating assemblies. For this reason, informative notes are added to the description of CLS rules to assist the reader in understanding the rule's implications for several different classes of tools and users. The different viewpoints used in the description are called **framework, consumer**, and **extender** and are described here.

7.2.1 CLS Framework

A library consisting of CLS-compliant code is herein referred to as a "framework". Frameworks (libraries) are designed for use by a wide range of programming languages and tools, including both CLS consumer and extender languages. By adhering to the rules of the CLS, authors of libraries ensure that the libraries will be usable by a larger class of tools than if they chose not to adhere to the CLS rules. The following are some additional guidelines that CLS-compliant frameworks should follow:

- Avoid the use of names commonly used as keywords in programming languages
- Should not expect users of the framework to be able to author nested types
- Should assume that implementations of methods of the same name and signature on different interfaces are independent.
- Should not rely on initialization of value types to be performed automatically based on specified initializer values.

7.2.2 CLS Consumer

A CLS consumer is a language or tool that is designed to allow access to all of the features supplied by CLS-compliant frameworks (libraries), but not necessarily be able to produce them. The following is a partial list of things CLS consumer tools are expected to be able to do:

- Support calling any CLS-compliant method or delegate
- Have a mechanism for calling methods that have names that are keywords in the language

- Support calling distinct methods supported by a type that have the same name and signature, but implement different interfaces
- Create an instance of any CLS-compliant type
- Read and modify any CLS-compliant field
- Access nested types
- Access any CLS-compliant property. This does not require any special support other than the ability to call the getter and setter methods of the property.
- Access any CLS-compliant event. This does not require any special support other than the ability to call methods defined for the event.

The following is a list of things CLS consumer tools need not support:

- Creation of new types or interfaces
- Initialization metadata (see Partition II) on fields and parameters other than static literal fields. Note that consumers may choose to use initialization metadata, but may also safely ignore such metadata on anything other than static literal fields.

7.2.3 CLS Extender

A CLS extender is a language or tool that is designed to allow programmers to both use and extend CLS-compliant frameworks. CLS extenders support a superset of the behavior supported by a CLS consumer, i.e., everything that applies to a CLS consumer also applies to CLS extenders. In addition to the requirements of a consumer, extenders are expected to be able to:

- Define new CLS-compliant types that extend any (non-sealed) CLS-compliant base class
- Have some mechanism for defining types with names that are keywords in the language
- Provide independent implementations for all methods of all interfaces supported by a type. That is, it is not sufficient for an extender to require a single code body to implement all interface methods of the same name and signature.
- Implement any CLS-compliant interface
- Place any CLS-compliant custom attribute on all appropriate elements of metadata

Extenders need not support the following:

- Definition of new CLS-compliant interfaces
- Definition of nested types

The common language specification is designed to be large enough that it is properly expressive and small enough that all languages can reasonably accommodate it.

End informative text

7.3 CLS Compliance

As these rules are introduced in detail, they are described in a common format. For an example, see the first rule below. The first paragraph specifies the rule itself. This is then followed by an informative description of the implications of the rule from the three different viewpoints as described above.

The CLS defines language interoperability rules, which apply only to “externally visible” items. The CLS unit of that language interoperability is the assembly– that is, within a single assembly there are no restrictions as to the programming techniques that are used. Thus, the CLS rules apply only to items that are visible (see clause 8.5.3) outside of their defining assembly and have **public**, **family**, or **family-or-assembly** accessibility (see clause 8.5.3.2).

CLS Rule 1: CLS rules apply only to those parts of a type that are accessible or visible outside of the defining assembly.

Note:

CLS (consumer): no impact.

CLS (extender): when checking CLS compliance at compile time, be sure to apply the rules only to information that will be exposed outside the assembly.

CLS (framework): CLS rules do not apply to internal implementation within an assembly. A type is **CLS-compliant** if all its publicly accessible parts (those classes, interfaces, methods, fields, properties, and events that are available to code executing in another assembly) either

- have signatures composed only of CLS-compliant types, or
- are specifically marked as not CLS-compliant

Any construct that would make it impossible to rapidly verify code is excluded from the CLS. This allows all CLS-compliant languages to produce verifiable code if they so choose.

7.3.1 Marking Items as CLS-Compliant

The CLS specifies how to mark externally visible parts of an assembly to indicate whether or not they comply with the CLS requirements. This is done using the custom attribute mechanism (see Section 9.7 and Partition II). The class `System.CLSCompliantAttribute` (see Partition IV) indicates which types and type members are CLS-compliant. It also can be attached to an assembly, to specify the default value for all top-level types it contains.

The constructor for `System.CLSCompliantAttribute` takes a Boolean argument indicating whether the item with which it is associated is or is not CLS-compliant. This allows any item (assembly, type, or type member) to be explicitly marked as CLS-compliant or not.

The rules for determining CLS compliance are:

- When an assembly does not carry an explicit `System.CLSCompliantAttribute`, it shall be assumed to carry `System.CLSCompliantAttribute(false)`.
- By default, a type inherits the CLS-compliance attribute of its enclosing type (for nested types) or acquires the value attached to its assembly (for top-level types). It may be marked as either CLS-compliant or not CLS-Compliant by attaching the `System.CLSCompliantAttribute` attribute.
- By default, other members (methods, fields, properties and events) inherit the CLS-compliance of their type. They may be marked as not CLS-compliant by attaching the attribute `System.CLSCompliantAttribute(false)`.

CLS Rule 2: Members of non-CLS compliant types shall not be marked CLS-compliant.

Note:

CLS (consumer): May ignore any member that is not CLS-compliant using the above rules.

CLS (extender): Should encourage correct labeling of newly authored assemblies, classes, interfaces, and methods. Compile-time enforcement of the CLS rules is strongly encouraged.

CLS (framework): Shall correctly label all publicly exposed members as to their CLS compliance. The rules specified here may be used to minimize the number of markers required (for example, label the entire assembly if all types and members are compliant or if there are only a few exceptions that need to be marked).

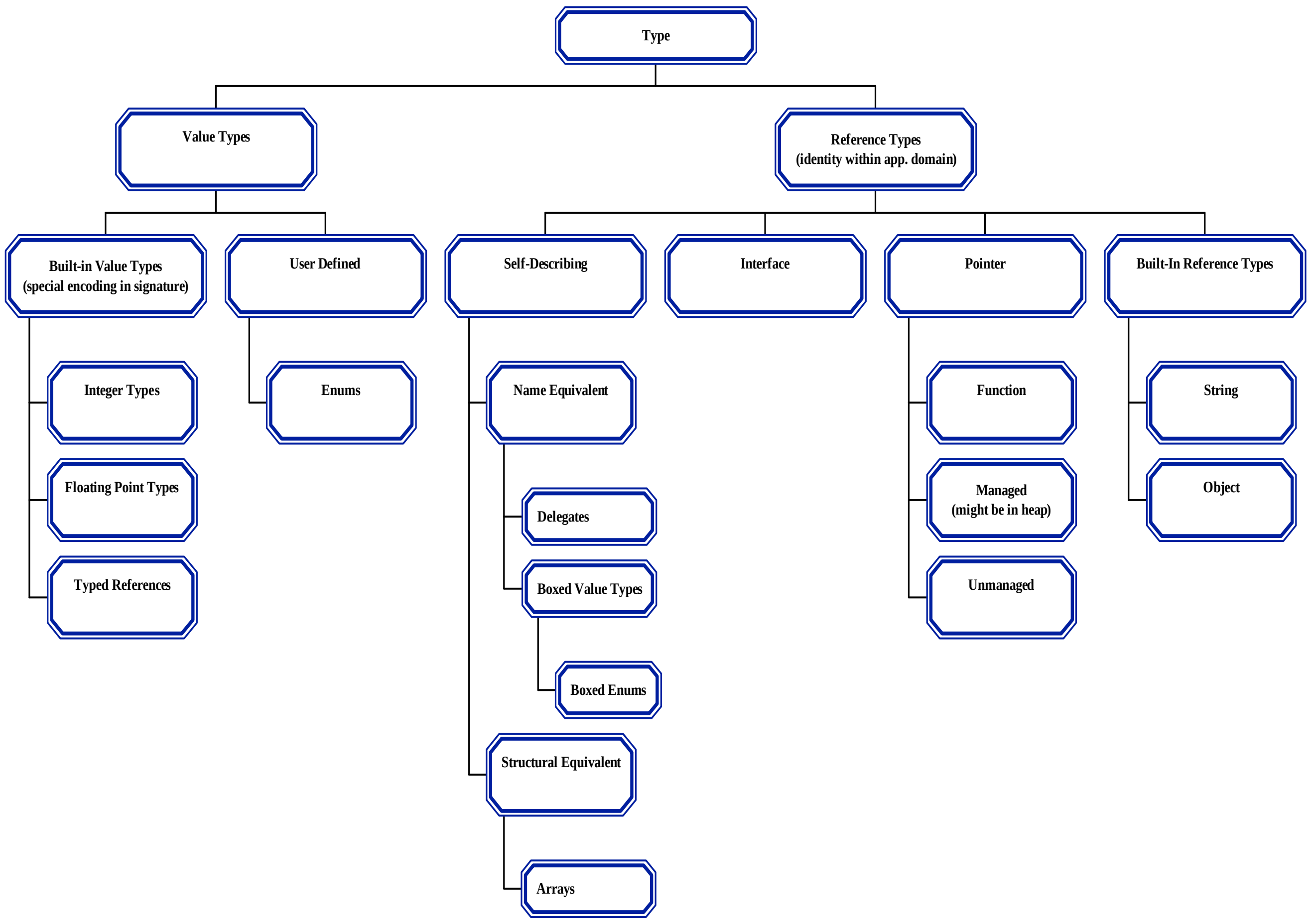
8 Common Type System

Types describe values and specify a contract (see Section 8.6) that all values of that type shall support. Because the CTS supports Object-Oriented Programming (OOP) as well as functional and procedural programming languages, it deals with two kinds of entities: Objects and Values. Values are simple bit patterns for things like integers and floats; each value has a type that describes both the storage that it occupies and the meanings of the bits in its representation, and also the operations that may be performed on that representation. Values are intended for representing the corresponding simple types in programming languages like C, and also for representing non-objects in languages like C++ and Java™.

Objects have rather more to them than do values. Each object is self-typing, that is, its type is explicitly stored in its representation. It has an identity that distinguishes it from all other objects, and it has slots that store other entities (which may be either objects or values). While the contents of its slots may be changed, the identity of an object never changes.

There are several kinds of Objects and Values, as shown in the following diagram.

Figure 1: Type System



8.1 Relationship to Object-Oriented Programming

This section contains only informative text

The term **type** is often used in the world of value-oriented programming to mean data representation. In the object-oriented world it usually refers to behavior rather than to representation. In the CTS, type is used to mean both of these things: two entities have the same type if and only if they have both compatible representations and behaviors. Thus, in the CTS, if one type is derived from a base type, then instances of the derived type may be substituted for instances of the base type because **both** the representation and the behavior are compatible.

In the CTS, unlike some OOP languages, two objects that have fundamentally different representations have different types. Some OOP languages use a different notion of type. They consider two objects to have the same type if they respond in the same way to the same set of messages. This notion is captured in the CTS by saying that the objects implement the same interface.

Similarly, some OOP languages (e.g. Smalltalk) consider message passing to be the fundamental model of computation. In the CTS, this corresponds to calling virtual methods (see clause 8.4.4), where the signature of the virtual method plays the role of the message.

The CTS itself does not directly capture the notion of "typeless programming." That is, there is no way to call a non-static method without knowing the type of the object. Nevertheless, typeless programming can be implemented based on the facilities provided by the reflection package (see Partition IV) if it is implemented.

End informative text

8.2 Values and Types

Types describe values. All places where values are stored, passed, or operated upon have a type, e.g. all variables, parameters, evaluation stack locations, and method results. The type defines the allowable values and the allowable operations supported by the values of the type. All operators and functions have expected types for each of the values accessed or used.

A value can be of more than one type. A value that supports many interfaces is an example of a value that is of more than one type, as is a value that inherits from another.

8.2.1 Value Types and Reference Types

There are two kinds of types: **Value Types** and **Reference Types**.

- Value Types - Value Types describe values that are represented as sequences of bits.
- Reference Types – Reference Types describe values that are represented as the location of a sequence of bits. There are four kinds of Reference Types:
    - An **object type** is a reference type of a self-describing value (see clause 8.2.3). Some object types (e.g. abstract classes) are only a partial description of a value.
    - An **interface type** is always a partial description of a value, potentially supported by many object types.
    - A **pointer type** is a compile time description of a value whose representation is a machine address of a location.
    - Built-in types

8.2.2 Built-in Types

The following data types are an integral part of the CTS and are supported directly by the Virtual Execution System (VES). They have special encoding in the persisted metadata:

Table 1: Special Encoding

Name in CIL assembler (see Partition II)	**CLS Type?**	**Name in class library (see Partition IV)**	**Description**
`bool`	Yes	`System.Boolean`	True/false value
`char`	Yes	`System.Char`	Unicode 16-bit char.
`object`	Yes	`System.Object`	Object or boxed value type
`string`	Yes	`System.String`	Unicode string
`float32`	Yes	`System.Single`	IEC 60559:1989 32-bit float
`float64`	Yes	`System.Double`	IEC 60559:1989 64-bit float
`int8`	No	`System.SByte`	Signed 8-bit integer
`int16`	Yes	`System.Int16`	Signed 16-bit integer
`int32`	Yes	`System.Int32`	Signed 32-bit integer
`int64`	Yes	`System.Int64`	Signed 64-bit integer
`native int`	Yes	`System.IntPtr`	Signed integer, native size
`native unsigned int`	No	`System.UIntPtr`	Unsigned integer, native size
`typedref`	No	`System.TypedReference`	Pointer plus runtime type
`unsigned int8`	Yes	`System.Byte`	Unsigned 8-bit integer
`unsigned int16`	No	`System.UInt16`	Unsigned 16-bit integer
`unsigned int32`	No	`System.UInt32`	Unsigned 32-bit integer
`unsigned int64`	No	`System.UInt64`	Unsigned 64-bit integer

8.2.3 Classes, Interfaces and Objects

Every value has an **exact type** that **fully describes** the value. A type fully describes a value if it unambiguously defines the value's representation and the operations defined on the value.

For a Value Type, defining the representation entails describing the sequence of bits that make up the value's representation. For a Reference Type, defining the representation entails describing the location and the sequence of bits that make up the value's representation.

A **method** describes an operation that may be performed on values of an exact type. Defining the set of operations allowed on values of an exact type entails specifying named methods for each operation.

Some types are only a partial description, e.g. **interface types**. Interface types describe a subset of the operations and none of the representation, and hence, cannot be an exact type of any value. Hence, while a value has only one exact type, it may also be a value of many other types as well. Furthermore, since the exact type fully describes the value, it also fully specifies all of the other types that a value of the exact type can have.

While it is true that every value has an exact type, it is not always possible to determine the exact type by inspecting the representation of the value. In particular, it is *never* possible to determine the exact type of a value of a Value Type. Consider two of the built-in Value Types, 32-bit signed and unsigned integers. While each type is a full specification of their respective values, i.e. an exact type, there is no way to derive that exact type from a value's particular 32-bit sequence.

For some values, called **objects**, it *is* always possible to determine the exact type from the value. Exact types of objects are also called **object types**. Objects are values of Reference Types, but not all Reference Types describe objects. Consider a value that is a pointer to a 32-bit integer, a kind of Reference Type. There is no way to discover the type of the value by examining the pointer bits, hence it is not an object. Now consider the built-in CTS Reference Type **System.String** (see Partition IV). The exact type of a value of this type is always determinable by examining the value, hence values of type **System.String** are objects and **System.String** is an object type.

8.2.4 Boxing and Unboxing of Values

For every Value Type, the CTS defines a corresponding Reference Type called the **boxed type**. The reverse is not true: Reference Types do not in general have a corresponding Value Type. The representation of a value of a boxed type (a **boxed value**) is a location where a value of the Value Type may be stored. A boxed type is an object type and a boxed value is an object.

All Value Types have an operation called **box**. Boxing a value of any Value Type produces its boxed value, i.e. a value of the corresponding boxed type containing a bit copy of the original value. All boxed types have an operation called **unbox**. Unboxing results in a managed pointer to the bit representation of the value.

Notice that interfaces and inheritance are defined only on Reference types. Thus, while a Value Type definition (see clause 8.9.7) can specify both interfaces that shall be implemented by the Value Type and the class (`System.ValueType` or `System.Enum`) from which it inherits, these apply only to boxed values.

CLS Rule 3: The CLS does not include boxed value types.

Note:

In lieu of boxed types, use `System.Object`, `System.ValueType` or `System.Enum`, as appropriate. (See Partition IV)

CLS (consumer): need not import boxed value types.

CLS (extender): need not provide syntax for defining or using boxed value types.

CLS (framework): shall not use boxed value types in their publicly exposed aspects.

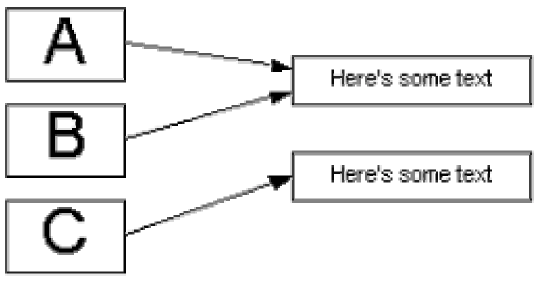
8.2.5 Identity and Equality of Values

There are two binary operators defined on all pairs of values, **identity** and **equality**, that return a Boolean result. Both of these operators are mathematical **equivalence** operators, i.e. they are:

- Reflexive - `a op a` is true.
- Symmetric - a op b is true if and only if `b op a` is true.
- Transitive - if `a op b` is true and `b op c` is true, then `a op c` is true

In addition, identity always implies equality, but not the reverse, i.e., the equality operator need not be the same as the identity operator as long as two identical values are also equal values.

To understand the difference between these operations, consider three variables whose type is `System.String`, where the arrow is intended to mean "is a reference to":



The values of the variables are **identical** if the locations of the sequences of characters are the same, i.e., there is in fact only one string in memory. The values stored in the variables are **equal** if the sequences of characters are the same. Thus, the values of variables A and B are identical, the values of variables A and C as well as B and C are not identical, and the values of all three of A, B, and C are equal.

8.2.5.1 Identity

The identity operator is defined by the CTS as follows.

- If the values have different exact types, then they are not identical.
- Otherwise, if their exact type is a Value Type, then they are identical if and only if the bit sequences of the values are the same, bit by bit.
- Otherwise, if their exact type is a Reference Type, then they are identical if and only if the locations of the values are the same.

Identity is implemented on `System.Object` via the `ReferenceEquals` method.

8.2.5.2 Equality

For value types, the equality operator is part of the definition of the exact type. Definitions of equality should obey the following rules:

- Equality should be an equivalence operator, as defined above.
- Identity should imply equality, as stated earlier.
- If either (or both) operand is a boxed value, equality should be computed by
  - first unboxing any boxed operand(s), and then
  - applying the usual rules for equality on the resulting values.

Equality is implemented on `System.Object` via the `Equals` method.

Note: Although two floating point NaNs are defined by IEC 60559:1989 to always compare as unequal, the contract for `System.Object.Equals`, requires that overrides must satisfy the requirements for an equivalence operator. Therefore, `System.Double.Equals` and `System.Single.Equals` return **True** when comparing two NaNs, while the equality operator returns False in that case, as required by the standard.

8.3 Locations

Values are stored in **locations**. A location can hold a single value at a time. All locations are typed. The type of the location embodies the requirements that shall be met by values that are stored in the location. Examples of locations are local variables and parameters.

More importantly, the type of the location specifies the restrictions on usage of any value that is loaded from the location. For example, a location can hold values of potentially many exact types as long as all of the values are assignment compatible with the type of the location (see below). All values loaded from a location are treated as if they are of the type of the location. Only operations valid for the type of the location may be invoked even if the exact type of the value stored in the location is capable of additional operations.

8.3.1 Assignment Compatible Locations

A value may be stored in a location only if one of the types of the value is **assignment compatible** with the type of the location. A type is always assignment compatible with itself. Assignment compatibility can often be determined at compile time, in which case there is no need for testing at run time. Assignment compatibility is described in detail in Section 8.7.

8.3.2 Coercion

Sometimes it is desirable to take a value of a type that is *not* assignment compatible with a location and convert the value to a type that *is* assignment compatible. This is accomplished through **coercion** of the value. Coercion takes a value of a particular type and a desired type and attempts to create a value of the desired type that has equivalent meaning to the original value. Coercion can result in representation changes as well as type changes, hence coercion does not necessarily preserve the identity of two objects.

There are two kinds of coercion: **widening**, which never loses information, and **narrowing**, in which information may be lost. An example of a widening coercion would be coercing a value that is a 32-bit signed integer to a value that is a 64-bit signed integer. An example of a narrowing coercion is the reverse: coercing a 64-bit signed integer to a 32-bit signed integer. Programming languages often implement widening coercions as **implicit conversions**, whereas narrowing coercions usually require an **explicit conversion**.

Some widening coercion is built directly into the VES operations on the built-in types (see Section 12.1). All other coercion shall be explicitly requested. For the built-in types, the CTS provides operations to perform widening coercions with no runtime checks and narrowing coercions with runtime checks.

8.3.3 Casting

Since a value can be of more than one type, a use of the value needs to clearly identify which of its types is being used. Since values are read from locations that are typed, the type of the value which is used is the type of the location from which the value was read. If a different type is to be used, the value is **cast** to one of its other types. Casting is usually a compile time operation, but if the compiler cannot statically know that the value is of the target type, a runtime cast check is done. Unlike coercion, a cast never changes the actual type of an object nor does it change the representation. Casting preserves the identity of objects.

For example, a runtime check may be needed when casting a value read from a location that is typed as holding values of a particular interface. Since an interface is an incomplete description of the value, casting that value to be of a different interface type will usually result in a runtime cast check.

8.4 Type Members

As stated above, the type defines the allowable values and the allowable operations supported by the values of the type. If the allowable values of the type have a substructure, that substructure is described via fields or array elements of the type. If there are operations that are part of the type, those operations are described via methods on the type. Fields, array elements, and methods are called **members** of the type. Properties and events are also members of the type.

8.4.1 Fields, Array Elements, and Values

The representation of a value (except for those of built-in types) can be subdivided into sub-values. These sub-values are either named, in which case they are called **fields**, or they are accessed by an indexing expression, in which case they are called **array elements**. Types that describe values composed of array elements are **array types**. Types that describe values composed of fields are **compound types**. A value cannot contain both fields and array elements, although a field of a compound type may be an array type and an array element may be a compound type.

Array elements and fields are typed, and these types never change. All of the array elements shall have the same type. Each field of a compound type may have a different type.

8.4.2 Methods

A type may associate operations with the type or with each instance of the type. Such operations are called methods. A method is named, and has a signature (see clause 8.6.1) that specifies the allowable types for all of its arguments and for its return value, if any.

A method that is associated only with the type itself (as opposed to a particular instance of the type) is called a static method (see clause 8.4.3).

A method that is associated with an instance of the type is either an instance method or a virtual method (see clause 8.4.4). When they are invoked, instance and virtual methods are passed the instance on which this invocation is to operate (known as **this** or a **this pointer**).

The fundamental difference between an instance method and a virtual method is in how the implementation is located. An instance method is invoked by specifying a class and the instance method within that class. The object passed as **this** may be **null** (a special value indicating that no instance is being specified) or an instance of any type that inherits (see clause 8.9.8) from the class that defines the method. A virtual method may also be called in this manner. This occurs, for example, when an implementation of a virtual method wishes to call the implementation supplied by its parent class. The CTS allows **this** to be **null** inside the body of a virtual method.

Rationale: *Allowing a virtual method to be called with a non-virtual call eliminates the need for a "call super" instruction and allows version changes between virtual and non-virtual methods. It requires CIL generators to insert explicit tests for a null pointer if they don't want the null this pointer to propagate to called methods.*

A virtual or instance method may also be called by a different mechanism, a **virtual call**. Any type that inherits from a type that defines a virtual method may provide its own implementation of that method (this is known as **overriding**, see clause 8.10.4). It is the exact type of the object (determined at runtime) that is used to decide which of the implementations to invoke

8.4.3 Static Fields and Static Methods

Types may declare locations that are associated with the type rather than any particular value of the type. Such locations are **static fields** of the type. As such, static fields declare a location that is shared by all values of the type. Just like non-static (instance) fields, a static field is typed and that type never changes. Static fields are always restricted to a single application domain basis (see Section 12.5), but they may also be allocated on a per-thread basis.

Similarly, types may also declare methods that are associated with the type rather than with values of the type. Such methods are **static methods** of the type. Since an invocation of a static method does not have an associated value on which the static method operates, there is no **this** pointer available within a static method.

8.4.4 Virtual Methods

An object type may declare any of its methods as **virtual**. Unlike other methods, each exact type that implements the type may provide its own implementation of a virtual method. A virtual method may be invoked through the ordinary method call mechanism that uses the static type, method name, and types of parameters to choose an implementation, in which case the **this** pointer may be **null**. In addition, however, a virtual method may be invoked by a special mechanism (a **virtual call**) that chooses the implementation based on the dynamically detected type of the instance used to make the virtual call rather than the type statically known at compile time. Virtual methods may be marked **final** (see clause 8.10.2).

8.5 Naming

Names are given to entities of the type system so that they can be referred to by other parts of the type system or by the implementations of the types. Types, fields, methods, properties and events have names. With respect to the type system values, locals, and parameters do not have names. An entity of the type system is given a single name, e.g. there is only one name for a type.

8.5.1 Valid Names

All comparisons are done on a byte-by-byte (i.e. case sensitive, locale-independent, also known as code-point comparison) basis. Where names are used to access built-in VES-supplied functionality (for example, the class initialization method) there is always an accompanying indication on the definition so as not to build in any set of reserved names.

CLS Rule 4: Assemblies shall follow Annex 7 of Technical Report 15 of the Unicode Standard 3.0 (ISBN 0-201-61633-5) governing the set of characters permitted to start and be included in identifiers, available on-line at http://www.unicode.org/unicode/reports/tr15/tr15-18.html. Identifiers shall be in the canonical format defined by Unicode Normalization Form C. For CLS purposes, two identifiers are the same if their lowercase mappings (as specified by the Unicode locale-insensitive, 1-1 lowercase mappings) are the same. That is, for two identifiers to be considered different under the CLS they shall differ in more than simply their case. However, in order to override an inherited definition the CLI requires the precise encoding of the original declaration be used.

Note:

CLS (consumer): need not consume types that violate CLS rule 4, but shall have a mechanism to allow access to named items that use one of its own keywords as the name.

CLS (extender): need not create types that violate CLS rule 4. Shall provide a mechanism for defining new names that obey these rules but are the same as a keyword in the language.

CLS (framework): shall not export types that violate CLS rule 4. Should avoid the use of names that are commonly used as keywords in programming languages (see Partition V Annex D)

8.5.2 Assemblies and Scoping

Generally, names are not unique. Names are collected into groupings called **scopes**. Within a scope, a name may refer to multiple entities as long as they are of different **kinds** (methods, fields, nested types, properties, and events) or have different signatures.

CLS Rule 5: All names introduced in a CLS-compliant scope shall be distinct independent of kind, except where the names are identical and resolved via overloading. That is, while the CTS allows a single type to use the same name for a method and a field, the CLS does not.

CLS Rule 6: Fields and nested types shall be distinct by identifier comparison alone, even though the CTS allows distinct signatures to be distinguished. Methods, properties, and events that have the same name (by identifier comparison) shall differ by more than just the return type, except as specified in CLS Rule 39.

Note:

CLS (consumer): need not consume types that violate these rules after ignoring any members that are marked as not CLS-compliant.

CLS (extender): need not provide syntax for defining types that violate these rules.

CLS (framework): shall not mark types as CLS-compliant if they violate these rules unless they mark sufficient offending items within the type as not CLS-compliant so that the remaining members do not conflict with one another.

A named entity has its name in exactly one scope. Hence, to identify a named entity, both a scope and a name need to be supplied. The scope is said to **qualify** the name. Types provide a scope for the names in the type; hence types qualify the names in the type. For example, consider a compound type `Point` that has a field named `x`. The name "field `x`" by itself does not uniquely identify the named field, but the **qualified name** "field `x` in type `Point`" does.

Since types are named, the names of types are also grouped into scopes. To fully identify a type, the type name shall be qualified by the scope that includes the type name. Type names are scoped by the **assembly** that contains the implementation of the type. An assembly is a configured set of loadable code modules and other resources that together implement a unit of functionality. The type name is said to be in the **assembly scope** of the assembly that implements the type. Assemblies themselves have names that form the basis of the CTS naming hierarchy.

The **type definition**:

- Defines a name for the type being defined, i.e. the **type name**, and specifies a scope in which that name will be found
- Defines a **member scope** in which the names of the different kinds of members (fields, methods, events, and properties) are bound. The tuple of (member name, member kind, and member signature) is unique within a member scope of a type.
- Implicitly assigns the type to the assembly scope of the assembly that contains the type definition.

The CTS supports an **enum** (also known as an **enumeration type**), an alternate name for an existing type. For purposes of matching signatures an enum shall not be the same as the underlying type. Instances of an enum, however, shall be assignment compatible with the underlying type and vice versa. That is: no cast (see clause 8.3.3) or coercion (see clause 8.3.2) is required to convert from the enum to the underlying type, nor are they required from the underlying type to the enum. An enum is considerably more restricted than a true type:

- It shall have exactly one instance field, and the type of that field defines the underlying type of the enumeration.
- It shall not have any methods of its own.
- It shall derive from `System.Enum` (see Partition IV).

- It shall not implement any interfaces of its own.
- It shall not have any properties or events of its own.
- It shall not have any static fields unless they are literal (see clause 8.6.1).

The underlying type shall be a built-in integer type. Enums shall derive from `System.Enum`, hence they are value types. Like all value types, they shall be sealed (see clause 8.9.9).

CLS Rule 7: The underlying type of an enum shall be a built-in CLS integer type.

CLS Rule 8: There are two distinct kinds of enums, indicated by the presence or absence of the `System.FlagsAttribute` (see Partition IV) custom attribute. One represents named integer values, the other named bit flags that can be combined to generate an unnamed value. The value of an enum is not limited to the specified values.

CLS Rule 9: Literal static fields (see clause 8.6.1) of an enum shall have the type of the enum itself.

Note:

CLS (consumer): Shall accept definition of enums that follow these rules, but need not distinguish flags from named values.

CLS (extender): Same as consumer. Extender languages are encouraged to allow the authoring of enums, but need not do so.

CLS (framework): shall not expose enums that violate these rules, and shall not assume that enums have only the specified values (even for enums that are named values).

8.5.3 Visibility, Accessibility, and Security

To refer to a named entity in a scope, both the scope and the name in the scope shall be **visible** (see clause 8.5.3.1). Visibility is determined by the relationship between the entity that contains the reference (the **referent**) and the entity that contains the name being referenced. Consider the following pseudo-code:

```
class A
{ int32 IntInsideA;
}
class B inherits from A
{ method X(int32, int32) returning Boolean
  { IntInsideA := 15;
  }
}
```

If we consider the reference to the field `IntInsideA in class A`:

- We call class B the **referent** because it has a method that refers to that field,
- We call `IntInsideA in class A` the **referenced entity**.

There are two fundamental questions that need to be answered in order to decide whether the referent is allowed to access the referenced entity. The first is whether the name of the referenced entity is **visible** to the referent. If it is visible, then there is a separate question of whether the referent is **accessible** (see clause 8.5.3.2).

Access to a member of a type is permitted only if all three of the following conditions are met:

1. The type is visible.
2. The member is accessible.
3. All relevant security demands (see clause 8.5.3.3) have been granted.

8.5.3.1 Visibility of Types

Only type names, not member names, have controlled visibility. Type names fall into one of the following three categories

- **Exported** from the assembly in which they are defined. While a type may be marked to *allow* it to be exported from the assembly, it is the configuration of the assembly that decides whether the type name *is* made available.
- **Not exported** outside the assembly in which they are defined.
- Nested within another type. In this case, the type itself has the visibility of the type inside of which it is nested (its **enclosing type**). See clause 8.5.3.4.

8.5.3.2 Accessibility of Members

A type scopes all of its members, and it also specifies the accessibility rules for its members. Except where noted, accessibility is decided based only on the statically visible type of the member being referenced and the type and assembly that is making the reference. The CTS supports seven different rules for accessibility:

- **Compiler-Controlled** – accessible only through use of a definition, not a reference, hence only accessible from within a single compilation unit and under the control of the compiler.
- **Private** – accessible only to referents in the implementation of the exact type that defines the member.
- **Family** – accessible to referents that support the same type, i.e. an exact type and all of the types that inherit from it. For verifiable code (see Section 8.8), there is an additional requirement that may require a runtime check: the reference shall be made through an item whose exact type supports the exact type of the referent. That is, the item whose member is being accessed shall inherit from the type performing the access.
- **Assembly** – accessible only to referents in the same assembly that contains the implementation of the type.
- **Family-and-Assembly** – accessible only to referents that qualify for both Family and Assembly access.
- **Family-or-Assembly** – accessible only to referents that qualify for either Family or Assembly access.
- **Public** – accessible to all referents.

In general, a member of a type can have any one of these accessibility rules assigned to it. There are two exceptions, however:

1. Members defined by an interface shall be public.
2. When a type defines a virtual method that overrides an inherited definition, the accessibility shall either be identical in the two definitions or the overriding definition shall permit more access than the original definition. For example, it is possible to override an **assembly virtual** method with a new implementation that is **public virtual**, but not with one that is **family virtual**. In the case of overriding a definition derived from another assembly, it is not considered restricting access if the base definition has **Family-or-Assembly** access and the override has only **family** access.

Rationale: *Languages including C++ allow this "widening" of access. Restricting access would provide an incorrect illusion of security since simply casting an object to the base class (which occurs implicitly on method call) would allow the method to be called despite the restricted accessibility. To prevent overriding a virtual method use **final** (see clause 8.10.2) rather than relying on limited accessibility.*

CLS Rule 10: Accessibility shall not be changed when overriding inherited methods, except when overriding a method inherited from a different assembly with accessibility **Family-or-Assembly**. In this case the override shall have accessibility **family**.

Note:

CLS (consumer): need not accept types that widen access to inherited virtual methods.

CLS (extender): need not provide syntax to widen access to inherited virtual methods.

CLS (frameworks): shall not rely on the ability to widen access to a virtual method, either in the exposed portion of the framework or by users of the framework.

8.5.3.3 Security Permissions

Access to members is also controlled by security demands that may be attached to an assembly, type, method, property, or event. Security demands are not part of a type contract (see Section 8.6), and hence are not inherited. There are two kinds of demands:

- An **inheritance demand.** When attached to a type it requires that any type that wishes to inherit from this type shall have the specified security permission. When attached to a non-final virtual method it requires that any type that wishes to override this method shall have the specified permission. It shall not be attached to any other member.
- A **reference demand**. Any attempt to resolve a reference to the marked item shall have specified security permission.

Only one demand of each kind may be attached to any item. Attaching a security demand to an assembly implies that it is attached to all types in the assembly unless another demand of the same kind is attached to the type. Similarly, a demand attached to a type implies the same demand for all members of the type unless another demand of the same kind is attached to the member. For additional information, see Declarative Security in Partition II, and the classes in the `System.Security` namespace in Partition IV.

8.5.3.4 Nested Types

A type (called a nested type) can be a member of an enclosing type. A nested type has the same visibility as the enclosing type and has an accessibility as would any other member of the enclosing type. This accessibility determines which other types may make references to the nested type. That is, for a class to define a field or array element of a nested type, have a method that takes a nested type as a parameter or returns one as value, etc., the nested type shall be both visible and accessible to the referencing type. A nested type is part of the enclosing type so its methods have access to all members of its enclosing type, as well as family access to members of the type from which it inherits (see clause 8.9.8). The names of nested types are scoped by their enclosing type, not their assembly (only top-level types are scoped by their assembly). There is no requirement that the names of nested types be unique within an assembly.

8.6 Contracts

Contracts are named. They are the shared assumptions on a set of **signatures** (see clause 8.6.1) between all implementers and all users of the contract. The signatures are the part of the contract that can be checked and enforced.

Contracts are not types; rather they specify requirements on the implementation of types. Types state which contracts they abide by, i.e. which contracts all implementations of the type shall support. An implementation of a type can be verified to check that the enforceable parts of a contract, the named signatures, have been implemented. The kinds of contracts are:

- **Class contract** – A class contract is specified with a class definition. Hence, a class definition defines both the class contract and the **class type**. The name of the class contract and the name of the class type are the same. A class contract specifies the representation of the values of the class type. Additionally, a class contract specifies the other contracts that the class type supports, e.g., which interfaces, methods, properties and events shall be implemented. A class contract, and hence the class type, can be supported by other class types as well. A class type that supports the class contract of another class type is said to **inherit** from that class type.
- **Interface contract** – An interface contract is specified with an interface definition. Hence, an interface definition defines both the interface contract and the **interface type**. The name of the interface contract and the name of the interface type are the same. Many types can support an interface contract. Like a class contract, interface contracts specify which other contracts the interface supports, e.g. which interfaces, methods, properties and events shall be implemented.

Note: An interface type can never fully describe the representation of a value. Therefore an interface type can never support a class contract, and hence can never be a class type or an exact type.

- **Method contract** – A method contract is specified with a method definition. A method contract is a named operation that specifies the contract between the implementation(s) of the method and the callers of the method. A method contract is always part of a type contract (class, value type, or interface), and describes how a particular named operation is implemented. The method contract specifies the contracts that each parameter to the method shall support and the contracts that the return value shall support, if there is a return value.
- **Property contract** – A property contract is specified with a property definition. There is an extensible set of operations for handling a named value, which includes a standard pair for reading the value and changing the value. A property contract specifies method contracts for the subset of these operations that shall be implemented by any type that supports the property contract. A type can support many property contracts, but any given property contract can be supported by exactly one type. Hence, property definitions are a part of the type definition of the type that supports the property.
- **Event contract** – An event contract is specified with an event definition. There is an extensible set of operations for managing a named event, which includes three standard methods (register interest in an event, revoke interest in an event, fire the event). An event contract specifies method contracts for all of the operations that shall be implemented by any type that supports the event contract. A type can support many event contracts, but any given event contract can be supported by exactly one type. Hence, event definitions are a part of the type definition of the type that supports the event.

8.6.1 Signatures

Signatures are the part of a contract that can be checked and automatically enforced. Signatures are formed by adding constraints to types and other signatures. A constraint is a limitation on the use of or allowed operations on a value or location. Example constraints would be whether a location may be overwritten with a different value or whether a value may ever be changed.

All locations have signatures, as do all values. Assignment compatibility requires that the signature of the value, including constraints, is compatible with the signature of the location, including constraints. There are four fundamental kinds of signatures: type signatures, location signatures, parameter signatures, and method signatures.

CLS Rule 11: All types appearing in a signature shall be CLS-compliant.

CLS Rule 12: The visibility and accessibility of types and members shall be such that types in the signature of any member shall be visible and accessible whenever the member itself is visible and accessible. For example, a public method that is visible outside its assembly shall not have an argument whose type is visible only within the assembly.

Note:

CLS (consumer): need not accept types whose members violate these rules.

CLS (extender): need not provide syntax to violate these rules.

CLS (framework): shall not violate this rule in its exposed types and their members.

The following sections describe the various kinds of signatures. These descriptions are cumulative: the simplest signature is a type signature; a location signature is a type signature plus (optionally) some additional attributes; and so forth.

8.6.1.1 Type Signatures

Type signatures define the constraints on a value and its usage. A type, by itself, is a valid type signature. The type signature of a value cannot be determined by examining the value or even by knowing the class type of the value. The type signature of a value is derived from the location signature (see below) of the location from which the value is loaded. Normally the type signature of a value is the type in the location signature from which the value is loaded.

Rationale: *The distinction between a Type Signature and a Location Signature (below) is not currently useful. It is made because certain constraints, such as "constant," are constraints on values not locations. Future versions of this standard, or non-standard extensions, may introduce type constraints, thus making the distinction meaningful.*

8.6.1.2 Location Signatures

All locations are typed. This means that all locations have a **location signature**, which defines constraints on the location, its usage, and on the usage of the values stored in the location. Any valid type signature is a valid location signature. Hence, a location signature contains a type and may additionally contain the constant constraint. The location signature may also contain **location constraints** that give further restrictions on the uses of the location. The location constraints are:

- The **init-only constraint** promises (hence, requires) that once the location has been initialized, its contents never change. Namely, the contents are initialized before any access, and after initialization, no value may be stored in the location. The contents are always identical to the initialized value (see clause 8.2.3). This constraint, while logically applicable to any location, shall only be placed on fields (static or instance) of compound types.
- The **literal constraint** promises that the value of the location is actually a fixed value of a built-in type. The value is specified as part of the constraint. Compilers are required to replace all references to the location with its value, and the VES therefore need not allocate space for the location. This constraint, while logically applicable to any location, shall only be placed on static fields of compound types. Fields that are so marked are not permitted to be referenced from CIL (they shall be in-lined to their constant value at compile time), but are available using Reflection and tools that directly deal with the metadata.

CLS Rule 13: The value of a literal static is specified through the use of field initialization metadata (see Partition II). A CLS compliant literal must have a value specified in field initialization metadata that is of exactly the same type as the literal (or of the underlying type, if that literal is an **enum**).

Note:

CLS (consumer): must be able to read field initialization metadata for static literal fields and inline the value specified when referenced. Consumers may assume that the type of the field initialization metadata is exactly the same as the type of the literal field, i.e., a consumer tool need not implement conversions of the values.

CLS (extender): must avoid producing field initialization metadata for static literal fields in which the type of the field initialization metadata does not exactly match the type of the field.

CLS (framework): should avoid the use of syntax specifying a value of a literal that requires conversion of the value. Note that compilers may do the conversion themselves before persisting the field initialization metadata resulting in a CLS compliant framework, but frameworks are encouraged not to rely on such implicit conversions.

Note: It might seem reasonable to provide a volatile constraint on a location that would require that the value stored in the location not be cached between accesses. Instead, CIL includes a **volatile.** prefix to certain instructions to specify that the value neither be cached nor computed using an existing cache. Such a constraint may be encoded using a custom attribute (see Section 9.7), although this standard does not specify such an attribute.

8.6.1.3 Local Signatures

A **local signature** specifies the contract on a local variable allocated during the running of a method. A local signature contains a full location signature, plus it may specify one additional constraint:

The **byref** constraint states that the content of the corresponding location is a **managed pointer**. A managed pointer may point to a local variable, parameter, field of a compound type, or element of an array. However, when a call crosses a remoting boundary (see Section 12.5) a conforming implementation may use a copy-in/copy-out mechanism instead of a managed pointer. Thus programs shall not rely on the aliasing behavior of true pointers.

In addition, there is one special local signature. The **typed reference** local variable signature states that the local will contain both a managed pointer to a location and a runtime representation of the type that may be stored at that location. A typed reference signature is similar to a byref constraint, but while the byref specifies the type as part of the byref constraint (and hence as part of the type description), a typed reference provides the type information dynamically. A typed reference is a full signature in itself and can not be combined with other constraints. In particular, it is not possible to specify a **byref** whose type is **typed reference**.

The typed reference signature is actually represented as a built-in value type, like the integer and floating point types. In the Base Class Library (see Partition IV) the type is known as **System.TypedReference** and in the assembly language used in Partition II it is designated by the keyword **typedref**. This type shall only be used for parameters and local variables. It shall not be boxed, nor shall it be used as the type of a field, element of an array, return value, etc.

CLS Rule 14: Typed references are not CLS-compliant.

Note:

CLS (consumer): there is no need to accept this type.

CLS (extender): there is no need to provide syntax to define this type or to extend interfaces or classes that use this type.

CLS (framework): this type shall not appear in exposed members.

8.6.1.4 Parameter Signatures

Parameter signatures define constraints on how an individual value is passed as part of a method invocation. Parameter signatures are declared by method definitions. Any valid local signature is a valid parameter signature.

8.6.1.5 Method Signatures

Method signatures are composed of

- a calling convention,
- a list of zero or more parameter signatures, one for each parameter of the method,
- and a type signature for the result value if one is produced.

Method signatures are declared by method definitions. Only one constraint can be added to a method signature in addition to those of parameter signatures:

- The **varargs** constraint may be included to indicate that all arguments past this point are optional. When it appears, the calling convention shall be one that supports variable argument lists.

Method signatures are used in two different ways. They are used as part of a method definition and as a description of a calling site when calling through a function pointer. In this latter case, the method signature indicates

- the calling convention (which may include platform-specific calling conventions)
- the type of all the argument values that are being passed,
- if needed, a varargs marker indicating where the fixed parameter list ends and the variable parameter list begins

When used as part of a method definition, the varargs constraint is represented by the choice of calling convention.

CLS Rule 15: The varargs constraint is not part of the CLS, and the only calling convention supported by the CLS is the standard managed calling convention.

Note:

CLS (consumer): there is no need to accept methods with variable argument lists or unmanaged calling convention.

CLS (extender): there is no need to provide syntax to declare varargs methods or unmanaged calling conventions.

CLS (framework): neither varargs methods nor methods with unmanaged calling conventions may be exposed externally.

8.7 Assignment Compatibility

The constraints in the type signature and the location signature affect assignment compatibility of a value to a location. Assignment compatibility of a value (described by a type signature) to a location (described by a location signature) is defined as follows:

One of the types supported by the exact type of the value is the same as the type in the location signature.

This allows, for example, an instance of a class that inherits from a base class (hence supports the base class's type contract) to be stored into a location whose type is that of the base class.

8.8 Type Safety and Verification

Since types specify contracts, it is important to know whether a given implementation lives up to these contracts. An implementation that lives up to the enforceable part of the contract (the named signatures) is said to be **typesafe**. An important part of the contract deals with restrictions on the visibility and accessibility of named items as well as the mapping of names to implementations and locations in memory.

Typesafe implementations only store values described by a type signature in a location that is assignment compatible with the location signature of the location (see clause 8.6.1). Typesafe implementations never apply an operation to a value that is not defined by the exact type of the value. Typesafe implementations only access locations that are both visible and accessible to them. In a typesafe implementation, the exact type of a value cannot change.

Verification is a mechanical process of examining an implementation and asserting that it is typesafe. Verification is said to succeed if the process proves that an implementation is typesafe. Verification is said to fail if that process does not prove the type safety of an implementation. Verification is necessarily conservative: it may report failure for a typesafe implementation, but it never reports success for an implementation that is not typesafe. For example, most verification processes report implementations that do pointer-based arithmetic as failing verification, even if the implementation is in fact typesafe.

There are many different processes that can be the basis of verification. The simplest possible process simply says that all implementations are not typesafe. While correct and efficient this is clearly not particularly useful. By spending more resources (time and space) a process can correctly identify more typesafe implementations. It has been proven, however, that no mechanical process can in finite time and with no errors correctly identify all implementations as either typesafe or not typesafe. The choice of a particular verification process is thus a matter of engineering, based on the resources available to make the decision and the importance of detecting the typesafety of different programming constructs.

8.9 Type Definers

Type definers construct a new type from existing types. **Implicit types** (e.g., built-in types, arrays, and pointers including function pointers) are defined when they are used. The mention of an implicit type in a signature is in and of itself a complete definition of the type. Implicit types allow the VES to manufacture instances with a standard set of members, interfaces, etc. Implicit types need not have user-supplied names.

All other types shall be explicitly defined using an explicit type definition. The explicit type definers are:

- interface definitions – used to define interface types
- class definitions – used to define:

- object types
- value types and their associated boxed types

Note: While class definitions always define class types, not all class types require a class definition. Array types and pointer types, which are implicitly defined, are also class types. See clause 8.2.3.

Similarly, not all types defined by a class definition are object types. Array types, explicitly defined object types, and boxed types are object types. Pointer types, function pointer types, and value types are not object types. See clause 8.2.3.

8.9.1 Array Types

An **array type** shall be defined by specifying the element type of the array, the **rank** (number of dimensions) of the array, and the upper and lower bounds of each dimension of the array. Hence, no separate definition of the array type is needed. The bounds (as well as indices into the array) shall be signed integers. While the actual bounds for each dimension are known at runtime, the signature may specify the information that is known at compile time: no bounds, a lower bound, or both an upper and lower bound.

Array elements shall be laid out within the array object in row-major order, i.e. the elements associated with the rightmost array dimension shall be laid out contiguously from lowest to highest index. The actual storage allocated for each array element may include platform-specific padding.

Values of an array type are objects; hence an array type is a kind of object type (see clause 8.2.3). Array objects are defined by the CTS to be a repetition of locations where values of the array element type are stored. The number of repeated values is determined by the rank and bounds of the array.

Only type signatures, not location signatures, are allowed as array element types.

Exact array types are created automatically by the VES when they are required. Hence, the operations on an array type are defined by the CTS. These generally are: allocating the array based on size and lower bound information, indexing the array to read and write a value, computing the address of an element of the array (a managed pointer), and querying for the rank, bounds, and the total number of values stored in the array.

CLS Rule 16: Arrays shall have elements with a CLS-compliant type and all dimensions of the array shall have lower bounds of zero. Only the fact that an item is an array and the element type of the array shall be required to distinguish between overloads. When overloading is based on two or more array types the element types shall be named types.

Note: so-called “jagged arrays” are CLS-compliant, but when overloading multiple array types they are one-dimensional, zero-based arrays of type System.Array.

CLS (consumer): there is no need to support arrays of non-CLS types, even when dealing with instances of **System.Array**. Overload resolution need not be aware of the full complexity of array types. Programmers should have access to the Get, Set, and Address methods on instances of System.Array if there is no language syntax for the full range of array types.

CLS (extender): there is no need to provide syntax to define non-CLS types of arrays or to extend interfaces or classes that use non-CLS array types. Shall provide access to the type `System.Array`, but may assume that all instances will have a CLS-compliant type. While the full array signature must be used to override an inherited method that has an array parameter, the full complexity of array types need not be made visible to programmers. Programmers should have access to the Get, Set, and Address methods on instances of System.Array if there is no language syntax for the full range of array types.

CLS (framework): non-CLS array types shall not appear in exposed members. Where possible, use only one-dimensional, zero-based arrays (vectors) of simple named types, since these are supported in the widest range of programming languages. Overloading on array types should be avoided, and when used shall obey the restrictions.

Array types form a hierarchy, with all array types inheriting from the type `System.Array`. This is an abstract class (see clause 8.9.6.2) that represents all arrays regardless of the type of their elements, their rank, or their upper and lower bounds. The VES creates one array type for each distinguishable array type. In general, array types are only distinguished by the type of their elements and their rank. The VES, however, treats single dimensional, zero-based arrays (also known as **vectors**) specially. Vectors are also distinguished by the type of their elements, but a vector is distinct from a single-dimensional array of the same element type that has a non-zero lower bound. Zero-dimensional arrays are not supported.

Consider the following examples, using the syntax of CIL as described in Partition II:

Table 2: Array Examples

Static specification of type	**Actual type constructed**	**Allowed in CLS?**
`int32[]`	vector of int32	Yes
`int32[0..5]`	vector of int32	Yes
`int32[1..5]`	array, rank 1, of int32	No
`int32[,]`	array, rank 2, of int32	Yes
`int32[0..3, 0..5]`	array, rank 2, of int32	Yes
`int32[0.., 0..]`	array, rank 2, of int32	Yes
`int32[1.., 0..]`	array, rank 2, of int32	No

8.9.2 Unmanaged Pointer Types

An **unmanaged pointer type** (also known simply as a “pointer type”) is defined by specifying a location signature for the location the pointer references. Any signature of a pointer type includes this location signature. Hence, no separate definition of the pointer type is needed.

While pointer types are Reference Types, values of a pointer type are not objects (see clause 8.2.3), and hence it is not possible, given a value of a pointer type, to determine its exact type. The CTS provides two typesafe operations on pointer types: one to load the value from the location referenced by the pointer and the other to store an assignment compatible value into that location. The CTS also provides three operations on pointer types (byte-based address arithmetic): adding and subtracting integers from pointers, and subtracting one pointer from another. The results of the first two operations are pointers to the same type signature as the original pointer. See Partition III for details.

CLS Rule 17: Unmanaged pointer types are not CLS-compliant.

Note:

CLS (consumer): there is no need to support unmanaged pointer types.

CLS (extender): there is no need to provide syntax to define or access unmanaged pointer types.

CLS (framework): unmanaged pointer types shall not be externally exposed.

8.9.3 Delegates

Delegates are the object-oriented equivalent of function pointers. Unlike function pointers, delegates are object-oriented, type-safe, and secure. Delegates are created by defining a class that derives from the base type `System.Delegate` (see Partition IV). Each delegate type shall provide a method named **Invoke** with appropriate parameters, and each instance of a delegate forwards calls to its **Invoke** method to a compatible static or instance method on a particular object. The object and method to which it delegates are chosen when the delegate instance is created.

In addition to an instance constructor and an **Invoke** method, delegates may optionally have two additional methods: **BeginInvoke** and **EndInvoke**. These are used for asynchronous calls.

While, for the most part, delegates appear to be simply another kind of user defined class, they are tightly controlled. The implementations of the methods are provided by the VES, not user code. The only additional members that may be defined on delegate types are static or instance methods.

8.9.4 Interface Type Definition

An **interface definition** defines an interface type. An interface type is a named group of methods, locations and other contracts that shall be implemented by any object type that supports the interface contract of the same name. An interface definition is always an incomplete description of a value, and as such can never define a class type or an exact type, nor can it be an object type.

Zero or more object types can support an interface type, and only object types can support an interface type. An interface type may require that objects that support it shall also support other (specified) interface types. An object type that supports the named interface contract shall provide a complete implementation of the methods, locations, and other contracts specified (but not implemented by) the interface type. Hence, a value of an object type is also a value of all of the interface types the object type supports. Support for an interface contract is declared, never inferred, i.e. the existence of implementations of the methods, locations, and other contracts required by the interface type does not imply support of the interface contract.

CLS Rule 18: CLS-compliant interfaces shall not require the definition of non-CLS compliant methods in order to implement them.

Note:

CLS (consumer): there is no need to deal with such interfaces.

CLS (extender): need not provide a mechanism for defining such interfaces..

CLS (framework): shall not expose any non-CLS compliant methods on interfaces it defines for external use.

Interfaces types are necessarily incomplete since they say nothing about the representation of the values of the interface type. For this reason, an interface type definition shall not provide field definitions for values of the interface type (i.e. instance fields), although it may declare static fields (see clause 8.4.3).

Similarly, an interface type definition shall not provide implementations for any methods on the values of its type. However, an interface type definition may and usually does define method contracts (method name and method signature) that shall be implemented by supporting types. An interface type definition may define and implement static methods (see clause 8.4.3) since static methods are associated with the interface type itself rather than with any value of the type.

Interfaces may have static or virtual methods, but shall not have instance methods.

CLS Rule 19: CLS-compliant interfaces shall not define static methods, nor shall they define fields.

Note:

CLS-compliant interfaces may define properties, events, and virtual methods.

CLS (consumer): need not accept interfaces that violate these rules.

CLS (extender): need not provide syntax to author interfaces that violate these rules.

CLS (framework): shall not externally expose interfaces that violate these rules. Where static methods, instance methods, or fields are required a separate class may be defined that provides them.

Interface types may also define event and property contracts that shall be implemented by object types that support the interface. Since event and property contracts reduce to sets of method contracts (Section 8.6), the above rules for method definitions apply. For more information, see clause 8.11.4 and clause 8.11.3.

Interface type definitions may specify other interface contracts that implementations of the interface type are required to support. See clause 8.9.11 for specifics.

An interface type is given a visibility attribute, as described in clause 8.5.3, that controls from where the interface type may be referenced. An interface type definition is separate from any object type definition that supports the interface type. Hence, it is possible, and often desirable, to have a different visibility for the interface type and the implementing object type. However, since accessibility attributes are relative to the implementing type rather than the interface itself, all members of an interface shall have public accessibility, and no security permissions may be attached to members or to the interface itself.

8.9.5 Class Type Definition

All types other than interfaces, and those types for which a definition is automatically supplied by the CTS, are defined by **class definitions**. A **class type** is a complete specification of the representation of the values of the class type and all of the contracts (class, interface, method, property, and event) that are supported by the class type. Hence, a class type is an exact type. A class definition, unless it specifies that the class is an **abstract object type**, not only defines the class type: it also provides implementations for all of the contracts supported by the class type.

A class definition, and hence the implementation of the class type, always resides in some assembly. An assembly is a configured set of loadable code modules and other resources that together implement a unit of functionality.

Note: While class definitions always define class types, not all class types require a class definition. Array types and pointer types, which are implicitly defined, are also class types. See clause 8.2.3.

An explicit class definition is used to define:

- An object type (see clause 8.2.3).
- A value type and its associated boxed type (see clause 8.2.4).

An explicit class definition:

- Names the class type.
- Implicitly assigns the class type name to a scope, i.e. the assembly that contains the class definition, (see clause 8.5.2).
- Defines the class contract of the same name (see Section 8.6).
- Defines the representations and valid operations of all values of the class type using member definitions for the fields, methods, properties, and events (see Section 8.11).
- Defines the static members of the class type (see Section 8.11).
- Specifies any other interface and class contracts also supported by the class type.
- Supplies implementations for member and interface contracts supported by the class type.
- Explicitly declares a visibility for the type, either public or assembly (see clause 8.5.3).
- May optionally specify a method to be called to initialize the type.

The semantics of when, and what triggers execution of such type initialization methods, is as follows:

1. A type may have a type-initializer method, or not.
2. A type may be specified as having a relaxed semantic for its type-initializer method (for convenience below, we call this relaxed semantic **BeforeFieldInit**)
3. If marked **BeforeFieldInit** then the type’s initializer method is executed at, or sometime before, first access to any static field defined for that type
4. If *not* marked **BeforeFieldInit** then that type’s initializer method is executed at (i.e., is triggered by):
    - o first access to any static or instance field of that type, or
    - o first invocation of any static, instance or virtual method of that type
5. Execution of any type's initializer method will *not* trigger automatic execution of any initializer methods defined by its base type, nor of any interfaces that the type implements

Note: **BeforeFieldInit** behavior is intended for initialization code with no interesting side-effects, where exact timing does not matter. Also, under **BeforeFieldInit** semantics, type initializers are allowed to be executed *at or before* first access to any static field of that Type -- at the discretion of the CLI

If a language wishes to provide more rigid behavior -- e.g. type initialization automatically triggers execution of parents initializers, in a top-to-bottom order, then it can do so by either:

- defining hidden static fields and code in each class constructor that touches the hidden static field of its parent and/or interfaces it implements, or
- by making explicit calls to `System.Runtime.CompilerServices.Runtime-Helpers.RunClassConstructor` (see Partition IV).

8.9.6 Object Type Definitions

All objects are instances of an **object type**. The object type of an object is set when the object is created and it is immutable. The object type describes the physical structure of the instance and the operations that are allowed on it. All instances of the same object type have the same structure and the same allowable operations. Object types are explicitly declared by a class type definition, with the exception of Array types, which are intrinsically provided by the VES.

8.9.6.1 Scope and Visibility

Since object type definitions are class type definitions, object type definitions implicitly specify the scope of the name of object type to be the assembly that contains the object type definition, see clause 8.5.2. Similarly, object type definitions shall also explicitly state the visibility attribute of the object type (either **public** or **assembly**); see clause 8.5.3.

8.9.6.2 Concreteness

An object type may be marked as **abstract** by the object type definition. An object type that is not marked **abstract** is by definition **concrete**. Only object types may be declared as abstract. Only an abstract object type is allowed to define method contracts for which the type or the VES does not also provide the implementation. Such method contracts are called abstract methods (see Section 8.11). All methods on an abstract class need not be abstract.

It is an error to attempt to create an instance of an abstract object type, whether or not the type has abstract methods. An object type that derives from an abstract object type may be concrete if it provides implementations for any abstract methods in the base object type and is not itself marked as abstract. Instances may be made of such a concrete derived class. Locations may have an abstract type, and instances of a concrete type that derives from the abstract type may be stored in them.

8.9.6.3 Type Members

Object type definitions include member definitions for all of the members of the type. Briefly, members of a type include fields into which values are stored, methods that may be invoked, properties that are available, and events that may be raised. Each member of a type may have attributes as described in Section 8.4.

- Fields of an object type specify the representation of values of the object type by specifying the component pieces from which it is composed (see clause 8.4.1). Static fields specify fields associated with the object type itself (see clause 8.4.3). The fields of an object type are named and they are typed via location signatures. The names of the members of the type are scoped to the type (see clause 8.5.2). Fields are declared using a field definition (see clause 8.11.2).
- Methods of an object type specify operations on values of the type (see clause 8.4.2). Static methods specify operations on the type itself (see clause 8.4.3). Methods are named and they have a method signature. The names of methods are scoped to the type (see clause 8.5.2). Methods are declared using a method definition (see clause 8.11.1).
- Properties of an object type specify named values that are accessible via methods that read and write the value. The name of the property is the grouping of the methods; the methods themselves are also named and typed via method signatures. The names of properties are scoped to the type (see clause 8.5.2). Properties are declared using a property definition (see clause 8.11.3).

- Events of an object type specify named state transitions in which subscribers may register/unregister interest via accessor methods. When the state changes, the subscribers are notified of the state transition. The name of the event is the grouping of the accessor methods; the methods themselves are also named and typed via method signatures. The names of events are scoped to the type (see clause 8.5.2). Events are declared using an event definition (see clause 8.11.4).

8.9.6.4 Supporting Interface Contracts

Object type definitions may declare that they support zero or more interface contracts. Declaring support for an interface contract places a requirement on the implementation of the object type to fully implement that interface contract. Implementing an interface contract always reduces to implementing the required set of methods, i.e. the methods required by the interface type.

The different types that the object type implements, i.e. the object type and any implemented interface types, are each a separate logical grouping of named members. If a class `Foo` implements an interface `IFoo` and `IFoo` declares a member method `int a()` and the class also declares a member method `int a()`, there are two members, one in the `IFoo` interface type and one in the `Foo` class type. An implementation of `Foo` will provide an implementation for both, potentially shared.

Similarly, if a class implements two interfaces `IFoo` and `IBar` each of which defines a method `int a()` the class will supply two method implementations, one for each interface, although they may share the actual code of the implementation.

CLS Rule 20: CLS-compliant classes, value types, and interfaces shall not require the implementation of non-CLS-compliant interfaces.

Note:

CLS (consumer): need not accept classes, value types or interfaces that violate this rule.

CLS (extender): need not provide syntax to author classes, value types, or interfaces that violate this rule.

CLS (framework): shall not externally expose classes, value types, or interfaces that violate this rule.

8.9.6.5 Supporting Class Contracts

Object type definitions may declare support for one other class contract. Declaring support for another class contract is synonymous with object type inheritance (see clause 8.9.9).

8.9.6.6 Constructors

New values of an object type are created via **constructors**. Constructors shall be instance methods, defined via a special form of method contract, which defines the method contract as a constructor for a particular object type. The constructors for an object type are part of the object type definition. While the CTS and VES ensure that only a properly defined constructor is used to make new values of an object type, the ultimate correctness of a newly constructed object is dependent on the implementation of the constructor itself.

Object types shall define at least one constructor method, but that method need not be public. Creating a new value of an object type by invoking a constructor involves the following steps in order:

1. Space for the new value is allocated in managed memory.
2. VES data structures of the new value are initialized and user-visible memory is zeroed.
3. The specified constructor for the object type is invoked.

Inside the constructor, the object type may do any initialization it chooses (possibly none).

CLS Rule 21: An object constructor shall call some class constructor of its base class before any access occurs to inherited instance data. This does not apply to value types, which need not have constructors.

CLS Rule 22: An object constructor shall not be called except as part of the creation of an object, and an object shall not be initialized twice.

Note:

CLS (consumer): Shall provide syntax for choosing the constructor to be called when an object is created.

CLS (extender): Shall provide syntax for defining constructor methods with different signatures. May issue a compiler error if the constructor does not obey these rules.

CLS (framework): May assume that object creation includes a call to one of the constructors, and that no object is initialized twice. `System.MemberwiseClone` (see Partition IV) and deserialization (including object remoting) may not run constructors.

8.9.6.7 Finalizers

A class definition that creates an object type may supply an instance method to be called when an instance of the class is no longer accessible. The class `System.GC` (see Partition IV) provides limited control over the behavior of finalizers through the methods `SuppressFinalize` and `ReRegisterForFinalize`. Conforming implementations of the CLI may specify and provide additional mechanisms that affect the behavior of finalizers.

A conforming implementation of the CLI shall not automatically call a finalizer twice for the same object unless

- there has been an intervening call to `ReRegisterForFinalize` (not followed by a call to `SuppressFinalize`), or
- the program has invoked an implementation-specific mechanism that is clearly specified to produce an alteration to this behavior

Rationale: *Programmers expect that finalizers are run precisely once on any given object unless they take an explicit action to cause the finalizer to be run multiple times.*

It is legal to define a finalizer for a Value Type. That finalizer however will only be run for *boxed* instances of that Value Type.

Note: Since programmers may depend on finalizers to be called, the CLI should make every effort to ensure that finalizers are called, before it shuts down, for all objects that have not been exempted from finalization by a call to `SuppressFinalize`. The implementation should specify any conditions under which this behavior cannot be guaranteed.

Note: Since resources may become exhausted if finalizers are not called expeditiously, the CLI should ensure that finalizers are called soon after the instance becomes inaccessible. While relying on memory pressure to trigger finalization is acceptable, implementers should consider the use of additional metrics.

8.9.7 Value Type Definition

Not all types defined by a class definition are object types (see clause 8.2.3); in particular, value types are not object types but they are defined using a class definition. A class definition for a value type defines both the (unboxed) value type and the associated boxed type (see clause 8.2.4). The members of the class definition define the representation of both:

1. When a non-static method (i.e. an instance or virtual method) is called on the value type its **this** pointer is a managed reference to the instance, whereas when the method is called on the associated boxed type the **this** pointer is an object reference.

Instance methods on value types receive a **this** pointer that is a managed pointer to the unboxed type whereas virtual methods (including those on interfaces implemented by the value type) receive an instance of the boxed type.

1. Value types do not support interface contracts, but their associated boxed types do.
2. A value type does not inherit; rather the base type specified in the class definition defines the base type of the boxed type.

3. The base type of a boxed type shall not have any fields.
4. Unlike object types, instances of value types do not require a constructor to be called when an instance is created. Instead, the verification rules require that verifiable code initialize instances to zero (null for object fields).

8.9.8 Type Inheritance

Inheritance of types is another way of saying that the derived type guarantees support for all of the type contracts of the base type. In addition, the derived type usually provides additional functionality or specialized behavior. A type inherits from a base type by implementing the type contract of the base type. An interface type inherits from zero or more other interfaces. Value types do not inherit, although the associated boxed type is an object type and hence inherits from other types

The derived class type shall support all of the supported interfaces contracts, class contracts, event contracts, method contracts, and property contracts of its base type. In addition, all of the locations defined by the base type are also defined in the derived type. The inheritance rules guarantee that code that was compiled to work with a value of a base type will still work when passed a value of the derived type. Because of this, a derived type also inherits the implementations of the base type. The derived type may extend, override, and/or hide these implementations.

8.9.9 Object Type Inheritance

With the sole exception of `System.Object`, which does not inherit from any other object type, all object types shall either explicitly or implicitly declare support for (inherit from) exactly one other object type. The graph of the inherits-relation shall form a singly rooted tree with `System.Object` at the base, i.e. all object types eventually inherit from the type `System.Object`.

An object type declares it shall not be used as a base type (be inherited from) by declaring that it is a **sealed** type.

CLS Rule 23: `System.Object` is CLS-compliant. Any other CLS-compliant class shall inherit from a CLS-compliant class.

Arrays are object types and as such inherit from other object types. Since arrays object types are manufactured by the VES, the inheritance of arrays is fixed. See clause 8.9.1.

8.9.10 Value Type Inheritance

Value Types, in their unboxed form, do not inherit from any type. Boxed value types shall inherit directly from System.ValueType unless they are enumerations, in which case they shall inherit from System.Enum. Boxed value types shall be sealed.

Logically, the boxed type corresponding to a value type

- Is an object type.
- Will specify which object type is its base type, i.e. the object type from which it inherits.
- Will have a base type that has no fields defined.
- Will be **sealed** to avoid dealing with the complications of value slicing

The more restrictive rules specified here allow for more efficient implementation without severely compromising functionality.

8.9.11 Interface Type Inheritance

Interface types may inherit from multiple interface types, i.e. an interface contract may list other interface contracts that shall also be supported. Any type that implements support for an interface type shall also implement support for all of the inherited interface types. This is different from object type inheritance in two ways.

- Object types form a single inheritance tree; interface types do not.

- Object type inheritance specifies how implementations are inherited; interface type inheritance does not, since interfaces do not define implementation. Interface type inheritance specifies additional contracts that an implementing object type shall support.

To highlight the last difference, consider an interface, `IFoo`, that has a single method. An interface, `IBar`, which inherits from it is requiring that any object type that supports `IBar` also support `IFoo`. It does not say anything about which methods `IBar` itself will have.

8.10 Member Inheritance

Only object types may inherit implementations, hence only object types may inherit members (see clause 8.9.8). Interface types, while they do inherit from other interface types, only inherit the requirement to implement method contracts, never fields or method implementations.

8.10.1 Field Inheritance

A derived object type inherits all of the non-static fields of its base object type. This allows instances of the derived type to be used wherever instances of the base type are expected (the shapes, or layouts, of the instances will be the same). Static fields are not inherited. Just because a field exists does not mean that it may be read or written. The type visibility, field accessibility, and security attributes of the field definition (see clause 8.5.3) determine if a field is accessible to the derived object type.

8.10.2 Method Inheritance

A derived object type inherits all of the instance and virtual methods of its base object type. It does not inherit constructors or static methods. Just because a method exists does not mean that it may be invoked. It shall be accessible via the typed reference that is being used by the referencing code. The type visibility, method accessibility, and security attributes of the method definition (see clause 8.5.3) determine if a method is accessible to the derived object type.

A derived object type may hide a non-virtual (i.e. static or instance) method of its base type by providing a new method definition with the same name or same name and signature. Either method may still be invoked, subject to method accessibility rules, since the type that contains the method always qualifies a method reference.

Virtual methods may be marked as **final**, in which case they shall not be overridden in a derived object type. This ensures that the implementation of the method is available, by a virtual call, on any object that supports the contract of the base class that supplied the final implementation. If a virtual method is not final it is possible to demand a security permission in order to override the virtual method, so that the ability to provide an implementation can be limited to classes that have particular permissions. When a derived type overrides a virtual method, it may specify a new accessibility for the virtual method, but the accessibility in the derived class shall permit at least as much access as the access granted to the method it is overriding. See clause 8.5.3.

8.10.3 Property and Event Inheritance

Properties and events are fundamentally constructs of the metadata intended for use by tools that target the CLI and are not directly supported by the VES itself. It is, therefore, the job of the source language compiler and the Reflection library [see Partition IV] to determine rules for name hiding, inheritance, and so forth. The source compiler shall generate CIL that directly accesses the methods named by the events and properties, not the events or properties themselves.

8.10.4 Hiding, Overriding, and Layout

There are two separate issues involved in inheritance. The first is which contracts a type shall implement and hence which member names and signatures it shall provide. The second is the layout of the instance so that an instance of a derived type can be substituted for an instance of any of its base types. Only the non-static fields and the virtual methods that are part of the derived type affect the layout of an object.

The CTS provides independent control over both the names that are visible from a base type (**hiding**) and the sharing of layout slots in the derived class (**overriding**). Hiding is controlled by marking a member in the derived class as either **hide by name** or **hide by name-and-signature**. Hiding is always performed based on the kind of member, that is, derived field names may hide base field names, but not method names, property names, or event names. If a derived member is marked **hide by name**, then members of the same kind in the base class with the same name are not visible in the derived class; if the member is marked **hide by name-and-signature** then only a member of the same kind with exactly the same name and type (for fields) or method signature (for methods) is hidden in the derived class. Implementation of the distinction between these two forms of hiding is provided entirely by source language compilers and the Reflection library; it has no direct impact on the VES itself.

For example:

```
class Base
{ field  int32         A;
  field  System.String A;
  method int32         A();
  method int32         A(int32);
}
class Derived inherits from Base
{ field  int32 A;
  hidebysig method int32 A();
}
```

The member names available in type `Derived` are:

Table 3: Member names

Kind of member	**Type / Signature of member**	**Name of member**
`Field`	`int32`	`A`
`Method`	`() -> int32`	`A`
`Method`	`(int32) -> int32`	`A`

While hiding applies to all members of a type, overriding deals with object layout and is applicable only to instance fields and virtual methods. The CTS provides two forms of member overriding, **new slot** and **expect existing slot**. A member of a derived type that is marked as a new slot will always get a new slot in the object's layout, guaranteeing that the base field or method is available in the object by using a qualified reference that combines the name of the base type with the name of the member and its type or signature. A member of a derived type that is marked as expect existing slot will re-use (i.e. share or override) a slot that corresponds to a member of the same kind (field or method), name, and type if one already exists from the base type; if no such slot exists, a new slot is allocated and used.

The general algorithm that is used for determining the names in a type and the layout of objects of the type is roughly as follows:

- Flatten the inherited names (using the **hide by name** or **hide by name-and-signature** rule) *ignoring* accessibility rules.
- For each new member that is marked "expect existing slot", look to see if an exact match on kind (i.e. field or method), name, and signature exists and use that slot if it is found, otherwise allocate a new slot.
- After doing this for all new members, add these new member-kind/name/signatures to the list of members of this type
- Finally, remove any inherited names that match the new members based on the **hide by name** or **hide by name-and-signature** rules.

8.11 Member Definitions

Object type definitions, interface type definitions, and value type definitions may include member definitions. Field definitions define the representation of values of the type by specifying the substructure of the value. Method definitions define operations on values of the type and operations on the type itself (static methods). Property and event definitions may only be defined on object types. Property and events define named groups of accessor method definitions that implement the named event or property behavior. Nested type declarations define types whose names are scoped by the enclosing type and whose instances have full access to all members of the enclosing class.

Depending on the kind of type definition, there are restrictions on the member definitions allowed.

8.11.1 Method Definitions

Method definitions are composed of a name, a method signature, and optionally an implementation of the method. The method signature defines the calling convention, type of the parameters to the method, and the return type of the method (see clause 8.6.1). The implementation is the code to execute when the method is invoked. A value type or object type may define only one method of a given name and signature. However, a derived object type may have methods that are of the same name and signature as its base object type. See clause 8.10.2 and clause 8.10.4.

The name of the method is scoped to the type (see clause 8.5.2). Methods may be given accessibility attributes (see clause 8.5.3). Methods may only be invoked with arguments that are assignment compatible with the parameters types of the method signature. The return value of the method shall also be assignment compatible with the location in which it is stored.

Methods may be marked as **static**, indicating that the method is not an operation on values of the type but rather an operation associated with the type as a whole. Methods not marked as static define the valid operations on a value of a type. When a non-static method is invoked, a particular value of the type, referred to as **this** or the **this pointer**, is passed as an implicit parameter.

A method definition that does not include a method implementation shall be marked as **abstract.** All non-static methods of an interface definition are abstract. Abstract method definitions are only allowed in object types that are marked as abstract.

A non-static method definition in an object type may be marked as **virtual**, indicating that an alternate implementation may be provided in derived types. All non-static method definitions in interface definitions shall be virtual methods. Virtual method may be marked as **final**, indicating that derived object types are not allowed to override the method implementation.

8.11.2 Field Definitions

Field definitions are composed of a name and a location signature. The location signature defines the type of the field and the accessing constraints, see clause 8.6.1. A value type or object type may define only one field of a given name and type. However, a derived object type may have fields that are of the same name and type as its base object type. See clause 8.10.1 and clause 8.10.4.

The name of the field is scoped to the type (see clause 8.5.2). Fields may be given accessibility attributes, see clause 8.5.3. Fields may only store values that are assignment compatible with the type of the field (see clause 8.3.1).

Fields may be marked as **static**, indicating that the field is not part of values of the type but rather a location associated with the type as a whole. Locations for the static fields are created when the type is loaded and initialized when the type is initialized.

Fields not marked as static define the representation of a value of a type by defining the substructure of the value (see clause 8.4.1). Locations for such fields are created within every value of the type whenever a new value is constructed. They are initialized during construction of the new value. A non-static field of a given name is always located at the same place within every value of the type.

A field that is marked **serializable** is to be serialized as part of the persistent state of a value of the type. This standard does not specify the mechanism by which this is accomplished.

8.11.3 Property Definitions

A property definition defines a named value and the methods that access the value. A property definition defines the accessing contracts on that value. Hence, the property definition specifies which accessing methods exist and their respective method contracts. An implementation of a type that declares support for a property contract shall implement the accessing methods required by the property contract. The implementation of the accessing methods defines how the value is retrieved and stored.

A property definition is always part of either an interface definition or a class definition. The name and value of a property definition is scoped to the object type or the interface type that includes the property definition. While all of the attributes of a member may be applied to a property (accessibility, static, etc.) these are not enforced by the CTS. Instead, the CTS requires that the method contracts that comprise the property shall match the method implementations, as with any other method contract. There are no CIL instructions associated with properties, just metadata.

By convention, properties define a **getter** method (for accessing the current value of the property) and optionally a **setter** method (for modifying the current value of the property). The CTS places no restrictions on the set of methods associated with a property, their names, or their usage.

CLS Rule 24: The methods that implement the `getter` and `setter` methods of a property shall be marked **SpecialName** in the metadata.

CLS Rule 25: The accessibility of a property and of its accessors shall be identical.

CLS Rule 26: A property and its accessors shall all be static, all be virtual, or all be instance.

CLS Rule 27: The type of a property shall be the return type of the `getter` and the type of the last argument of the **setter**. The types of the parameters of the property shall be the types of the parameters to the `getter` and the types of all but the final parameter of the `setter`. All of these types shall be CLS-compliant, and shall not be managed pointers (i.e. shall not be passed by reference).

CLS Rule 28: Properties shall adhere to a specific naming pattern. See Section 10.4. The `SpecialName` attribute referred to in CLS rule 26 shall be ignored in appropriate name comparisons and shall adhere to identifier rules.

Note:

CLS (consumer): Shall ignore the `SpecialName` bit in appropriate name comparisons and shall adhere to identifier rules. Otherwise, no direct support other than the usual access to the methods that define the property.

CLS (extender): Shall ignore the `SpecialName` bit in appropriate name comparisons and shall adhere to identifier rules. Otherwise, no direct support other than the usual access to the methods that define the property. In particular, an extender need not be able to define properties.

CLS (framework): Shall design understanding that not all CLS languages will access the property using special syntax.

8.11.4 Event Definitions

The CTS supports events in precisely the same way that it supports properties (see clause 8.11.3). The conventional methods, however, are different and include means for subscribing and unsubscribing to events as well as for firing the event.

CLS Rule 29: The methods that implement an event shall be marked `SpecialName` in the metadata.

CLS Rule 30: The accessibility of an event and of its accessors shall be identical.

CLS Rule 31: The `add` and `remove` methods for an event shall both either be present or absent.

CLS Rule 32: The `add` and `remove` methods for an event shall each take one parameter whose type defines the type of the event and that shall be derived from `System.Delegate`.

CLS Rule 33: Events shall adhere to a specific naming pattern. See Section 10.4. The `SpecialName` attribute referred to in CLS rule 31 shall be ignored in appropriate name comparisons and shall adhere to identifier rules.

Note:

CLS (consumer): Shall ignore the `SpecialName` bit in appropriate name comparisons and shall adhere to identifier rules. Otherwise, no direct support other than the usual access to the methods that define the event.

CLS (extender): Shall ignore the `SpecialName` bit in appropriate name comparisons and shall adhere to identifier rules. Otherwise, no direct support other than the usual access to the methods that define the event. In particular, an extender need not be able to define events.

CLS (framework): Shall design based on the understanding that not all CLS languages will access the event using special syntax.

8.11.5 Nested Type Definitions

A nested type definition is identical to a top-level type definition, with one exception: a top-level type has a visibility attribute, while the visibility of a nested type is the same as the visibility of the enclosing type. See clause 8.5.3.

9 Metadata

This section and its subsections contain only informative text, with the exception of the CLS rules introduced here and repeated in Chapter 11. The metadata format is specified in Partition II

New types – value types and reference types – are introduced into the CTS via type declarations expressed in **metadata**. In addition, metadata is a structured way to represent all information that the CLI uses to locate and load classes, lay out instances in memory, resolve method invocations, translate CIL to native code, enforce security, and set up runtime context boundaries. Every CLI PE/COFF module (see Partition II) carries a compact metadata binary that is emitted into the module by the CLI-enabled development tool or compiler.

Each CLI-enabled language will expose a language-appropriate syntax for declaring types and members and for annotating them with attributes that express which services they require of the infrastructure. Type imports are also handled in a language-appropriate way, and it is the development tool or compiler that consumes the metadata to expose the types that the developer sees.

Note that the typical component or application developer will not need to be aware of the rules for emitting and consuming CLI metadata. While it may help a developer to understand the structure of metadata, the rules outlined in this section are primarily of interest to tool builders and compiler writers.

9.1 Components and Assemblies

Each CLI component carries the metadata for declarations, implementations, and references specific to that component. Therefore, the component-specific metadata is referred to as **component metadata**, and the resulting component is said to be **self-describing**. In object models such as COM or CORBA, this information is represented by a combination of typelibs, IDL files, DLLRegisterServer, and a myriad of custom files in disparate formats and separate from the actual executable file. In contrast, the metadata is a fundamental part of a CLI component.

Collections of CLI components and other files are packaged together for deployment into **assemblies**, discussed in more detail in a later section. An assembly is a logical unit of functionality that serves as the primary unit of reuse in the CLI. Assemblies establish a name scope for types.

Types declared and implemented in individual components are exported for use by other implementations via the assembly in which the component participates. All references to a type are scoped by the identity of the assembly in whose context the type is being used. The CLI provides services to locate a referenced assembly and request resolution of the type reference. It is this mechanism that provides an isolation scope for applications: the assembly alone controls its composition.

9.2 Accessing Metadata

Metadata is emitted into and read from a CLI module using either direct access to the file format as described in Partition II or through the Reflection library. It is possible to create a tool that verifies a CLI module, including the metadata, during development, based on the specifications supplied in Partition III and Partition II.

When a class is loaded at runtime, the CLI loader imports the metadata into its own in-memory data structures, which can be browsed via the CLI Reflection services. The Reflection services should be considered as similar to a compiler; they automatically walk the inheritance hierarchy to obtain information about inherited methods and fields, they have rules about hiding by name or name-and-signature, rules about inheritance of methods and properties, and so forth.

9.2.1 Metadata Tokens

A metadata token is an implementation dependent encoding mechanism. Partition II describes the manner in which metadata tokens are embedded in various sections of a CLI PE/COFF module. Metadata tokens are embedded in CIL and native code to encode method invocations and field accesses at call sites; the token is used by various infrastructure services to retrieve information from metadata about the reference and the type on which it was scoped in order to resolve the reference.

A metadata token is a typed identifier of a metadata object (type declaration, member declaration, etc.). Given a token, its type can be determined and it is possible to retrieve the specific metadata attributes for that metadata object. However, a metadata token is not a persistent identifier. Rather it is scoped to a specific metadata binary. A metadata token is represented as an index into a metadata data structure, so access is fast and direct.

9.2.2 Member Signatures in Metadata

Every location — including fields, parameters, method return values, and properties — has a type, and a specification for its type is carried in metadata.

A value type describes values that are represented as a sequence of bits. A reference type describes values that are represented as the location of a sequence of bits. The CLI provides an explicit set of built-in types, each of which has a default runtime form as either a value type or a reference type. The metadata APIs may be used to declare additional types, and part of the type specification of a variable encodes the identity of the type as well as which form (value or reference) the type is to take at runtime.

Metadata tokens representing encoded types are passed to CIL instructions that accept a type (**newobj**, **newarray**, **ldtoken**). See the CIL instruction set specification in Partition III.

These encoded type metadata tokens are also embedded in member signatures. To optimize runtime binding of field accesses and method invocations, the type and location signatures associated with fields and methods are encoded into member signatures in metadata. A member signature embodies all of the contract information that is used to decide whether a reference to a member succeeds or fails.

9.3 Unmanaged Code

It is possible to pass data from CLI managed code to unmanaged code. This always involves a transition from managed to unmanaged code, which has some runtime cost, but data can often be transferred without copying. When data must be reformatted the VES provides a reasonable specification of default behavior, but it is possible to use metadata to explicitly require other forms of **marshalling** (i.e. reformatted copying). The metadata also allows access to unmanaged methods through implementation-specific pre-existing mechanisms.

9.4 Method Implementation Metadata

For each method for which an implementation is supplied in the current CLI module, the tool or compiler will emit information used by the CIL-to-native code compilers, the CLI loader, and other infrastructure services. This information includes:

- Whether the code is managed or unmanaged.
- Whether the implementation is in native code or CIL (note that all CIL code is managed).
- The location of the method body in the current module, as an address relative to the start of the module file in which it is located (a **Relative Virtual Address**, or **RVA**). Or, alternatively, the RVA is encoded as 0 and other metadata is used to tell the infrastructure where the method implementation will be found, including:
    - An implementation to be located via the CLI Interoperability Services. See related specifications for details.
    - Forwarding calls through an imported global static method.

9.5 Class Layout

In the general case, the CLI loader is free to lay out the instances of a class in any way it chooses, consistent with the rules of the CTS. However, there are times when a tool or compiler needs more control over the layout. In the metadata, a class is marked with an attribute indicating whether its layout rule is:

- **autolayout**: A class marked "autolayout" indicates that the loader is free to lay out the class in any way it sees fit; any layout information that may have been specified is ignored. This is the default.
- **layoutsequential**: A class marked "layoutsequential" guides the loader to preserve field order as emitted, but otherwise the specific offsets are calculated based on the CLI type of the field; these may be shifted by explicit offset, padding, and/or alignment information.
- **explicitlayout**: A class marked "explicitlayout" causes the loader to ignore field sequence and to use the explicit layout rules provided, in the form of field offsets and/or overall class size or alignment. There are restrictions on legal layouts, specified in Partition II.

It is also possible to specify an overall size for a class. This enables a tool or compiler to emit a value type specification where only the size of the type is supplied. This is useful in declaring CLI built-in types (such as 32 bit integer). It is also useful in situations where the data type of a member of a structured value type does not have a representation in CLI metadata (e.g., C++ bit fields). In the latter case, as long as the tool or compiler controls the layout, and CLI doesn't need to know the details or play a role in the layout, this is sufficient. Note that this means that the VES can move bits around but can't marshal across machines – the emitting tool or compiler will need to handle the marshaling.

Optionally, a developer may specify a packing size for a class. This is layout information that is not often used but it allows a developer to control the alignment of the fields. It is not an alignment specification, per se, but rather serves as a modifier that places a ceiling on all alignments. Typical values are 1, 2, 4, 8, or 16.

For the full specification of class layout attributes, see the classes in `System.Runtime.InteropServices` in Partition IV.

9.6 Assemblies: Name Scopes for Types

An assembly is a collection of resources that are built to work together to deliver a cohesive set of functionality. An assembly carries all of the rules necessary to ensure that cohesion. It is the unit of access to resources in the CLI.

Externally, an assembly is a collection of exported resources, including types. Resources are exported by name. Internally, an assembly is a collection of public (exported) and private (internal to the assembly) resources. It is the assembly that determines which resources are to be exposed outside of the assembly and which resources are accessible only within the current assembly scope. It is the assembly that controls how a reference to a resource, public or private, is mapped onto the bits that implement the resource. For types in particular, the assembly may also supply runtime configuration information. A CLI module can be thought of as a packaging of type declarations and implementations, where the packaging decisions may change under the covers without affecting clients of the assembly.

The identity of a type is its assembly scope and its declared name. A type defined identically in two different assemblies is considered two different types.

Assembly Dependencies: An assembly may depend on other assemblies. This happens when implementations in the scope of one assembly reference resources that are scoped in or owned by another assembly.

- All references to other assemblies are resolved under the control of the current assembly scope. This gives an assembly an opportunity to control how a reference to another assembly is mapped onto a particular version (or other characteristic) of that referenced assembly (although that target assembly has sole control over how the referenced resource is resolved to an implementation).
- It is always possible to determine which assembly scope a particular implementation is running in. All requests originating from that assembly scope are resolved relative to that scope.

From a deployment perspective, an assembly may be deployed by itself, with the assumption that any other referenced assemblies will be available in the deployed environment. Or, it may be deployed with its dependent assemblies.

Manifests: Every assembly has a manifest that declares what files make up the assembly, what types are exported, and what other assemblies are required to resolve type references within the assembly. Just as CLI components are self-describing via metadata in the CLI component, so are assemblies self-describing via their manifests. When a single file makes up an assembly it contains both the metadata describing the types defined in the assembly and the metadata describing the assembly itself. When an assembly contains more than one file with metadata, each of the files describes the types defined in the file, if any, and one of these files also contains the metadata describing the assembly (including the names of the other files, their cryptographic hashes, and the types they export outside of the assembly).

Applications: Assemblies introduce isolation semantics for applications. An application is simply an assembly that has an external entry point that triggers (or causes a hosting environment such as a browser to trigger) the creation of a new Application Domain. This entry point is effectively the root of a tree of request invocations and resolutions. Some applications are a single, self-contained assembly. Others require the availability of other assemblies to provide needed resources. In either case, when a request is resolved to a module to load, the module is loaded into the same Application Domain from which the request originated. It is possible to monitor or stop an application via the Application Domain.

References: A reference to a type always qualifies a type name with the assembly scope within which the reference is to be resolved – that is, an assembly establishes the name scope of available resources. However, rather than establishing relationships between individual modules and referenced assemblies, every reference is resolved through the current assembly. This allows each assembly to have absolute control over how references are resolved. See Partition II.

9.7 Metadata Extensibility

CLI metadata is extensible. There are three reasons this is important:

- The Common Language Specification (CLS) is a specification for conventions that languages and tools agree to support in a uniform way for better language integration. The CLS constrains parts of the CTS model, and the CLS introduces higher-level abstractions that are layered over the CTS. It is important that the metadata be able to capture these sorts of development-time abstractions that are used by tools even though they are not recognized or supported explicitly by the CLI.
- It should be possible to represent language-specific abstractions in metadata that are neither CLI nor CLS language abstractions. For example, it should be possible, over time, to enable languages like C++ to not require separate header files or IDL files in order to use types, methods, and data members exported by compiled modules.
- It should be possible, in member signatures, to encode types and type modifiers that are used in language-specific overloading. For example, to allow C++ to distinguish **int** from **long** even on 32-bit machines where both map to the underlying type **int32**.

This extensibility comes in the following forms:

- Every metadata object can carry custom attributes, and the metadata APIs provide a way to declare, enumerate, and retrieve custom attributes. Custom attributes may be identified by a simple name, where the value encoding is opaque and known only to the specific tool, language, or service that defined it. Or, custom attributes may be identified by a type reference, where the structure of the attribute is self-describing (via data members declared on the type) and any tool including the CLI Reflection services may browse the value encoding.

CLS Rule 34: The CLS only allows a subset of the encodings of custom attributes. The only types that shall appear in these encodings are (see Partition IV): `System.Type, System.String, System.Char, System.Boolean, System.Byte, System.Int16, System.Int32, System.Int64, System.Single, System.Double,` and any enumeration type based on a CLS-compliant base integer type.

Note:

CLS (consumer): Shall be able to read attributes encoded using the restricted scheme.

CLS (extender): Must meet all requirements for CLS consumer and be able to author new classes and new attributes. Shall be able to attach attributes based on existing attribute classes to any metadata that is emitted. Shall implement the rules for the `System.AttributeUsageAttribute` (see Partition IV).

CLS (framework): Shall externally expose only attributes that are encoded within the CLS rules and following the conventions specified for `System.AttributeUsageAttribute`

- In addition to CTS type extensibility, it is possible to emit custom modifiers into member signatures (see Types in Partition II). The CLI will honor these modifiers for purposes of method overloading and hiding, as well as for binding, but will not enforce any of the language-specific semantics. These modifiers can reference the return type or any parameter of a method, or the type of a field. They come in two kinds: **required modifiers** that anyone using the member must understand in order to correctly use it, and **optional modifiers** that may be ignored if the modifier is not understood.

CLS Rule 35: The CLS does not allow publicly visible required modifiers (modreq, see Partition II), but does allow optional modifiers (modopt, see Partition II) they do not understand.

Note:

CLS (consumer): Shall be able to read metadata containing optional modifiers and correctly copy signatures that include them. May ignore these modifiers in type matching and overload resolution. May ignore types that become ambiguous when the optional modifiers are ignored, or that use required modifiers.

CLS (extender): Shall be able to author overrides for inherited methods with signatures that include optional modifiers. Consequently, an extender must be able to copy such modifiers from metadata that it imports. There is no requirement to support required modifiers, nor to author new methods that have any kind of modifier in their signature.

CLS (framework): Shall not use required modifiers in externally visible signatures unless they are marked as not CLS-compliant. Shall not expose two members on a class that differ only by the use of optional modifiers in their signature unless only one is marked CLS-compliant.

9.8 Globals, Imports, and Exports

The CTS does not have the notion of **global statics**: all statics are associated with a particular class. Nonetheless, the metadata is designed to support languages that rely on static data that is stored directly in a PE/COFF file and accessed by its relative virtual address. In addition, while access to managed data and managed functions is mediated entirely through the metadata itself, the metadata provides a mechanism for accessing unmanaged data and unmanaged code.

CLS Rule 36: Global static fields and methods are not CLS-compliant.

Note:

CLS (consumer): Need not support global static fields or methods.

CLS (extender): Need not author global static fields or methods.

CLS (framework): Shall not define global static fields or methods.

9.9 Scoped Statics

The CTS does not include a model for file- or function-scoped static functions or data members. However, there are times when a compiler needs a metadata token to emit into CIL for a scoped function or data member. The metadata allows members to be marked so that they are never visible/accessible outside of the PE/COFF file in which they are declared and for which the compiler guarantees to enforce all access rules.

End informative text

10 Name and Type Rules for the Common Language Specification

10.1 Identifiers

Languages that are either case-sensitive or case-insensitive can support the CLS. Since its rules apply only to items exposed to other languages, **private** members or types that aren't exported from an assembly may use any names they choose. For interoperation, however, there are some restrictions.

In order to make tools work well with a case-sensitive language it is important that the exact case of identifiers be maintained. At the same time, when dealing with non-English languages encoded in Unicode, there may be more than one way to represent precisely the same identifier that includes combining characters. The CLS requires that identifiers obey the restrictions of the appropriate Unicode standard and persist them in Canonical form C, which preserves case but forces combining characters into a standard representation. See CLS Rule 4, in Section 8.5.1.

At the same time, it is important that externally visible names not conflict with one another when used from a case-insensitive programming language. As a result, all identifier comparisons shall be done internally to CLS-compliant tools using the Canonical form KC, which first transforms characters to their case-canonical representation. See CLS Rule 4, in Section 8.5.1.

When a compiler for a CLS-compliant language supports interoperability with a non-CLS-compliant language it must be aware that the CTS and VES perform all comparisons using code-point (i.e. byte-by-byte) comparison. Thus, even though the CLS requires that persisted identifiers be in Canonical form C, references to non-CLS identifiers will have to be persisted using whatever encoding the non-CLS language chose to use. It is a language design issue, not covered by the CTS or the CLS, precisely how this should be handled.

10.2 Overloading

Note: The CTS, while it describes inheritance, object layout, name hiding, and overriding of virtual methods, does not discuss overloading at all. While this is surprising, it arises from the fact that overloading is entirely handled by compilers that target the CTS and not the type system itself. In the metadata, all references to types and type members are fully resolved and include the precise signature that is intended. This choice was made since every programming language has its own set of rules for coercing types and the VES does not provide a means for expressing those rules.

Following the rules of the CTS, it is possible for duplicate names to be defined in the same scope as long as they differ in either kind (field, method, etc.) or signature. The CLS imposes a stronger restriction for overloading methods. Within a single scope, a given name may refer to any number of methods provided they differ in any of the following:

- Number of parameters
- Type of each argument

Notice that the signature includes more information but CLS-compliant languages need not produce or consume classes that differ only by that additional information (see Partition II for the complete list of information carried in a signature):

- Calling convention
- Custom modifiers
- Return type
- Whether a parameter is passed by value or by reference (i.e. as a managed pointer or by-ref)

There is one exception to this rule. For the special names `op_Implicit` and `op_Explicit` described in clause 10.3.3 methods may be provided that differ only by their return type. These are marked specially and may be ignored by compilers that don't support operator overloading.

Properties shall not be overloaded by type (that is, by the return type of their `getter` method), but they may be overloaded with different number or types of indices (that is, by the number and types of the parameters of its **getter** method). The overloading rules for properties are identical to the method overloading rules.

CLS Rule 37: Only properties and methods may be overloaded.

CLS Rule 38: Properties, instance methods, and virtual methods may be overloaded based only on the number and types of their parameters, except the conversion operators named **op_Implicit** and **op_Explicit** which may also be overloaded based on their return type.

Note:

CLS (consumer): May assume that only properties and methods are overloaded, and need not support overloading based on return type unless providing special syntax for operator overloading. If return type overloading isn't supported, then the **op_Implicit** and **op_Explicit** may be ignored since the functionality shall be provided in some other way by a CLS-compliant framework.

CLS (extender): Should not permit the authoring of overloads other than those specified here. It is not necessary to support operator overloading at all, hence it is possible to entirely avoid support for overloading on return type.

CLS (framework): Shall not publicly expose overloading except as specified here. Frameworks authors should bear in mind that many programming languages, including Object-Oriented languages, do not support overloading and will expose overloaded methods or properties through mangled names. Most languages support neither operator overloading nor overloading based on return type, so **op_Implicit** and **op_Explicit** shall always be augmented with some alternative way to gain the same functionality.

10.3 Operator Overloading

CLS-compliant consumer and extender tools are under no obligation to allow defining of operator overloading. CLS-compliant consumer and extender tools do not have to provide a special mechanism to call these methods.

Note: This topic is addressed by the CLS so that

- languages that do provide operator overloading can describe their rules in a way that other languages can understand, and
- languages that do not provide operator overloading can still access the underlying functionality without the addition of special syntax.

Operator overloading is described by using the names specified below, and by setting a special bit in the metadata (**SpecialName**) so that they do not collide with the user's name space. A CLS-compliant producer tool shall provide some means for setting this bit. If these names are used, they shall have precisely the semantics described here.

10.3.1 Unary Operators

Unary operators take one argument, perform some operation on it, and return the result. They are represented as static methods on the class that defines the type of their one operand or their return type. Table 4: Unary Operator Names shows the names that are defined.

Table 4: Unary Operator Names

Name	ISO/IEC 14882:1998 C++ Operator Symbol
`op_Decrement`	`Similar to --`[1]
`op_Increment`	`Similar to ++`[1]
`op_UnaryNegation`	`-` (unary)
`op_UnaryPlus`	`+` (unary)
`op_LogicalNot`	`!`
`op_True`[2]	`Not defined`

op_False[2]	Not defined
op_AddressOf	& (unary)
op_OnesComplement	~
op_PointerDereference	* (unary)

[1] From a pure C++ point of view, the way one must write these functions for the CLI differs in one very important aspect. In C++, these methods must increment or decrement their operand directly, whereas, in CLI, they must not; instead, they simply return the value of their operand +/- 1, as appropriate, without modifying their operand. The operand must be incremented or decremented by the compiler that generates the code for the ++/-- operator, separate from the call to these methods.

[2] The op_True and op_False operators do not exist in C++. They are provided to support tri-state boolean types, such as those used in database languages.

10.3.2 Binary Operators

Binary operators take two arguments, perform some operation and return a value. They are represented as static methods on the class that defines the type of one of their two operands or the return type. Table 5: Binary Operator Names shows the names that are defined.

Table 5: Binary Operator Names

Name	**C++ Operator Symbol**
op_Addition	+ (binary)
op_Subtraction	- (binary)
op_Multiply	* (binary)
op_Division	/
op_Modulus	%
op_ExclusiveOr	^
op_BitwiseAnd	& (binary)
op_BitwiseOr	|
op_LogicalAnd	&&
op_LogicalOr	||
op_Assign	=
op_LeftShift	<<
op_RightShift	>>
op_SignedRightShift	Not defined
op_UnsignedRightShift	Not defined
op_Equality	==
op_GreaterThan	>
op_LessThan	<
op_Inequality	!=
op_GreaterThanOrEqual	>=
op_LessThanOrEqual	<=
op_UnsignedRightShiftAssignment	Not defined
op_MemberSelection	->
op_RightShiftAssignment	>>=
op_MultiplicationAssignment	*=

op_PointerToMemberSelection	->*
op_SubtractionAssignment	-=
op_ExclusiveOrAssignment	^=
op_LeftShiftAssignment	<<=
op_ModulusAssignment	%=
op_AdditionAssignment	+=
op_BitwiseAndAssignment	&=
op_BitwiseOrAssignment	|=
op_Comma	,
op_DivisionAssignment	/=

10.3.3 Conversion Operators

Conversion operators are unary operations that allow conversion from one type to another. The operator method shall be defined as a static method on either the operand or return type. There are two types of conversions:

- An implicit (**widening**) coercion shall not lose any magnitude or precision. These should be provided using a method named `op_Implicit`
- An explicit (**narrowing**) coercion may lose magnitude or precision. These should be provided using a method named `op_Explicit`

Note: Conversions provide functionality that can't be generated in other ways, and many languages will not support the use of the conversion operators through special syntax. Therefore, CLS rules require that the same functionality be made available through an alternate mechanism. Using the more common ToXxx (where Xxx is the target type) and FromYyy (where Yyy is the name of the source type) naming pattern is recommended.

Because these operations may exist on the class of their operand type (so-called "from" conversions) and would therefore differ on their return type only, the CLS specifically allows that these two operators be overloaded based on their return type. The CLS, however, also requires that if this form of overloading is used then the language shall provide an alternate means for providing the same functionality since not all CLS languages will implement operators with special syntax.

CLS Rule 39: If either `op_Implicit` or `op_Explicit` is provided, an alternate means of providing the coercion shall be provided.

Note:

CLS (consumer): Where appropriate to the language design, use the existence of `op_Implicit` and/or **op_Explicit** in choosing method overloads and generating automatic coercions.

CLS (extender): Where appropriate to the language design, implement user-defined implicit or explicit coercion operators using the corresponding `op_Implicit`, `op_Explicit`, `ToXxx`, and/or `FromXxx` methods.

CLS (framework): If coercion operations are supported, they shall be provided as `FromXxx` and `ToXxx`, and optionally `op_Implicit` and `op_Explicit` as well. CLS frameworks are encouraged to provide such coercion operations.

10.4 Naming Patterns

See also Partition V.

While the CTS does not dictate the naming of properties or events, the CLS does specify a pattern to be observed.

For Events:

An individual event is created by choosing or defining a delegate type that is used to signal the event. Then, three methods are created with names based on the name of the event and with a fixed signature. For the examples below we define an event named **`Click`** that uses a delegate type named **`EventHandler`**.

```
EventAdd, used to add a handler for an event

        Pattern: void add_<EventName> (<DelegateType> handler)

        Example: void add_Click (EventHandler handler);

EventRemove, used to remove a handler for an event

        Pattern: void remove_<EventName> (<DelegateType> handler)

        Example: void remove_Click (EventHandler handler);

EventRaise, used to signal that an event has occurred

        Pattern: void family raise_<EventName> (Event e)
```

For Properties:

An individual property is created by deciding on the type returned by its getter method and the types of the getter's parameters (if any). Then, two methods are created with names based on the name of the property and these types. For the examples below we define two properties: **`Name`** takes no parameters and returns a `System.String`, while **`Item`** takes a `System.Object` parameter and returns a `System.Object`. Item is referred to as an indexed property, meaning that it takes parameters and thus may appear to the user as through it were an array with indices

```
PropertyGet, used to read the value of the property

        Pattern: <PropType> get_<PropName> (<Indices>)

        Example: System.String get_Name ();

        Example: System.Object get_Item (System.Object key);

PropertySet, used to modify the value of the property

        Pattern: void set_<PropName> (<Indices>, <PropType>)

        Example: void set_Name (System.String name);

        Example: void set_Item (System.Object key, System.Object value);
```

10.5 Exceptions

The CLI supports an exception handling model, which is introduced in clause 12.4.2. CLS compliant frameworks may define and throw externally visible exceptions, but there are restrictions on the type of objects thrown:

CLS Rule 40: Objects that are thrown shall be of type `System.Exception` or inherit from it. Nonetheless, CLS compliant methods are not required to block the propagation of other types of exceptions.

Note:

CLS (consumer): Need not support throwing or catching of objects that are not of the specified type.

CLS (extender): Must support throwing of objects of type `System.Exception` or a type inheriting from it. Need not support throwing of objects of other types.

CLS (framework): Shall not publicly expose thrown objects that are not of type `System.Exception` or a type inheriting from it.

10.6 Custom Attributes

In order to allow languages to provide a consistent view of custom attributes across language boundaries, the Base Class Library provides support for the following rules defined by the CLS:

CLS Rule 41: Attributes shall be of type `System.Attribute`, or inherit from it.

Note:

CLS (consumer): Need not support attributes that are not of the specified type.

CLS (extender): Must support the authoring of custom attributes.

CLS (framework): Shall not publicly expose attributes that are not of type `System.Attribute` or a type inheriting from it.

The use of a particular attribute class may be restricted in various ways by placing an attribute on the attribute class. The `System.AttributeUsageAttribute` is used to specify these restrictions. The restrictions supported by the `System.AttributeUsageAttribute` are:

- What kinds of constructs (types, methods, assemblies, etc.) may have the attribute applied to them. By default, instances of an attribute class can be applied to any construct. This is specified by setting the value of the `ValidOn` property of `System.AttributeUsageAttribute`. Several constructs may be combined.
- Multiple instances of the attribute class may be applied to a given piece of metadata. By default, only one instance of any given attribute class can be applied to a single metadata item. The `AllowMultiple` property of the attribute is used to specify the desired value.
- Do not inherit the attribute when applied to a type. By default, any attribute attached to a type should be inherited to types that derive from it. If multiple instances of the attribute class are allowed, the inheritance performs a union of the attributes inherited from the parent and those explicitly applied to the child type. If multiple instance are not allowed, then an attribute of that type applied directly to the child overrides the attribute supplied by the parent. This is specified by setting the `Inherited` property of `System.AttributeUsageAttribute` to the desired value.

Note: Since these are CLS rules and not part of the CTS itself, tools are required to specify explicitly the custom attributes they intend to apply to any given metadata item. That is, compilers or other tools that generate metadata must implement the `AllowMultiple` and `Inherit` rules. The CLI does not supply attributes automatically.The usage of attributes in the CLI is further described in Partition II.

11 Collected CLS Rules

The complete set of CLS rules are collected here for reference. Recall that these rules apply only to "externally visible" items – types that are visible outside of their own assembly and members of those types that have `public`, `family`, or `family-or-assembly` accessibility. Furthermore, items may be explicitly marked as CLS-compliant or not using the `System.CLSCompliantAttribute`. The CLS rules apply only to items that are marked as CLS-compliant.

1. CLS rules apply only to those parts of a type that are accessible or visible outside of the defining assembly (see Section 7.3).
2. Members of non-CLS compliant types shall not be marked CLS-compliant. (see clause 7.3.1).
3. The CLS does not include boxed value types (see clause 8.2.4).
4. Assemblies shall follow Annex 7 of Technical Report 15 of the Unicode Standard 3.0 (ISBN 0-201-61633-5) governing the set of characters permitted to start and be included in identifiers, available on-line at http://www.unicode.org/unicode/reports/tr15/tr15-18.html. For CLS purposes, two identifiers are the same if their lowercase mappings (as specified by the Unicode locale-insensitive, 1-1 lowercase mappings) are the same. That is, for two identifiers to be considered different under the CLS they shall differ in more than simply their case. However, in order to override an inherited definition the CLI requires the precise encoding of the original declaration be used (see clause 8.5.1).
5. All names introduced in a CLS-compliant scope shall be distinct independent of kind, except where the names are identical and resolved via overloading. That is, while the CTS allows a single type to use the same name for a method and a field, the CLS does not (see clause 8.5.2).
6. Fields and nested types shall be distinct by identifier comparison alone, even though the CTS allows distinct signatures to be distinguished. Methods, properties, and events that have the same name (by identifier comparison) shall differ by more than just the return type, except as specified in CLS Rule 39 (see clause 8.5.2).
7. The underlying type of an enum shall be a built-in CLS integer type (see clause 8.5.2).
8. There are two distinct kinds of enums, indicated by the presence or absence of the `System.FlagsAttribute` custom attribute. One represents named integer values, the other named bit flags that can be combined to generate an unnamed value. The value of an enum is not limited to the specified values (see clause 8.5.2).
9. Literal static fields of an enum shall have the type of the enum itself (see clause 8.5.2).
10. Accessibility shall not be changed when overriding inherited methods, except when overriding a method inherited from a different assembly with accessibility Family-or-Assembly. In this case the override shall have accessibility family (see clause 8.5.3.2).
11. All types appearing in a signature shall be CLS-compliant (see clause 8.6.1).
12. The visibility and accessibility of types and members shall be such that types in the signature of any member shall be visible and accessible whenever the member itself is visible and accessible. For example, a public method that is visible outside its assembly shall not have an argument whose type is visible only within the assembly (see clause 8.6.1).
13. The value of a literal static is specified through the use of field initialization metadata (see Partition II). A CLS compliant literal must have a value specified in field initialization metadata that is of exactly the same type as the literal (or of the underlying type, if that literal is an **enum**). (see clause 8.6.1.2).
14. Typed references are not CLS-compliant (see clause 8.6.1.3).
15. The varargs constraint is not part of the CLS, and the only calling convention supported by the CLS is the standard managed calling convention (see clause 8.6.1.5).

16. Arrays shall have elements with a CLS-compliant type and all dimensions of the array shall have lower bounds of zero. Only the fact that an item is an array and the element type of the array shall be required to distinguish between overloads. When overloading is based on two or more array types the element types shall be named types. (see clause 8.9.1).
17. Unmanaged pointer types are not CLS-compliant (see clause 8.9.2).
18. CLS-compliant interfaces shall not require the definition of non-CLS compliant methods in order to implement them (see clause 8.9.4).
19. CLS-compliant interfaces shall not define static methods, nor shall they define fields (see clause 8.9.4).
20. CLS-compliant classes, value types, and interfaces shall not require the implementation of non-CLS-compliant interfaces (see clause 8.9.6.4).
21. An object constructor shall call some class constructor of its base class before any access occurs to inherited instance data. This does not apply to value types, which need not have constructors (see clause 8.9.6.6).
22. An object constructor shall not be called except as part of the creation of an object, and an object shall not be initialized twice (see clause 8.9.6.6).
23. `System.Object` is CLS-compliant. Any other CLS-compliant class shall inherit from a CLS-compliant class (see clause 8.9.9).
24. The methods that implement the getter and setter methods of a property shall be marked SpecialName in the metadata (see Partition II) (see clause 8.11.3).
25. The accessibility of a property and of its accessors shall be identical (see clause 8.11.3).
26. A property and its accessors shall all be static, all be virtual, or all be instance (see clause 8.11.3).
27. The type of a property shall be the return type of the `getter` and the type of the last argument of the `setter`. The types of the parameters of the property shall be the types of the parameters to the `getter` and the types of all but the final parameter of the `setter`. All of these types shall be CLS-compliant, and shall not be managed pointers (i.e. shall not be passed by reference) (see clause 8.11.3).
28. Properties shall adhere to a specific naming pattern. See Section 10.4. The SpecialName attribute referred to in CLS rule 24 shall be ignored in appropriate name comparisons and shall adhere to identifier rules (see clause 8.11.3).
29. The methods that implement an event shall be marked SpecialName in the metadata (see Partition II) (see clause 8.11.4).
30. The accessibility of an event and of its accessors shall be identical (see clause 8.11.4).
31. The `add` and `remove` methods for an event shall both either be present or absent (see clause 8.11.4).
32. The `add` and `remove` methods for an event shall each take one parameter whose type defines the type of the event and that shall be derived from `System.Delegate` (see clause 8.11.4).
33. Events shall adhere to a specific naming pattern. See Section 10.4. The SpecialName attribute referred to in CLS rule 29 shall be ignored in appropriate name comparisons and shall adhere to identifier rules (see clause 8.11.4).
34. The CLS only allows a subset of the encodings of custom attributes. The only types that shall appear in these encodings are: `System.Type, System.String, System.Char, System.Boolean, System.Byte, System.Int16, System.Int32, System.Int64, System.Single, System.Double`, and any enumeration type based on a CLS-compliant base integer type (see Section 9.7).
35. The CLS does not allow publicly visible required modifiers (modreq, see Partition II), but does allow optional modifiers (modopt, see Partition II) they do not understand(see Section 9.7).
36. Global static fields and methods are not CLS-compliant (see Section 9.8).

37. Only properties and methods may be overloaded (see Section 10.2).
38. Properties, instance methods, and virtual methods may be overloaded based only on the number and types of their parameters, except the conversion operators named `op_Implicit` and `op_Explicit` which may also be overloaded based on their return type (see Section 10.2).
39. If either `op_Implicit` or `op_Explicit` is overloaded on its return type, an alternate means of providing the coercion shall be provided (see clause 10.3.3).
40. Objects that are thrown shall be of type `System.Exception` or inherit from it (see Section 10.5). Nonetheless, CLS compliant methods are not required to block the propagation of other types of exceptions.
41. Attributes shall be of type `System.Attribute`, or inherit from it (see Section 10.6).

12 Virtual Execution System

The Virtual Execution System (VES) provides an environment for executing managed code. It provides direct support for a set of built-in data types, defines a hypothetical machine with an associated machine model and state, a set of control flow constructs, and an exception handling model.To a large extent, the purpose of the VES is to provide the support required to execute the Common Intermediate Language instruction set (see Partition III).

12.1 Supported Data Types

The CLI directly supports the data types shown in Table 6: Data Types Directly Supported by the CLI. That is, these data types can be manipulated using the CIL instruction set (see Partition III).

Table 6: Data Types Directly Supported by the CLI

Data Type	Description
`int8`	8-bit 2's complement signed value
`unsigned int8`	8-bit unsigned binary value
`int16`	16-bit 2's complement signed value
`unsigned int16`	16-bit unsigned binary value
`int32`	32-bit 2's complement signed value
`unsigned int32`	32-bit unsigned binary value
`int64`	64-bit 2's complement signed value
`unsigned int64`	64-bit unsigned binary value
`float32`	32-bit IEC 60559:1989 floating point value
`float64`	64-bit IEC 60559:1989 floating point value
`native int`	native size 2's complement signed value
`native unsigned int`	native size unsigned binary value, also unmanaged pointer
`F`	native size floating point number (internal to VES, not user visible)
`O`	native size object reference to managed memory
`&`	native size managed pointer (may point into managed memory)

The CLI model uses an evaluation stack. Instructions that copy values from memory to the evaluation stack are "loads"; instructions that copy values from the stack back to memory are "stores". The full set of data types in Table 6: Data Types Directly Supported by the CLI can be represented in memory. However, the CLI supports only a subset of these types in its operations upon values stored on its evaluation stack – int32, int64, native int. In addition the CLI supports an internal data type to represent floating point values on the internal evaluation stack. The size of the internal data type is implementation-dependent. For further information on the treatment of floating-point values on the evaluation stack, see clause 12.1.3 and Partition III. Short numeric values (int8, int16, unsigned int8, unsigned int16) are widened when loaded (memory-to-stack) and narrowed when stored (stack-to-memory). This reflects a computer model that assumes, for numeric and object references, memory cells are 1, 2, 4, or 8 bytes wide but stack locations are either 4 or 8 bytes wide. User-defined value types may appear in memory locations or on the stack and have no size limitation; the only built-in operations on them are those that compute their address and copy them between the stack and memory.

The only CIL instructions with special support for short numeric values (rather than support for simply the 4 or 8 byte integral values) are:

- Load and store instructions to/from memory: **ldelem, ldind, stind, stelem**

- Data conversion: **conv**, **conv.ovf**
- Array creation: **newarr**

The signed integer (int8, int16, int32, int64, and native int) and the respective unsigned integer (unsigned int8, unsigned int16, unsigned int32, unsigned int64, and native unsigned int) types differ only in how the bits of the integer are interpreted. For those operations where an unsigned integer is treated differently from a signed integer (e.g. comparisons or arithmetic with overflow) there are separate instructions for treating an integer as unsigned (e.g. **cgt.un** and **add.ovf.u**).

This instruction set design simplifies CIL-to-native code (eg. JIT) compilers and interpreters of CIL by allowing them to internally track a smaller number of data types. See clause 12.3.2.1.

As described below, CIL instructions do not specify their operand types. Instead, the CLI keeps track of operand types based on data flow and aided by a stack consistency requirement described below. For example, the single **add** instruction will add two integers or two floats from the stack.

12.1.1 Native Size: native int, native unsigned int, O and &

The native-size, or generic, types (native int, native unsigned int, O, and &) are a mechanism in the CLI for deferring the choice of a value's size. These data types exist as CIL types. But the CLI maps each to the native size for a specific processor. (For example, data type I would map to int32 on a Pentium processor, but to int64 on an IA64 processor). So, the choice of size is deferred until JIT compilation or runtime, when the CLI has been initialized and the architecture is known. This implies that field and stack frame offsets are also not known at compile time. For languages like Visual Basic, where field offsets are not computed early anyway, this is not a hardship. In languages like C or C++, where sizes must be known when source code is compiled, a conservative assumption that they occupy 8 bytes is sometimes acceptable (for example, when laying out compile-time storage).

12.1.1.1 Unmanaged Pointers as Type Native Unsigned Int

Rationale: *For languages like C, when compiling all the way to native code, where the size of a pointer is known at compile time and there are no managed objects, the fixed-size unsigned integer types (unsigned int32 or unsigned int64) may serve as pointers. However choosing pointer size at compile time has its disadvantages. If pointers were chosen to be 32 bit quantities at compile time, the code would be restricted to 4 gigabytes of address space, even if it were run on a 64 bit machine. Moreover, a 64 bit CLI would need to take special care so those pointers passed back to 32-bit code would always fit in 32 bits. If pointers were chosen at compile time to be 64 bits, the code would run on a 32 bit machine, but pointers in every data structure would be twice as large as necessary on that CLI.*

For other languages, where the size of a data type need not be known at compile time, it is desirable to defer the choice of pointer size from compile time to CLI initialization time. In that way, the same CIL code can handle large address spaces for those applications that need them, while also being able to reap the size benefit of 32 bit pointers for those applications that do not need a large address space.

The native unsigned int type is used to represent unmanaged pointers with the VES. The metadata allows unmanaged pointers to be represented in a strongly typed manner, but these types are translated into type native unsigned int for use by the VES.

12.1.1.2 Managed Pointer Types: O and &

The **O** datatype represents an object reference that is managed by the CLI. As such, the number of specified operations is severely limited. In particular, references shall only be used on operations that indicate that they operate on reference types (e.g. **ceq** and **ldind.ref**), or on operations whose metadata indicates that references are allowed (e.g. **call**, **ldsfld**, and **stfld**).

The **&** datatype (managed pointer) is similar to the **O** type, but points to the interior of an object. That is, a managed pointer is allowed to point to a field within an object or an element within an array, rather than to point to the 'start' of object or array.

Object references (**O**) and managed pointers (**&**) may be changed during garbage collection, since the data to which they refer may be moved.

Note: In summary, object references, or **O** types, refer to the 'outside' of an object, or to an object as-a-whole. But managed pointers, or **&** types, refer to the interior of an object. The **&** types are sometimes called "by-ref types" in source languages, since passing a field of an object by reference is represented in the VES by using an **&** type to represent the type of the parameter.

In order to allow managed pointers to be used more flexibly, they are also permitted to point to areas that aren't under the control of the CLI garbage collector, such as the evaluation stack, static variables, and unmanaged memory. This allows them to be used in many of the same ways that unmanaged pointers (**U**) are used. Verification restrictions guarantee that, if all code is verifiable, a managed pointer to a value on the evaluation stack doesn't outlast the life of the location to which it points.

12.1.1.3 Portability: Storing Pointers in Memory

Several instructions, including **calli**, **cpblk**, **initblk**, **ldind.***, and **stind.***, expect an address on the top of the stack. If this address is derived from a pointer stored in memory, there is an important portability consideration.

1. Code that stores pointers in a native sized integer or pointer location (types **native int**, **O**, **native unsigned int**, or **&**) is always fully portable.
2. Code that stores pointers in an 8 byte integer (type **int64** or **unsigned int64**) *can* be portable. But this requires that a **conv.ovf.u** instruction be used to convert the pointer from its memory format before its use as a pointer. This may cause a runtime exception if run on a 32-bit machine.
3. Code that uses any smaller integer type to store a pointer in memory (**int8**, **unsigned int8**, **int16**, **unsigned int16**, **int32**, **unsigned int32**) is *never* portable, even though the use of a unsigned int32 or int32 will work correctly on a 32-bit machine.

12.1.2 Handling of Short Integer Data Types

The CLI defines an evaluation stack that contains either 4-byte or 8-byte integers, but a memory model that encompasses in addition 1-byte and 2-byte integers. To be more precise, the following rules are part of the CLI model:

- Loading from 1-byte or 2-byte locations (arguments, locals, fields, statics, pointers) expands to 4-byte values. For locations with a known type (e.g. local variables) the type being accessed determines whether the load sign-extends (signed locations) or zero-extends (unsigned locations). For pointer dereference (**ldind.***), the instruction itself identifies the type of the location (e.g. **ldind.u1** indicates an unsigned location, while **ldind.i1** indicates a signed location).
- Storing into a 1-byte or 2-byte location truncates to fit and will not generate an overflow error. Specific instructions (**conv.ovf.***) can be used to test for overflow before storing.
- Calling a method assigns values from the evaluation stack to the arguments for the method, hence it truncates just as any other store would when the actual argument is larger than the formal argument.
- Returning from a method assigns a value to an invisible return variable, so it also truncates as a store would when the type of the value returned is larger than the return type of the method. Since the value of this return variable is then placed on the evaluation stack, it is then sign-extended or zero-extended as would any other load. Note that this truncation followed by extending is *not* identical to simply leaving the computed value unchanged.

It is the responsibility of any translator from CIL to native machine instructions to make sure that these rules are faithfully modeled through the native conventions of the target machine. The CLI does not specify, for example, whether truncation of short integer arguments occurs at the call site or in the target method.

12.1.3 Handling of Floating Point Datatypes

Floating-point calculations shall be handled as described in IEC 60559:1989. This standard describes encoding of floating point numbers, definitions of the basic operations and conversion, rounding control, and exception handling.

The standard defines special values, **NaN**, (not a number), **+infinity**, and **–infinity**. These values are returned on overflow conditions. A general principle is that operations that have a value in the limit return an appropriate infinity while those that have no limiting value return **NaN**, but see the standard for details.

Note: The following examples show the most commonly encountered cases.

X **rem** 0 = **NaN**
0 * **+infinity** = 0 * **-infinity** = **NaN**
(X / 0) = **+infinity**, if X>0
NaN, if X=0
-infinity, if X < 0
NaN op X = X op **NaN** = **NaN** for all operations
(**+infinity**) + (**+infinity**) = (**+infinity**)
X / (**+infinity**) = 0
X mod (-**infinity**) = -X
(**+infinity**) - (**+infinity**) = NaN

Note: This standard does not specify the behavior of arithmetic operations on denormalized floating point numbers, nor does it specify when or whether such representations should be created. This is in keeping with IEC 60559:1989. In addition, this standard does not specify how to access the exact bit pattern of NaNs that are created, nor the behavior when converting a NaN between 32-bit and 64-bit representation. All of this behavior is deliberately left implementation-specific.

For purposes of comparison, infinite values act like a number of the correct sign but with a very large magnitude when compared with finite values. **NaN** is 'unordered' for comparisons (see **clt**, **clt.un**).

While the IEC 60559:1989 standard also allows for exceptions to be thrown under unusual conditions (such as overflow and invalid operand), the CLI does not generate these exceptions. Instead, the CLI uses the **NaN, +infinity**, and **–infinity** return values and provides the instruction **ckfinite** to allow users to generate an exception if a result is **NaN**, **+infinity**, or **–infinity**.

The rounding mode defined in IEC 60559:1989 shall be set by the CLI to "round to the nearest number," and neither the CIL nor the class library provide a mechanism for modifying this setting. Conforming implementations of the CLI need not be resilient to external interference with this setting. That is, they need not restore the mode prior to performing floating-point operations, but rather may rely on it having been set as part of their initialization.

For conversion to integers, the default operation supplied by the CIL is "truncate towards zero". There are class libraries supplied to allow floating-point numbers to be converted to integers using any of the other three traditional operations (**round** to nearest integer, **floor** (truncate towards **–infinity**), **ceiling** (truncate towards **+infinity**)).

Storage locations for floating point numbers (statics, array elements, and fields of classes) are of fixed size. The supported storage sizes are **float32** and **float64**. Everywhere else (on the evaluation stack, as arguments, as return types, and as local variables) floating point numbers are represented using an internal floating-point type. In each such instance, the nominal type of the variable or expression is either R4 or R8, but its value may be represented internally with additional range and/or precision. The size of the internal floating-point representation is implementation-dependent, may vary, and shall have precision at least as great as that of the variable or expression being represented. An implicit widening conversion to the internal representation from **float32** or **float64** is performed when those types are loaded from storage. The internal representation is typically the native size for the hardware, or as required for efficient implementation of an operation. The internal representation shall have the following characteristics:

- The internal representation shall have precision and range greater than or equal to the nominal type.
- Conversions to and from the internal representation shall preserve value.

Note: This implies that an implicit widening conversion from **float32** (or **float64**) to the internal representation, followed by an explicit conversion from the internal representation to **float32** (or **float64**), will result in a value that is identical to the original **float32** (or **float64**) value.

Rationale: *This design allows the CLI to choose a platform-specific high-performance representation for floating point numbers until they are placed in storage locations. For example, it may be able to leave floating point variables in hardware registers that provide more precision than a user has requested. At the same time, CIL generators can force operations to respect language-specific rules for representations through the use of conversion instructions.*

When a floating-point value whose internal representation has greater range and/or precision than its nominal type is put in a storage location it is automatically coerced to the type of the storage location. This may involve a loss of precision or the creation of an out-of-range value (NaN, +infinity, or -infinity). However, the value may be retained in the internal representation for future use, if it is reloaded from the storage location without having been modified. It is the responsibility of the compiler to ensure that the retained value is still valid at the time of a subsequent load, taking into account the effects of aliasing and other execution threads (see memory model section). This freedom to carry extra precision is not permitted, however, following the execution of an explicit conversion (conv.r4 or conv.r8), at which time the internal representation must be exactly representable in the associated type.

Note: To detect values that cannot be converted to a particular storage type, a conversion instruction (**conv.r4**, or **conv.r8**) may be used, followed by a check for a non-finite value using **ckfinite**. To detect underflow when converting to a particular storage type, a comparison to zero is required before and after the conversion.

Note: The use of an internal representation that is wider than **float32** or **float64** may cause differences in computational results when a developer makes seemingly unrelated modifications to their code, the result of which may be that a value is spilled from the internal representation (e.g. in a register) to a location on the stack.

12.1.4 CIL Instructions and Numeric Types

This clause contains only informative text

Most CIL instructions that deal with numbers take their operands from the evaluation stack (see clause 12.3.2.1), and these inputs have an associated type that is known to the VES. As a result, a single operation like **add** can have inputs of any numeric data type, although not all instructions can deal with all combinations of operand types. Binary operations other than addition and subtraction require that both operands be of the same type. Addition and subtraction allow an integer to be added to or subtracted from a managed pointer (types **&** and **O**). Details are specified in Partition II.

Instructions fall into the following categories:

Numeric: These instructions deal with both integers and floating point numbers, and consider integers to be signed. Simple arithmetic, conditional branch, and comparison instructions fit in this category.

Integer: These instructions deal only with integers. Bit operations and unsigned integer division/remainder fit in this category.

Floating point: These instructions deal only with floating point numbers.

Specific: These instructions deal with integer and/or floating point numbers, but have variants that deal specially with different sizes and unsigned integers. Integer operations with overflow detection, data conversion instructions, and operations that transfer data between the evaluation stack and other parts of memory (see clause 12.3.2) fit into this category.

Unsigned/unordered: There are special comparison and branch instructions that treat integers as unsigned and consider unordered floating point numbers specially (as in "branch if greater than or unordered"):

Load constant: The load constant (**ldc.***) instructions are used to load constants of type int32, int64, float32 or float64. Native size constants (type native int) shall be created by conversion from int32 (conversion from int64 would not be portable) using **conv.i** or **conv.u.**

Table 7: CIL Instructions by Numeric Category shows the CIL instructions that deal with numeric values, along with the category to which they belong. Instructions that end in ".*" indicate all variants of the instruction (based on size of data and whether the data is treated as signed or unsigned).

Table 7: CIL Instructions by Numeric Category

`add`	Numeric	`div`	Numeric
`add.ovf.*`	Specific	`div.un`	Integer
`and`	Integer	`ldc.*`	Load constant
`beq[.s]`	Numeric	`ldelem.*`	Specific
`bge[.s]`	Numeric	`ldind.*`	Specific
`bge.un[.s]`	Unsigned/unordered	`mul`	Numeric
`bgt[.s]`	Numeric	`mul.ovf.*`	Specific
`bgt.un[.s]`	Unsigned/unordered	`neg`	Integer
`ble[.s]`	Numeric	`newarr.*`	Specific
`ble.un[.s]`	Unsigned/unordered	`not`	Integer
`blt[.s]`	Numeric	`or`	Integer
`blt.un[.s]`	Unsigned/unordered	`rem`	Numeric
`bne.un[.s]`	Unsigned/unordered	`rem.un`	Integer
`ceq`	Numeric	`shl`	Integer

`cgt`	Numeric
`cgt.un`	Unsigned/unordered
`ckfinite`	Floating point
`clt`	Numeric
`clt.un`	Unsigned/unordered
`conv.*`	Specific
`conv.ovf.*`	Specific
`shr`	Integer
`shr.un`	Specific
`stelem.*`	Specific
`stind.*`	Specific
`sub`	Numeric
`sub.ovf.*`	Specific
`xor`	Integer

End informative text

12.1.5 CIL Instructions and Pointer Types

This clause contains only informative text

Rationale: *Some implementations of the CLI will require the ability to track pointers to objects and to collect objects that are no longer reachable (thus providing memory management by "garbage collection"). This process moves objects in order to reduce the working set and thus will modify all pointers to those objects as they move. For this to work correctly, pointers to objects may only be used in certain ways. The* ***O*** *(object reference) and* ***&*** *(managed pointer) datatypes are the formalization of these restrictions.*

The use of object references is tightly restricted in the CIL. They are used almost exclusively with the "virtual object system" instructions, which are specifically designed to deal with objects. In addition, a few of the base instructions of the CIL handle object references. In particular, object references can be:

1. Loaded onto the evaluation stack to be passed as arguments to methods (**ldloc**, **ldarg**), and stored from the stack to their home locations (**stloc, starg**)
2. Duplicated or popped off the evaluation stack (**dup**, **pop**)
3. Tested for equality with one another, but not other data types (**beq**, **beq.s**, **bne**, **bne.s**, **ceq**)
4. Loaded-from / stored-into unmanaged memory, in type unmanaged code only (**ldind.ref**, **stind.ref**)
5. Created as a null reference (**ldnull**)
6. Returned as a value (**ret**)

Managed pointers have several additional base operations.

1. Addition and subtraction of integers, in units of *bytes*, returning a managed pointer (**add**, **add.ovf.u, sub, sub.ovf.u**)
2. Subtraction of two managed pointers to elements of the same array, returning the number of *bytes* between them (**sub**, **sub.ovf.u**)
3. Unsigned comparison and conditional branches based on two managed pointers (**bge.un, bge.un.s, bgt.un, bgt.un.s, ble.un, ble.un.s, blt.un, blt.un.s, cgt.un**, **clt.un**)

Arithmetic operations upon managed pointers are intended *only* for use on pointers to elements of the same array. Other uses of arithmetic on managed pointers is unspecified.

Rationale: *Since the memory manager runs asynchronously with respect to programs and updates managed pointers, both the distance between distinct objects and their relative position can change.*

End informative text

12.1.6 Aggregate Data

This clause contains only informative text

The CLI supports *aggregate data*, that is, data items that have sub-components (arrays, structures, or object instances) but are passed by copying the value. The sub-components can include references to managed memory. Aggregate data is represented using a *value type*, which can be instantiated in two different ways:

- **Boxed**: as an Object, carrying full type information at runtime, and typically allocated on the heap by the CLI memory manager.
- **Unboxed**: as a "value type instance" that does *not* carry type information at runtime and that is never allocated directly on the heap. It can be part of a larger structure on the heap – a field of a class, a field of a boxed value type, or an element of an array. Or it can be in the local variables or incoming arguments array (see clause 12.3.2). Or it can be allocated as a static variable or static member of a class or a static member of another value type.

Because value type instances, specified as method arguments, are copied on method call, they do not have "identity" in the sense that Objects (boxed instances of classes) have.

12.1.6.1 Homes for Values

The **home** of a data value is where it is stored for possible reuse. The CLI directly supports the following home locations:

- An incoming **argument**
- A **local variable** of a method
- An instance **field** of an object or value type
- A **static** field of a class, interface, or module
- An **array element**

For each home location, there is a means to compute (at runtime) the address of the home location and a means to determine (at JIT compile time) the type of a home location. These are summarized in Table 8: Address and Type of Home Locations.

Table 8: Address and Type of Home Locations

Type of Home	**Runtime Address Computation**	**JITtime Type Determination**
Argument	**ldarga** for by-value arguments or **ldarg** for by-reference arguments	Method signature
Local Variable	**ldloca** for by-value locals or **ldloc** for by-reference locals	Locals signature in method header
Field	**ldflda**	Type of field in the class, interface, or module
Static	**ldsflda**	Type of field in the class, interface, or module
Array Element	**ldelema** for single-dimensional zero-based arrays or call the instance method **Address**	Element type of array

In addition to homes, built-in values can exist in two additional ways (i.e. without homes):

1. as constant values (typically embedded in the CIL instruction stream using **ldc.*** instructions)
2. as an intermediate value on the evaluation stack, when returned by a method or CIL instruction.

12.1.6.2 Operations on Value Type Instances

Value type instances can be created, passed as arguments, returned as values, and stored into and extracted from locals, fields, and elements of arrays (i.e., copied). Like classes, value types may have both static and non-static members (methods and fields). But, because they carry no type information at runtime, value type instances are not substitutable for items of type Object; in this respect, they act like the built-in types int, long, and so forth. There are two operations, box and unbox, that convert between value type instances and Objects.

12.1.6.2.1 Initializing Instances of Value Types

There are three options for initializing the home of a value type instance. You can zero it by loading the address of the home (see Table 8: Address and Type of Home Locations) and using the **initobj** instruction (for local variables this is also accomplished by setting the **zero initialize** bit in the method's header). You can call a user-defined constructor by loading the address of the home (see Table 8: Address and Type of Home Locations) and then calling the constructor directly. Or you can copy an existing instance into the home, as described in clause 12.1.6.2.

12.1.6.2.2 Loading and Storing Instances of Value Types

There are two ways to load a value type onto the evaluation stack:

- Directly load the value from a home that has the appropriate type, using an **ldarg**, **ldloc**, **ldfld**, or **ldsfld** instruction
- Compute the address of the value type, then use an **ldobj** instruction

Similarly, there are two ways to store a value type from the evaluation stack:

- Directly store the value into a home of the appropriate type, using a **starg**, **stloc**, **stfld**, or **stsfld** instruction
- Compute the address of the value type, then use a **stobj** instruction

12.1.6.2.3 Passing and Returning Value Types

Value types are treated just as any other value would be treated:

- **To pass a value type by value**, simply load it onto the stack as you would any other argument: use **ldloc**, **ldarg**, etc., or call a method that returns a value type. To access a value type parameter that has been passed by value use the **ldarga** instruction to compute its address or the **ldarg** instruction to load the value onto the evaluation stack.
- **To pass a value type by reference**, load the address of the value type as you normally would (see Table 8: Address and Type of Home Locations). To access a value type parameter that has been passed by reference use the **ldarg** instruction to load the address of the value type and then the **ldobj** instruction to load the value type onto the evaluation stack.
- **To return a value type**, just load the value onto an otherwise empty evaluation stack and then issue a **ret** instruction.

12.1.6.2.4 Calling Methods

Static methods on value types are handled no differently from static methods on an ordinary class: use a **call** instruction with a metadata token specifying the value type as the class of the method. Non-static methods (i.e. instance and virtual methods) are supported on value types, but they are given special treatment. A non-static method on a class (rather than a value type) expects a **this** pointer that is an instance of that class. This makes sense for classes, since they have identity and the **this** pointer represents that identity. Value types, however, have identity only when boxed. To address this issue, the **this** pointer on a non-static method of a value type is a by-ref parameter of the value type rather than an ordinary by-value parameter.

A non-static method on a value type may be called in the following ways:

- Given an unboxed instance of a value type, the compiler will know the exact type of the object statically. The **call** instruction can be used to invoke the function, passing as the first parameter (the **this** pointer) the address of the instance. The metadata token used with the **call** instruction shall specify the value type itself as the class of the method.

- Given a boxed instance of a value type, there are three cases to consider:
  - Instance or virtual methods introduced on the value type itself: unbox the instance and call the method directly using the value type as the class of the method.
  - Virtual methods inherited from a parent class: use the **callvirt** instruction and specify the method on the `System.Object`, `System.ValueType` or `System.Enum` class as appropriate.
  - Virtual methods on interfaces implemented by the value type: use the **callvirt** instruction and specify the method on the interface type.

12.1.6.2.5 Boxing and Unboxing

Box and **unbox** are conceptually equivalent to (and may be seen in higher-level languages as) casting between a value type instance and `System.Object`. Because they change data representations, however, boxing and unboxing are like the widening and narrowing of various sizes of integers (the **conv** and **conv.ovf** instructions) rather than the casting of reference types (the **isinst** and **castclass** instructions). The **box** instruction is a widening (always typesafe) operation that converts a value type instance to `System.Object` by making a copy of the instance and embedding it in a newly allocated object. **Unbox** is a narrowing (runtime exception may be generated) operation that converts a `System.Object` (whose runtime type is a value type) to a value type instance. This is done by computing the address of the embedded value type instance without making a copy of the instance.

12.1.6.2.6 Castclass and IsInst on Value Types

Casting to and from value type instances isn't permitted (the equivalent operations are **box** and **unbox**). When boxed, however, it is possible to use the **isinst** instruction to see whether a value of type `System.Object` is the boxed representation of a particular class.

12.1.6.3 Opaque Classes

Some languages provide multi-byte data structures whose contents are manipulated directly by address arithmetic and indirection operations. To support this feature, the CLI allows value types to be created with a specified size but no information about their data members. Instances of these "opaque classes" are handled in precisely the same way as instances of any other class, but the **ldfld**, **stfld**, **ldflda**, **ldsfld**, and **stsfld** instructions shall not be used to access their contents.

End informative text

12.2 Module Information

Partition II provides details of the CLI PE file format. The CLI relies on the following information about each method defined in a PE file:

- The *instructions* composing the method body, including all exception handlers.
- The *signature* of the method, which specifies the return type and the number, order, parameter passing convention, and built-in data type of each of the arguments. It also specifies the native calling convention (this does *not* affect the CIL virtual calling convention, just the native code).
- The *exception handling array*. This array holds information delineating the ranges over which exceptions are filtered and caught. See Partition II and clause 12.4.2.
- The size of evaluation stack that the method will require.
- The size of the locals array that the method will require.
- A “zero init flag” that indicates whether the local variables and memory pool should be initialized by the CLI (see also **localloc**).
- Type of each local variable in the form of a signature of the local variable array (called the “locals signature”).

In addition, the file format is capable of indicating the degree of portability of the file. There is one kind of restriction that may be described:

- Restriction to a specific (32-bit) native size for integers.

By stating which restrictions are placed on executing the code, the CLI class loader can prevent non-portable code from running on an architecture that it cannot support.

12.3 Machine State

One of the design goals of the CLI is to hide the details of a method call frame from the CIL code generator. This allows the CLI (and not the CIL code generator) to choose the most efficient calling convention and stack layout. To achieve this abstraction, the call frame is integrated into the CLI. The machine state definitions below reflect these design choices, where machine state consists primarily of global state and method state.

12.3.1 The Global State

The CLI manages multiple concurrent threads of control (not necessarily the same as the threads provided by a host operating system), multiple managed heaps, and a shared memory address space.

Note: A thread of control can be thought of, somewhat simplistically, as a singly linked list of *method states*, where a new state is created and linked back to the current state by a method call instruction – the traditional model of a stack-based calling sequence. Notice that this model of the thread of control doesn’t correctly explain the operation of **tail.**, **jmp**, or **throw** instructions.

Figure 2: Machine State Model illustrates the machine state model, which includes threads of control, method states, and multiple heaps in a shared address space. Method state, shown separately in Figure 3: Method State, is an abstraction of the stack frame. Arguments and local variables are part of the method state, but they can contain Object References that refer to data stored in any of the managed heaps. In general, arguments and local variables are only visible to the executing thread, while instance and static fields and array elements may be visible to multiple threads, and modification of such values is considered a side-effect.

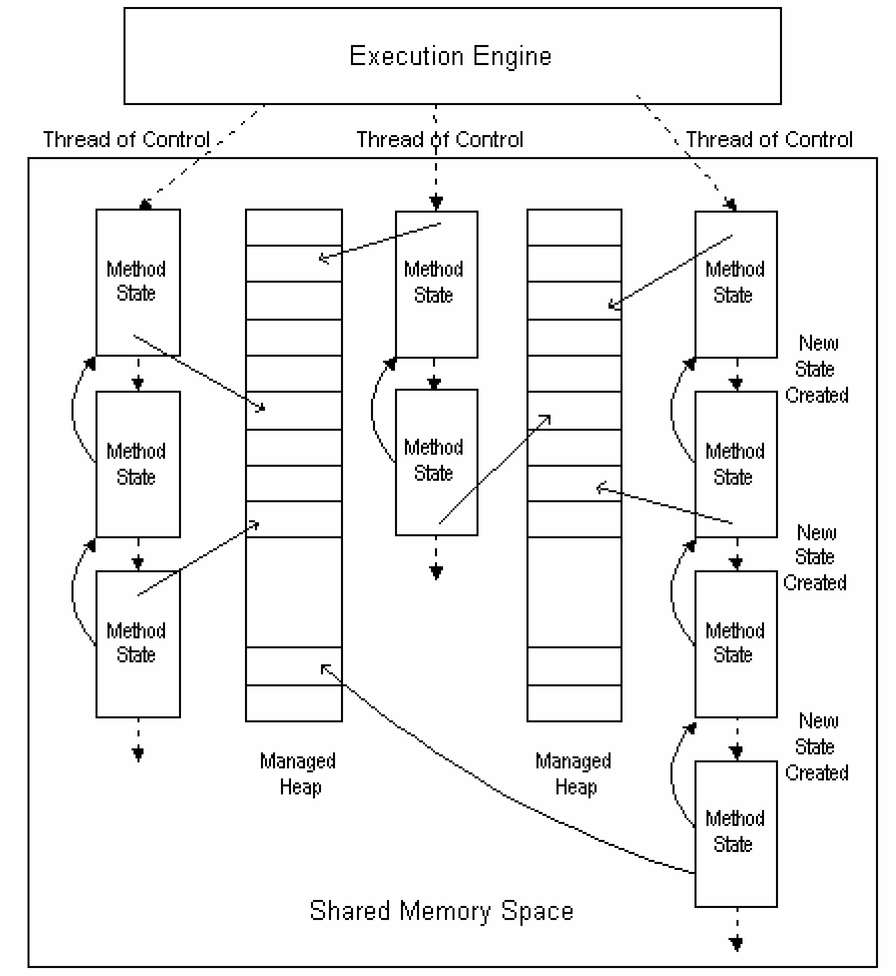


Figure 2: Machine State Model

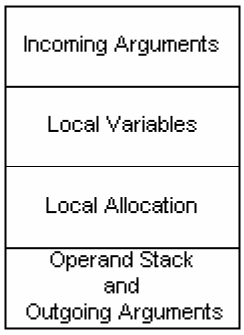


Figure 3: Method State

12.3.2 Method State

Method state describes the environment within which a method executes. (In conventional compiler terminology, it corresponds to a superset of the information captured in the “invocation stack frame”). The CLI method state consists of the following items:

- An instruction pointer (**IP**). This points to the next CIL instruction to be executed by the CLI in the present method.

- An *evaluation stack*. The stack is empty upon method entry. Its contents are entirely local to the method and are preserved across call instructions (that's to say, if this method calls another, once that other method returns, our evaluation stack contents are "still there"). The evaluation stack is not addressable. At all times it is possible to deduce which one of a reduced set of types is stored in any stack location at a specific point in the CIL instruction stream (see clause 12.3.2.1).
- A *local variable array* (starting at index 0). Values of local variables are preserved across calls (in the same sense as for the evaluation stack). A local variable may hold any data type. However, a particular slot shall be used in a type consistent way (where the type system is the one described in clause 12.3.2.1). Local variables are initialized to 0 before entry if the initialize flag for the method is set (see Section 12.2). The address of an individual local variable may be taken using the **ldloca** instruction.
- An *argument array*. The values of the current method's incoming arguments (starting at index 0). These can be read and written by logical index. The address of an argument can be taken using the **ldarga** instruction. The address of an argument is also implicitly taken by the **arglist** instruction for use in conjunction with typesafe iteration through variable-length argument lists.
- A *methodInfo* handle. This contains read-only information about the method. In particular it holds the signature of the method, the types of its local variables, and data about its exception handlers.
- A *local memory pool*. The CLI includes instructions for dynamic allocation of objects from the local memory pool (**localloc**). Memory allocated in the local memory pool is *addressable*. The memory allocated in the local memory pool is reclaimed upon method context termination.
- A *return state* handle. This handle is used to restore the method state on return from the current method. Typically, this would be the state of the method's caller. This corresponds to what in conventional compiler terminology would be the *dynamic link*.
- A *security descriptor*. This descriptor is not directly accessible to managed code but is used by the CLI security system to record security overrides (**assert**, **permit-only**, and **deny**).

The four areas of the method state – incoming arguments array, local variables array, local memory pool and evaluation stack – are specified as if logically distinct areas. A conforming implementation of the CLI may map these areas into one contiguous array of memory, held as a conventional stack frame on the underlying target architecture, or use any other equivalent representation technique.

12.3.2.1 The Evaluation Stack

Associated with each method state is an evaluation stack. Most CLI instructions retrieve their arguments from the evaluation stack and place their return values on the stack. Arguments to other methods and their return values are also placed on the evaluation stack. When a procedure call is made the arguments to the called methods become the incoming arguments array (see clause 12.3.2.2) to the method. This may require a memory copy, or simply a sharing of these two areas by the two methods.

The evaluation stack is made up of slots that can hold any data type, including an unboxed instance of a value type. The type state of the stack (the stack depth and types of each element on the stack) at any given point in a program shall be identical for all possible control flow paths. For example, a program that loops an unknown number of times and pushes a new element on the stack at each iteration would be prohibited.

While the CLI, in general, supports the full set of types described in Section 12.1, the CLI treats the evaluation stack in a special way. While some JIT compilers may track the types on the stack in more detail, the CLI only requires that values be one of:

- int64, an 8-byte signed integer
- int32, a 4-byte signed integer
- native int, a signed integer of either 4 or 8 bytes, whichever is more convenient for the target architecture

- F, a floating point value (float32, float64, or other representation supported by the underlying hardware)
- &, a managed pointer
- O, an object reference
- *, a "transient pointer," which may be used only within the body of a single method, that points to a value known to be in unmanaged memory (see the CIL Instruction Set specification for more details. * types are generated internally within the CLI; they are not created by the user).
- A user-defined value type

The other types are synthesized through a combination of techniques:

- Shorter integer types in other memory locations are zero-extended or sign-extended when loaded onto the evaluation stack; these values are truncated when stored back to their home location.
- Special instructions perform numeric conversions, with or without overflow detection, between different sizes and between signed and unsigned integers.
- Special instructions treat an integer on the stack as though it were unsigned.
- Instructions that create pointers which are guaranteed not to point into the memory manager's heaps (e.g. **ldloca**, **ldarga**, and **ldsflda**) produce transient pointers (type *****) that may be used wherever a managed pointer (type **&**) or unmanaged pointer (type **native unsigned int**) is expected.
- When a method is called, an unmanaged pointer (type **native unsigned int** or *****) is permitted to match a parameter that requires a managed pointer (type **&**). The reverse, however, is *not* permitted since it would allow a managed pointer to be "lost" by the memory manager.
- A managed pointer (type **&**) may be explicitly converted to an unmanaged pointer (type **native unsigned int**), although this is not verifiable and may produce a runtime exception.

12.3.2.2 Local Variables and Arguments

Part of each method state is an array that holds local variables and an array that holds arguments. Like the evaluation stack, each element of these arrays can hold any single data type or an instance of a value type. Both arrays start at 0 (that is, the first argument or local variable is numbered 0). The address of a local variable can be computed using the **ldloca** instruction, and the address of an argument using the **ldarga** instruction.

Associated with each method is metadata that specifies:

- whether the local variables and memory pool memory will be initialized when the method is entered
- the type of each argument and the length of the argument array (but see below for variable argument lists)
- the type of each local variable and the length of the local variable array.

The CLI inserts padding as appropriate for the target architecture. That is, on some 64-bit architectures all local variables may be 64-bit aligned, while on others they may be 8-, 16-, or 32-bit aligned. The CIL generator shall make no assumptions about the offsets of local variables within the array. In fact, the CLI is free to reorder the elements in the local variable array, and different JITters may choose to order them in different ways.

12.3.2.3 Variable Argument Lists

The CLI works in conjunction with the class library to implement methods that accept argument lists of unknown length and type ("varargs methods"). Access to these arguments is through a typesafe iterator in the Class Library, called `System.ArgIterator` (see Partition IV).

The CIL includes one instruction provided specifically to support the argument iterator, **arglist**. This instruction may be used only within a method that is declared to take a variable number of arguments. It returns a value that is needed by the constructor for a `System.ArgIterator` object. Basically, the value created by **arglist** provides access both to the address of the argument list that was passed to the method and a runtime data structure that specifies the number and type of the arguments that were provided. This is sufficient for the class library to implement the user visible iteration mechanism.

From the CLI point of view, varargs methods have an array of arguments like other methods. But only the initial portion of the array has a fixed set of types and only these may be accessed directly using the **ldarg**, **starg**, and **ldarga** instructions. The argument iterator allows access to both this initial segment and the remaining entries in the array.

12.3.2.4 Local Memory Pool

Part of each method state is a local memory pool. Memory can be explicitly allocated from the local memory pool using the **localloc** instruction. All memory in the local memory pool is reclaimed on method exit, and that is the only way local memory pool memory is reclaimed (there is no instruction provided to *free* local memory that was allocated during this method invocation). The local memory pool is used to allocate objects whose type or size is not known at compile time and which the programmer does not wish to allocate in the managed heap.

Because the local memory pool cannot be shrunk during the lifetime of the method, a language implementation cannot use the local memory pool for general-purpose memory allocation.

12.4 Control Flow

The CIL instruction set provides a rich set of instructions to alter the normal flow of control from one CIL instruction to the next.

- **Conditional and Unconditional Branch** instructions for use within a method, provided the transfer doesn't cross a protected region boundary (see clause 12.4.2).
- **Method call** instructions to compute new arguments, transfer them and control to a known or computed destination method (see clause 12.4.1).
- **Tail call** prefix to indicate that a method should relinquish its stack frame before executing a method call (see clause 12.4.1).
- **Return** from a method, returning a value if necessary.
- **Method jump** instructions to transfer the current method's arguments to a known or computed destination method (see clause 12.4.1).
- **Exception-related** instructions (see clause 12.4.2). These include instructions to initiate an exception, transfer control out of a protected region, and end a filter, catch clause, or finally clause.

While the CLI supports control transfers within a method, there are several restrictions that shall be observed:

1. Control transfer is never permitted to enter a catch handler or finally clause (see clause 12.4.2) except through the exception handling mechanism.
2. Control transfer out of a protected region (see clause 12.4.2) is only permitted through an exception instruction (**leave**, **end.filter**, **end.catch**, or **end.finally**).
3. The evaluation stack shall be empty after the return value is popped by a **ret** instruction.
4. Each slot on the stack shall have the same data type at any given point within the method body, regardless of the control flow that allows execution to arrive there.
5. In order for the JIT compilers to efficiently track the data types stored on the stack, the stack shall normally be empty at the instruction following an unconditional control transfer instruction (**br**, **br.s**, **ret**, **jmp**, **throw**, **end.filter**, **end.catch**, or **end.finally**). The stack may be non-empty at such an instruction only if at some earlier location within the method there has been a forward branch to that instruction.

6. Control is not permitted to simply "fall through" the end of a method. All paths shall terminate with one of these instructions: **ret**, **throw**, **jmp**, or (**tail.** followed by **call**, **calli**, or **callvirt**).

12.4.1 Method Calls

Instructions emitted by the CIL code generator contain sufficient information for different implementations of the CLI to use different native calling convention. All method calls initialize the method state areas (see clause 12.3.2) as follows:

1. The incoming arguments array is set by the caller to the desired values.
2. The local variables array always has **null** for Object types and for fields within value types that hold objects. In addition, if the "zero init flag" is set in the method header, then the local variables array is initialized to 0 for all integer types and 0.0 for all floating point types. Value Types are not initialized by the CLI, but verified code will supply a call to an initializer as part of the method's entry point code.
3. The evaluation stack is empty.

12.4.1.1 Call Site Descriptors

Call sites specify additional information that enables an interpreter or JIT compiler to synthesize any native calling convention. All CIL calling instructions (**call, calli,** and **callvirt**) include a description of the call site. This description can take one of two forms. The simpler form, used with the **calli** instruction, is a "call site description" (represented as a metadata token for a stand-alone call signature) that provides:

- The number of arguments being passed.
- The data type of each argument.
- The order in which they have been placed on the call stack.
- The native calling convention to be used

The more complicated form, used for the **call** and **callvirt** instructions, is a "method reference" (a metadata **methodref** token) that augments the call site description with an identifier for the target of the call instruction.

12.4.1.2 Calling Instructions

The CIL has three call instructions that are used to transfer new argument values to a destination method. Under normal circumstances, the called method will terminate and return control to the calling method.

- **call** is designed to be used when the destination address is fixed at the time the CIL is linked. In this case, a method reference is placed directly in the instruction. This is comparable to a direct call to a static function in C. It may be used to call static or instance methods or the (statically known) superclass method within an instance method body.
- **calli** is designed for use when the destination address is calculated at run time. A method pointer is passed on the stack and the instruction contains only the call site description.
- **callvirt**, part of the CIL common type system instruction set, uses the class of an object (known only at runtime) to determine the method to be called. The instruction includes a method reference, but the particular method isn't computed until the call actually occurs. This allows an instance of a subclass to be supplied and the method appropriate for that subclass to be invoked. The **callvirt** instruction is used both for instance methods and methods on interfaces. For further details, see the Common Type System specification and the CIL Instruction Set specification.

In addition, each of these instructions may be immediately preceded by a `tail.` instruction prefix. This specifies that the calling method terminates with this method call (and returns whatever value is returned by the called method). The `tail.` prefix instructs the JIT compiler to discard the caller's method state prior to making the call (if the call is from untrusted code to trusted code the frame cannot be fully discarded for security reasons). When the called method executes a **ret** instruction, control returns not to the calling method but rather to wherever that method would itself have returned (typically, return to caller's caller). Notice that the `tail.`

instruction shortens the lifetime of the caller's frame so it is unsafe to pass managed pointers (type **&**) as arguments.

Finally, there are two instructions that indicate an optimization of the `tail.` case:

- **jmp** is followed by a **methodref** or **methoddef** token and indicates that the current method's state should be discarded, its arguments should be transferred intact to the destination method, and control should be transferred to the destination. The signature of the calling method shall exactly match the signature of the destination method.

12.4.1.3 Computed Destinations

The destination of a method call may be either encoded directly in the CIL instruction stream (the **call** and **jmp** instructions) or computed (the **callvirt**, and **calli** instructions). The destination address for a **callvirt** instruction is automatically computed by the CLI based on the method token and the value of the first argument (the **this** pointer). The method token shall refer to a virtual method on a class that is a direct ancestor of the class of the first argument. The CLI computes the correct destination by locating the nearest ancestor of the first argument's class that supplies an implementation of the desired method.

Note: The implementation can be assumed to be more efficient than the linear search implied here).

For the **calli** instruction the CIL code is responsible for computing a destination address and pushing it on the stack. This is typically done through the use of a **ldftn** or **ldvirtfn** instruction at some earlier time. The **ldftn** instruction includes a metadata token in the CIL stream that specifies a method, and the instruction pushes the address of that method. The **ldvirtfn** instruction takes a metadata token for a virtual method in the CIL stream and an object on the stack. It performs the same computation described above for the **callvirt** instruction but pushes the resulting destination on the stack rather than calling the method.

The **calli** instruction includes a call site description that includes information about the native calling convention that should be used to invoke the method. Correct CIL code shall specify a calling convention specified in the **calli** instruction that matches the calling convention for the method that is being called.

12.4.1.4 Virtual Calling Convention

The CIL provides a "virtual calling convention" that is converted by the JIT into a native calling convention. The JIT determines the optimal native calling convention for the target architecture. This allows the native calling convention to differ from machine to machine, including details of register usage, local variable homes, copying conventions for large call-by-value objects (as well as deciding, based on the target machine, what is considered "large"). This also allows the JIT to reorder the values placed on the CIL virtual stack to match the location and order of arguments passed in the native calling convention.

The CLI uses a single uniform calling convention for all method calls. It is the responsibility of the JITters to convert this into the appropriate native calling convention. The contents of the stack at the time of a call instruction (call, calli, or callvirt any of which may be preceded by `tail.`) are as follows:

1. If the method being called is an instance method (class or interface) or a virtual method, the this pointer is the first object on the stack at the time of the call instruction. For methods on Objects (including boxed value types), the this pointer is of type O (object reference). For methods on value types, the this pointer is provided as a by-ref parameter; that is, the value is a pointer (managed, &, or unmanaged, * or native int) to the instance.

2. The remaining arguments appear on the stack in left-to-right order (that is, the lexically leftmost argument is the lowest on the stack, immediately following the this pointer, if any). clause 12.4.1.5 describes how each of the three parameter passing conventions (by-value, by-reference, and typed reference) should be implemented.

12.4.1.5 Parameter Passing

The CLI supports three kinds of parameter passing, all indicated in metadata as part of the signature of the method. Each parameter to a method has its own passing convention (e.g., the first parameter may be passed by-value while all others are passed by-ref). Parameters shall be passed in one of the following ways (see detailed descriptions below):

- **By-value** parameters, where the **value** of an object is passed from the caller to the callee.
- **By-ref** parameters, where the **address** of the data is passed from the caller to the callee, and the type of the parameter is therefore a managed or unmanaged pointer.
- **Typed reference** parameters, where a runtime representation of the data type is passed along with the address of the data, and the type of the parameter is therefore one specially supplied for this purpose.

It is the responsibility of the CIL generator to follow these conventions. Verification checks that the types of parameters match the types of values passed, but is otherwise unaware of the details of the calling convention.

12.4.1.5.1 By-Value Parameters

For built-in types (integers, floats, etc.) the caller copies the value onto the stack before the call. For objects the object reference (type **O**) is pushed on the stack. For managed pointers (type **&**) or unmanaged pointers (type **native unsigned int**), the address is passed from the caller to the callee. For value types, see the protocol in clause 12.1.6.2.

12.4.1.5.2 By-Ref Parameters

By-Ref Parameters are the equivalent of C++ reference parameters or PASCAL **var** parameters: instead of passing as an argument the value of a variable, field, or array element, its address is passed instead; and any assignment to the corresponding parameter actually modifies the corresponding caller's variable, field, or array element. Much of this work is done by the higher-level language, which hides from the user the need to compute addresses to pass a value and the use of indirection to reference or update values.

Passing a value by reference requires that the value have a home (see clause 12.1.6.1) and it is the address of this home that is passed. Constants, and intermediate values on the evaluation stack, cannot be passed as by-ref parameters because they have no home.

The CLI provides instructions to support by-ref parameters:

- calculate addresses of home locations (see Table 8: Address and Type of Home Locations)
- load and store built-in data types through these address pointers (**ldind.***, **stind.*, ldfld,** etc.)
- copy value types (**ldobj** and **cpobj**).

Some addresses (e.g., local variables and arguments) have lifetimes tied to that method invocation. These shall not be referenced outside their lifetimes, and so they should not be stored in locations that last beyond their lifetime. The CIL does not (and cannot) enforce this restriction, so the CIL generator shall enforce this restriction or the resulting CIL will not work correctly. For code to be verifiable (see Section 8.8) by-ref parameters may **only** be passed to other methods or referenced via the appropriate **stind** or **ldind** instructions.

12.4.1.5.3 Typed Reference Parameters

By-ref parameters and value types are sufficient to support statically typed languages (C++, Pascal, etc.). They also support dynamically typed languages that pay a performance penalty to box value types before passing them to polymorphic methods (Lisp, Scheme, Smalltalk, etc.). Unfortunately, they are not sufficient to support languages like Visual Basic that require by-reference passing of unboxed data to methods that are not statically restricted as to the type of data they accept. These languages require a way of passing *both* the address of the home of the data *and* the static type of the home. This is exactly the information that would be provided if the data were boxed, but without the heap allocation required of a box operation.

Typed reference parameters address this requirement. A typed reference parameter is very similar to a standard by-ref parameter but the static data type is passed as well as the address of the data. Like by-ref parameters, the argument corresponding to a typed reference parameter will have a home.

Note: If it were not for the fact that verification and the memory manager need to be aware of the data type and the corresponding address, a by-ref parameter could be implemented as a standard value type with two fields: the address of the data and the type of the data.

Like a regular by-ref parameter, a typed reference parameter can refer to a home that is on the stack, and that home will have a lifetime limited by the call stack. Thus, the CIL generator shall apply appropriate checks on the lifetime of by-ref parameters; and verification imposes the same restrictions on the use of typed reference parameters as it does on by-ref parameters (see clause 12.4.1.5.2).

A typed reference is passed by either creating a new typed reference (using the **mkrefany** instruction) or by copying an existing typed reference. Given a typed reference argument, the address to which it refers can be extracted using the **refanyval** instruction; the type to which it refers can be extracted using the **refanytype** instruction.

12.4.1.5.4 Parameter Interactions

A given parameter may be passed using any one of the parameter passing conventions: by-value, by-ref, or typed reference. No combination of these is allowed for a single parameter, although a method may have different parameters with different calling mechanisms.

A parameter that has been passed in as typed reference shall not be passed on as by-ref or by-value without a runtime type check and (in the case of by-value) a copy.

A by-ref parameter may be passed on as a typed reference by attaching the static type.

Table 9: Parameter Passing Conventions illustrates the parameter passing convention used for each data type.

Table 9: Parameter Passing Conventions

Type of data	Pass By	How data is sent
Built-in value type (int, float, etc.)	Value	Copied to called method, type statically known at both sides
	Reference	Address sent to called method, type statically known at both sides
	Typed reference	Address sent along with type information to called method
User-defined value type	Value	Called method receives a copy; type statically known at both sides
	Reference	Address sent to called method, type statically known at both sides
	Typed reference	Address sent along with type information to called method
Object	Value	Reference to data sent to called method, type statically known and class available from reference
	Reference	Address of reference sent to called method, type statically known and class available from reference
	Typed reference	Address of reference sent to called method along with static type information, class (i.e. dynamic type) available from reference

12.4.2 Exception Handling

Exception handling is supported in the CLI through exception objects and protected blocks of code. When an exception occurs, an object is created to represent the exception. All exceptions objects are instances of some class (i.e. they can be boxed value types, but not pointers, unboxed value types, etc.). Users may create their own exception classes, typically by subclassing `System.Exception` (see Partition IV).

There are four kinds of handlers for protected blocks. A single protected block shall have exactly one handler associated with it:

- A **finally handler** that shall be executed whenever the block exits, regardless of whether that occurs by normal control flow or by an unhandled exception.
- A **fault handler** that shall be executed if an exception occurs, but not on completion of normal control flow.
- A **type-filtered handler** that handles any exception of a specified class or any of its sub-classes.

- A **user-filtered handler** that runs a user-specified set of CIL instructions to determine whether the exception should be ignored (i.e. execution should resume), handled by the associated handler, or passed on to the next protected block.

Protected regions, the type of the associated handler, and the location of the associated handler and (if needed) user-supplied filter code are described through an Exception Handler Table associated with each method. The exact format of the Exception Handler Table is specified in detail in Partition II. Details of the exception handling mechanism are also specified in Partition II.

12.4.2.1 Exceptions Thrown by the CLI

CLI instructions can throw the following exceptions as part of executing individual instructions. The documentation for each instruction lists all the exceptions the instruction can throw (except for the general purpose **ExecutionEngineException** described below that may be generated by all instructions).

Base Instructions (see Partition III)

- ArithmeticException
- DivideByZeroException
- ExecutionEngineException
- InvalidAddressException
- OverflowException
- SecurityException
- StackOverflowException

Object Model Instructions (see Partition III)

- TypeLoadException
- IndexOutOfRangeException
- InvalidAddressException
- InvalidCastException
- MissingFieldException
- MissingMethodException
- NullReferenceException
- OutOfMemoryException
- SecurityException
- StackOverflowException

The `ExecutionEngineException` is special. It can be thrown by any instruction and indicates an unexpected inconsistency in the CLI. Running exclusively verified code can never cause this exception to be thrown by a conforming implementation of the CLI. However, unverified code (even though that code is conforming CIL) can cause this exception to be thrown if it corrupts memory. Any attempt to execute non-conforming CIL or non-conforming file formats can cause completely unspecified behavior: a conforming implementation of the CLI need not make any provision for these cases.

There are no exceptions for things like ‘MetaDataTokenNotFound.’ CIL verification (see Partition V) will detect this inconsistency before the instruction is executed, leading to a verification violation. If the CIL is not verified this type of inconsistency shall raise the generic ExecutionEngineException.

Exceptions can also be thrown by the CLI, as well as by user code, using the **throw** instruction. The handing of an exception is identical, regardless of the source.

12.4.2.2 Subclassing Of Exceptions

Certain types of exceptions thrown by the CLI may be subclassed to provide more information to the user. The specification of CIL instructions in Partition III describes what types of exceptions should be thrown by the runtime environment when an abnormal situation occurs. Each of these descriptions allows a conforming implementation to throw an object of the type described or an object of a subclass of that type.

Note: For instance, the specification of the `ckfinite` instruction requires that an exception of type `ArithmeticException` or a subclass of `ArithmeticException` be thrown by the CLI. A conforming implementation may simply throw an exception of type `ArithmeticException`, but it may also choose to provide more information to the programmer by throwing an exception of type `NotFiniteNumberException` with the offending number.

12.4.2.3 Resolution Exceptions

CIL allows types to reference, among other things, interfaces, classes, methods, and fields. Resolution errors occur when references are not found or are mismatched. Resolution exceptions can be generated by references from CIL instructions, references to base classes, to implemented interfaces, and by references from signatures of fields, methods and other class members.

To allow scalability with respect to optimization, detection of resolution exceptions is given latitude such that it may occur as early as install time and as late as execution time.

The latest opportunity to check for resolution exceptions from all references except CIL instructions is as part of initialization of the type that is doing the referencing (see Partition II). If such a resolution exception is detected the static initializer for that type, if present, shall not be executed.

The latest opportunity to check for resolution exceptions in CIL instructions is as part of the first execution of the associated CIL instruction. When an implementation chooses to perform resolution exception checking in CIL instructions as late as possible, these exceptions, if they occur, shall be thrown prior to any other non-resolution exception that the VES may throw for that CIL instruction. Once a CIL instruction has passed the point of throwing resolution errors (it has completed without exception, or has completed by throwing a non-resolution exception), subsequent executions of that instruction shall no longer throw resolution exceptions.

If an implementation chooses to detect some resolution errors, from any references, earlier than the latest opportunity for that kind of reference, it is not required to detect all resolution exceptions early.

An implementation that detects resolution errors early is allowed to prevent a class from being installed, loaded or initialized as a result of resolution exceptions detected in the class itself or in the transitive closure of types from following references of any kind.

For example, each of the following represents a permitted scenario. An installation program can throw resolution exceptions (thus failing the installation) as a result of checking CIL instructions for resolution errors in the set of items being installed. An implementation is allowed to fail to load a class as a result of checking CIL instructions in a referenced class for resolution errors. An implementation is permitted to load and initialize a class that has resolution errors in its CIL instructions.

The following exceptions are among those considered resolution exceptions:

- `BadImageFormatException`
- `EntryPointNotFoundException`
- `MissingFieldException`
- `MissingMemberException`
- `MissingMethodException`
- `NotSupportedException`
- `TypeLoadException`
- `TypeUnloadedException`

For example, when a referenced class cannot be found, a `TypeLoadException` is thrown. When a referenced method (whose class is found) cannot be found, a `MissingMethodException` is thrown. If a matching method being used consistently is accessible, but violates declared security policy, a `SecurityException` is thrown.

12.4.2.4 Timing of Exceptions

Certain types of exceptions thrown by CIL instructions may be detected before the instruction is executed. In these cases, the specific time of the throw is not precisely defined, but the exception should be thrown no later than the instruction is executed. That relaxation of the timing of exceptions is provided so that an implementation may choose to detect and throw an exception before any code is run, e.g., at the time of CIL to native code conversion.

There is a distinction between the time of detecting the error condition and throwing the associated exception. An error condition may be detected early (e.g., at JIT time), but the condition may be signaled later (e.g. at the execution time of the offending instruction) by throwing an exception.

The following exceptions are among those that may be thrown early by the runtime:

- `MissingFieldException`,
- `MissingMethodException`,
- `SecurityException`,
- `TypeLoadException`

12.4.2.5 Overview of Exception Handling

See the Exception Handling specification in Partition II for details.

Each method in an executable has associated with it a (possibly empty) array of exception handling information. Each entry in the array describes a protected block, its filter, and its handler (which may be a **catch** handler, a **filter** handler, a **finally** handler, or a **fault** handler). When an exception occurs, the CLI searches the array for the first protected block that

- Protects a region including the current instruction pointer *and*
- Is a catch handler block *and*
- Whose filter wishes to handle the exception

If a match is not found in the current method, the calling method is searched, and so on. If no match is found the CLI will dump a stack trace and abort the program.

Note: A debugger can intervene and treat this situation like a breakpoint, before performing any stack unwinding, so that the stack is still available for inspection through the debugger.

If a match is found, the CLI walks the stack back to the point just located, but this time calling the **finally** and **fault** handlers. It then starts the corresponding exception handler. Stack frames are discarded either as this second walk occurs or after the handler completes, depending on information in the exception handler array entry associated with the handling block.

Some things to notice are:

- The ordering of the exception clauses in the Exception Handler Table is important. If handlers are nested, the most deeply nested try blocks shall come before the try blocks that enclose them.
- Exception handlers may access the local variables and the local memory pool of the routine that catches the exception, but any intermediate results on the evaluation stack at the time the exception was thrown are lost.
- An exception object describing the exception is automatically created by the CLI and pushed onto the evaluation stack as the first item upon entry of a filter or catch clause.
- Execution cannot be resumed at the location of the exception, except with a **user-filtered handler**.

12.4.2.6 CIL Support for Exceptions

The CIL has special instructions to:

- **Throw** and **rethrow** a user-defined exception.

- **Leave** a protected block and execute the appropriate **finally** clauses within a method, without throwing an exception. This is also used to exit a **catch** clause. Notice that leaving a protected block does **not** cause the fault clauses to be called.
- End a user-supplied filter clause (**endfilter**) and return a value indicating whether to handle the exception.
- End a finally clause (**endfinally**) and continue unwinding the stack.

12.4.2.7 Lexical Nesting of Protected Blocks

A protected region (also called a "try block") is described by two addresses: the trystart is the address of the first instruction to be protected and tryend is the address immediately following the last instruction to be protected. A handler region is described by two addresses: the **handlerstart** is the address of the first instruction of the handler and the **handlerend** is the address immediately following the last instruction of the handler.

There are three kinds of handlers: catch, finally, and fault. A single exception entry consists of

- Optional: a type token (the type of exception to be handled) or **filterstart** (the address of the first instruction of the user-supplied filter code)
- Required: **protected region**
- Required: **handler region**.

Every method has associated with it a set of exception entries, called the **exception set**.

If an exception entry contains a **filterstart**, then **filterstart < handlerstart**. The **filter region** starts at the instruction specified by **filterstart** and contains all instructions up to (but not including) that specified by **handlerstart**. If there is no **filterstart** then the filter region is empty (hence does not overlap with any region).

No two regions (protected region, handler region, filter region) of a single exception entry may overlap with one another.

For every pair of exception entries in an exception set, one of the following must be true:

- They **nest**: all three regions of one entry must be within a single region of the other entry.
- They are **disjoint**: all six regions of the two entries are pairwise disjoint (no addresses overlap)
- They **mutually protect**: the protected regions are the same and the other regions are pairwise disjoint.

The encoding of an exception entry in the file format (see Partition II) guarantees that only a catch handler (not a fault handler or finally handler) can have a filter region.

12.4.2.8 Control Flow Restrictions on Protected Blocks

The following restrictions govern control flow into, out of, and between **try** blocks and their associated handlers.

1. CIL code shall not enter a **filter**, **catch**, **fault** or **finally** block except through the CLI exception handling mechanism.
2. There are only two ways to enter a **try** block from outside its lexical body:
   a. **Branching to or falling into the try block's first instruction**. The branch may be made using a conditional branch, an unconditional branch, or a **leave** instruction.
   b. **Using a leave instruction from that try's catch block.** In this case, correct CIL code may branch to any instruction within the **try** block, not just its first instruction, so long as that branch target is not protected by yet another **try**, nested withing the first
3. Upon entry to a **try** block the evaluation stack shall be empty.
4. The only ways CIL code may leave a **try**, **filter**, **catch, finally** or **fault** block are as follows:

a. **throw** from any of them.

b. **leave** from the body of a **try** or **catch** (in this case the destination of the **leave** shall have an empty evaluation stack and the **leave** instruction has the side-effect of emptying the evaluation stack).

c. **endfilter** may appear only as the lexically last instruction of a **filter** block, and it shall always be present (even if it is immediately preceded by a **throw** or other unconditional control flow). If reached, the evaluation stack shall contain an **int32** when the **endfilter** is executed, and the value is used to determine how exception handling should proceed.

d. **endfinally** from anywhere within a **finally** or **fault**, with the side-effect of emptying the evaluation stack.

e. **rethrow** from within a **catch** block, with the side-effect of emptying the evaluation stack.

5. When the try block is exited with a leave instruction, the evaluation stack shall be empty.
6. When a catch or filter clause is exited with a leave instruction, the evaluation stack shall be empty. This involves popping, from the evaluation stack, the exception object that was automatically pushed onto the stack.
7. CIL code shall not exit any try, filter, catch finally or fault block using a **ret** instruction.
8. The `localloc` instruction cannot occur within an exception block: **filter**, **catch**, **finally**, or **fault**

12.5 Proxies and Remoting

A **remoting boundary** exists if it is not possible to share the identity of an object directly across the boundary. For example, if two objects exist on physically separate machines that do not share a common address space, then a remoting boundary will exist between them. There are other administrative mechanisms for creating remoting boundaries.

The VES provides a mechanism, called the **application domain**, to isolate applications running in the same operating system process from one another. Types loaded into one application domain are distinct from the same type loaded into another application domain, and instances of objects shall not be directly shared from one application domain to another. Hence, the application domain itself forms a remoting boundary.

The VES implements remoting boundaries based on the concept of a **proxy**. A proxy is an object that exists on one side of the boundary and represents an object on the other side. The proxy forwards references to instance fields and methods to the actual object for interpretation. Proxies do not forward references to static fields or calls to static methods.

The implementation of proxies is provided automatically for instances of types that derive from **System.MarshalByRefObject** (see Partition IV).

12.6 Memory Model and Optimizations

12.6.1 The Memory Store

By “memory store” we mean the regular process memory that the CLI operates within. Conceptually, this store is simply an array of bytes. The index into this array is the address of a data object. The CLI accesses data objects in the memory store via the **ldind.*** and **stind.*** instructions.

12.6.2 Alignment

Built-in datatypes shall be *properly aligned*, which is defined as follows:

- 1-byte, 2-byte, and 4-byte data is properly aligned when it is stored at a 1-byte, 2-byte, or 4-byte boundary, respectively.
- 8-byte data is properly aligned when it is stored on the same boundary required by the underlying hardware for atomic access to a **native int**.

Thus, **int16** and **unsigned int16** start on even address; **int32**, **unsigned int32**, and **float32** start on an address divisible by 4; and **int64**, **unsigned int64**, and **float64** start on an address divisible by 4 or 8, depending upon the target architecture. The native size types (**native int**, **native unsigned int**, and **&**) are always naturally aligned (4 bytes or 8 bytes, depending on architecture). When generated externally, these should also be aligned to their natural size, although portable code may use 8 byte alignment to guarantee architecture independence. It is strongly recommended that **float64** be aligned on an 8-byte boundary, even when the size of **native int** is 32 bits.

There is a special prefix instruction, **unaligned.**, that may immediately precede a **ldind**, **stind, initblk**, or **cpblk** instruction. This prefix indicates that the data may have arbitrary alignment; the JIT is required to generate code that correctly performs the effect of the instructions regardless of the actual alignment. Otherwise, if the data is not properly aligned and no unligned. prefix has been specified, executing the instruction may generate unaligned memory faults or incorrect data.

12.6.3 Byte Ordering

For datatypes larger than 1 byte, the byte ordering is dependent on the target CPU. Code that depends on byte ordering may not run on all platforms. The PE file format (see Section 12.2) allows the file to be marked to indicate that it depends on a particular type ordering.

12.6.4 Optimization

Conforming implementations of the CLI are free to execute programs using any technology that guarantees, within a single thread of execution, that side-effects and exceptions generated by a thread are visible in the order specified by the CIL. For this purpose volatile operations (including volatile reads) constitute side-effects. Volatile operations are specified in clause 12.6.7. There are no ordering guarantees relative to exceptions injected into a thread by another thread (such exceptions are sometimes called "asynchronous exceptions," e.g., **System.Threading.ThreadAbortException**).

Rationale: *An optimizing compiler is free to reorder side-effects and synchronous exceptions to the extent that this reordering does not change any observable program behavior.*

Note: An implementation of the CLI is permitted to use an optimizing compiler, for example, to convert CIL to native machine code provided the compiler maintains (within each single thread of execution) the same order of side-effects and synchronous exceptions.

This is a stronger condition than ISO C++ (which permits reordering between a pair of sequence points) or ISO Scheme (which permits reordering of arguments to functions).

12.6.5 Locks and Threads

The logical abstraction of a thread of control is captured by an instance of the `System.Threading.Thread` object in the class library. Classes beginning with the string "`System.Threading`" (see Partition IV) provide much of the user visible support for this abstraction.

To create consistency across threads of execution, the CLI provides the following mechanisms:

1. **Synchronized methods**. A lock that is visible across threads controls entry to the body of a synchronized method. For instance and virtual methods the lock is associated with the *this* pointer. For static methods the lock is associated with the type to which the method belongs. The lock is taken by the logical thread (see `System.Threading.-Thread` in Partition IV) and may be entered any number of times by the same thread; entry by other threads is prohibited while the first thread is still holding the lock. The CLI shall release the lock when control exits (by any means) the method invocation that first acquired the lock.

2. **Explicit locks and monitors.** These are provided in the class library, see `System.Threading.Monitor`**.** Many of the methods in the `System.Threading.Monitor` class accept an **`Object`** as argument, allowing direct access to the same lock that is used by synchronized methods. While the CLI is responsible for ensuring correct protocol when this lock is only used by synchronized methods, the user must accept this responsibility when using explicit monitors on these same objects.

3. **Volatile reads and writes.** The CIL includes a prefix, `volatile.`, that specifies that the subsequent operation is to be performed with the cross-thread visibility constraints described in clause 12.6.7. In addition, the class library provides methods to perform explicit volatile reads and writes, as well as barrier synchronization. (See `VolatileRead`, `VolatileWrite`, and `MemoryBarrier`, respectively, in `System.Threading.Thread`.)
4. **Built-in atomic reads and writes.** All reads and writes of certain properly aligned data types are guaranteed to occur atomically. See clause 12.6.6.
5. **Explicit atomic operations.** The class library provides a variety of atomic operations in the `System.Threading.Interlocked` class.

Acquiring a lock (`System.Threading.Monitor.Enter` or entering a synchronized method) shall implicitly perform a volatile read operation, and releasing a lock (`System.Threading.Monitor.Exit` or leaving a synchronized method) shall implicitly perform a volatile write operation. See clause 12.6.7.

12.6.6 Atomic Reads and Writes

A conforming CLI shall guarantee that read and write access to *properly aligned* memory locations no larger than the native word size (the size of type **native int**) is atomic (see clause 12.6.2). Atomic writes shall alter no bits other than those written. Unless explicit layout control (see Partition II (Controlling Instance Layout)) is used to alter the default behavior, data elements no larger than the natural word size (the size of a **native int**) shall be properly aligned. Object references shall be treated as though they are stored in the native word size.

Note: There is no guarantee about atomic update (read-modify-write) of memory, except for methods provided for that purpose as part of the class library (see Partition IV). An atomic write of a “small data item” (an item no larger than the native word size) *is* required to do an atomic read/write/modify on hardware that does not support direct writes to small data items.

Note: There is no guaranteed atomic access to 8-byte data when the size of a **native int** is 32 bits even though some implementations may perform atomic operations when the data is aligned on an 8-byte boundary.

12.6.7 Volatile Reads and Writes

The **volatile.** prefix on certain instructions shall guarantee cross-thread memory ordering rules. They do not provide atomicity, other than that guaranteed by the specification of clause 12.6.6.

A volatile read has “acquire semantics” meaning that the read is guaranteed to occur prior to any references to memory that occur after the read instruction in the CIL instruction sequence. A volatile write has “release semantics” meaning that the write is guaranteed to happen after any memory references prior to the write instruction in the CIL instruction sequence.

A conforming implementation of the CLI shall guarantee this semantics of volatile operations. This ensures that all threads will observe volatile writes performed by any other thread in the order they were performed. But a conforming implementation is *not* required to provide a single total ordering of volatile writes as seen from all threads of execution.

An optimizing compiler that converts CIL to native code shall not remove any volatile operation, nor may it coalesce multiple volatile operations into a single operation.

Rationale: *One traditional use of volatile operations is to model hardware registers that are visible through direct memory access. In these cases, removing or coalescing the operations may change the behavior of the program.*

Note: An optimizing compiler from CIL to native code is permitted to reorder code, provided that it guarantees both the single-thread semantics described in Section 12.6 and the cross-thread semantics of volatile operations.

12.6.8 Other Memory Model Issues

All memory allocated for static variables (other than those assigned RVAs within a PE file, see Partition II) and objects shall be zeroed before they are made visible to any user code.

A conforming implementation of the CLI shall ensure that, even in a multi-threaded environment and without proper user synchronization, objects are allocated in a manner that prevents unauthorized memory access and prevents illegal operations from occurring. In particular, on multiprocessor memory systems where explicit synchronization is required to ensure that all relevant data structures are visible (for example, vtable pointers) the EE shall be responsible for either enforcing this synchronization automatically or for converting errors due to lack of synchronization into non-fatal, non-corrupting, user-visible exceptions.

It is explicitly *not* a requirement that a conforming implementation of the CLI guarantee that all state updates performed within a constructor be uniformly visible before the constructor completes. CIL generators may ensure this requirement themselves by inserting appropriate calls to the memory barrier or volatile write instructions.

13 Index

Common Language Infrastructure (CLI)

Partition II:
Metadata Definition and Semantics

Table of contents

1 Scope

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 document focuses on the structure and semantics of metadata. The semantics of metadata, which dictate much of the operation of the VES, are described using the syntax of ilasm, an assembler language for CIL. The *ilasm* syntax itself is considered a normative part of this International standard. This constitutes Chapters 5 through 20. A complete syntax for *ilasm* is included in Partition V. The structure (both logical and physical) is covered in Chapters 21 through 24.

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 and also provides a convenient way to express examples.*

The semantics of the metadata also can be described independently of the actual format in which the metadata is stored. This point is important because the storage format as specified Chapters 21 through 24 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.

3 Validation and Verification

Validation refers to a set of tests that can be performed on any file to check that the file format, metadata, and CIL are self-consistent. These tests are intended to ensure that the file conforms to the mandatory requirements of this specification. The behavior of conforming implementations of the CLI when presented with non-conforming files is unspecified.

Verification refers to a check 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 valid 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) as well as containing 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 does not specify:

- at what time (if ever) such an algorithm should be performed
- what a conforming implementation should do in case of failure of verification.

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

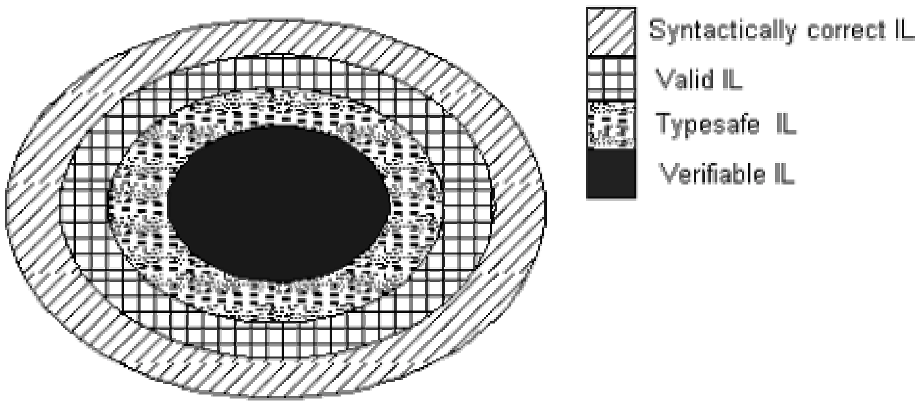


Figure 1: Relationship between valid and verifiable CIL

In the above figure, the outer circle contains all code permitted by the ilasm syntax. The next circle represents all code that is valid CIL. The dotted inner circle represents all type safe code. Finally, the black innermost circle contains all code that is verifiable. (The difference between typesafe 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 typesafe). Note that even if a program follows the syntax described in Partition V, the code may still not be valid, because valid code shall adhere to restrictions presented in this document and in Partition III.

Verification is a very stringent test. 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 may 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. A conforming implementation of the CLI may allow *unverifiable code* (valid code that does not pass verification) to be executed, although this may 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 may be subject to additional implementation-specified trust controls.

4 Introductory Examples

This section and its subsections contain only informative text.

Before diving into the details, it is useful to see an introductory sample program to get a feeling for the ilasm assembly language. The next section shows the famous Hello World program, this time in the ilasm assembly language.

4.1 Hello World Example

This section gives a simple example to illustrate the general feel of ilasm. Below is code that 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 assembly `mscorlib` (see Partition IV).

```
Example (informative):
.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
}
```

The **.assembly extern** declaration references an external assembly, mscorlib, which defines `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 the method is 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**).

The method contains only three instructions. 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 returns (**ret**) from `main`.

4.2 Examples

This document contains integrated examples for most features of the CLI metadata. Many sections conclude with an example showing a typical use of the feature. All these examples are written using the ilasm assembly language. In addition, Partition V 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 section describes aspects of the ilasm syntax that are common to many parts of the grammar. The term "ASCII" refers to the American Standard Code for Information Interchange, a standard seven-bit code that was proposed by ANSI in 1963, and finalized in 1968. The ASCII repertoire of Unicode is the set of 128 Unicode characters from U+0000 to U+007F.

5.1 General Syntax Notation

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

Bold items are terminals. Items placed in angle brackets (e.g. <int64>) are names of syntax classes and shall be replaced by actual instances of the class. Items placed in square brackets (e.g. [<float>]) are optional, and any item followed by * can appear zero or more times. The character "|" means that the items on either side of it are acceptable. The options are sorted in alphabetical order (to be more specific: in ASCII order, ignoring "<" for syntax classes, 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 reference.

Example (informative):

```
A grammar such as
<top> ::= <int32> | float <float> |
          floats [<float> [, <float>]*] | else <QSTRING>
would consider the following all to be legal:
     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 illegal:
     else 3
     3, 4
     float 4.3, 2.4
     float else
     stuff
```

5.2 Terminals

The basic syntax classes used in the grammar are used to describe syntactic constraints on the input intended to convey logical restrictions on the information encoded in the metadata.

The syntactic constraints described in this clause are informative only. The semantic constraints (e.g. "shall be represented in 32 bits") are normative.

`<int32>` is either a decimal number or "0x" followed by a hexadecimal number, and shall be represented in 32 bits.

`<int64>` is either a decimal number or "0x" followed by a hexadecimal number, and shall be represented in 64 bits.

`<hexbyte>` is a 2-digit hexadecimal number that fits into one byte.

<realnumber> is any syntactic representation for a floating point number that is distinct from that for all other terminal nodes. In this document, a period (.) is used to separate the integer and fractional parts, and "e" or "E" separates the mantissa from the exponent. Either (but not both) may be omitted.

Note: A complete assembler may also provide syntax for infinities and NaNs.

`<QSTRING>` is a string surrounded by double quote (″) marks. Within the quoted string the character "\" can be used as an escape character, with "\t" for a tab character, "\n" for a new line character, or followed by three octal digits in order to insert an arbitrary byte into the string. 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 using "\" as the last character in a line, in which case the line break is not entered into the generated string. Any white characters (space, line feed, carriage return, and tab) between the "\" and the first character on the next line are ignored. See also examples below.

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).

`<SQSTRING>` is similar to `<QSTRING>` with the difference that it is surround by single quote (′) marks instead of double quote marks.

`<ID>` is a contiguous string of characters which starts with either an alphabetic character or one of "_", "$", "@" or "?" and is followed by any number of alphanumeric characters or any of "_", "$", "@", or "?". An `<ID>` is used in only two ways:

- As a label of a CIL instruction
- As an `<id>` which can either be an `<ID>` or an `<SQSTRING>`, so that special characters can be included.

Example (informative):

The following examples shows breaking of strings:

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

and

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

become both `"Hello World from CIL!"`.

5.3 Identifiers

Identifiers are used to name entities. Simple identifiers are just 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 is placed within single quotation marks. This is summarized in the following grammar.

```
<id> ::=
  <ID>
   | <SQSTRING>
```

Keywords may only be used as identifiers if they appear in single quotes (see Partition V for a list of all keywords).

Several <id>'s may be combined to form a larger <id>. The <id>'s are separated by 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. fully qualified type names)*

```
 Examples (informative):
The following shows some simple identifiers:
     A
     Test
     $Test
     @Foo?
     ?_X_
The following shows identifiers in single quotes:
     'Weird Identifier'
     'Odd\102Char'
     'Embedded\nReturn'
The following shows dotted names:
     System.Console
     A.B.C
     'My Project'.'My Component'.'My Name'
```

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 Chapters 21 through 24; 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 may be also single quoted and may contain Unicode characters.

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

```
<labeloroffset> ::= <id>
<labels> ::= <labeloroffset> [, <labeloroffset>]*
```

Rationale: *In a real assembler the syntax for <labeloroffset> might allow the direct specification of a number rather than requiring symbolic labels.*

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 may 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. The data label may not be used as a code label, and a code label may not be used as a data label. A particular code label name may not be declared more than once in a module.

```
<codeLabel> ::= <id> :
<dataLabel> ::= <id>
```

Example (informative):

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

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

5.5 Lists of Hex Bytes

A list of bytes consists simply of one or more hex bytes. Hex bytes are pairs of characters 0 – 9, a – f, and A – F.

```
<bytes> ::= <hexbyte> [<hexbyte>*]
```

5.6 Floating point numbers

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

1. Use the dot (".") for the decimal point and "e" or "E" in front of the exponent. Both the decimal point and the exponent are optional.
2. Indicate that the floating-point value is derived from an integer using the keyword float32 or float64 and indicating the integer in parentheses.

```
<float64> ::=
  float32 ( <int32> )
| float64 ( <int64> )
| <realnumber>
```

Example (informative):

```
5.5
1.1e10
float64(128)    // note: this converts the integer 128 to its fp value
```

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, and optional column number (preceded by a colon) and single quoted string that specifies the name of the file the line number is referring to

```
<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> ::=`	**Section**
`<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 (see Chapter 22). 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 should be set to allow access only within the defining assembly.
- Using a general-purpose syntax in ilasm. The non-terminal `<customDecl>` describes this grammar (see Chapter 20). For some attributes, called *pseudo-custom attributes*, this grammar actually results in setting special encodings within the metadata (see clause 20.2.1).
- Some attributes are required to be set based on the settings of other attributes or information within the metadata and are not visible from the syntax of ilasm at all. These attributes, called *hidden attributes*
- Security attributes are treated specially. There is special syntax in ilasm that allows the XML representing security attributes to be described directly (see Chapter 19). 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 Section 21.10, 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 Section 21.11.

5.10 *ilasm* Source Files

An input to ilasm is a sequence of declarations, defined as follows:

`<ILFile> ::=`	**Reference**
`<decl>*`	5.10

The complete grammar for a top level declaration is shown below. The following sections will concentrate on the various parts of this grammar.

`<decl> ::=`	**Reference**
`.assembly` `<dottedname>` **`{`** `<asmDecl>*` **`}`**	6.1
`|` **`.assembly extern`** `<dottedname>` **`{`** `<asmRefDecl>*` **`}`**	6.3
`|` **`.class`** `<classHead>` **`{`** `<classMember>*` **`}`**	9
`|` **`.class extern`** `<exportAttr>` `<dottedname>` **`{`** `<externClassDecl>*` **`}`**	6.7
`|` **`.corflags`** `<int32>`	6.1
`|` **`.custom`** `<customDecl>`	20

| **.data** <datadecl>	15.3.1
| **.field** <fieldDecl>	15
| **.file** [**nometadata**] <filename> [**.hash = (** <bytes> **)**] [**.entrypoint**]	6.2.3
| **.mresource** [**public** | **private**] <dottedname> [**(** <QSTRING> **)**] **{** <manResDecl>* **}**	6.2.2
| **.method** <methodHead> **{** <methodBodyItem>* **}**	14
| **.module** [<filename>]	6.4
| **.module extern** <filename>	6.5
| **.subsystem <**int32>	6.2
| **.vtfixup** <vtfixupDecl>	14.5.1
| <externSourceDecl>	5.7
| <securityDecl>	18

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 (see Section 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.
- Which of the types defined in other files of the assembly 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 to be executed (rather than dynamically loaded) the manifest shall reside in the module that contains the entry point.

While some programming languages introduce the concept of a *namespace*, there is no support in the CLI for this concept. 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 section contains informative text only.

The following picture should clarify the various forms of references:

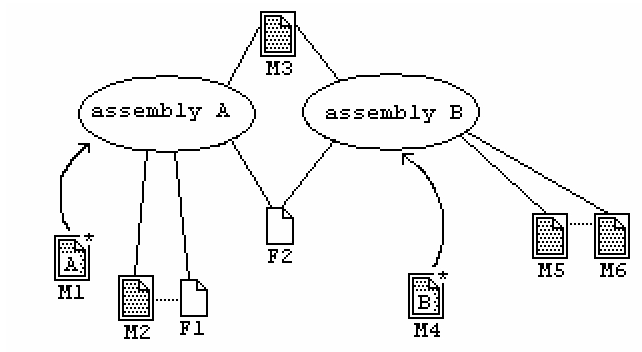


Figure 2: References

Eight files are shown in the picture. The name of each file is shown below the file. Files that declare a module have an additional border around them and have names beginning with M. The other two files have a name beginning with F. These files may be resource files, like 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. Assembly A references M2 and M3. 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 Section 21.2. The information for the manifest is created from the following portions of the grammar:

`<decl> ::=`	**Section**
`.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> [( <QSTRING> )] { <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: Since some platforms treat names in a case insensitive manner, two assemblies that have names that differ only in case should not be declared.

The **.corflags** directive sets a field in the CLI header of the output PE file (see clause 24.3.3.1). A conforming implementation of the CLI shall expect it to be 1. For backwards compatibility, the three least significant bits are reserved. Future versions of this standard may 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 directly executed (as opposed to used as a library for another program). It specifies the kind of application environment required for the program, by storing the specified value in the PE file header (see clause 24.2.2). While a full 32 bit integer may be supplied, a conforming implementation of the CLI need only respect two possible 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 (informative):
.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)
```

6.2.1 Information about the Assembly (<asmDecl>)

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

`<asmDecl> ::=`	**Description**	**Section**
`.custom <customDecl>`	Custom attributes	20
`.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, revision, and build	6.2.1.4
`| <securityDecl>`	Permissions needed, desired, or prohibited	19

6.2.1.1 Hash Algorithm

```
<asmDecl> ::= .hash algorithm <int32> | …
```

When an assembly consists of more than one file (see clause 6.2.3), the manifest for the assembly specifies both the name of the file and the cryptographic hash of the contents of the file. 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 SHA1(see Partition I) hash function and shall specify this algorithm by using a value of 32772 (0x8004).

Rationale: *SHA1 was chosen as the best widely available technology at the time of standardization (see Partition I). A single* algorithm *is chosen since all conforming implementations of the CLI would be required to implement all* algorithms *to ensure portability of executable images.*

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.

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

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 the assembly (using the SHA1 hash function) and then 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 "SHA1/RSA digital signature") can then be stored in the metadata 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, then the result stored into the signature.

A reference to an assembly (see Section 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).

6.2.1.4 Version Numbers

```
<asmDecl> ::= .ver <int32> : <int32> : <int32> : <int32> | …
```

The version number of the assembly, 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 standards.

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 virtual machine are required to implement it. Furthermore, this standard shall change one or both of the first two 32-bit integers specified for the **mscorlib** assembly so that it's version number may be used (if desired) to distinguish between different versions of the Execution Engine required to run programs conforming to that version of the standard.

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

the first of these is considered 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.

the second of these is considered the minor version number and assemblies with the same name and major version but different minor versions indicate significant enhancements but with intention to be backward compatible. This would be appropriate, for example, on a "point release" of a product or a fully backward compatible new version of a product.

the third of these is considered 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.

the fourth of these is considered 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.

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> ::=`	**Section**

`.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 and hence only available from within the assembly. The <dottedname> is the name of the resource, and the optional quoted string is a description of the resource.

`<manResDecl> ::=`	**Description**	**Section**
`.assembly extern <dottedname>`	Manifest resource is in external assembly with name <dottedname>.	6.3
`| .custom <customDecl>`	Custom attribute.	20
`| .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 clause 6.2.3) and the byte offset shall be zero Similarly, 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 (see Section 6.3).

6.2.3 Files in the Assembly

Assemblies may be associated with other files, e.g. 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 Section 21.19)

`<decl> ::=`	**Section**
`.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** may 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 shall generate an exception if it does not match. The algorithm used to calculate this hash value is specified with **.hash algorithm** (see clause 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

```
<asmRefDecl> ::= .assembly extern <dottedname> [ as <dottedname> ]
                 { <asmRefDecl>* }
```

An assembly mediates all accesses from the files that it contains to other assemblies. 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. The syntax **.assembly extern** as a top-level 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 **.assembly** directive in a case sensitive manner. (So, even though an assembly might be stored within a file, within a filesystem that is case-blind, the names stored internally within metadata are case-sensitive, and shall match exactly.)

`<asmRefDecl> ::=`	**Description**	**Section**
`.hash = ( <bytes> )`	Hash of referenced assembly	6.2.3
`| .custom <customDecl>`	Custom attributes	20
`| .culture <QSTRING>`	Culture of the referenced assembly	6.2.1.2
`| .publickeytoken = ( <bytes> )`	The low 8 bytes of the SHA1 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, revision, and build	6.2.1.4

These declarations are the same as those for **.assembly** declarations (clause 6.2.1), except for the addition of **.publickeytoken.** This declaration is used to store the low 8 bytes of the SHA1 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 "publickeytoken." Either can be used to validate that the same private key used to sign the assembly at compile time 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 may refuse to load an assembly for which the validation fails. A conforming implementation of the CLI may 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 full public key is cryptographically safer, but requires more storage space in the assembly reference.*

```
Example (informative):
.assembly extern MyComponents
{ .publickey = (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
}
```

6.4 Declaring Modules

All CIL files are modules and are referenced by a logical name carried in the metadata rather than their file name. See Section 21.16.

<decl> ::=	Section
| **.module** <filename>	
| …	5.10

```
Example (informative):
.module CountDown.exe
```

6.5 Referencing Modules

When an item is in the current assembly but part of a different module 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 (see Section 6.4) of the defining module. See Section 21.28.

<decl> ::=	Section
| **.module extern** <filename>	
| …	5.10

```
Example (informative):
.module extern Counter.dll
```

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 section.

<decl> ::=	Section
| **.class** <classHead> **{** <classMember>* **}**	9
| **.custom** <customDecl>	20

| **.data** <datadecl>	15.3.1
| **.field** <fieldDecl>	15
| **.method** <methodHead> **{** <methodBodyItem>* **}**	14
| <externSourceDecl>	5.7
| <securityDecl>	18
| …	

6.7 Exported Type Definitions

The manifest module, of which there can only be one per assembly, includes the **.assembly** statement. 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> ::=	**Section**
.class extern <exportAttr> <dottedname> **{** <externClassDecl>* **}**	

<externClassDecl> ::=	**Section**
.file <dottedname>	
| **.class extern** <dottedname>	
| **.custom** <customDecl>	20

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** statement shall be included in A.EXE.

Conversely, a public class "Bar" defined in A.EXE does not need any **.class extern** statement.

Rationale: *Tools should be able to retrieve a single module, the manifest module, to determine the complete set 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.*

7 Types and Signatures

The metadata provides mechanisms to both *define* types and *reference* types. Chapter 9 describes the metadata associated with a type definition, regardless of whether the type is an interface, class or a value type.

The mechanism used to reference types is divided into two parts. The first is the creation of a logical description of user-defined types that are referenced but (typically) not defined in the current module. These are stored in a logical table in the metadata (see Section 21.35).

The second is 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 Section 22.1.15

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**	**Section**
`bool`	Boolean	7.2
`|` **`boxed`** `<typeReference>`	Boxed user-defined value type	
`|` **`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	13.5

`|` **`native int`**	Signed integer whose size varies depending on platform (32- or 64-bit)	7.2
`|` **`native unsigned int`**	Unsigned integer whose size varies depending on platform (32- or 64-bit)	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**	13.4
`|` `<type>` **`*`**	Unmanaged pointer to <type>	13.4
`|` `<type>` **`[`** `[<bound> [,<bound>]*]` **`]`**	Array of <type> with optional rank (number of dimensions) and bounds.	13.1and 13.2
`|` `<type>` **`modopt (`** `<typeReference>` **`)`**	Custom modifier that may 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, created by **mkrefany** and used by **refanytype** or **refanyval**.	7.2
`| valuetype <typeReference>`	User defined value type (unboxed)	12
`| unsigned int8`	Unsigned 8-bit integers	7.2
`| unsigned int16`	Unsigned 16-bit integers	7.2
`| unsigned int32`	Unsigned 32-bit integers	7.2
`| unsigned int64`	Unsigned 64-bit integers	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 mechanism for specifying types, by just allowing type names (e.g. "`System.GC`") to be used instead of the full algebra (e.g. "class `System.GC`"). These are called *type specifications*:

`<typeSpec> ::=`	**Section**
`[ [.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 (see Chapter 20) except that modifiers are part of a signature rather than 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 concept of const in the C programming language can be modelled with an optional modifier since the caller of a method that has a constant 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.

7.1.2 pinned

The signature encoding for pinned shall appear only in signatures that describe local variables (see clause 14.4.1.3). 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.*

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 (see Chapter 9).

A `<typeReference>` is used to capture the full name and resolution scope.

`<typeReference> ::=`
`[<resolutionScope>] <dottedname> [/ <dottedname>]*`

`<resolutionScope> ::=`
`[ .module <filename> ]`
`| [ <assemblyRefName> ]`

`<assemblyRefName> ::=`	**Section**
`<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 (for example, when referencing a nested type that has **compilercontrolled** accessibility) or convenient (for example, in some one-pass compilers) a type reference is equivalent and may be used.

- **Different module, current assembly**. The resolution scope shall be a module reference syntactically reprented using the notation **[.module** `<filename>`**]**. The type shall be resolved to a definition only if the referenced module (see Section 6.4) and type (see Section 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 (see Section 6.3), the type shall be declared in the referenced assembly's manifest, and the type shall be marked as exported from that assembly (see section 6.7 and clause 9.1.1).

- For nested types, the resolution scope is always the enclosing type. **(**See Section 9.6). This is indicated syntactically by using a slash ("/") to separate the enclosing type name from the nested type's name

```
Example (informative):
The proper way to refer to a type defined in the base class library. The name of the
type is System.Console and it is 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
```

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 the types. All encoding values from 0 through 63 are reserved for backward compatibility with existing implementations of the CLI. Values 64 through 127 are reserved for future use in this and related Standards.

`<nativeType> ::=`	**Description**	**Name in class library**
`[ ]`	Native array. Type and size are determined at runtime from the actual marshaled array.	LPArray
`| bool`	Boolean. 4-byte integer value where a non-zero value represents TRUE and 0 represents FALSE.	Bool
`| float32`	32-bit floating point number.	FLOAT32
`| float64`	64-bit floating point number.	FLOAT64
`| [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	unsigned int8 or int8
`| [unsigned] int16`	Signed or unsigned 16-bit integer	unsigned int16 or int16
`| [unsigned] int32`	Signed or unsigned 32-bit integer	unsigned int32 or int32
`| [unsigned] int64`	Signed or unsigned 64-bit integer	unsigned int64 or int64

| **lpstr**	A pointer to a null terminated array of ANSI characters. Code page is implementation specific.	LPStr
| **lptstr**	A pointer to a null terminated array of platform characters (ANSI or Unicode). Code page and character encoding are implementation specific.	LPTStr
| **lpvoid**	An untyped pointer, platform specifies size.	LPVoid
| **lpwstr**	A pointer to a null terminated array of Unicode characters. 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 1) will specify the additional number of elements in the array. Can only be applied to methods, not fields	LPArray

Example (informative):

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

Method M1 takes two arguments: an int32, and an array of 5 bools

++++++++++

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

Method M2 takes two arguments: an int32, and an array of bools: 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 bools: the number of elements
in that array is given as 7 plus the value of the first parameter
```

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 may have any of the 7 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 the 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 clause 22.1.14.

8.2 Accessibility

Accessibility is encoded directly in the metadata. See, for example, clause 21.24.

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 (method or field) 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 (for fields) or signature (for 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 (see clause 14.4.2.3) and correctly chosing the type used to resolve a method reference (see clause 14.1.3).

9 Defining Types

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

`<decl> ::=`	**Section**
`.class <classHead> { <classMember>* }`	9
`| …`	

The logical metadata table created by this declaration is specified in Section 21.34.

Rationale: *For historical reasons, many of the syntactic classes 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.*

9.1 Type Header (<classHead>)

A type header consists of

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

`<classHead> ::=`
`<classAttr>* <id> [extends <typeReference>] [implements <typeReference> [, <typeReference>]*]`

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

The implements keyword defines 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 Chapter 10 and Chapter 11.

Example (informative):

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

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 sections.

A type can have any number of custom attributes attached. Custom attributes are attached as described in Chapter 20. The other (predefined) attributes of a type may be grouped into attributes that specify visibility, type layout information, type semantics information, inheritance rules, interoperation information, and information on special handling. The following subsections provide additional information on each group of predefined attributes.

`<classAttr> ::=`	**Description**	**Section**
abstract	Type is **abstract**.	9.1.4
| **ansi**	Marshal strings to platform as **ANSI**.	9.1.5

`| auto`	Auto layout of type.	9.1.2
`| autochar`	Marshal strings to platform based on platform.	9.1.5
`| beforefieldinit`	Calling static methods does not initialize type.	9.1.6
`| explicit`	Layout of fields is provided explicitly.	9.1.2
`| interface`	Interface declaration.	9.1.3
`| nested assembly`	Assembly accessibility for nested type.	9.1.1
`| nested famandassem`	Family and Assembly accessibility for nested type.	9.1.1
`| nested family`	Family accessibility for nested type.	9.1.1
`| nested famorassem`	Family or Assembly accessibility for nested type.	9.1.1
`| nested private`	Private accessibility for nested type.	9.1.1
`| nested public`	Public accessibility for nested type.	9.1.1
`| private`	Private visibility of top-level type.	9.1.1
`| public`	Public visibility of top-level type.	9.1.1
`| rtspecialname`	Special treatment by runtime.	9.1.6
`| sealed`	The type cannot be subclassed.	9.1.4
`| sequential`	The type is laid out sequentially.	9.1.2
`| serializable`	Type may be serialized.	9.1.6
`| specialname`	Special treatment by tools.	9.1.6
`| unicode`	Marshal strings to platform as Unicode.	9.1.5

9.1.1 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 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.

9.1.2 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.

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

explicit: the layout of the fields is explicitly provided (see Section 9.7).

sequential: the CLI shall lay out the fields in sequential order, based on the order of the fields in the logical metadata table (see Section 21.15).

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.*

9.1.3 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` a value type shall be defined (see Chapter 12). Otherwise, a class shall be defined (see Chapter 10).

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

9.1.4 Inheritance Attributes

```
<classAttr> ::= …
| abstract
| sealed
```

Attributes that specify special semantics are **abstract** and **sealed**. These attributes may be used together.

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

sealed specifies that a type shall not have subclasses. 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 may be necessary when the implementation of a set of virtual methods for a single class (typically inherited from different interfaces) becomes interdependent or depends critically on implementation details not visible to potential subclasses.*

*A type that is both **abstract** and **sealed** should have only static members, and serves as what some languages call a namespace.*

9.1.5 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**.

ansi specifies that marshalling shall be to and from ANSI strings

unicode specifies that marshalling shall be to and from Unicode strings

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

9.1.6 Special Handling Attributes

`<classAttr> ::= …`
`| **beforefieldinit**`
`| **serializable**`
`| **specialname**`
`| **rtspecialname**`

These attributes may be combined in any way.

beforefieldinit instructs the CLI that it need not initialize the type before a static method is called. See clause 9.5.3.

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

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**

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

9.2 Body of a Type Definition

A type may 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 Section 9.6.

`<classMember> ::=`	**Description**	**Section**
`.class <classHead> { <classMember>* }`	Defines a nested type.	9.6
`| .custom <customDecl>`	Custom attribute.	20
`| .data <datadecl>`	Defines static data associated with the type.	15.3
`| .event <eventHead> { <eventMember>* }`	Declares an event.	17
`| .field <fieldDecl>`	Declares a field belonging to the type.	15
`| .method <methodHead> { <methodBodyItem>* }`	Declares a method of the type.	14
`| .override <typeSpec> :: <methodName> with <callConv> <type> <typeSpec> :: <methodName> ( <parameters> )`	Specifies that the first method is overridden by the definition of the second method.	9.3.2
`| .pack <int32>`	Used for explicit layout of fields.	9.7

`| .property <propHead> { <propMember>* }`	Declares a property of the type.	16
`| .size <int32>`	Used for explicit layout of fields.	9.7
`| <externSourceDecl>`	**.line**	5.7
`| <securityDecl>`	**.permission** or **.capability**	19

9.3 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 Section 14.4) and not specifying it to be newslot (see clause 14.4.2.3). An existing method body may also be used to implement a given virtual declaration using the **.override** directive (see clause 9.3.2).

9.3.1 Introducing a Virtual Method

A virtual method is introduced in the inheritance hierarchy by defining a virtual method (see Section 14.4). The versioning semantics differ depending on whether or not the definition is marked as newslot (see clause 14.4.2.3):

If the definition is marked **newslot** then the definition *always* creates a new virtual method, even if a base class provides a matching virtual method. Any reference to the virtual method created before the new virtual function was defined will continue to refer to the original definition.

If the definition is not marked **newslot** then it creates a new virtual method only if there is no virtual method of the same name and signature inherited from a base class. If the inheritance hierarchy changes so that the definition matches an inherited virtual function the definition will be treated as a new implementation of the inherited function.

9.3.2 The .override Directive

The **.override** directive specifies that a virtual method should be implemented (overridden), in this type, by a virtual method with a different name but with the same signature. It 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 (see clause 14.1.4).

`<classMember> ::=`	**Section**
`.override <typeSpec> :: <methodName> with <callConv> <type> <typeSpec> :: <methodName> ( <parameters> )`	
`| …`	9.2

The first `<typeSpec> :: <methodName>` pair specifies the virtual method that is being overridden. It shall reference 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 and the actual metadata format (see Section 21.25)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.*

Example (informative):

The following example shows a typical use of the **.override** directive. A method implementation is provided for a method declared in an interface (see Chapter 11).

```
.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.

9.3.3 Accessibility and Overriding

If a type overrides an inherited method, it may *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 subclass can apply that accessibility, a "No" denotes it is illegal.

Table 7.1: Legal Widening of Access to a Virtual Method

Subclass	**Base type Accessibility**					
	private	family	assembly	famandassem	famorassem	public
private	Yes	No	No	No	No	No
family	Yes	Yes	No	No	If not in same assembly	No
assembly	Yes	No	Same assembly	No	No	No
famandassem	Yes	No	No	Same assembly	No	No
famorassem	Yes	Yes	Same assembly	Yes	Same assembly	No
public	Yes	Yes	Yes	Yes	Yes	Yes

Note: A method may be overridden even if it may not be accessed by the subclass.

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, may be used inside the same assembly.

A special rule applies to **famorassem**, as shown in the table. This is the only case where the accessibility is apparently narrowed by the subclass. A **famorassem** method may 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.*

9.4 Method Implementation Requirements

A type (concrete or abstract) may 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 parent
- 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

9.5 Special Members

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

9.5.1 Instance constructors

Instance constructors initialize an instance of a type. An instance constructor is called when an instance of a type is created by the newobj instruction (see Partition III). Instance constructors shall be instance (not static or virtual) methods, they shall be named **.ctor** and marked both **rtspecialname** and **specialname** (see clause 14.4.2.6). Instance constructors may take parameters, but shall not return a value. Instance constructors may be overloaded (i.e. a type may have several instance constructors). Each instance constructor shall have a unique signature. Unlike other methods, instance constructors may write into fields of the type that are marked with the **initonly** attribute (see clause 15.1.2).

Example (informative):

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
  }
}
```

9.5.2 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.

9.5.3 Type Initializer

Types may contain special methods called *type initializers* to initialize the type itself.

All types (classes, interfaces, and value types) may have a type initializer. This method shall be static, take no parameters, return no value, be marked with **rtspecialname** and **specialname** (see clause 14.4.2.6), and be named **.cctor**.

Like instance initializers, type initializers may write into static fields of their type that are marked with the initonly attribute (see clause 15.1.2).

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.

9.5.3.1 Type Initialization Guarantees

The CLI shall provide the following guarantees regarding type initialization (but see also clause 9.5.3.2 and clause 9.5.3.3):

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

9.5.3.2 Relaxed Guarantees

A type can be marked with the attribute **beforefieldinit** (see clause 9.1.6) to indicate that all the guarantees specified in clause 9.5.3.1 are not required. In particular, the final requirement of guarantee 1 need not be provided: the type initializer need not run 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.*

9.5.3.3 Races and Deadlocks

In addition to the type initialization guarantees specified in clause 9.5.3.1 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
    stfld 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 <u>*completed*</u> *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 the algorithm 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, 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 parent 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.

9.6 Nested Types

Nested types are specified in Partition I. Interfaces may be nested inside of classes and value types, but classes and value types shall not be nested inside of interfaces. For information about the logical tables associated with nested types, see Section 21.29.

Note: A nested type is not associated with an instance of its enclosing type. The nested type has its own base type and may be instantiated independent of the 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 may access any members of its enclosing type, including private members, as long as the member is static or the nested type has a reference to an instance of the enclosing type. Thus, by using nested types a type may give access to its private members to another type.

On the other side, the enclosing type may not 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.

```
Example (informative):

The following example shows a class declared inside another class. Both classes declare
a field. The nested class may access both fields, while the enclosing class does not
have access to the field b.

.class private auto autochar CounterTextBox

       extends [System.Windows.Forms]System.Windows.Forms.TextBox

       implements [.module Counter]IcountDisplay

{ .field static private int32 a

  /* Nested class. Declares the NegativeNumberException */

  .class nested assembly NonPositiveNumberException extends [mscorlib]System.Exception

  { .field static private int32 b

    // body of nested class

  } // end of nested class NegativeNumberException

}
```

9.7 Controlling Instance Layout

The CLI supports both sequential and explicit layout control, see clause 9.1.2. For explicit layout it is also necessary to specify the precise layout of an instance, see also Section 21.18 and Section 21.16.

```
<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. This form of explicit layout control shall not be used with global fields specified using the at notation (see clause 15.3.2).

Offset values shall be 0 or greater; they cannot be negative. It is possible to overlap fields in this way, even though it is not recommended. The field may be accessed using pointer arithmetic and **ldind** to load the field indirectly or **stind** to store the field indirectly (see Partition III). See Section 21.18 and Section 21.16 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 addresses which are a multiple of the specified number, or at natural alignment for that field type, whichever is *smaller*. e.g., **.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 to say, placed on addresses that are a multiple of 4. The integer following **.pack** shall be one of 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 directive **.size** specifies that a memory block of the specified amount of bytes shall be allocated for an instance of the type. e.g., **.size** 32 would create a block of 32 bytes for the instance. The value specified shall be greater than or equal to the calculated size of the class, based upon its field sizes and any **.pack** 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.

```
Example (informative):

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
}
```

9.8 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.

It is simplest to understand global fields and methods in the CLI by imagining 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. 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 in the output class.
- If there are duplicates and two or more have an accessibility other than **compilercontrolled** an error has occurred.

10 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`) shall declare exactly one parent class. A class shall declare zero or more interfaces that it implements (see Chapter 11). A concrete class may be instantiated to create an object, but an **abstract** class (see clause 9.1.4) shall not be instantiated. A class may define fields (static or instance), methods (static, instance, or virtual), events, properties, and nested types (classes, value types, or interfaces).

Instances of a class (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 is assignment compatible with all class types even though it is not an instance of any particular class.

11 Semantics of Interfaces

Interfaces, as specified in Partition I, define a contract that other types may implement. Interfaces may have static fields and methods, but they shall not have instance fields or methods. Interfaces may define virtual methods, but only if they are **abstract** (see Partition I and clause 14.4.2.4).

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.*

Interfaces may be nested inside any type (interface, class, or value type). Classes and value types shall not be nested inside of interfaces.

11.1 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 Section 9.1) 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 that fact that an interface represents a contract that may 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.

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.

Example (informative):

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
}
```

11.2 Implementing Virtual Methods on Interfaces

Classes that implement an interface (see Section 11.1) are required to provide implementations for the **abstract** virtual methods defined by the 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` (see clause 14.1.4).

The Virtual Execution System shall determine the appropriate implementation of a virtual method to be used for an interface **abstract** method using the following algorithm.

- If the parent 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
  - if the class defines any **public virtual newslot** functions 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 available on this class (directly or inherited) and use these to implement the corresponding methods on the interface.
- Apply all `MethodImpls` 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 not valid.

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.*

12 Semantics of Value Types

In contrast to classes, 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, or dates.*

Like other types, value types may have fields (static or instance), methods (static, instance, or virtual), properties, events, and nested types. A value type may be converted into 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 may 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 (see Section 12.3).

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 may be used for boxed value types. 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 reference types do (see Section 9.7). This is especially important when values are imported from native code.

12.1 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>
```

12.2 Initializing Value Types

Like classes, value types may have both instance constructors (see clause 9.5.1) and type initializers (see clause 9.5.3). Unlike classes that are automatically initialized to null, however, the following rules constitute the only guarantee about the initilisation of (unboxed) value types:

- Static variables shall be initialized to zero when a type is loaded (see clause 9.5.3.3), 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 appropriate bit in the method header (see clause 24.4.4) 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 allow threads to be created outside of the CLI and then passed to the CLI for management.*

Note: Boxed value types are classes and follow the rules for classes.

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 (informative):

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
}
```

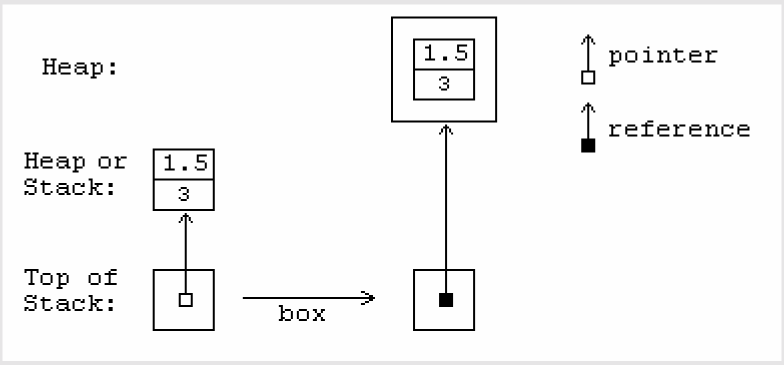
12.3 Methods of Value Types

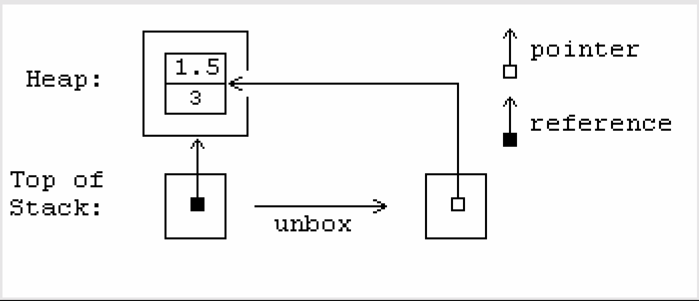
Value types may have static, instance and virtual methods. static methods of value types are defined and called the same way as static methods of class types. As with classes, both instance and virtual methods of a boxed or unboxed value type may be called using the **call** instruction. The **callvirt**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





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.*

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

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

```
Example (informative):

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()
```

13 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.

13.1 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, or single-dimension arrays with a non-zero lower bound (see Section 13.2). Two vectors are 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 index beyond the end of the vector, attempts to store the incorrect type of data into an element of a vector, and attempts to take addresses of elements of a vector with an incorrect data type. See Partition III.

Example (informative):

Declaring a vector of Strings:

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

Declaring 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 four 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
```

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.

13.2 Arrays

While vectors (see Section 13.1) have direct support through CIL instructions, all other arrays are supported by the VES by creating subtypes of the abstract class System.Arrray (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 may be omitted

`| <int32>`	zero lower bound, <int32> upper bound
`| <int32> ...`	lower bound only specified
`| <int32> ... <int32>`	both bounds specified

The fundamental operations provided by the CIL instruction set for vectors are provided by methods on the class created by the VES.

The VES shall provide two constructors for arrays. One takes a sequence of numbers giving the number of elements in each dimension (a lower bound of zero is assumed). The second takes twice as many arguments: a sequence of lower bounds, one for each dimension; followed by a sequence of lengths, one for each dimension (where length is the number of elements required).

In addition to array constructors, the VES shall provide the instance methods `Get`, `Set`, and `Address` to access specific elements and compute their addresses. These methods take a number for each dimension, to specify the target element. In addition, `Set` takes an additional final argument specifying the value to store into the target element.

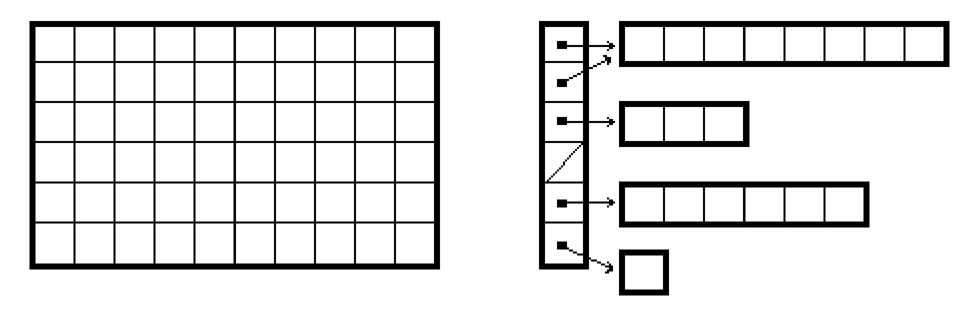
Example (informative):

Creates an array, MyArray, of strings with two dimensions, with indexes 5..10 and 3..7. Stores the string "One" into MyArray[5, 3], retrieves it and prints it out. Then 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
}
```

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 be of 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, 6 and 1.

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 are themselves either the base elements or (recursively) jagged arrays.

End of informative text

13.3 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 (see Chapter 12). The symbols of an enum are represented by an *underlying* type: one of { **bool**, **char**, **int8**, **unsigned int8**, **int16**, **unsigned int16**, **int32**, **unsigned int32**, **int64**, **unsigned int64**, **float32**, **float64**, **native int**, **unsigned native int** }

Note: The CLI does *not* provide a guarantee that values of the enum type are integers corresponding to one of the symbols (unlike Pascal). 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.

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 (see clause 9.1.2); 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 (see Section 15.1); and they shall not be initialized with the **initobj** instruction.

Rationale: *These restrictions allow a very efficient implementation of enums.*

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 (see Section 15.2).

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 (see Chapter 12) 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 (informative):

Declare an enum type, then create a local variable of that type. Store a constant of the underlying type into the enum (showing automatic coercsion 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
 }
```

13.4 Pointer Types

`<type> ::= …`	**Section**
`  | <type> &`	13.4.2
`  | <type> *`	13.4.1

A *pointer type* shall be defined by specifying a signature that includes the type for the location it points at. A *pointer* may be *managed* (reported to the CLI garbage collector, denoted by &, see clause 13.4.2) or *unmanaged* (not reported, denoted by *, see clause 13.4.1)

Pointers may contain the address of a field (of an object or value type) or 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 pointer:

- *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, results in a pointer of the same kind
- Subtracting an integer value (number of bytes) from a pointer 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 may be used to load the data item to which it points. Similarly, the **stind** class 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).

13.4.1 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 legal 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

13.4.2 Managed Pointers

Managed pointers (&) may 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 by-ref 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.

    e. 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

13.5 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 may 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 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.

Example (informative):

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 call 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
}
```

13.6 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 an immediate base type of `System.MulticastDelegate`, which in turns has an immediate base type of `System.Delegate` (see Partition IV).

Delegates shall be declared sealed, and the only members a Delegate shall have are either two or four methods as specified here. These methods shall be declared **runtime** and **managed** (see clause 14.4.3). They shall not have a body, since it shall be automatically created by the VES. Other methods available on delegates are inherited from the classes `System.Delegate` and `System.MulticastDelegate` in the Base Class Library (see Partition IV).

Rationale: *A better design would be to simply have delegate classes derive directly from* `System.Delegate`*. Unfortunately, backward compatibility with an existing CLI does not permit this design.*

The instance constructor (named **.ctor** and marked **specialname** and **rtspecialname**, see clause 9.5.1) shall take exactly two parameters. The first parameter shall be of type `System.Object` and the second parameter shall be of 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 subclasses) 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 have the same signature (return type, parameter types, calling convention, and modifiers, see Section 7.1) as the target method. When actually called the arguments passed shall match the types specified in this signature.

The `BeginInvoke` method (see clause 13.6.2.1), if present, shall be **virtual** 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 (see clause 13.6.2) shall be **virtual** 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`.

Example (informative):

The following example declares a Delegate used to call functions that take a single integer and return void. It provides all four methods so it can be called either synchronously or asynchronously. Because there are no parameters that 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.MulticastDelegate
 { .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 {}
}
```

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.

13.6.1 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 (informative):

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 parent 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 instead 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
```

Note: The *code sequence above shall use* ***dup*** *–not* **ldloc** *InstanceOne twice. The* ***dup*** *code sequence is easily recognized as typesafe, whereas alternatives would require more complex analysis.* Verifiability of code is discussed in Partition III

13.6.2 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 may throw exceptions. Any unhandled exception propagates to the caller via the `EndInvoke` method.

13.6.2.1 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 (see clause 13.6.1), but has three differences:

- It has a 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 may 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.*

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.

13.6.2.2 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 may execute concurrently) as well as any managed pointers that were passed as arguments (so their return values can be provided).

14 Defining, Referencing, and Calling Methods

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

```
<decl> ::= …
   | .method <methodHead>  { <methodBodyItem>* }
```

as well as inside a type:

```
<classMember> ::= …
   | .method <methodHead>  { <methodBodyItem>* }
```

14.1 Method Descriptors

There are four constructs in *ilasm* connected with methods. These correspond with different metadata constructs, as described in Chapter 21.

14.1.1 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 <methodHead> but no <methodBodyItem>s. These are used at callsites 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.*

14.1.2 Method Definitions

A *Method*, or method definition, supplies the method name, attributes, signature and body. That is, a method definition provides a <methodHead> 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 Chapter 11.

14.1.3 Method References

A *MethodRef*, or method reference, is a reference to a method. It is used when a method is called whose 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 Chapter 21.23.

14.1.4 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 Section 21.25.

14.2 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 may only be defined in classes or value types, but not in interfaces or outside of a type (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 Section 14.3) to specify the type of the *this* pointer, overriding the default values specified here
4. the *this* pointer may 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 depends 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 subclass may *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.*

14.3 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. It 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:

- **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 portable. Four more are reserved for future standardization, and two are available for non-standard experimental use.

(By "portable" is meant a feature that is available on all conforming implementations of the CLI)

14.4 Defining Methods

```
<methodHead> ::=
  <methAttr>* [<callConv>] [<paramAttr>*] <type>
              [marshal ( [<nativeType>] )]
              <methodName> ( <parameters> ) <implAttr>*
```

The method head (see also Chapter 11) consists of

- the calling convention (`<callConv>`, see Section 14.3)
- any number of predefined method attributes (`<paramAttr>`, see clause 14.4.2)
- a return type with optional attributes
- optional marshalling information (see Section 7.4)
- a method name
- a signature
- and any number of implementation attributes (`<implAttr>`, see clause 14.4.3)

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 may 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 (see Section 21.30) 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 are not.

`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 may 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 (see clause 14.4.1.4).

14.4.1 Method Body

The method body shall contain the instructions of a program. However, it may 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**	**Section**
`.custom <customDecl>`	Definition of custom attributes.	20
`| .data <datadecl>`	Emits data to the data section	15.3
`| .emitbyte <unsigned int8>`	Emits a byte to the code section of the method.	14.4.1.1
`| .entrypoint`	Specifies that this method is the entry point to the application (only one such method is allowed).	14.4.1.2
`| .locals [init] ( <localsSignature> )`	Defines a set of local variables for this method.	14.4.1.3
`| .maxstack <int32>`	int32 specifies the maximum number of elements on the evaluation stack during the execution of the method	14.4.1
`| .override <typeSpec>::<methodName>`	Use current method as the implementation for the method specified.	9.3.2
`| .param [ <int32> ] [= <fieldInit>]`	Store a constant <fieldInit> value for parameter <int32>	14.4.1.4
`| <externSourceDecl>`	**.line** or **#line**	5.7
`| <instr>`	An instruction	Partition V
`| <id> :`	A label	5.4
`| <securityDecl>`	**.permission** or **.permissionset**	19
`| <sehBlock>`	An exception block	18

14.4.1.1 .emitbyte

```
<methodBodyItem> ::= …
   | .emitbyte <unsigned int8>
```

Emits an unsigned 8 bit value directly into the CIL stream of the method. The value is emitted at the position where the directive appears.

Note: the **.emitbyte** directive is used for generating tests. It is not required in generating regular programs

14.4.1.2 .entrypoint

```
<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 may be a global method or may 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 may 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.

Example (informative):

The following example prints the first argument and return 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
}
```

14.4.1.3 .locals

The **.locals** statement declares local variables (see Partition I) for the current method.

```
<methodBodyItem> ::= …
   | .locals  [init] ( <localsSignature> )
<localsSignature> ::= <local> [, <local>]*
<local> ::= <type> [<id>]
```

The `<id>`, if present, is the name of the local.

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.

14.4.1.4 .param

```
<methodBodyItem> ::= …
   | .param  [ <int32> ] [= <fieldInit>]
```

Stores in the metadata a constant value associated with method parameter number <int32>, see Section 21.9. While the CLI requires that a value be supplied for the parameter, some tools may 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 is the first parameter of the method, and so forth.

Note: The CLI attaches no semantic whatsoever to these values – it is entirely up to compilers to implement any semantic they wish (eg so-called default argument values)

14.4.2 Predefined Attributes on Methods

`<methAttr> ::=`	**Description**	**Section**
abstract	The method is **abstract** (shall also be virtual).	14.4.2.4
| **assembly**	Assembly accessibility	14.4.2.1
| **compilercontrolled**	Compiler-controlled accessibility.	14.4.2.1
| **famandassem**	Family and Assembly accessibility	14.4.2.1
| **family**	Family accessibility	14.4.2.1
| **famorassem**	Family or Assembly accessibility	14.4.2.1
| **final**	This virtual method cannot be overridden by subclasses.	14.4.2.2
| **hidebysig**	Hide by signature. Ignored by the runtime.	14.4.2.2
| **newslot**	Specifies that this method shall get a new slot in the virtual method table.	14.4.2.3
| **pinvokeimpl (** <QSTRING> [**as** <QSTRING>] <pinvAttr>* **)**	Method is actually implemented in native code on the underlying platform	14.4.2.5
| **private**	Private accessibility	14.4.2.1
| **public**	Public accessibility.	14.4.2.1
| **rtspecialname**	The method name needs to be treated in a special way by the runtime.	14.4.2.6
| **specialname**	The method name needs to be treated in a special way by some tool.	14.4.2.6
| **static**	Method is static.	14.4.2.2
| **virtual**	Method is virtual.	14.4.2.2

The following combinations of predefined attributes are illegal:

- **static** combined with any of **final**, **virtual**, or **newslot**
- **abstract** combined with any of **final** or **pinvokeimpl**
- **compilercontrolled** combined with any of **virtual**, **final**, **specialname** or **rtspecialname**

14.4.2.1 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.

14.4.2.2 Method Contract Attributes

```
<methAttr> ::= …
| final
| hidebysig
| static
| virtual
```

These attributes may be combined, except a method shall not be both **static** and **virtual**; only **virtual** methods may be **final;** and abstract methods shall not be **final**.

final methods shall not be overridden by subclasses 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 parent 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 use a hide-by-name semantics (C++) while others use a hide-by-name-and-signature semantics (C#, Java™)*

Static and **virtual** are described in Section 14.2.

14.4.2.3 Overriding Behavior

```
<methAttr> ::= …
   | newslot
```

newslot shall only be used with virtual methods. See Section 9.3.

14.4.2.4 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 subclass. Abstract methods shall only appear in **abstract** types (see clause 9.1.4).

14.4.2.5 Interoperation Attributes

```
<methAttr> ::= …
   | pinvokeimpl ( <QSTRING> [as <QSTRING>] <pinvAttr>* )
```

See clause 14.5.2and Section 21.20.

14.4.2.6 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.

14.4.3 Implementation Attributes of Methods

`<implAttr> ::=`	**Description**	**Section**
`cil`	The method contains standard CIL code.	14.4.3.1
`| forwardref`	The body of this method is not specified with this declaration.	14.4.3.3
`| internalcall`	Denotes the method body is provided by the CLI itself	14.4.3.3
`| managed`	The method is a managed method.	14.4.3.2
`| native`	The method contains native code.	14.4.3.1
`| noinlining`	The runtime shall not expand the method inline.	14.4.3.3
`| runtime`	The body of the method is not defined but produced by the runtime.	14.4.3.1
`| synchronized`	The method shall be executed in a single threaded fashion.	14.4.3.3
`| unmanaged`	Specifies that the method is unmanaged.	14.4.3.2

14.4.3.1 Code Implementation Attributes

```
<implAttr> ::= …
   | cil
   | native
   | runtime
```

These attributes are 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 (see clause 14.5.2) 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 method of delegates (see Section 13.6).

14.4.3.2 Managed or Unmanaged

```
<implAttr> ::= …
```

```
    | managed
    | unmanaged
```

These shall not be combined. Methods implemented using CIL are managed. Unmanaged is used primarily with PInvoke (see clause 14.5.2).

14.4.3.3 Implementation Information

```
<implAttr> ::= …
    | forwardref
    | internalcall
    | noinlining
    | synchronized
```

These attributes may 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.

Rationale: *specifying that a method not be inlined ensures that it remains 'visible' for debugging (eg 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.*

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 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 may 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 may be done by the runtime for optimization purposes.

14.4.4 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.

14.4.5 vararg Methods

vararg methods accept a variable number of arguments. They shall use the **vararg** calling convention (see Section 14.3).

At each call site, a method reference shall be used to describe the types of the actual arguments that are passed. The fixed part of the argument list shall be separated from the additional arguments with an ellipsis (see Partition I).

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

Example (informative):

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 System.ArgIterator it, int32 x)
  ldloca it                           // initialize the iterator
  initobj valuetype System.ArgIterator
  ldloca it
  arglist                             // obtain the argument handle
  call   instance void System.ArgIterator::.ctor(valuetype
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 System.ArgIterator::GetNextArg()
  call   object System.TypedReference::ToObject(typedref) // retrieve the object
  castclass System.Int32              // cast and unbox
  unbox  int32
  cpobj  int32                        // copy the value into x
  // first vararg argument is stored in x
  ret
}
```

14.5 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 dependent 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 (see clause 14.5.1).
- A mechanism for marking certain method definitions as being implemented in unmanaged code (called *platform invoke*, see clause 14.5.2).
- A mechanism for marking call sites used with method pointers to indicate that the call is to an unmanaged method (see clause 14.5.3).
- A small set of pre-defined data types that can be passed (marshaled) using these mechanisms on all implementations of the CLI (see clause 14.5.4). The set of types is extensible through the use of custom attributes and modifiers, but these extensions are platform-specific.

14.5.1 Method Transition Thunks

Note: This mechanism is not part of the Kernel Profile, so it may not be present in all conforming implementations of the CLI. See Partition IV.

In order to call from unmanaged code into managed 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 marshalling). Both of these problems are solved by the **.vtfixup** directive. This directive may appear several times only at the top level of a CIL assembly file, as shown by the following grammar:

`<decl> ::=`	**Section**
`.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 is loaded into memory for execution. The declaration specifies the number of entries in the table, what kind of method pointer 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 and **int32** is 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 may simply emit the tokens as byte literals into a block specified using the **.data** directive.

14.5.2 Platform Invoke

Methods defined in native code may 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** (see clause 14.4.2.4).

`<methAttr> ::=`	**Description**	**Section**
`pinvokeimpl (` `<QSTRING>` `[`**`as`** `<QSTRING>]` `<pinvAttr>*` **`)`**	`Implemented in native code`	
`| …`		14.4.2

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 may 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 (ie, outside of any type), may 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 specificy the calling conventions for native code.

Example (informative):

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 {}
```

14.5.3 Via Function Pointers

Unmanaged functions can also be called via function pointers. There is no difference between calling managed or unmanaged functions with pointers. However, the unmanaged function needs to be declared with **pinvokeimpl** as described in clause 14.5.2. Calling managed methods with function pointers is described in Section 13.5

14.5.4 Data Type Marshaling

While data type marshaling is necessarily platform-dependent, this standard specifies a minimum set of data types that shall be supported by all conforming implementations of the CLI. Additional data types may be supported in an implementation-dependent 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 *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.

15 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 shared across all instances of that type, instance fields are associated with a particular instance of that type. When instantiated, the instance has its own copy of that 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: (see Section 21.15)

```
<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 (see Section 9.7). 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 (see Section 15.2)
- type
- name
- optionally either a <fieldInit> form or a data label

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 may, but do not need to, be assigned a data label.

```
Example (informative):
.field private class [.module Counter.dll]Counter counter
```

15.1 Attributes of Fields

Attributes of a field specify information about accessibility, contract information, interoperation attributes, as well as information on special handling.

The following subsections contain additional information on each group of predefined attributes of a field.

`<fieldAttr> ::=`	**Description**	**Section**
`assembly`	Assembly accessibility.	15.1.1
`| famandassem`	Family and Assembly accessibility.	15.1.1
`| family`	Family accessibility.	15.1.1
`| famorassem`	Family or Assembly accessibility.	15.1.1
`| initonly`	Marks a constant field.	15.1.2
`| literal`	Specifies metadata field. No memory is allocated at runtime for this field.	15.1.2
`| marshal(``<nativeType>`**`)`**	Marshaling information.	15.1.3
`| notserialized`	Field is not serialized with other fields of the type.	15.1.2
`| private`	Private accessibility.	15.1.1

`| compilercontrolled`	Compiler controlled accessibility.	15.1.1
`| public`	Public accessibility.	15.1.1
`| rtspecialname`	Special treatment by runtime.	15.1.4
`| specialname`	Special name for other tools.	15.1.4
`| static`	Static field.	15.1.2

15.1.1 Accessibility Information

The accessibility attributes are **assembly**, **famandassem**, **family**, **famorassem**, **private**, **compilercontrolled** and **public**. These attributes are mutually exclusive.

Accessibility attributes are described in Section 8.2.

15.1.2 Field Contract Attributes

Field contract attributes are **initonly**, **literal**, **static** and **notserialized**. These attributes may be combined. Only static fields may be literal. The default is an instance field that may 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, a static field is shared, within an application domain, 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 may only be mutated inside a constructor. If the field is a static field, then it may be mutated only inside the type initializer of the type in which it was declared. If it is an instance field, then it may be mutated only in one of the instance constructors of the type in which it was defined. It may not be mutated in any other method or in any other constructor, including constructors of subclasses.

Note: 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 and is not verifiable.

literal specifies that this field represents a constant value; they 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 (see Section 15.2).

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.

15.1.3 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 clause 14.5.4

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.

15.1.4 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).*

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 Common Language Specification (CLS, see Partition I).

15.2 Field Init Metadata

The <fieldInit> metadata can be optionally added to a field declaration. The use of this feature may 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 (see clause 15.1.2) or parameters with default values. See Section 21.9

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. May be 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 ( <int8> )`	8 bit integer with the integer specified in parentheses.
`| [ unsigned ] int16 ( <int16> )`	16 bit integer with the integer specified in parentheses.
`| [ unsigned ] int32 ( <int32> )`	32 bit integer with the integer specified in parentheses.
`| [ unsigned ] int64 ( <int64> )`	64 bit integer with the integer specified in parentheses.
`| <QSTRING>`	String. <QSTRING> is stored as Unicode
`| nullref`	Null object reference

Example (informative):

The following example 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.

15.3 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> ::=	**Section**
.data <datadecl>	
| …	6.6

Data may also be declared as part of a type:

<classMember> ::=	**Section**
.data <datadecl>	
| …	9.2

Yet another alternative is to declare data inside a method:

<methodBodyItem> ::=	**Section**
.data <datadecl>	
| …	14.4.1

15.3.1 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 may 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 not 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, may be replicated
| **float64** [**(** <float64> **)**] [**[** <int32> **]**]	64-bit floating point number, may be replicated
| **int8** [**(** <int8> **)**] [**[** <int32> **]**]	8-bit integer, may be replicated

| **int16** [(<int16>)] [[<int32>]]	16-bit integer, may be replicated
| **int32** [(<int32>)] [[<int32>]]	32-bit integer, may be replicated
| **int64** [(<int64>)] [[<int32>]]	64-bit integer, may be replicated

Example (informative):

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]
```

15.3.2 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 may be subject to platform-specific rules, typically related to section access permissions within the PE File format itself.

Example (informative):

The following accesses the data declared in the example of clause 15.3.1. 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
```

15.4 Initialization of Non-Literal Static Data

This section and its subsections contain only informative text.

Many languages that support static data (i.e. variables that have a lifetime that is the entire program) provide for a means to initialize that data before the program begins running. There are three common mechanisms for doing this, and each is supported in the CLI.

15.4.1 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 (see Section 15.3). 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.

15.5 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 may be possible to supply arbitrary code to run as the PE file is loaded, but this mechanism is platform-specific and may not be available in all conforming implementations of the CLI.

15.5.1 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 clause 9.5.3.1. 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 the data out 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 clause 9.5.3.

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 may 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 may 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

16 Defining Properties

A Property is declared by the using the **.property** directive. Properties may only be declared inside of types (ie global Properties are not supported)

<classMember> ::=
.property <propHead> **{** <propMember>* **}**

See Section 21.31 and Section 21.32 for how Property information is stored in metadata.

<propHead> ::=
[**specialname**][**rtspecialname**] <callConv> <type> <id> **(** <parameters> **)**

The property directive specifies a calling convention (see Section 14.3), type, name, and parameter in parentheses. **specialname** marks the Property as *special* to other tools, while **rtspecialname** marks Property as *special* to the CLI. The signature for the property (i.e., the <propHead> 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).*

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 may contain any number of methods in its body. The following table shows these and provides short descriptions of each item:

<propMember> ::=	**Description**	**Section**
| **.custom** <customDecl>	Custom attribute.	20
| **.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* may 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* may 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 (see Chapter 20) or source line declarations may be specified.

Example (informative):

This example shows the declaration of the property used in the example in Part 5.

```
.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 get_Count()
         .set instance void set_Count(int32)
         .other instance void reset_Count()
  }
}
```

17 Defining Events

Events are declared inside types with the **.event** directive; there are no global events.

`<classMember> ::=`	**Section**
`.event <eventHead> { <eventMember>* }`	
`| …`	9

See Section 21.13 and Section 21.11

`<eventHead> ::=`
`[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 may 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).*

`<eventMember> ::=`	**Description**	**Section**
`.addon <callConv> <type> [<typeSpec> ::] <methodName> ( <parameters> )`	**Add** method for event.	
`| .custom <customDecl>`	Custom attribute.	20
`| .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 may 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 may also have custom attributes (Chapter 20) associated with them and they may declare source line information.

Example (informative):

```
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.MulticastDelegate {

  .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 add_TimeUp(class TimeUpEventHandler 'handler')

  .removeon instance void remove_TimeUp(class TimeUpEventHandler 'handler')

  .fire instance void 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 multicast 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
```

18 Exception Handling

In the CLI, a method may define a range of CIL instructions that are said to be *protected.* This is called the 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 CIL 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.

```
<sehBlock> ::=
  <tryBlock> <sehClause> [<sehClause>*]
```

The next few sections 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> may 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

18.1 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, 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 uses a scope block (see clause 14.4.4) after the **.try** directive – the instructions within that scope are the ones to be protected.

18.2 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 syntax, 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. Alternatively, the handler block is just a scope block.

18.3 Catch

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 (informative):
.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
```

18.4 Filter

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 (informative):
.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:
    ...
}
```

18.5 Finally

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 (informative):
.try {
  ...                         // protected instructions
  leave exitTry         // shall use leave
} finally {
  ...                         // finally handler
  endfinally
}
exitTry:                // back to normal
```

18.6 Fault Handler

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 (informative):
.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
}
```

19 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 Section 21.11. By contrast, *ilasm* requires the conversion information to be represented in its input.

<securityDecl> ::=
.permissionset <secAction> **=** **(** <bytes> **)**
| **.permission** <secAction> <typeReference> **(** <nameValPairs> **)**

In **.permission**, <typeReference> specifies the permission class and <nameValPairs> specifies the settings. See Section 21.11

In **.permissionset** the bytes specify the serialized 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 subclass.
| **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 may be required.

```
<nameValPairs> ::= <nameValPair> [, <nameValPair>]*
```

```
<nameValPair> ::= <SQSTRING> = <SQSTRING>
```

20 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 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 parent 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 this directive is the method declaration for a type constructor, optionally followed by a <bytes> in parentheses:

```
<customDecl> ::=
  <ctor> [ = ( <bytes> ) ]
```

The <ctor> item represents a method declaration (see Section 14.4), specific for the case where the method's name is **.ctor**.

For example:

```
.custom instance void myAttribute::.ctor(bool, bool) = ( 01 00 00 01 00 00 )
```

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 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 these 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 Section 22.3.

Example (informative):

The following example shows a class that is marked with the `System.SerializableAttribute` and a method that is marked with the `System.Runtime.Remoting.OneWayAttribute`. The keyword serializable corresponds to the `System.SerializableAttribute`.

```
.class public MyClass {
  .custom void [mscorlib]System.SerializableAttribute::.ctor ()
  .method public static void main() {
         .custom void [mscorlib]System.Runtime.Remoting.OneWayAttribute::.ctor ()
         ret
  }
}
```

20.1 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.

20.2 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 regular 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 (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 regular 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 regular Custom Attribute 'blob', but reflection uses this Custom Attribute when called to invoke the default member (property) for a type.

The following subsections 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.

20.2.1 Pseudo Custom Attributes

The following table lists the CLI Pseudo Custom Attributes. They are defined in either the `System` or the `System.Reflection` 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 Section 22.4.
`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

Not all of these Pseudo Custom Attributes are specified in this standard, but all of them 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.

The Pseudo Custom Attributes above 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.PackingSiz* ,and *ClassLayout.ClassSize* fields for that type.

20.2.2 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.

20.2.3 Custom Attributes for Security

The following Custom Attributes affect the security checks performed upon method invocations at runtime. They are defined in the `System.Security` namespace.

Attribute	**Description**
`DynamicSecurityMethodAttribute`	Indicates to the CLI that the method requires space to be allocated for a security object
`SuppressUnmanagedCodeSecurityAttribute`	Indicates the target method, implemented as unmanaged code, should skip per-call checks

The following Custom Attributes are defined in the `System.Security.Permissions`. namespace. 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
`SiteIdentityPermissionAttribute`	Custom attribute class for declarative security with SiteIdentityPermission.
`SocketPermissionAttribute`	Custom attribute class for declarative security with SocketPermission.
`StrongNameIdentityPermissionAttribute`	Custom attribute class for declarative security with StrongNameIdentityPermission.
`WebPermissionAttribute`	Custom attribute class for declarative security with WebPermission.

Note that any other security-related Custom Attributes (ie, any Custom Attributes that derive from `System.Security.Permissions.SecurityAttribute`) included into an assembly, may 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 in fact holds true for any Custom Attributes that cannot be resolved; security-related Custom Attributes are just one particular case)

20.2.4 Custom Attributes for TLS

A Custom Attribute that denotes a TLS (thread-local storage) field is defined in the `System`. namespace

Attribute	Description
`ThreadStaticAttribute`	Provides for type member fields that are relative for the thread.

20.2.5 Custom Attributes, Various

The following Custom Attributes control various aspects of the CLI:

Attribute	Description
`ConditionalAttribute`	Used to mark methods as callable, based on some compile-time condition. If the condition is false, the method will not be called
`DecimalConstantAttribute`	Stores the value of a decimal constant in metadata
`DefaultMemberAttribute`	Defines the member of a type that is the default member used by reflection’s InvokeMember.
`FlagsAttribute`	Custom attribute indicating an enumeration should be treated as a bitfield; that is, a set of flags
`IndexerNameAttribute`	Indicates the name by which an indexer will be known in programming languages that do not support indexers directly
`ParamArrayAttribute`	Indicates that the method will allow a variable number of arguments in its invocation

21 Metadata Logical Format: Tables

This section 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 Chapter 23)

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 (eg 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 (eg the *Flags* column in the *Field* table), but there are a few that are 4 bytes (eg the *Flags* column in the *TypeDef* table)

Each index is either 2 bytes wide, or 4 bytes wide. The index points into another (or the same) 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 huge tables, containing 64K rows or more, an indexer of that table will be 4 bytes wide.

Note that indexes begin at 1, meaning the first row in any given metadata table. An index value of zero denotes that it does not index a row at all (it behaves like a null reference).

The columns that index a metadata table are of two sorts:

- Simple – that column indexes one, and only one, table. e.g., 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 – that column indexes any of several tables. e.g., the *Extends* column in the *TypeDef* table can index into the *TypeDef* table, or into the *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 (Chapter 23).

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 size 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) is 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 (eg "Assembly") and numbered (eg 0x20). The number for each table is listed immediately with its title in the following sections.

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 specification 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
Constant	Parent
FieldMarshal	Parent
MethodSemantics	Association
ClassLayout	Parent
FieldLayout	Field
ImplMap	MemberForwarded
FieldRVA	Field
NestedClass	NestedClass
MethodImpl	Class
CustomAttribute	Parent
DeclSecurity	Parent

Furthermore, the InterfaceImpl table is subsorted using the Interface column as a secondary key.

Finally, the TypeDef table has a special ordering constraint: the definition of an enclosing class must precede the definition of all classes it encloses.

21.1 Metadata Validation Rules

This contains informative text only

The sections 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 - checking the CIL instruction stream for type safety, building method tables, CIL-to-native-code compilation, data marshalling, etc 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 often the metadata is held only in memory, but never actually saved to a file on disk).

The rules address intra-module validation only. So, validator software, for example, that checks conformance with this spec, need not resolve references or walk type hierarchies defined in other modules. However, it should be clear that even if two modules, A and B, analyzed separately, contain only valid metadata, they may still be in error when viewed together (e.g., a call from Module A, to a method defined in module B, might specify a callsite signature that does not match the signatures defined for that method in B)

All checks are categorized as ERROR, WARNING or CLS.

- An ERROR reports something that might cause a CLI to crash or hang, it might run but produce wrong answers; or it might be entirely benign. There may exist conforming implementations of the CLI that will not accept metadata that violates an ERROR rule, and therefore such metadata is invalid and is not portable.
- A WARNING 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 will support metadata that violate only WARNING rules; hence such metadata is both valid and portable.

- A CLS reports lack of compliance with common language specification (see Partition I). Such metadata is both valid and portable, but there may exist programming languages that cannot process it, even though all conforming implementations of the CLI support the constructs.

Validation rules fall into a few broad categories, as follows:

- **Number of Rows** A few tables are allowed only one row (e.g. Module table). Most have no such restriction.
- **Unique Rows** No table may 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:
  - o Every index into the String, Blob or Userstring heaps shall point *into* that heap, neither before its start (offset 0), nor after its end
  - o Every index into the Guid heap shall lie between 1 and the maximum element number in this module, inclusive
  - o Every index (row number) into another metadata table shall lie between 0 and that table's row count + 1 (for some tables, the index may point just past the end of any target table, meaning it indexes nothing)
- **Valid Bitmasks** Columns which are bitmasks shall only have valid permutations of bits set
- **Valid RVAs** There are restrictions upon fields and methods that are assigned RVAs (Relative Virtual Addresses; these 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 say "nothing" - for example, some rules state that a particular table is allowed zero or more rows - so 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 size of names stored in metadata, and subsequently processed by a CLI implementation.

21.2 Assembly : 0x20

The *Assembly* table has the following columns:

- *HashAlgId* (a 4 byte constant of type AssemblyHashAlgorithm, clause 22.1.1)
- MajorVersion, MinorVersion, BuildNumber, RevisionNumber (2 byte constants)
- *Flags* (a 4 byte bitmask of type AssemblyFlags, clause 22.1.2)
- *PublicKey* (index into **Blob** heap)
- *Name* (index into String heap)
- *Culture* (index into String heap)

The *Assembly* table is defined using the **.assembly** directive (see Section 6.2); its columns are obtained from the respective **.hash algorithm**, **.ver**, **.publickey**, and **.culture** (see clause 6.2.1 For an example see Section 6.2.

This contains informative text only

1. The *Assembly* table may contain zero or one row [ERROR]
2. *HashAlgId* should be one of the specified values [ERROR]
3. *Flags* may have only those values set that are specified [ERROR]

4. *PublicKey* may be null or non-null
5. *Name* shall index a non-null string in the String heap [ERROR]
6. The string indexed by *Name* can be of unlimited length
7. *Culture* may be null or non-null
8. If *Culture* is non-null, it shall index a single string from the list specified (see clause 0) [ERROR]

Note: *Name* is a simple name (e.g., “Foo” - no drive letter, no path, no file extension); on POSIX-compliant systems *Name* contains no colon, no forward-slash, no backslash, no period.

End informative text

21.3 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. If present in a PE file, it should be treated as if all its fields were zero. It should be ignored by the CLI.

21.4 AssemblyProcessor : 0x21

The *AssemblyProcessor* table has the following column:

- *Processor* (a 4 byte constant)

This record should not be emitted into any PE file. If present in a PE file, it should be treated as if its field were zero. It should be ignored by the CLI.

21.5 AssemblyRef : 0x23

The *AssemblyRef* table has the following columns:

- MajorVersion, MinorVersion, BuildNumber, RevisionNumber (2 byte constants)
- *Flags* (a 4 byte bitmask of type AssemblyFlags, clause 22.1.2)
- *PublicKeyOrToken* (index into Blob heap – the public key or token that identifies the author of this Assembly)
- *Name* (index into String heap)
- *Culture* (index into String heap)
- *HashValue* (index into Blob heap)

The table is defined by the **.assembly extern** directive (see Section 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 Section 6.3.

This contains informative text only

1. MajorVersion, MinorVersion, BuildNumber, RevisionNumber can each have any value
2. Flags may have only one possible bit set – the **PublicKey** bit (see clause 22.1.2). All other bits shall be zero. [ERROR]
3. *PublicKeyOrToken* my 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-null string, in the String heap (there is no limit to its length). [ERROR]
6. *Culture* may be null or non-null. If non-null, it shall index a single string from the list specified (see clause 0) [ERROR]
7. *HashValue* may be null or non-null
8. If non-null, then *HashValue* shall index a non-empty 'blob' in the Blob heap [ERROR]
9. The *AssemblyRef* table shall contain no duplicates, where duplicate rows have the same *MajorVersion*, *MinorVersion*, *BuildNumber*, *RevisionNumber*, *PublicKeyOrToken*, *Name* and *Culture* [WARNING]

Note: *Name* is a simple name (e.g., "Foo" - no drive letter, no path, no file extension); on POSIX-compliant systems *Name* contains no colon, no forward-slash, no backslash, no period.End informative text

End informative text

21.6 AssemblyRefOS : 0x25

The *AssemblyRefOS* table has the following columns:

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

These records should not be emitted into any PE file. If present in a PE file, they should be treated as-if their fields were zero. They should be ignored by the CLI.

21.7 AssemblyRefProcessor : 0x24

The *AssemblyRefProcessor* table has the following columns:

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

These records should not be emitted into any PE file. If present in a PE file, they should be treated as-if their fields were zero. They should be ignored by the CLI.

21.8 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 make a managed value type be laid out in exactly the same way as an unmanaged C struct – with this condition true, the managed value type can be handed to unmanaged code, which accesses the fields exactly as if that block of memory had been laid out by unmanaged code.*

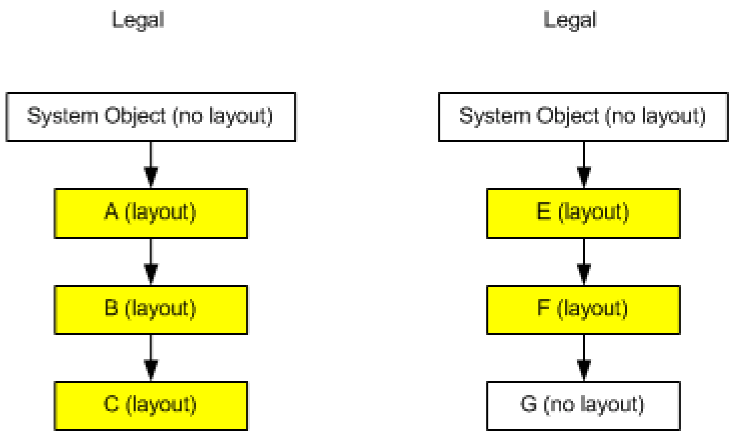
The information held in the *ClassLayout* table depends upon the *Flags* value for {*AutoLayout, SequentialLayout, ExplicitLayout*} in the owner class or value type.

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 parents, up to the one that descends immediately from System.Object, or from System.ValueType.

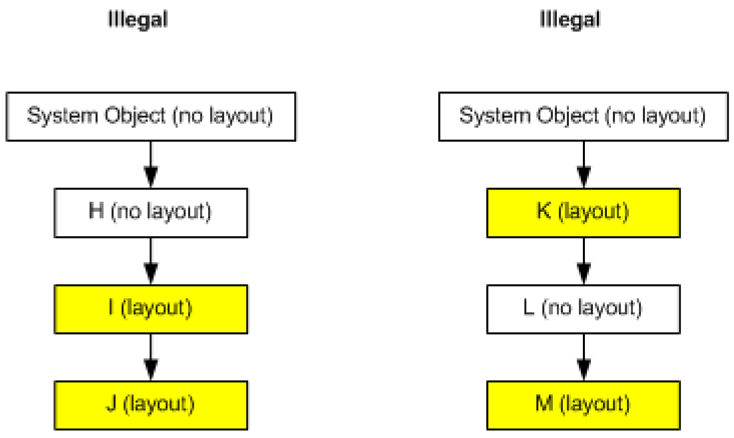
This contains informative text only

Layout cannot begin part way down the chain. But it *is* legal 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 legal.

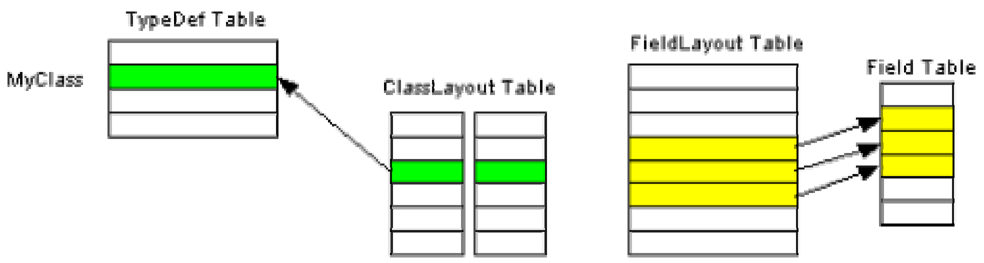


Similarly with Classes E, F and G. G has no layout. This too is legal. The following picture shows two *illegal* setups:



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 – the *ClassLayout* and *FieldLayout* tables, as shown in this diagram:



This example shows how row 3 of the *ClassLayout* table points to row 2 in the *TypeDef* table (the definition for a Class, called "MyClass"). Rows 4 through 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, this design overall saves space

End informative text

The *ClassLayout* table has the following columns:

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

The rows of the *ClassLayout* table are defined by placing **.pack** and **.size** directives on the body of a parent type declaration (see Section 9.2). For an example see Section 9.7.

This contains informative text only

1. A *ClassLayout* table may contain zero or more or rows
2. *Parent* shall index a valid row in the *TypeDef* table, corresponding to a Class or ValueType (not to an Interface) [ERROR]
3. The Class or ValueType indexed by *Parent* shall **not** be *AutoLayout* - i.e., it shall be one of *SequentialLayout* or *ExplicitLayout*. (See clause 22.1.14). Put another way, *AutoLayout* types shall not own any rows in the *ClassLayout* table. [ERROR]
4. If *Parent* indexes a *SequentialLayout* type, then: [ERROR]
   - o *PackingSize* shall be one of {0, 1, 2, 4, 8, 16, 32, 64, 128} (0 means use the default pack size for the platform that the application is running on)
   - o if *ClassSize* is non-zero, then it shall be greater than or equal to the calculated size of the class, based upon its field sizes and *PackingSize* (compilers request padding at the end of a class by providing a value for *ClassSize* that is larger than its calculated size) [ERROR]
   - o a *ClassSize* of zero does not mean the class has zero size. It means, no size was specified at definition time. Instead, 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
   - o if *Parent* indexes a ValueType, then *ClassSize* shall be less than 1 MByte (0x100000 bytes)
5. Note that *ExplicitLayout* types *might* result in verifiable types, so long as that layout does not create *union* types.
6. If *Parent* indexes an *ExplicitLayout* type, then [ERROR]
   - o if *ClassSize* is non-zero, then it shall be greater than or equal to the calculated size of the class, based upon the rows it owns in the *FieldLayout* table (compilers create padding at the end of a class by providing a value for *ClassSize* that is larger than its calculated size)

- a *ClassSize* of zero does not mean the class has zero size. It means, no size was specified at definition time. Instead, the actual size is calculated from the field types, their specified offsets, and any beyond-end **alignment** packing performed by the target platform
- if *Parent* indexes a ValueType, then *ClassSize* shall be less than 1 MByte (0x100000 bytes)
- *PackingSize* shall be 0 (because it makes no sense to provide explicit offsets for each field, as well as a packing size)

7. Layout along the length of an inheritance chain shall follow the rules specified above (starts at 'highest' Type, with no 'holes', etc) [ERROR]

End informative text

21.9 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 Clause 22.1.15 . The encoding of *Type* for the **nullref** value for <fieldInit> in *ilasm* (see Section 15.2) 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* (index into the *Param* or *Field* or *Property* table; more precisely, a *HasConstant* coded index)
- *Value* (index into Blob heap)

Note that *Constant* information does not directly influence runtime behavior, although it is visible via Reflection (and hence may 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 Section 15.2.

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, ELEMENT_TYPE_STRING`; or `ELEMENT_TYPE_CLASS` with a *Value* of zero (See clause 22.1.15) [ERROR]
2. *Type* shall not be any of: `ELEMENT_TYPE_I1, ELEMENT_TYPE_U2, ELEMENT_TYPE_U4, ELEMENT_TYPE_U8` (See clause 22.1.15) [CLS]
3. *Parent* shall index a valid row in the *Field* or *Property* or *Param* table [ERROR]
4. There shall be no duplicate rows, based upon *Parent* [ERROR]
5. *Constant.Type* must match exactly the declared type of the *Param*, *Field* or *Property* identified by *Parent* (in the case where the parent is an enum, it must match exactly the underlying type of that enum) [CLS]

End informative text

21.10 CustomAttribute : 0x0C

The *CustomAttribute* table has the following columns:

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

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 (see Chapter 20)

This contains informative text only

All binary values are stored in little-endian format (except *PackedLen* items - used only as counts for the number of bytes to follow in a UTF8 string)

1. It is legal for there to be no *CustomAttribute* present at all - that is, for the *CustomAttribute.Value* field 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 *MethodRef* table. That row shall be a constructor method (for the class of which this information forms an instance) [ERROR]
4. *Value* may 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'(see Section 22.3):
   - o *Prolog* shall be 0x0001 [ERROR]
   - o There shall be as many occurrences of *FixedArg* as are declared in the Constructor method [ERROR]
   - o *NumNamed* may be zero or more
   - o There shall be exactly *NumNamed* occurrences of *NamedArg* [ERROR]
   - o Each *NamedArg* shall be accessible by the caller [ERROR]
   - o If *NumNamed* = 0 then there shall be no further items in the *CustomAttrib* [ERROR]
7. The following rules apply to the structure of *FixedArg* (see Section 22.3):
   - o 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]
   - o If this item is for a vector, then:
     - o *NumElem* shall be 1 or more [ERROR]
     - o This shall be followed by *NumElem* occurrences of *Elem* [ERROR]
8. The following rules apply to the structure of *Elem* (see Section 22.3):
   - o If this is a simple type or an enum (see Section 22.3 for how this is defined), then *Elem* consists simply of its value [ERROR]
   - o If this is a string, or a Type, then *Elem* consists of a *SerString* – *PackedLen* count of bytes, followed by the UTF8 characters [ERROR]
   - o 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, ELEMENT_TYPE_R8`), followed by its value. [ERROR]

9. The following rules apply to the structure of *NamedArg* (see Section 22.3):

   o The single byte `FIELD` (0x53) or `PROPERTY` (0x54) [ERROR]

   o The type of the field or property -- 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`)

   o 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]

   o A *FixedArg* (see above) [ERROR]

End informative text

21.11 DeclSecurity : 0x0E

Security attributes, which derive from `System.Security.Permissions.SecurityAttribute` (see Partition IV), can be attached to a *TypeDef*, a *Method* or to 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`, may 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 Section 21.10). 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.

The *DeclSecurity* table has the following columns:

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

Action is a 2-byte representation of Security Actions, see `System.Security.SecurityAction` in Partition IV. The values 0 through 0xFF are reserved for future standards use. Values 0x20 through 0x7F and 0x100 through 0x07FF are for uses where the action may be ignored if it is not understood or supported. Values 0x80 through 0xFF and 0x0800 through 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**	**Legal Scope**
Assert	1	Without further checks satisfy Demand for specified permission	Method, Type
Demand	1	Check 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 specified permission	Method, Type
InheritanceDemand	1	Specified permission shall be granted in order to inherit from class or override virtual method.	Method, Type
LinkDemand	1	Check immediate caller has been granted specified permission, throw `SecurityException` (see Partition IV) on failure	Method, Type
PermitOnly	1	Without further checks refuse Demand for all permissions other than those specified.	Method, Type
RequestMinimum		Specify minimum permissions required to run	Assembly
RequestOptional		Specify optional permissions to grant	Assembly
RequestRefuse		Specify permissions not to be granted	Assembly
NonCasDemand	2	Check that current assembly has been granted specified permission, throw `SecurityException` (see Partition IV) otherwise	Method, Type
NonCasLinkDemand	2	Check that immediate caller has been granted specified permission, throw `SecurityException` (see Partition IV) otherwise	Method, Type
PrejitGrant		Reserved for implementation-specific use	Assembly

Note 1: Specified attribute shall derive from `System.Security.Permissions.CodeAccess-SecurityAttribute`

Note 2: Attribute shall derive from `System.Security.Permissions.SecurityAttribute`, but shall not derive from `System.Security.Permissions.CodeAccessSecurityAttribute`

Parent is a Meta Data token that identifies the *Method*, *Type* or *Assembly* on which security custom attributes serialized in *PermissionSet* was defined.

PermissionSet is a 'blob' that contains the XML serialization of a permission set. The permission set contains the permissions that were requested with an *Action* on a specific *Method*, *Type* or *Assembly* (see *Parent*).

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 (see Section 6.6) or parent type or method (see Section 9.2).

This contains informative text only

1. *Action* may 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 should be set [ERROR]

5. If *Parent* indexes a MethodDef, then its *MethodDef.Flags.HasSecurity* bit should be set [ERROR]
6. *PermissionSet* should index a 'blob' in the Blob heap [ERROR]
7. The format of the 'blob' indexed by *PermissionSet* should represent a valid, serialized CLI object graph. The serialized form of all standardized permissions is specified in Partition IV. [ERROR]

End informative text

21.12 EventMap : 0x12

The *EventMap* table has the following columns:

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

Note that *EventMap* info does not directly influence runtime behavior; what counts is the info stored for each method that the event comprises.

This contains informative text only

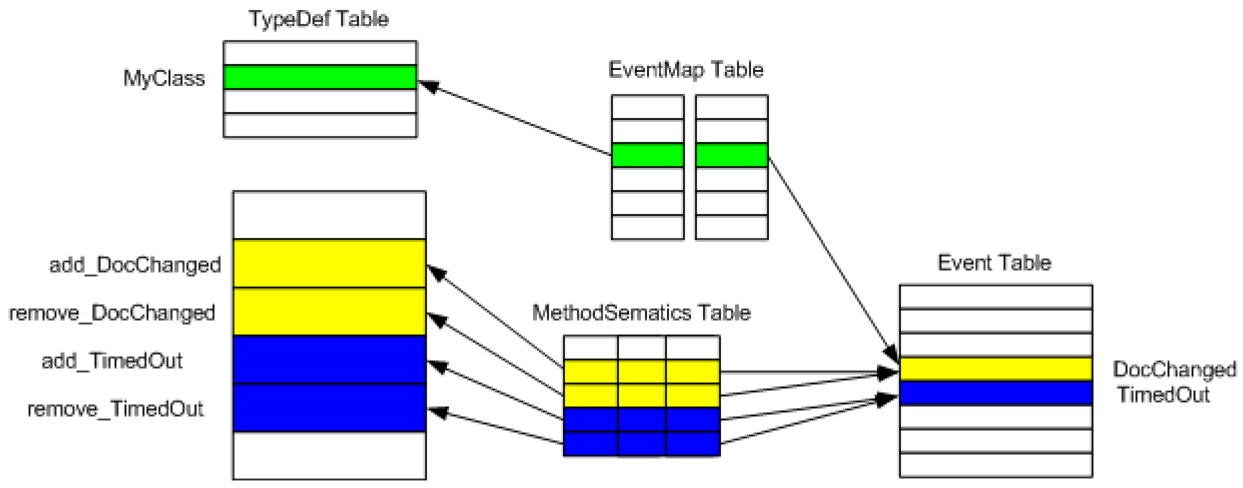
1. *EventMap* table may contain zero or more rows
2. There shall be no duplicate rows, based upon *Parent* (a given class has only one 'pointer' to the start of its event list) [ERROR]
3. There shall be no duplicate rows, based upon *EventList* (different classes cannot share rows in the *Event* table) [ERROR]

End informative text

21.13 Event : 0x14

Events are treated within metadata much like Properties – a way to associate a collection of methods defined on given class. There are two required methods – *add_* and *remove_*, plus optional *raise_* and *others*. 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 that used for Properties) – see the below:



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* (eg *DocChanged* and *TimedOut* in the example here) and *EventType.* In addition, the *MethodSemantics* table has a column to record whether the method it points at is an *add_*, a *remove_*, a *raise_*, or *other.*

The *Event* table has the following columns:

- *EventFlags* (a 2 byte bitmask of type *EventAttribute*, clause 22.1.4)
- *Name* (index into String heap)
- *EventType* (index into *TypeDef, TypeRef* or *TypeSpec* tables; more precisely, a *TypeDefOrRef* coded index) [this corresponds to the Type of the Event; it is *not* the Type that owns this event]

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 (see Chapter 17).

This contains informative text only

1. The *Event* table may contain zero or more rows
2. Each row shall have one, and only one, owner row in the *EventMap* table [ERROR]
3. *EventFlags* may have only those values set that are specified (all combinations valid) [ERROR]
4. *Name* shall index a non-null string in the String heap [ERROR]
5. The *Name* string shall be a valid CLS identifier [CLS]
6. *EventType* may 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 *TypeDef* , *TypeRef,* or *TypeSpec* table that it indexes shall be a Class (not an Interface; not 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

21.14 ExportedType : 0x27

The *ExportedType* table holds a row for each type, defined within *other* modules of this Assembly, that is exported out of this Assembly. In essence, it stores *TypeDef* row numbers of all types that are marked public in *other* modules that this Assembly comprises.

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. (Regular token values are indexes into table in the *current* module)

The full name of the type need not be stored directly. Instead, it may be split into two parts at any included "." (although typically this 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*, clause 22.1.14)
- *TypeDefId* (4 byte index into a *TypeDef* table of another module in this Assembly). This field is used as a hint only. If the entry in the target *TypeDef* table matches the *TypeName* and *TypeNamespace* entries in this table, resolution has succeeded. But if there is a mismatch, the CLI shall fall back to a search of the target *TypeDef* table
- *TypeName* (index into the String heap)
- *TypeNamespace* (index into the String heap)
- *Implementation.* This can be an index (more precisely, an *Implementation coded index*) into one of 2 tables, as follows:
    - *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

The rows in the *ExportedType* table are the result of the **.class extern** directive (see Section 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 may 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* may have only those values set that are specified [ERROR]

4. If *Implementation* indexes the *File* table, then *Flags.VisibilityMask* shall be `public` (see clause 22.1.14) [ERROR]
5. If *Implementation* indexes the *ExportedType* table, then *Flags.VisibilityMask* shall be `NestedPublic` (see see clause 22.1.14) [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 (see see clause 22.1.14) [WARNING]
7. *TypeName* shall index a non-null string in the String heap [ERROR]
8. *TypeNamespace* may be null, or non-null
9. If *TypeNamespace* is non-null, then it shall index a non-null 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: [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.tdVisibilityMask* 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]

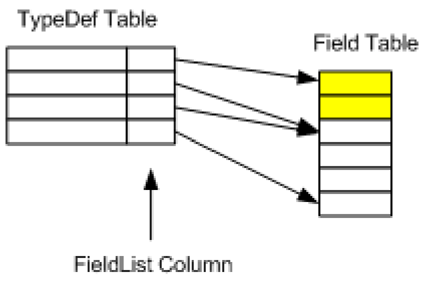
End informative text

21.15 Field : 0x04

The *Field* table has the following columns:

- *Flags* (a 2 byte bitmask of type *FieldAttributes*, clause 22.1.5)
- *Name* (index into String heap)
- *Signature* (index into 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 through 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 through 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 through 5 in the *Field* table. Type 4 owns row 6 in the *Field* table. (The *next* pointers in the diagram show the next free row in each table) So, in the *Field* table, rows 1 and 2 belong to Type 1 (global variables); rows 3 through 5 belong to Type 3; row 6 belongs to Type 4.

Each row in the *Field* table results from a toplevel **.field** directive (see Section 5.10), or a **.field** directive inside a Type (see Section 9.2). For an example see Section 13.5.

This contains informative text only

1. *Field* table may 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* may have only those set that are specified [ERROR]
5. The *FieldAccessMask* subfield of Flags shall contain precisely one of `CompilerControlled`, `Private`, `FamANDAssem`, `Assembly`, `Family`, `FamORAssem`, or `Public` (see clause 22.1.5) [ERROR]
6. *Flags* may set 0 or 1 of `Literal` or `InitOnly` (not both) (see clause 22.1.5) [ERROR]
7. If *Flags.Literal* = 1 then *Flags.Static* shall be 1 too (see clause 22.1.5) [ERROR]
8. If *Flags.RTSpecialName* = 1, then *Flags.SpecialName* shall also be 1 (see clause 22.1.5) [ERROR]
9. If *Flags.HasFieldMarshal* = 1, then this row shall 'own' exactly one row in the *FieldMarshal* table (see clause 22.1.5) [ERROR]
10. If *Flags.HasDefault* = 1, then this row shall 'own' exactly one row in the *Constant* table (see clause 22.1.5) [ERROR]
11. If *Flags.HasFieldRVA* = 1, then this row shall 'own' exactly one row in the *Field's RVA* table (see clause 22.1.5) [ERROR]
12. *Name* shall index a non-null 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 (see clause 22.1.5), 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:
    - o *Flags.Static* shall be 1 [ERROR]
    - o *Flags.MemberAccessMask* subfield shall be one of `Public`, `CompilerControlled`, or `Private` (see clause 22.1.5) [ERROR]

  - o 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, and *Name* string = "value__" then:
    a. `RTSpecialName` shall be 1 [ERROR]
    b. owner row in *TypeDef* table shall derive directly from `System.Enum` [ERROR]
    c. the owner row in *TypeDef* table shall have no other instance fields [CLS]
    d. its *Signature* shall be one of (see clause 22.1.15): [CLS]
       - `ELEMENT_TYPE_U1`
       - `ELEMENT_TYPE_I2`
       - `ELEMENT_TYPE_I4`
       - `ELEMENT_TYPE_I8`
20. its *Signature* shall be an integral type.

End informative text

21.16 FieldLayout : 0x10

The *FieldLayout* table has the following columns:

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

Note that each Field in any Type is defined by its Signature. When a Type instance (ie, an object) is laid out by the CLI, each Field is one of three kinds:

- Scalar – for any member of built-in, 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 `CLASS, STRING, OBJECT, ARRAY, SZARRAY`
- Pointer – for `PTR, FNPTR`
- ValueType – for `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

(This lists above use an abbreviation – each all-caps name should be prefixed by `ELEMENT_TYPE_` so, for example, `STRING` is actually `ELEMENT_TYPE_STRING`. See clause 22.1.15)

Note that metadata specifying explicit structure layout may be valid for use on one platform but not 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 (see Section 9.7).

This contains informative text only

1. A *FieldLayout* table may contain zero or more or rows

2. The Type whose Fields are described by each row of the *FieldLayout* table shall have *Flags.ExplicitLayout* (see clause 22.1.14) set [ERROR]
3. *Offset* shall be zero or more (cannot be negative) [ERROR]
4. *Field* shall index a valid row in the *Field* table [ERROR]
5. The row in the *Field* table indexed by *Field* shall be non-static (ie its *Flags.Static* shall be 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. No Field of kind *ObjectRef* may overlap any other Field no matter what its kind, wholly or partially [ERROR]
9. Among the rows owned by a given Type it is perfectly legal for several rows to have the same value of *Offset*, so long as they are *not* of type ObjectRef (used to define C *union*s, for example) [ERROR]
10. If *ClassSize* in the owner *ClassLayout* row is non-zero, then no Field may extend beyond that *ClassSize* (ie, the Field *Offset* value plus the Field's calculated size shall not exceed *ClassSize*) (note that it is legal, and common, for *ClassSize* to be supplied as *larger* than the calculated object size - the CLI pads the object with trailing bytes up to the *ClassSize* value) [ERROR]
11. 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

21.17 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) should 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* (index into *Field* or *Param* table; more precisely, a *HasFieldMarshal* coded index)
- *NativeType* (index into the Blob heap)

For the detailed format of the 'blob', see Section 22.4

A row in the *FieldMarshal* table is created if the **.field** directive for the parent field has specified a **.marshall** attribute (see Section 15.1).

This contains informative text only

1. A *FieldMarshal* table may 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 can 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' (see Section 22.4):
    a. *NativeIntrinsic* shall be exactly one of the constant values in its production [ERROR]
    b. If *NativeIntrinsic* has the value `BYVALSTR`, then *Parent* shall point to a row in the Field table, not the Param table [ERROR]
    c. If `FIXEDARRAY`, then *Parent* shall point to a row in the Field table, not the Param table [ERROR]
    d. If `FIXEDARRAY`, then *NumElem* shall be 1 or more [ERROR]
    e. If `FIXEDARRAY`, then *ArrayElemType* shall be exactly one of the constant values in its production [ERROR]
    f. If `ARRAY`, then ArrayElemType shall be exactly one of the constant values in its production [ERROR]
    g. If `ARRAY`, then *ParamNum* may be zero
    h. If `ARRAY`, then *ParamNum* cannot be < 0 [ERROR]
    i. If `ARRAY`, and *ParamNum* > 0, then *Parent* shall point to a row in the Param table, not in the Field table [ERROR]
    j. 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]
    k. If `ARRAY`, then *ElemMult* shall be >= 1 [ERROR]
    l. If `ARRAY` and *ElemMult* <> 1 issue a warning, because it is probably a mistake [WARNING]
    m. If `ARRAY` and *ParamNum* == 0, then *NumElem* shall be >= 1 [ERROR]
    n. If `ARRAY` and *ParamNum* != 0 and NumElem != 0 then issue a warning, because it is probably a mistake [WARNING]

End informative text

21.18 FieldRVA : 0x1D

The *FieldRVA* table has the following columns:

- *RVA* (a 4 byte constant)
- *Field* (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 (see Chapter 15). The RVA column is the relative virtual address of the data in the PE file (see Section 15.3).

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 table in the *Field* table [ERROR]
4. Any field with an RVA shall be a ValueType (not a Class, and not 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 may 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

21.19 File : 0x26

The *File* table has the following columns:

- *Flags* (a 4 byte bitmask of type FileAttributes, clause 22.1.6)
- *Name* (index into String heap)
- *HashValue* (index into Blob heap)

The rows of the *File* table result from **.file** directives in an Assembly (see clause 6.2.3)

This contains informative text only

1. *Flags* may have only those values set that are specified (all combinations valid) [ERROR]
2. *Name* shall index a non-null string in the String heap. It shall be in the format <filename>.<extension> (eg "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 - 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

21.20 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 *Method* 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.

The *ImplMap* table has the following columns:

- *MappingFlags* (a 2 byte bitmask of type *PInvokeAttributes*, clause 22.1.7)
- *MemberForwarded* (index into the *Field* or *MethodDef* table; more precisely, a *MemberForwarded* coded index. However, it only ever indexes the *MethodDef* table, since *Field* export is not supported.

- *ImportName* (index into the String heap)
- *ImportScope* (index into the *ModuleRef* table)

A row is entered in the *ImplMap* table for each parent Method (see Section 14.5) that is defined with a **.pinvokeimpl** interoperation attribute specifying the *MappingFlags*, *ImportName* and *ImportScope*. For an example see Section 14.5.

This contains informative text only

1. *ImplMap* may contain zero or more rows
2. *MappingFlags* may 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* (see clause 22.1.7) 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 set, the default is `CharSetNotSpec`) [ERROR]
5. *ImportName* shall index a non-null 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

21.21 InterfaceImpl : 0x09

The *InterfaceImpl* table has the following columns:

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

The *InterfaceImpl* table records which interfaces a Type implements. Conceptually, each row in the *InterfaceImpl* table says that *Class* implements *Interface.*

This contains informative text only

1. The *InterfaceImpl* table may 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* (a class can implement many interfaces)
6. There can be many rows with the same value for *Interface* (many classes can implement the same interface)

End informative text

21.22 ManifestResource : 0x28

The *ManifestResource* table has the following columns:

- *Offset* (a 4 byte constant)
- *Flags* (a 4 byte bitmask of type *ManifestResourceAttributes*, clause 22.1.8)
- *Name* (index into the String heap)
- *Implementation* (index into *File* table, or *AssemblyRef* table, or null; more precisely, an *Implementation* 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 (see clause 6.2.2).

This contains informative text only

1. The *ManifestResource* table may contain zero or more rows
2. *Offset* shall be a valid offset into the target file, starting from the Resource entry in the COR header [ERROR]
3. *Flags* may have only those values set that are specified [ERROR]
4. The *VisibilityMask* (see clause 22.1.8) subfield of *Flags* shall be one of `Public` or `Private` [ERROR]
5. *Name* shall index a non-null string in the String heap [ERROR]
6. *Implementation* may 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

21.23 MemberRef : 0x0A

The *MemberRef* table combines two sorts of references – to Fields and to Methods of a class, known as 'MethodRef' and 'FieldRef', respectively. The *MemberRef* table has the following columns:

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

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 callsite)

This contains informative text only

1. *Class* shall be one of ... [ERROR]

   a. a *TypeRef* token, if the class that defines the member is defined in another module. (**Note:** it is unusual, but legal, to use a *TypeRef* token when the member is defined in this same module - 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 varargs 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 constructed type

2. *Class* shall not be null (this would indicate an unresolved reference to a global function or variable) [ERROR]
3. *Name* shall index a non-null 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)

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 [CLS]
9. There shall be no duplicate rows, where *Name* fields are compared using CLS conflicting-identifier-rules. (note, particular, that the return type, and whether parameters are marked `ELEMENT_TYPE_BYREF` (see clause 22.1.15) are ignored in the CLS. For example, `int foo()` and `double foo()` result in duplicate rows by CLS rules. Similarly, `void bar(int i)` and `void bar(int& i)` also result in duplicate rows by CLS rules) [CLS]
10. If *Class* and *Name* resolve to a field, then that field shall not have a value of `CompilerControlled` (see clause 22.1.5) in its *Flags.FieldAccessMask* subfield [ERROR]
11. If *Class* and *Name* resolve to a method, then that method shall not have a value of he `CompilerControlled` in its *Flags.MemberAccessMask* (see clause 22.1.9) subfield [ERROR]

End informative text

21.24 MethodDef : 0x06

The *MethodDef* table has the following columns:

- *RVA* (a 4 byte constant)
- *ImplFlags* (a 2 byte bitmask of type *MethodImplAttributes*, clause 22.1.9)
- *Flags* (a 2 byte bitmask of type *MethodAttribute*, clause 22.1.9)
- *Name* (index into String heap)
- *Signature* (index into Blob heap)
- *ParamList* (index into *Param* table). It marks the first of a contiguous run of Parameters owned by this method. The run continues to the smaller of:

- o the last row of the *Param* table
- o the next run of Parameters, found by inspecting the *ParamList* of the next row in the *MethodDef* table

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 (see Chapter 14). 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 (see Chapter 24.4)

This contains informative text only

1. The *MethodDef* table may contain zero or more rows
2. Each row shall have one, and only one, owner row in the *TypeDef* table [ERROR]
3. *ImplFlags* may have only those values set that are specified [ERROR]
4. *Flags* may have only those values set that are specified [ERROR]
5. The *MemberAccessMask* (see clause 22.1.9) subfield of *Flags* shall contain precisely one of `CompilerControlled`, `Private`, `FamANDAssem`, `Assem`, `Family`, `FamORAssem`, or `Public` [ERROR]
6. The following combined bit settings in *Flags* are illegal [ERROR]
   a. `Static | Final`
   b. `Static | Virtual`
   c. `Static | NewSlot`
   d. `Final  | Abstract`
   e. `Abstract | PinvokeImpl`
   f. `CompilerControlled | Virtual`
   g. `CompilerControlled | Final`
   h. `CompilerControlled | SpecialName`
   i. `CompilerControlled | RTSpecialName`
7. An abstract method shall be virtual. So: if *Flags.Abstract* = 1 then *Flags.Virtual* shall also be 1 [ERROR]
8. If *Flags.RTSpecialName* = 1 then *Flags.SpecialName* shall also be 1 [ERROR]
9. If *Flags.HasSecurity* = 1, then at least one of the following conditions shall be true: [ERROR]
   - o this Method owns at least row in the *DeclSecurity* table
   - o this Method has a custom attribute called *SuppressUnmanagedCodeSecurityAttribute*
10. If this Method owns one (or more) rows in the *DeclSecurity* table then *Flags.HasSecurity* shall be 1 [ERROR]
11. If this Method has a custom attribute called *SuppressUnmanagedCodeSecurityAttribute* then *Flags.HasSecurity* shall be 1 [ERROR]
12. A Method may have a custom attribute called *DynamicSecurityMethodAttribute* - but this has no effect whatsoever upon the value of its *Flags.HasSecurity*
13. *Name* shall index a non-null string in the String heap [ERROR]
14. Interfaces cannot have instance constructors. So, if this Method is owned by an Interface, then its *Name* cannot be **.ctor** [ERROR]

15. Interfaces can only own virtual methods (not static or instance methods). So, if this Method is owned by an Interface, *Flags.Static* shall be clear [ERROR]
16. The *Name* string shall be a valid CLS identifier (unless *Flags.RTSpecialName* is set - for example, **.cctor** is legal) [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. (In C++, the method is called *global* and can be referenced only within its compiland, from its point of declaration forwards.) 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. i.e. *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 however that if *Flags.CompilerControlled* = 1, then this row is completely 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, then this row is completely excluded from duplicate checking as explained above) [CLS]
23. If `Final` or `NewSlot` 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]
    o RVA != 0
    o *Flags.PInvokeImpl* = 1
    o *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` (x5)
28. *Signature* shall have the calling conventions `DEFAULT` (0x0). [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 byte shall be 0 [ERROR]

31. If `EXPLICITTHIS` (0x40) in the signature is set, then `HASTHIS` (0x20) shall also be set (note in passing: 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]

    o *Flags.Abstract* = 1, or

    o *ImplFlags.Runtime* = 1, or

    o *Flags.PinvokeImpl* = 1, or

34. If *RVA* != 0, then: [ERROR]

    a. *Flags.Abstract* shall be 0, and

    b. *ImplFlags.CodeTypeMask* shall be 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]

    o *RVA* = 0 *and* the method owns a row in the *ImplMap* table, **OR**

36. If *Flags.RTSpecialName* = 1 then *Name* shall be one of: [ERROR]

    a. **.ctor** (object constructor method)

    b. **.cctor** (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** (object constructor method) then:

    a. return type in *Signature* shall be ELEMENT_TYPE_VOID (see clause 22.1.15) [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 0 or more **.ctors** for any given 'owner'

39. If *Name* = **.cctor** (class constructor method) then:

    a. return type in *Signature* shall be `ELEMENT_TYPE_VOID` (see clause 22.1.15) [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 be 0 or 1 methods named **.cctor** (never 2 or more) [ERROR]

End informative text

21.25 MethodImpl : 0x19

*MethodImpl*s let a compiler override the default inheritance rules provided by the CLI. Their original use was to allow a class "C", that inherited method "Foo" from interfaces I *and* J, to provide implementations for *both* methods (rather than have only *one* slot for "Foo" in its vtable). But *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::Foo, *MethodBody* specifies the method which provides the implementation for I::Foo (either a method body within "C", or a method body implemented by a superclass of "C")

The *MethodImpl* table has the following columns:

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

ilasm uses the **.override** directive to specify the rows of the *MethodImpl* table (see clause 9.3.2).

This contains informative text only

1. The *MethodImpl* table may 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 *Method* or *MethodRef* 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 superclass 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. 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

21.26 MethodSemantics : 0x18

The *MethodSemantics* table has the following columns:

- *Semantics* (a 2-byte bitmask of type *MethodSemanticsAttributes*, clause 22.1.11)
- *Method* (index into the *MethodDef* table)

- *Association* (index into the *Event* or *Property* table; more precisely, a *HasSemantics* coded index)

The rows of the *MethodSemantics* table are filled by **.property** (see Chapter 16) and **.event** directives (see Chapter 17). See clause 21.13 for more information.

This contains informative text only

1. *MethodSemantics* table may contain zero or more rows
2. *Semantics* may 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 *Method* table by *Method* may 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` (see clause 22.1.15) 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

21.27 Module : 0x00

The *Module* table has the following columns:

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

The *Mvid* column shall index a unique GUID in the GUID heap (see Section 23.2.5) that identifies this instance of the module. The *Mvid* may 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 may 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.

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.*

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 (see Section 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-null 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

21.28 ModuleRef : 0x1A

The *ModuleRef* table has the following column:

- *Name* (index into String heap)

The rows in the *ModuleRef* table result from **.module extern** directives in the Assembly (see Section 6.5).

This contains informative text only

1. *Name* shall index a non-null 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

21.29 NestedClass : 0x29

The *NestedClass* table has the following columns:

- *NestedClass* (index into the *TypeDef* table)
- *EnclosingClass* (index into the *TypeDef* table)

The *NestedClass* table records which Type definitions are nested within which other Type definition. In a typical high-level language, including *ilasm*, the nested class 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 may 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 legal to have two or more rows with the same value for *EnclosingClass* but different values for *NestedClass*

End informative text

21.30 Param : 0x08

The *Param* table has the following columns:

- *Flags* (a 2 byte bitmask of type ParamAttributes, clause 22.1.12)
- *Sequence* (a 2 byte constant)
- *Name* (index into 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 (see Section 14.4), or from a **.param** attribute attached to a method (see clause 14.4.1).

This contains informative text only

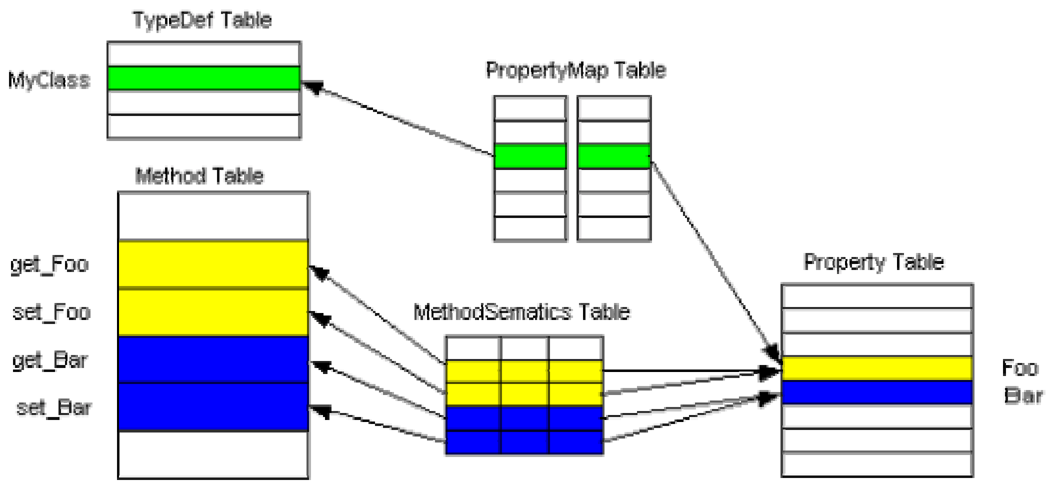
1. *Param* table may contain zero or more rows
2. Each row shall have one, and only one, owner row in the *MethodDef* table [ERROR]
3. *Flags* may have only those values set that are specified (all combinations valid) [ERROR]
4. *Sequence* shall have a value >= 0 and <= number of parameters in owner 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* may be null or non-null
11. If *Name* is non-null, then it shall index a non-null string in the String heap [WARNING]

End informative text

21.31 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 – see 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 *Method* 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 (see clause 22.2.1) 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.

The *Property* (0x17) table has the following columns:

- *Flags* (a 2 byte bitmask of type PropertyAttributes, clause 22.1.13)
- *Name* (index into String heap)

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

This contains informative text only

1. *Property* table may 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* may have only those values set that are specified (all combinations valid) [ERROR]
4. *Name* shall index a non-null 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

21.32 PropertyMap : 0x15

The *PropertyMap* table has the following columns:

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

The *PropertyMap* and *Property* tables result from putting the **.property** directive on a class (see Chapter 16).

This contains informative text only

1. *PropertyMap* table may 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

21.33 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, that 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 in *ilasm* generates a row in the *StandAloneSig* table.

The*StandAloneSig* table has the following column:

- *Signature* (index into the Blob heap)

Example (informative):

```
// 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
  }
}
```

This contains informative text only

1. The *StandAloneSig* table may 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

21.34 TypeDef : 0x02

The *TypeDef* table has the following columns:

- *Flags* (a 4 byte bitmask of type *TypeAttributes*, clause 22.1.14)
- *Name* (index into String heap)
- *Namespace* (index into String heap)
- *Extends* (index into *TypeDef*, *TypeRef* or *TypeSpec* table; more precisely, a *TypeDefOrRef* coded index)
- *FieldList* (index into *Field* table; it marks the first of a continguous run of Fields owned by this Type). The run continues to the smaller of:
    - o the last row of the *Field* table
    - o the next run of Fields, found by inspecting the *FieldList* of the next row in this *TypeDef* table
- *MethodList* (index into *MethodDef* table; it marks the first of a continguous run of Methods owned by this Type). The run continues to the smaller of:
    - o the last row of the *MethodDef* table
    - o the next run of Methods, found by inspecting the *MethodList* of the next row in this *TypeDef* table

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 quite distinct 'inheritance' 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 Interfaces

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 may represent one-to-many relations – that is, one Class might well implement two or more Interfaces.

Example (informative, written in C#):

```
interface IA {void m1(int i);        }
interface IB {void m2(int i, int j); }
class C : IA, IB {
  int f1, f2;
  public void m1(int i)        {f1 = i;        }
  public void m2(int i, int j) {f1 = i; f2 = j;}
}
// In metadata, Interface IA extends nothing; Interface IB
// extends nothing; class C extends System.Object and implements
// Interfaces IA and IB
```

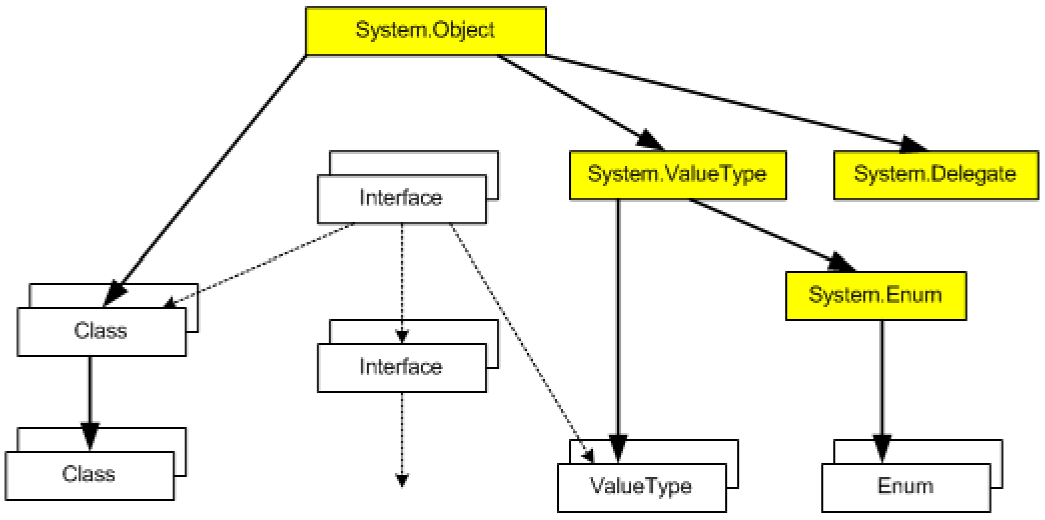
An Interface can also 'inherit' from one or more other Interfaces – metadata stores those links via the *InterfaceImpl* table (the nomenclature is a little inappropriate here – there is no "implementation" involved – perhaps a clearer name might have been *Interface* table, or *InterfaceInherit* table)

```
Example (informative, written in C#):
interface IA          {void m1(int i);        }
interface IB          {void m2(int i, int j); }
interface IC : IA, IB {void m3(int i, int j, int k);}
class C : IC {
  int f1, f2, f3;
  public void m1(int i)               {f1 = i;                }
  public void m2(int i, int j)        {f1 = i; f2 = j;        }
  public void m3(int i, int j, int k) {f1 = i; f2 = j; f3 = k;}
}
// In metadata, Interface IA extends nothing; Interface IB extends
// nothing; Interface IC "inherits" Interfaces IA and IB (defined via
// the InterfaceImpl table); Class C extends System.Object and
// implements Interface IC (see InterfaceImpl table)
```

There are also a few specialized types. One is the user-defined Enum – which shall derive directly from System.Enum (via the *Extends* field)

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*}.

The roots of the inheritance hierarchies look like this:



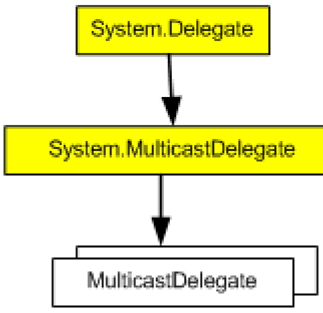
There is one system-defined root – System.Object. All Classes and ValueTypes shall derive, ultimately, from System.Object; Classes can derive from other Classes (through a single, non-looping chain) to any depth required. This *Extends* inheritance chain is shown with heavy arrows.

(See below for details of the `System.Delegate` Class)

Interfaces do not inherit from one another, however, they specify zero or more other 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 (ie 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.

The hierarchy below System.Delegate is as follows:



User-defined delegates derive directly from `System.MulticastDelegate`. Delegates cannot be derived to a depth of more than one.

For the directives to declare types see Chapter 9.

This contains informative text only

1. *TypeDef* table may contain one or more rows. There is always one row (row zero) that represents the pseudo class that acts as *parent* for functions and variables defined at module scope.
2. Flags:
   a. *Flags* may 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 legal 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-null string in the String heap [ERROR]
4. The *Name* string shall be a valid CLS identifier [CLS]
5. *Namespace* may be null or non-null
6. If non-null, then *Namespace* shall index a non-null string in the String heap [ERROR]
7. If non-null, *Namespace*'s string shall be a valid CLS Identifier [CLS]
8. Every Class (with the sole exception of `System.Object`) 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 sole exception of `System.Object`, for any Class, *Extends* shall index a valid row in the *TypeDef* or *TypeRef* 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 (ie its derived Classes), since this would introduce loops in the hierarchy tree [ERROR]
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 may '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 may '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 legal 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 may be inherited from its base class, from the interfaces it implements, or defined by itself. The implementations may 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 *Namespace+Name* (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 *Namespace+Name+OwnerRowInNestedClassTable* [ERROR]
31. There shall be no duplicate rows, where *Namespace+Name* 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 *Namespace+Name+OwnerRowInNestedClassTable* and where *Namespace+Name* fields are compared using CLS conflicting-identifier-rules [CLS]

33. If *Extends* = System.Enum (ie, 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 at least one static, literal field. If more than one, they shall all be of the same type. Any such static literal fields shall be of the type of the Enum [CLS]
    h. shall be at least one instance field, of integral type [ERROR]
    i. shall be exactly one instance field [CLS]
    j. the *Name* string of the instance field shall be "value__"; it shall marked `RTSpecialName`*;* its type shall be one of (see clause 22.1.15): [CLS]
        - `ELEMENT_TYPE_U1`
        - `ELEMENT_TYPE_I2`
        - `ELEMENT_TYPE_I4`
        - `ELEMENT_TYPE_I8`
    k. shall be no other members (ie, apart from any static literals, and the one instance field called "value__") [CLS]
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

21.35 TypeRef : 0x01

The *TypeRef* table has the following columns:

- ResolutionScope (index into *Module*, *ModuleRef*, *AssemblyRef* or *TypeRef* tables, or null; more precisely, a *ResolutionScope* coded index)
- Name (index into String heap)
- Namespace (index into 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. *Name* shall index a non-null string in the String heap [ERROR]
3. *Namespace* may be null, or non-null
4. If non-null, *Namespace* shall index a non-null string in the String heap [ERROR]
5. The *Name* string shall be a valid CLS identifier [CLS]
6. There shall be no duplicate rows, where a duplicate has the same *ResolutionScope, Name* and *Namespace* [ERROR]
7. There shall be no duplicate rows, where *Name* and *Namespace* fields are compared using CLS conflicting-identifier-rules [CLS]

End informative text

21.36 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 – 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 clause 22.2.14)

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, unbox

This contains informative text only

The *TypeSpec* table may contain zero or more rows

Signature shall index a valid Type specification in the Blob heap [ERROR]

There shall be no duplicate rows, based upon *Signature* [ERROR]

End informative text

22 Metadata Logical Format: Other Structures

22.1 Bitmasks and Flags

This section explains the various flags and bitmasks used in the various metadata tables.

22.1.1 Values for AssemblyHashAlgorithm

Algorithm	Value
`None`	0x0000
`Reserved (MD5)`	0x8003
`SHA1`	0x8004

22.1.2 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 may 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 may 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.

22.1.3 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). In other words, the familiar ASCII characters – a-z and A-Z respectively. However, whilst RFC 1766 recommends the first part is lowercase, the second part 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

22.1.4 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

22.1.5 Flags for Fields [FieldAttributes]

Flag	Value	Description
FieldAccessMask	0x0007	
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 may only be initialized, not written to after init
Literal	0x0040	Value is compile time constant
NotSerialized	0x0080	Field does not have to 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

22.1.6 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

22.1.7 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
CharSetNotSpec	0x0000	
CharSetAnsi	0x0002	
CharSetUnicode	0x0004	
CharSetAuto	0x0006	
SupportsLastError	0x0040	Information about target function. Not relevant for fields
Calling convention		
CallConvMask	0x0700	
CallConvWinapi	0x0100	

CallConvCdecl	0x0200	
CallConvStdcall	0x0300	
CallConvThiscall	0x0400	
CallConvFastcall	0x0500	

22.1.8 Flags for ManifestResource [ManifestResourceAttributes]

Flag	Value	Description
VisibilityMask	0x0007	
Public	0x0001	The Resource is exported from the Assembly
Private	0x0002	The Resource is private to the Assembly

22.1.9 Flags for Methods [MethodAttributes]

Flag	Value	Description
MemberAccessMask	0x0007	
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 may not 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
ReuseSlot	0x0000	Method reuses existing slot in vtable
NewSlot	0x0100	Method always gets a new slot in the vtable
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.

22.1.10 Flags for Methods [MethodImplAttributes]

Flag	Value	Description
CodeTypeMask	0x0003	
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.
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 may ignore
InternalCall	0x1000	Reserved: shall be zero in conforming implementations
Synchronized	0x0020	Method is single threaded through the body
NoInlining	0x0008	Method may not be inlined
MaxMethodImplVal	0xffff	Range check value

22.1.11 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

22.1.12 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

22.1.13 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

22.1.14 Flags for Types [TypeAttributes]

Flag	Value	Description
Visibility attributes		
VisibilityMask	0x00000007	Use this mask to retrieve visibility information
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
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
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	Class is serializable
String formatting Attributes		
`StringFormatMask`	0x00030000	Use this mask to retrieve string information for native interop
`AnsiClass`	0x00000000	LPSTR is interpreted as ANSI
`UnicodeClass`	0x00010000	LPSTR is interpreted as Unicode
`AutoClass`	0x00020000	LPSTR is interpreted automatically
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

22.1.15 Element Types used in Signatures

The following table lists the values for ELEMENT_TYPE constants. These are used extensively in metadata signature *blobs* – see Section 22.2

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> token
`ELEMENT_TYPE_BYREF`	0x10	Followed by <type> token
`ELEMENT_TYPE_VALUETYPE`	0x11	Followed by TypeDef or TypeRef token
`ELEMENT_TYPE_CLASS`	0x12	Followed by TypeDef or TypeRef token
`ELEMENT_TYPE_ARRAY`	0x14	<type> <rank> <boundsCount> <bound1> … <loCount> <lo1> …
`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_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 varargs method signature
`ELEMENT_TYPE_PINNED`	0x45	Denotes a local variable that points at a pinned object

22.2 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 five kinds of Signature, as follows:

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

The value of the leading byte of a Signature 'blob' indicates what kind of Signature it is. This section 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 what would otherwise be multiple diagrams into a single diagram; the accompanying text describes the use of shading.

Note that Signatures are compressed before being stored into the Blob heap (described below) by compressing the integers embedded in the signature. The maximum encodable integer is 29 bits long, 0x1FFFFFFF. The compression algorithm used is as follows (bit 0 is the least significant bit):

- 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 two-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)
- A null string should be represented with the reserved single byte 0xFF, and no following data

Note: The table below shows 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)

Original Value	**Compressed Representation**
0x03	03
0x7F	7F (7 bits set)
0x80	8080
0x2E57	AE57
0x3FFF	BFFF
0x4000	C000 4000
0x1FFF FFFF	DFFF FFFF

Thus, 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 - with the most significant byte at the smallest offset within the file.

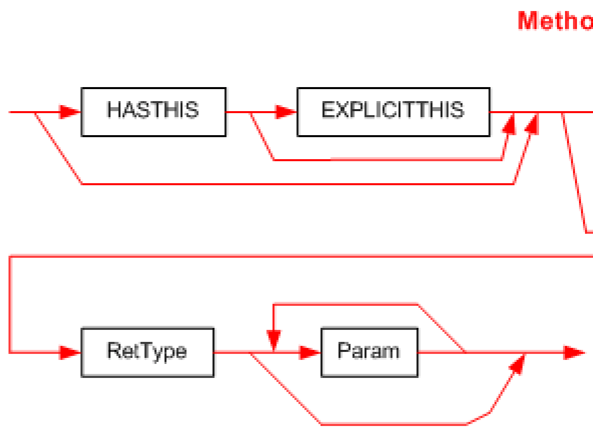
Signatures make extensive use of constant values called `ELEMENT_TYPE_xxx` – see Clause 22.1.15. In particular, signatures include two modifiers called:

`ELEMENT_TYPE_BYREF` – this element is a managed pointer (see Partition I). This modifier can only occur in the definition of Param (clause 22.2.10) or RetType (clause 22.2.11). It shall not occur within the definition of a Field (clause 22.2.4)

`ELEMENT_TYPE_PTR` – this element is an unmanaged pointer (see Partition I). This modifier can occur in the definition of Param (clause 22.2.10) or RetType (clause 22.2.11) or Field (clause 22.2.4)

22.2.1 MethodDefSig

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



This chart uses the following abbreviations:

`HASTHIS = 0x20, used to encode the keyword` **`instance`** `in the calling convention, see` Section 14.3

`EXPLICITTHIS = 0x40, used to encode the keyword` **`explicit`** `in the calling convention, see` Section 14.3

`DEFAULT = 0x0, used to encode the keyword` **`default`** `in the calling convention, see` Section 14.3

`VARARG = for 0x5, used to encode the keyword` **`vararg`** `in the calling convention, see` Section 14.3

The first byte of the Signature holds bits for `HASTHIS`, `EXPLICITTHIS` and calling convention – `DEFAULT` or `VARARG`. These are OR'd together.

ParamCount is an 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)

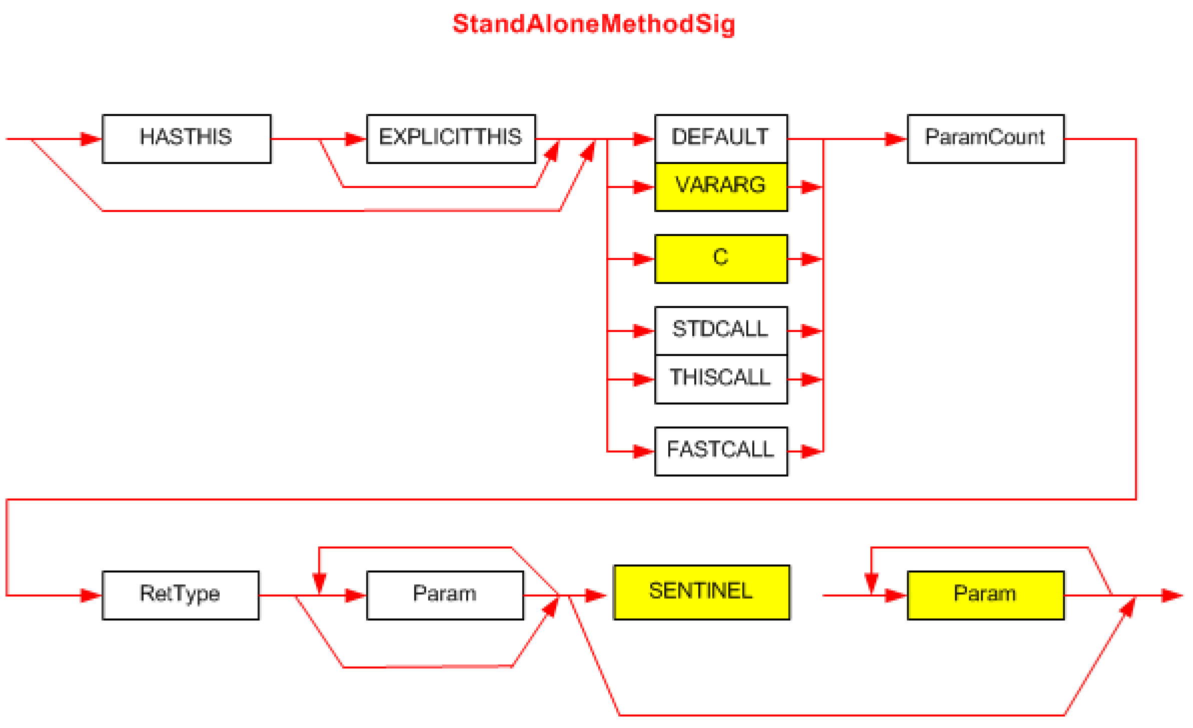
The *RetType* item describes the type of the method's return value (see clause 22.2.11)

The *Param* item describes the type of each of the method's parameters. There shall be *ParamCount* instances of the *Param* item (see clause 22.2.10).

22.2.2 MethodRefSig

A MethodRefSig is indexed by the `MemberRef.Signature` column. This provides the *callsite* Signature for a method. Normally, this callsite 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 uint32s and returns void; then any callsite shall index a signature that takes exactly two uint32s and returns void. In this case, the syntax chart for a MethodRefSig is identical with that for a MethodDefSig – see clause 22.2.1

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



This chart uses the following abbreviations:

`HASTHIS = 0x20, used to encode the keyword` **`instance`** `in the calling convention, see` Section 14.3

`EXPLICITTHIS = 0x40, used to encode the keyword` **`explicit`** `in the calling convention, see` Section 14.3

`DEFAULT = 0x0, used to encode the keyword` **`default`** `in the calling convention, see` Section 14.3

`VARARG = for 0x5, used to encode the keyword` **`vararg`** `in the calling convention, see` Section 14.3

`SENTINEL = 0x41 (see` clause 22.1.15`), used to encode "..." in the parameter list, see` Section 14.3

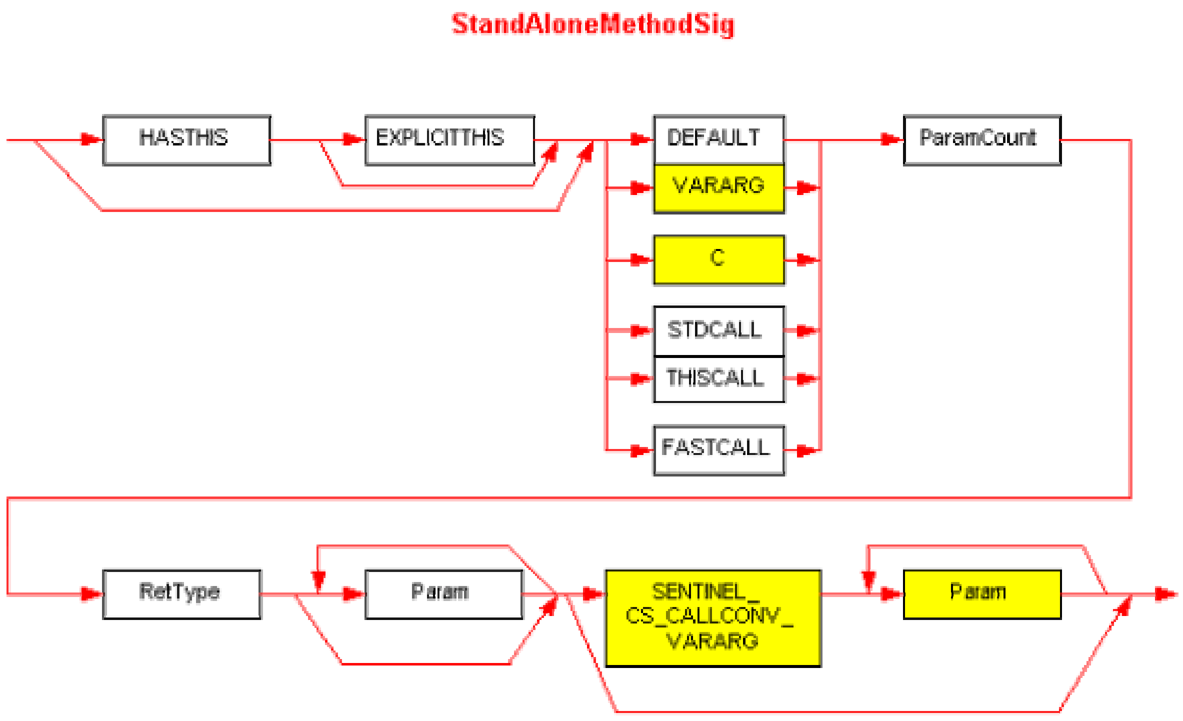
- The first byte of the Signature holds bits for `HASTHIS`, `EXPLICITTHIS` and calling convention – `DEFAULT`, `VARARG`, `C`, `STDCALL`, `THISCALL`, or `FASTCALL`. These are OR'd together.
- *ParamCount* is an integer that holds the number of parameters (0 or more). It can be any number between 0 and 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 (see clause 22.2.11)
- The *Param* item describes the type of each of the method's parameters. There shall be *ParamCount* instances of the *Param* item (see clause 22.2.10).

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 (see clause 22.2.1). 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 callsite 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)

22.2.3 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 callsite signature, but its calling convention may specify an unmanaged target (the *calli* instruction invokes either managed, or unmanaged code). Its syntax chart is:



This chart uses the following abbreviations (see Section 14.3):

`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` (see clause 22.1.15 and Section 14.3)

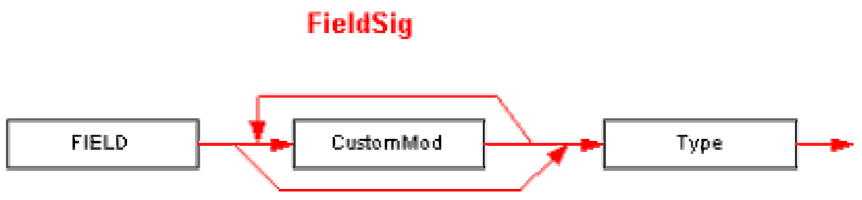
- The first byte of the Signature holds bits for `HASTHIS`, `EXPLICITTHIS` and calling convention – `DEFAULT`, `VARARG`, `C`, `STDCALL`, `THISCALL`, or `FASTCALL`. These are OR'd together.
- *ParamCount* is an integer that holds the number of parameters (0 or more). It can be any number between 0 and 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 (see clause 22.2.11)
- The *Param* item describes the type of each of the method's parameters. There shall be *ParamCount* instances of the *Param* item (see clause 22.2.10).

This is the most complex of the various method signatures. Two separate charts 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 chart.

22.2.4 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 may be a static or instance field in a class, or it may be a global variable. The syntax chart for a FieldSig looks like this:



This chart uses the following abbreviations:

`FIELD for 0x6`

CustomMod is defined in clause 22.2.7. *Type* is defined in clause 22.2.12

22.2.5 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:

how many parameters are supplied to its *getter* method

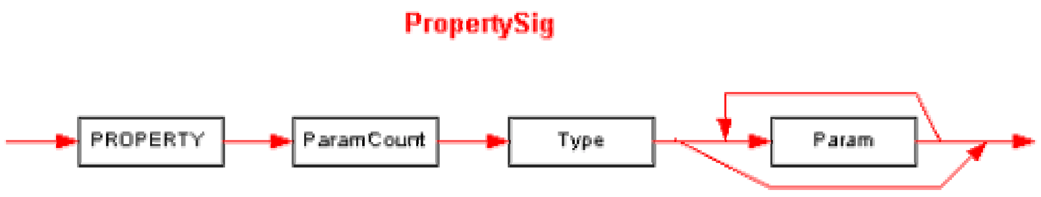
the base type of the Property – the type returned by its *getter* method

type information for each parameter in the *getter* 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 chart for a PropertySig looks like this:



This chart uses the following abbreviations:

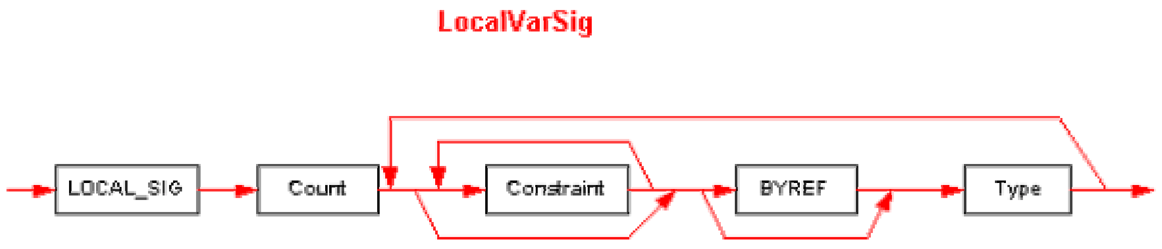
`PROPERTY for 0x8`

Type specifies the type returned by the *Getter* method for this property. *Type* is defined in clause 22.2.12. *Param* is defined in clause 22.2.10.

ParamCount is an integer that holds the number of index parameters in the *getter* methods (0 or more). (See clause 22.2.1) (*ParamCount* counts just the method parameters – it does not include the method's base type of the Property)

22.2.6 LocalVarSig

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



This chart uses the following abbreviations:

`LOCAL_SIG` for 0x7, used for the **.locals** directive, see clause 14.4.1.3

`BYREF` for `ELEMENT_TYPE_BYREF` (see clause 22.1.15)

Constraint is defined in clause 22.2.9.

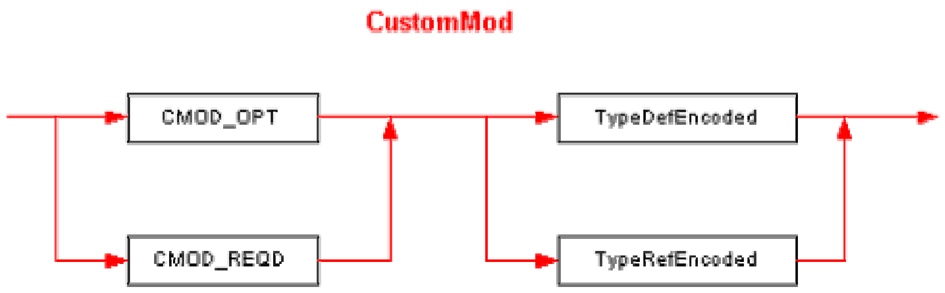
Type is defined in clause 22.2.12

Count is an 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

22.2.7 CustomMod

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



This chart uses the following abbreviations:

`CMOD_OPT`	for	`ELEMENT_TYPE_CMOD_OPT` (see clause 22.1.15)
`CMOD_REQD`	for	`ELEMENT_TYPE_CMOD_REQD` (see clause 22.1.15)

The `CMOD_OPT` or `CMOD_REQD` value is compressed, see Section 22.2.

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 clause 22.2.8 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.

22.2.8 TypeDefOrRefEncoded

These items are compact ways to store a TypeDef or TypeRef token in a Signature (see clause 22.2.12).

Consider a regular TypeRef token, such as 0x01000012. The top byte of 0x01 indicates that this is a TypeRef token (see Partition V 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

8. shift the 3-byte row index (0x000012 in this example) left by 2 bits and OR into the 2-bit encoding from step 1

9. compress the resulting value (see Section 22.2). 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.

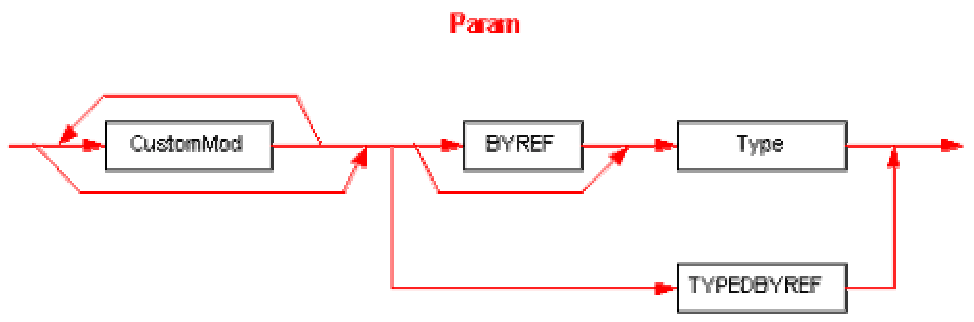
22.2.9 Constraint

The *Constraint* item in Signatures currently has only one possible value – `ELEMENT_TYPE_PINNED` (see clause 22.1.15), 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.

22.2.10 Param

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



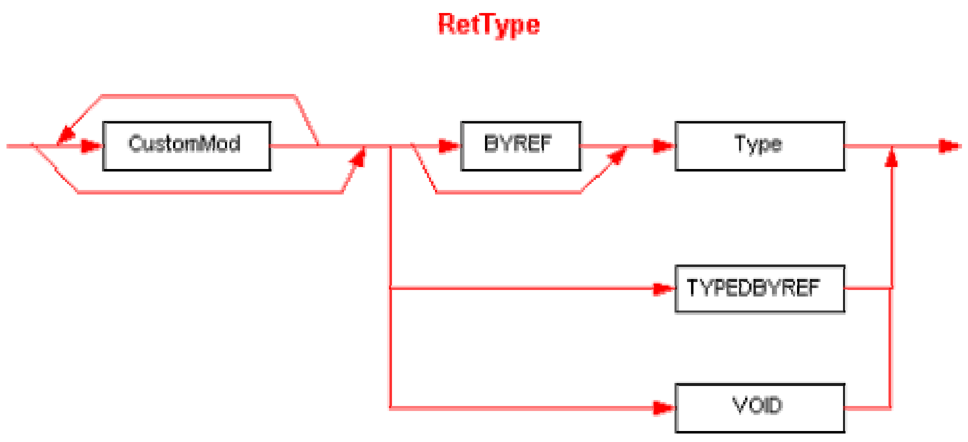
This chart uses the following abbreviations:

BYREF	for	0x10 (See clause 22.1.15)
TYPEDBYREF	for	0x16 (See clause 22.1.15)

CustomMod is defined in clause 22.2.7. *Type* is defined in clause 22.2.12

22.2.11 RetType

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



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

BYREF	for	ELEMENT_TYPE_BYREF (see clause 22.1.15)
TYPEDBYREF	for	ELEMENT_TYPE_TYPEDBYREF (see clause 22.1.15)
VOID	for	ELEMENT_TYPE_VOID (see clause 22.1.15)

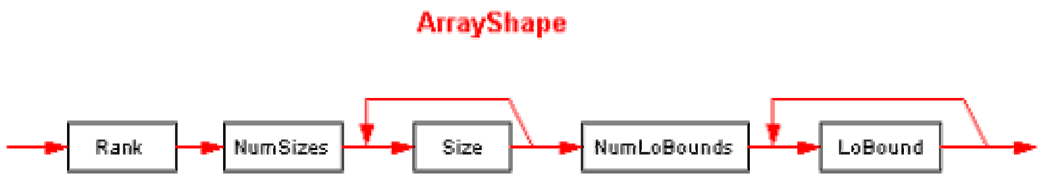
22.2.12 Type

Type is encoded in signatures as follows (I1 is an abbreviation for `ELEMENT_TYPE_I1`, etc., see clause 22.1.15):

```
Type ::=
BOOLEAN | CHAR | I1 | U1 | I2 | U2 | I4 | U4 | I8 | U8 | R4 | R8 | I  | U |
| VALUETYPE TypeDefOrRefEncoded
| CLASS TypeDefOrRefEncoded
| STRING
| OBJECT
| PTR CustomMod* VOID
| PTR CustomMod* Type
| FNPTR MethodDefSig
| FNPTR MethodRefSig
| ARRAY Type ArrayShape  (general array, see clause 22.2.13)
| SZARRAY CustomMod* Type (single dimensional, zero-based array i.e. vector)
```

22.2.13 ArrayShape

An ArrayShape has the following syntax chart:



Rank is an integer (stored in compressed form, see Section 22.2) that specifies the number of dimensions in the array (shall be 1 or more). *NumSizes* is a compressed integer that says how many dimensions have specified sizes (it shall be 0 or more). *Size* is a compressed integer specifying the size of that dimension – the sequence starts at the first dimension, and goes on for a total of *NumSizes* items. Similarly, *NumLoBounds* is a compressed integer that says how many dimensions have specified lower bounds (it shall be 0 or more). And *LoBound* is a compressed integer specifying the lower bound of that dimension – the sequence starts at the first dimension, and goes on for a total of *NumLoBounds* items. 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 may itself be an array

22.2.14 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
```

For compactness, the `ELEMENT_TYPE_` prefixes have been omitted from this list. So, for example, "PTR" is shorthand for `ELEMENT_TYPE_PTR`. (see clause 22.1.15) 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.

22.2.15 Short Form Signatures

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

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

short-form: `ELEMENT_TYPE_STRING`

Only the short form is valid. 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`)

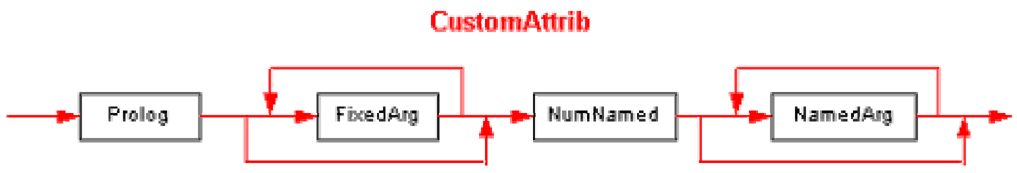
Long Form		**Short Form**
Prefix	**TypeRef to:**	
`CLASS`	`System.String`	`STRING`
`CLASS`	`System.Object`	`OBJECT`
`VALUETYPE`	`System.Void`	`VOID`
`VALUETYPE`	`System.Boolean`	`BOOLEAN`
`VALUETYPE`	`System.Char`	`CHAR`
`VALUETYPE`	`System.Byte`	`U1`
`VALUETYPE`	`System.Sbyte`	`I1`

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`

22.3 Custom Attributes

A Custom Attribute has the following syntax chart:

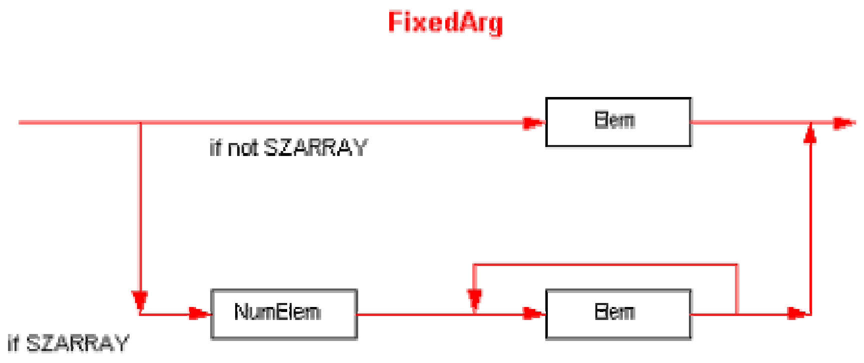


All binary values are stored in little-endian format (except PackedLen items – used only as counts for the number of bytes to follow in a UTF8 string). If there are no fields, parameters, or properties specified the entire attribute may be represented as an empty blob.

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 MethodDef; this info is *not* repeated in the *CustomAttrib* itself. As the syntax chart shows, there can be zero or more *FixedArgs*. (Note that `VARARG` constructor methods are not allowed in the definition of Custom Attributes.)

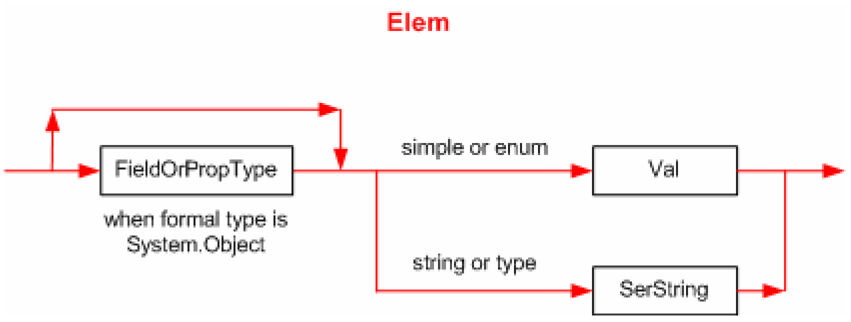
Next is a description of the optional "named" fields and properties. This starts with *NumNamed* – an unsigned int16 giving the number of "named" properties or fields that follow. Note that *NumNamed* shall always be present. If its value is zero, there are no "named" properties or fields to follow (and 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 *NamedArgs.*



The format for each *FixedArg* depends upon whether that argument is single, or an `SZARRAY` – this is shown in the upper and lower paths, respectively, of the syntax chart. 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.

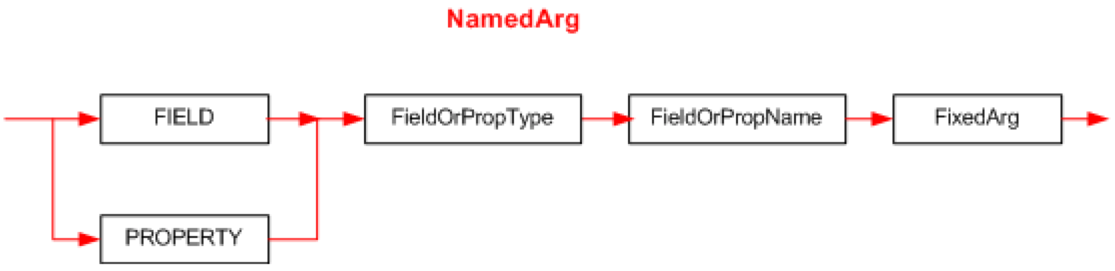


An *Elem* takes one of three forms:

- If the parameter kind is simple (**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*). 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 or type, then the blob contains a *SerString* – a *PackedLen* count of bytes, followed by the UTF8 characters. A type is stored as a string giving the full name of that type. Where the actual argument value is **null**, it shall be encoded as the single byte 0xFF.

For parameters, fields, or properties whose formal (static) type is **System.Object** the blob contains the actual type's **FieldOrPropType** (see below), followed by the representation of the actual parameter. [Note: it is not possible to pass a value of **null** in this case.]

Val is the binary value for a simple type. 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.



A *NamedArg* is simply a *FixedArg* (discussed above) preceded by information to identify which field or property it represents.

`FIELD` is the single byte 0x53.

`PROPERTY` is the single byte 0x54.

If the parameter kind 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` or the constant 0x50 (for an argument of type Type). A single-dimensional, zero-based array is specified as a single byte 0x1D followed by the **FieldOrPropType** of the element type. (See clause 22.1.15)

The *FieldOrPropName* is the name of the field or property, stored as a *SerString* (defined above).

The *SerString* used to encode an argument of type `Type` includes the 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 this assembly, and then the assembly named **mscorlib**.

For example, consider the Type string "Ozzy.OutBack.Kangaroo+Wallaby, MyAssembly" for a class "Wallaby" nested within class "Ozzy.OutBack.Kangaroo", defined in the assembly "MyAssembly".

22.4 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 clause 20.2.1)

Note that a conforming implementation of the CLI need only support marshalling of the types specified earlier – see clause 14.5.4.

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_INT`	0x1f
`NATIVE_TYPE_UINT`	0x20
`NATIVE_TYPE_FUNC`	0x26
`NATIVE_TYPE_ARRAY`	0x2a

The 'blob' has the following format –

```
MarshalSpec ::=
  NativeInstrinsic
| ARRAY ArrayElemType ParamNum ElemMult NumElem
NativeInstrinsic ::=
  BOOLEAN | I1 | U1 | I2 | U2 | I4 | U4 | I8 | U8 | R4 | R8
| CURRENCY | BSTR | LPSTR | LPWSTR | LPTSTR
| INT | UINT | FUNC | LPVOID
```

For compactness, the `NATIVE_TYPE_` prefixes have been omitted in the above lists. So, for example, "`ARRAY`" is shorthand for `NATIVE_TYPE_ARRAY`

NumElem is an integer (compressed as described in Section 22.2) that specifies how many elements are in the array

```
ArrayElemType :==
   NativeInstrinsic | BOOLEAN | I1 | U1 | I2 | U2
|  I4 | U4 | I8 | U8 | R4 | R8 | LPSTR | INT | UINT | FUNC | LPVOID
```

ParamNum is an integer (compressed as described in Section 22.2) specifying the parameter in the method call that provides the number of elements in the array – see below

ElemMult is an integer compressed as described in Section 22.2 (says by what factor to multiply – see below)

Note:

For example, in the method declaration:

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

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

`ARRAY  MAX` 2 1 0

This says the parameter is marshalled to a `NATIVE_TYPE_ARRAY`. There is no additional info about the type of each element (signified by that `NATIVE_TYPE_MAX`). The value of *ParamNum* is 2, which 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 *ElemMult* is 1. The value of *NumElem* is 0. The calculated total size, in bytes, of the array is given by the formula:

```
if ParamNum == 0
  SizeInBytes = NumElem * sizeof (elem)
else
  SizeInBytes = ( @ParamNum * ElemMult + 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` (see clause 22.1.15) of size 4 bytes.

23 Metadata Physical Layout

The physical on-disk representation of metadata is a direct reflection of the logical representation described in Chapter 21 and Chapter 22. That is, data is stored in streams representating the meta data 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.

23.1 Fixed Fields

Complete CLI components (metadata and CIL instructions) are stored in a subset of the current Portable Executable (PE) File Format (see Chapter 24). Because of this heritage, some of the fields in the physical representation of metadata have fixed values. When writing these fields they shall be set to the value indicated, on reading they may be ignored.

23.2 File Headers

23.2.1 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 (see Section 23.1).
`12`	`4`	**Length**	Length of version string in bytes, say *m* (<= 255), rounded up to a multiple of four.
`16`	*m*	**Version**	UTF8-encoded version string of length *m* (see below)
`16+`*m*			Padding to next 4 byte boundary, say *x*.
x	`2`	**Flags**	Reserved, always 0 (see Section 23.1).
x`+2`	`2`	**Streams**	Number of streams, say *n*.
x`+4`		**StreamHeaders**	Array of *n* StreamHdr structures.

The Version string shall be “Standard CLI 2002” 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 ”. Vendors that provide implementation specific extensions shall provide a version string that does *not* begin with “Standard ”.

23.2.2 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 (see clause 23.2.1)
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.

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. (see Chapter 22)

Each kind of stream may occur at most once, that is, a meta-data file may not contain two "#US" streams, or five "#Blob" streams. Streams need not be there if they are empty.

The next sections will describe the structure of each kind of stream in more detail.

23.2.3 #Strings heap

The stream of bytes pointed to by a "#Strings" header is the physical representation of the logical string heap. The physical heap may contain garbage, that is, it may 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 (ie \0).

23.2.4 #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 may 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 0*bs*, then the rest of the 'blob' contains the (*bs*) bytes of actual data.
- If the first two bytes of the 'blob' are 10*bs* and *x*, then the rest of the 'blob' contains the (*bs* << 8 + *x*) bytes of actual data.
- If the first four bytes of the 'blob' are 110*bs*, *x*, *y*, and *z*, then the rest of the 'blob' contains the (*bs* << 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.

23.2.5 #GUID heap

The "#GUID" header points to a sequence of 128-bit GUIDs. There might be unreachable GUIDs stored in the stream.

23.2.6 #~ stream

The “#~” streams contain the actual physical representations of the logical metadata tables (see Chapter 21). A “#~” stream has the following top-level structure:

Offset	**Size**	**Field**	**Description**
`0`	`4`	**Reserved**	Reserved, always 0 (see Section 23.1).
`4`	`1`	**MajorVersion**	Major version of table schemata, always 1 (see Section 23.1).
`5`	`1`	**MinorVersion**	Minor version of table schemata, always 0 (see Section 23.1).
`6`	`1`	**HeapSizes**	Bit vector for heap sizes.
`7`	`1`	**Reserved**	Reserved, always 1 (see Section 23.1).
`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* four 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 how wide indexes into the various heaps are. If bit 0 is set, indexes into the “#String” heap are 4 bytes wide; if bit 1 is set, indexes into the “#GUID” heap are 4 bytes wide; bit 2 is not used; if bit 3 is set, indexes into the “#Blob” heap are 4 bytes wide. Conversely, if the HeapSize bit for a particular heap is not set, indexes into that heap are 2 bytes wide.

Bit position	**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 bits wide 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 Chapter 21. For example when the DeclSecurity table is present in the logical metadata, bit 0x0e should be set in the Valid vector. It is illegal to include non-existent tables in Valid, so all bits above 0x2b 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 Chapter 21. For example, the table with assigned index 0x02 is a TypeDef table, which, according to its specification in Section 21.34, has the following columns: 4 byte-wide flags, index into the String heap, another index into String heap, index into TypeDef or TypeRef table, index into Field table, index into MethodDef table.

The physical representation of a table with schema ($C_0,\ldots,C_{n-1}$) with *n* rows consists of the concatenation of the physical representation of each of its rows. The physical representation of a row with schema ($C_0,\ldots,C_{n-1}$) 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 byte bitmask 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**
`FieldDef`	0
`ParamDef`	1
`Property`	2

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

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

ParamDef	1

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

MemberRefParent: 3 bits to encode tag	**Tag**
Not used	0
TypeRef	1
ModuleRef	2
MethodDef	3
TypeSpec	4

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

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

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

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

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

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

24 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 $s \le r < s+l$. The file position of the item is then given by $p+(r-s)$.

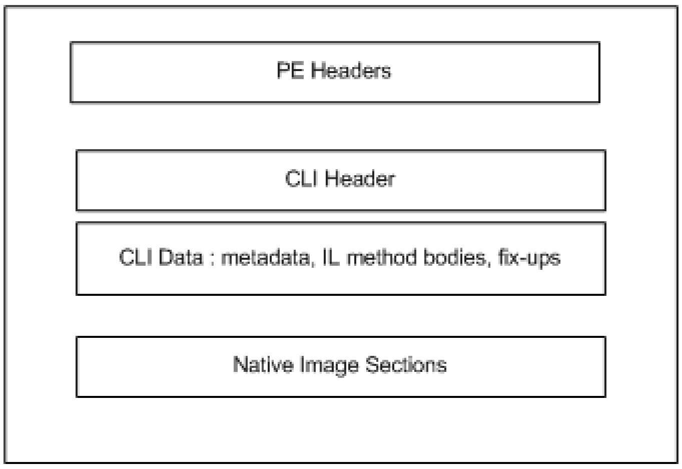
Unless stated otherwise, all binary values are stored in little-endian format.

End informative text

24.1 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 (see clause 24.3.3) 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 (metadata see Chapter 23 and CIL see Chapter 24.4) 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.).

24.2 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.

24.2.1 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

24.2.2 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 (see Section 23.1).
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 (see Section 23.1).
12	4	Number of Symbols	Always 0 (see Section 23.1).
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 Characteristics.

24.2.2.1 Characteristics

A CIL-only DLL sets flag 0x2000 to 1, while an 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, 0x0100 and 0x0020 shall all be set, while all others shall always be zero (see Section 23.1).

24.2.3 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 24.2.3.1.
28	68	NT-specific fields	These include additional fields to support specific features of Windows, see 24.2.3.2.
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).

24.2.3.1 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 (see Section 23.1).
2	1	LMajor	Always 6 (see Section 23.1).
3	1	LMinor	Always 0 (see Section 23.1).
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, always 0x00400000 for exes and 0x10000000 for DLL.
24	4	Base Of Data	RVA of the data section.

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 meta data from the image, and invoke the entry point specified in vthe CLI header.

End informative text

24.2.3.2 PE Header Windows NT-Specific Fields

These fields are Windows NT specific:

Offset	Size	Field	Description
28	4	Image Base	Always 0x400000 (see Section 23.1).
32	4	Section Alignment	Always 0x2000 (see Section 23.1).
36	4	File Alignment	Either 0x200 or 0x1000.

40	2	OS Major	Always 4 (see Section 23.1).
42	2	OS Minor	Always 0 (see Section 23.1).
44	2	User Major	Always 0 (see Section 23.1).
46	2	User Minor	Always 0 (see Section 23.1).
48	2	SubSys Major	Always 4 (see Section 23.1).
50	2	SubSys Minor	Always 0 (see Section 23.1).
52	4	Reserved	Always 0 (see Section 23.1).
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 (see Section 23.1).
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 (see Section 23.1).
72	4	Stack Reserve Size	Always 0x100000 (1Mb) (see Section 23.1).
76	4	Stack Commit Size	Always 0x1000 (4Kb) (see Section 23.1).
80	4	Heap Reserve Size	Always 0x100000 (1Mb) (see Section 23.1).
84	4	Heap Commit Size	Always 0x1000 (4Kb) (see Section 23.1).
88	4	Loader Flags	Always 0 (see Section 23.1)
92	4	Number of Data Directories	Always 0x10 (see Section 23.1).

24.2.3.3 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.

Offset	**Size**	**Field**	**Description**
96	8	Export Table	Always 0 (see Section 23.1).
104	8	Import Table	RVA of Import Table, (see clause 24.3.1).
112	8	Resource Table	Always 0 (see Section 23.1).
120	8	Exception Table	Always 0 (see Section 23.1).
128	8	Certificate Table	Always 0 (see Section 23.1).
136	8	Base Relocation Table	Relocation Table, set to 0 if unused (see clause 24.3.1).
144	8	Debug	Always 0 (see Section 23.1).
152	8	Copyright	Always 0 (see Section 23.1).
160	8	Global Ptr	Always 0 (see Section 23.1).
168	8	TLS Table	Always 0 (see Section 23.1).

176	8	Load Config Table	Always 0 (see Section 23.1).
184	8	Bound Import	Always 0 (see Section 23.1).
192	8	IAT	RVA of Import Address Table, (see clause 24.3.1).
200	8	Delay Import Descriptor	Always 0 (see Section 23.1).
208	8	CLI Header	CLI Header with directories for runtime data, (see clause 24.3.1).
216	8	Reserved	Always 0 (see Section 23.1).

The tables pointed to by the directory entries are stored in on of the PE file's sections; these sections themselves are described by section headers.

24.3 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 when loaded into memory in bytes rounded to Section Alignment. If this value is greater than Size of Raw Data, 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 (see Section 23.1).
32	2	NumberOfRelocations	Number of relocations, set to 0 if unused.
34	2	NumberOfLinenumbers	Always 0 (see Section 23.1).
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.

24.3.1 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 (see Section 23.1).
8	4	ForwarderChain	Always 0 (see Section 23.1).
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.

24.3.2 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 entries each containing a 2-byte Type/Offset pair having 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 below)
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.

24.3.3 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 meta data (see Chapter 23).
16	4	Flags	Flags describing this runtime image. (see clause 24.3.3.1).
20	4	EntryPointToken	Token for the MethodDef or File of the entry point for the image
24	8	Resources	Location of CLI resources. (See Partition V).
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 (see Section 23.1).
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 (see Section 23.1).
64	8	ManagedNativeHeader	Always 0 (see Section 23.1).

24.3.3.1 Runtime Flags

The following flags describe this runtime image and are used by the loader.

Flag	Value	Description
COMIMAGE_FLAGS_ILONLY	0x00000001	Always 1 (see Section 23.1).
COMIMAGE_FLAGS_32BITREQUIRED	0x00000002	Image may 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 (see Section 23.1).

24.3.3.2 Entry Point Meta Data Token

- The entry point token (see Clause 14.4.1.2) is always a MethodDef token (see Section 21.24) or File token (see Section 21.19) 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

24.3.3.3 Vtable Fixup

Certain languages, which choose not to follow the common type system runtime model, may have virtual functions which need to be represented in a v-table. These v-tables are laid out by the compiler, not by the runtime. Finding the correct v-table slot and calling indirectly through the value held in that slot is also done by the compiler. The **VtableFixups** field in the runtime header contains the location and size of an array of Vtable Fixups (see clause 14.5.1). V-tables shall be emitted into a *read-write* section of the PE file.

Each entry in this array describes a contiguous array of v-table slots of the specified size. Each slot starts out initialized to the metadata token value for the method they need to call. At image load time, the runtime Loader will turn each entry into a pointer to machine code for the CPU and can be called directly.

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).

24.3.3.4 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 (see Section 6.2.1.3).

24.4 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 (see Section 21.24) records in metadata carry each method's RVA.

A method consists of a method header immediately followed by the method body, possible followed by extra method data sections (see Section 24.4.5), typically exception handling data. If exception-handling data is present, then CorILMethod_MoreSects flag (see clause 24.4.4) shall be specified in the method header and for each chained item after that.

There are two flavors of method headers - tiny (see clause 24.4.2) and fat (see clause 24.4.3). The three least significant bits in a method header indicate which type is present (see clause 24.4.1). The tiny header is 1 byte long and represents 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. Method headers shall be 4-byte aligned.

24.4.1 Method Header Type Values

The three least significant bits of the first byte of the method header indicate what type of header is present. These 3 bits will be one and only one of the following:

Value	Value	Description
`CorILMethod_TinyFormat`	`0x2`	The method header is tiny (see clause 24.4.2) .
`CorILMethod_FatFormat`	`0x3`	The method header is fat (see clause 24.4.3).

24.4.2 Tiny Format

Tiny headers use a 5 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 need be no bigger than 8 entries

The first encoding has the following format:

Start Bit	Count of Bits	Description
`0`	`2`	Flags (CorILMethod_TinyFormat shall be set, see clause 24.4.4).
`2`	`6`	Size of the method body immediately following this header. Used only when the size of the method is less than 2^6 bytes.

24.4.3 Fat Format

The fat format is used whenever the tiny format is not sufficient. This may be true for one or more of the following reasons:

- The method is too large to encode the size
- 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, see clause 24.4.4)

12 (bits)	4 (bits)	**Size**	Size of this header expressed as the count of 4-byte integers occupied
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

24.4.4 Flags for Method Headers

The first byte of a method header may 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 (see Section 24.4.5).
`CorILMethod_InitLocals`	`0x10`	Call default constructor on all local variables.

24.4.5 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 flags, 1 byte for the length of the actual data) or a four 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. 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 (see Chapter 18). 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 (see Section 24.4.6).

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 (see Section 24.4.6).

24.4.6 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 size for the try block and handler code is smaller than or equal to 256 bytes. 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 the header.
4	1	**TryLength**	Length in bytes of the try block
5	2	**HandlerOffset**	Location of the handler for this try block
7	1	**HandlerLength**	Size of the handler code in bytes
8	4	**ClassToken**	Meta data token for a type-based exception handler
8	4	**FilterOffset**	Offset in method body for filter-based exception handler

The layout of 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 the header.
8	4	**TryLength**	Length in bytes of the try block
12	4	**HandlerOffset**	Location of the handler for this try block
16	4	**HandlerLength**	Size of the handler code in bytes
20	4	**ClassToken**	Meta data token for a type-based exception handler
20	4	**FilterOffset**	Offset in method body for filter-based exception handler

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)

Common Language Infrastructure (CLI)

Partition III:
CIL Instruction Set

Table of Contents

1 Scope

This specification is a detailed description of the Common Intermediate Language (CIL) instruction set, part of the specification of the Common Language Infrastructure. Partition I describes the architecture of the CLI and provides an overview of a large number of issues relating to the CIL instruction set. That overview is essential to an understanding of the instruction set as described here.

Each instruction description describes a set of related CLI machine instructions. Each instruction definition consists of five parts:

- A table describing the binary format, assembly language notation and description of each variant of the instruction. See Section 1.2 .
- A stack transition diagram that describes the state of the evaluation stack before and after the instruction is executed. See Section 1.3.
- An English description of the instruction. See Section 1.4.
- A list of exceptions that might be thrown by the instruction. See Partition I for details. There are three exceptions which may be thrown by any instruction and are not listed with the instruction:

  `ExecutionEngineException` indicates that the internal state of the Execution Engine is corrupted and execution cannot continue. [**Note:** in a system that executes only verifiable code this exception is not thrown.]

  `StackOverflowException` indicates that the hardware stack size has been exceeded. The precise timing of this exception and the conditions under which it occurs are implementation specific. [**Note:** this exception is unrelated to the maximum stack size described in clause 1.7.4. That size relates to the depth of the evaluation stack that is part of the method state described in Partition I, while this exception has to do with the implementation of that method state on physical hardware.]

  `OutOfMemoryException` indicates that the available memory space has been exhausted, either because the instruction inherently allocates memory (`newobj`, `newarr`) or for an implementation-specific reason (for example, an implementation based on just-in-time compilation to native code may run out of space to store the translated method while executing the first `call` or `callvirt` to a given method).
- A section describing the verifiability conditions associated with the instruction. See Section 1.8.

In addition, operations that have a numeric operand also specify an operand type table that describes how they operate based on the type of the operand. See Section 1.5.

Note that not all instructions are included in all CLI Profiles. See Partition IV for details.

1.1 Data Types

While the Common Type System (CTS) defines a rich type system and the Common Language Specification (CLS) specifies a subset that can be used for language interoperability, the CLI itself deals with a much simpler set of types. These types include user-defined value types and a subset of the built-in types. The subset is collectively known as the “basic CLI types”:

- A subset of the full numeric types (**`int32`**, **`int64`**, **`native int`**, and **`F`**)
- Object references (**`O`**) without distinction between the type of object referenced
- Pointer types (**`native unsigned int`** and **`&`**) without distinction as to the type pointed to

Note that object references and pointer types may be assigned the value **`null`**. This is defined throughout the CLI to be zero (a bit pattern of all bits zero)

1.1.1 Numeric Data Types

- The CLI only operates on the numeric types **`int32`** (4-byte signed integers), **`int64`** (8-byte signed integers), **`native int`** (native size integers), and **`F`** (native size floating-point numbers). The CIL instruction set, however, allows additional data types to be implemented:

- **Short integers**. The evaluation stack only holds 4- or 8-byte integers, but other locations (arguments, local variables, statics, array elements, fields) may hold 1- or 2-byte integers. Loading from these locations onto the stack either zero-extends (`ldind.u*`, `ldelem.u*`, etc.) or sign-extends (`ldind.i*`, `ldelem.i*`, etc.) to a 4-byte value. Storing to integers (`stind.u1`, `stelem.i2`, etc.) truncates. Use the `conv.ovf.*` instructions to detect when this truncation results in a value that doesn't correctly represent the original value.

Note: Short integers are loaded as 4-byte numbers on all architectures and these 4-byte numbers must always be tracked as distinct from 8-byte numbers. This helps portability of code by ensuring that the default arithmetic behavior (i.e when no `conv` or `conv.ovf` instruction are executed) will have identical results on all implementations.

Convert instructions that yield short integer values actually leave an `int32` (32-bit) value on the stack, but it is guaranteed that only the low bits have meaning (i.e. the more significant bits are all zero for the unsigned conversions or a sign extension for the signed conversions). To correctly simulate the full set of short integer operations a conversion to the short form is required before the `div`, `rem`, `shr`, comparison and conditional branch instructions.

In addition to the explicit conversion instructions there are four cases where the CLI handles short integers in a special way:

1. Assignment to a local (`stloc`) or argument (`starg`) whose type is declared to be a short integer type automatically truncates to the size specified for the local or argument.
2. Loading from a local (`ldloc`) or argument (`ldarg`) whose type is declared to be a short signed integer type automatically sign extends.
3. Calling a procedure with an argument that is a short integer type is equivalent to assignment to the argument value, so it truncates.
4. Returning a value from a method whose return type is a short integer is modeled as storing into a short integer within the called procedure (i.e. the CLI automatically truncates) and then loading from a short integer within the calling procedure (i.e. the CLI automatically zero- or sign-extends).

In the last two cases it is up to the native calling convention to determine whether values are actually truncated or extended, as well as whether this is done in the called procedure or the calling procedure. The CIL instruction sequence is unaffected and it is as though the CIL sequence included an appropriate `conv` instruction.

- **4-byte integers**. The shortest value actually stored on the stack is a 4-byte integer. These can be converted to 8-byte integers or native-size integers using `conv.*` instructions. Native-size integers can be converted to 4-byte integers, but doing so is not portable across architectures. The `conv.i4` and `conv.u4` can be used for this conversion if the excess significant bits should be ignored; the `conv.ovf.i4` and `conv.ovf.u4` instructions can be used to detect the loss of information. Arithmetic operations allow 4-byte integers to be combined with native size integers, resulting in native size integers. 4-byte integers may not be directly combined with 8-byte integers (they must be converted to 8-byte integers first).
- **Native size integers**. Native size integers can be combined with 4-byte integers using any of the normal arithmetic instructions, and the result will be a native-size integer. Native size integers must be explicitly converted to 8-byte integers before they can be combined with 8-byte integers.
- **8-byte integers**. Supporting 8-byte integers on 32-bit hardware may be expensive, whereas 32-bit arithmetic is available and efficient on current 64-bit hardware. For this reason, numeric instructions allow `int32` and `I` data types to be intermixed (yielding the largest type used as input), but these types *cannot* be combined with `int64`s. Instead, a `native int` or `int32` must be explicitly converted to `int64` before it can be combined with an `int64`.
- **Unsigned integers**. Special instructions are used to interpret integers on the stack as though they were unsigned, rather than tagging the stack locations as being unsigned.

- **Floating-point numbers**. See also Partition I, Handling of Floating Point Datatypes. Storage locations for floating-point numbers (statics, array elements, and fields of classes) are of fixed size. The supported storage sizes are **`float32`** and **`float64`**. Everywhere else (on the evaluation stack, as arguments, as return types, and as local variables) floating-point numbers are represented using an internal floating-point type. In each such instance, the nominal type of the variable or expression is either **`float32`** or **`float64`**, but its value may be represented internally with additional range and/or precision. The size of the internal floating-point representation is implementation-dependent, may vary, and shall have precision at least as great as that of the variable or expression being represented. An implicit widening conversion to the internal representation from **`float32`** or **`float64`** is performed when those types are loaded from storage. The internal representation is typically the natural size for the hardware, or as required for efficient implementation of an operation. The internal representation shall have the following characteristics:
  - The internal representation shall have precision and range greater than or equal to the nominal type.
  - Conversions to and from the internal representation shall preserve value. [Note: This implies that an implicit widening conversion from **`float32`** (or **`float64`**) to the internal representation, followed by an explicit conversion from the internal representation to **`float32`** (or **`float64`**), will result in a value that is identical to the original **`float32`** (or **`float64`**) value.]

Note: The above specification allows a compliant implementation to avoid rounding to the precision of the target type on intermediate computations, and thus permits the use of wider precision hardware registers, as well as the application of optimizing transformations which result in the same or greater precision, such as contractions. Where exactly reproducible behavior is required by a language or application, explicit conversions may be used.

When a floating-point value whose internal representation has greater range and/or precision than its nominal type is put in a storage location, it is automatically coerced to the type of the storage location. This may involve a loss of precision or the creation of an out-of-range value (NaN, +infinity, or -infinity). However, the value may be retained in the internal representation for future use, if it is reloaded from the storage location without having been modified. It is the responsibility of the compiler to ensure that the memory location is still valid at the time of a subsequent load, taking into account the effects of aliasing and other execution threads (see memory model section). This freedom to carry extra precision is not permitted, however, following the execution of an explicit conversion (**`conv.r4`** or **`conv.r8`**), at which time the internal representation must be exactly representable in the associated type.

Note: To detect values that cannot be converted to a particular storage type, use a conversion instruction (**`conv.r4`**, or **`conv.r8`**) and then check for an out-of-range value using **`ckfinite`**. To detect underflow when converting to a particular storage type, a comparison to zero is required before and after the conversion.

Note: This standard does not specify the behavior of arithmetic operations on denormalized floating point numbers, nor does it specify when or whether such representations should be created. This is in keeping with IEC 60559:1989. In addition, this standard does not specify how to access the exact bit pattern of NaNs that are created, nor the behavior when converting a NaN between 32-bit and 64-bit representation. All of this behavior is deliberately left implementation-specific.

1.1.2 Boolean Data Type

A CLI Boolean type occupies one byte in memory. A bit pattern of all zeroes denotes a value of false. A bit pattern with any one or more bits set (analogous to a non-zero integer) denotes a value of true.

1.1.3 Object References

Object references (type O) are completely opaque. There are no arithmetic instructions that allow object references as operands, and the only comparison operations permitted are equality (and inequality) between two object references. There are no conversion operations defined on object references. Object references are created by certain CIL object instructions (notably **`newobj`** and **`newarr`**). Object references can be passed as arguments, stored as local variables, returned as values, and stored in arrays and as fields of objects.

1.1.4 Runtime Pointer Types

There are two kinds of pointers: unmanaged pointers and managed pointers. For pointers into the same array or object (see Partition I), the following arithmetic operations are defined:

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

None of these operations is allowed in verifiable code.

It is important to understand the impact on the garbage collector of using arithmetic on the different kinds of pointers. Since unmanaged pointers must never reference memory that is controlled by the garbage collector, performing arithmetic on them can endanger the memory safety of the system (hence it is not verifiable) but since they are not reported to the garbage collector there is no impact on its operation.

Managed pointers, however, are reported to the garbage collector. As part of garbage collection both the contents of the location to which they point *and* the pointer itself can be modified. The garbage collector will ignore managed pointers if they point into memory that is not under its control (the evaluation stack, the call stack, static memory, or memory under the control of another allocator). If, however, a managed pointer refers to memory controlled by the garbage collector it must point to either a field of an object, an element of an array, or the address of the element just past the end of an array. If address arithmetic is used to create a managed pointer that refers to any other location (an object header or a gap in the allocated memory) the garbage collector's operation is unspecified.

1.1.4.1 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 legal to mark locations that contain unmanaged pointers as though they were unsigned integers (and this is, in fact, how they are treated by the CLI), it is often better to mark them as unmanaged pointers to a specific type of data. This is done by using `ELEMENT_TYPE_PTR` 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.

- Unmanaged pointers should be treated as unsigned (i.e. use **`conv.ovf.u`** rather than **`conv.ovf.i`**, etc.).
- Verifiable code cannot use unmanaged pointers to reference memory.
- 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 managed by the garbage collector
  b. The unmanaged pointer refers to a field within an object
  c. The unmanaged pointer refers to an element within an array
  d. The unmanaged pointer refers to the location where the element following the last element in an array would be located

1.1.4.2 Managed Pointers (type &)

Managed pointers (**`&`**) may point to a local variable, a method argument, 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`. (They must be reported to the garbage collector, even if they do not point to managed memory)

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

- Managed pointers can be passed as arguments and stored in local variables.
- If you pass a parameter 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.
- 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. This conversion is safe if any of the following is known to be true:
    a. the managed pointer does not point into the garbage collector's memory area
    b. the memory referred to has been pinned for the entire time that the unmanaged pointer is in use
    c. a garbage collection cannot occur while the unmanaged pointer is in use
    d. the garbage collector for the given implementation of the CLI is known to not move the referenced memory

1.2 Instruction Variant Table

In Chapter 3 an Instruction Variant Table is presented for each instruction. It describes each variant of the instructions. The "Format" column of the table lists the opcode for the instruction variant, along with any arguments that follow the instruction in the instruction stream. For example:

Format	**Assembly Format**	**Description**
FE 0A <*unsigned int16*>	Ldarga *argNum*	fetch the address of argument *argNum*.
0F <*unsigned int8*>	Ldarga.s *argNum*	fetch the address of argument *argNum*, short form

The first one or two hex numbers in the "Format" column show how this instruction is encoded (its "opcode"). So, the **`ldarga`** instruction is encoded as a byte holding FE, followed by another holding 0A. Italicized type names represent numbers that should follow in the instruction stream. In this example a 2-byte quantity that is to be treated as an unsigned integer directly follows the FE 0A opcode.

Any of the fixed size built-in types (**`int8`**, **`unsigned int8`**, **`int16`**, **`unsigned int16,`** **`int32`**, **`unsigned int32`**, **`int64,`** **`unsigned in64`**, **`float32`**, and **`float64`**) can appear in format descriptions. These types define the number of bytes for the argument and how it should be interpreted (signed, unsigned or floating-point). In addition, a metadata token can appear, indicated as <*T*>. Tokens are encoded as 4-byte integers. All argument numbers are encoded least-significant-byte-at-smallest-address (a pattern commonly termed "little-endian"). Bytes for instruction opcodes and arguments are packed as tightly as possible (no alignment padding is done).

The assembly format column defines an assembly code mnemonic for each instruction variant. For those instructions that have instruction stream arguments, this column also assigns names to each of the arguments to the instruction. For each instruction argument, there is a name in the assembly format. These names are used later in the instruction description.

1.2.1 Opcode Encodings

CIL opcodes are one or more bytes long; they may be followed by zero or more operand bytes. All opcodes whose first byte lies in the ranges 0x00 through 0xEF, or 0xFC through 0xFF are reserved for standardization. Opcodes whose first byte lies in the range 0xF0 through 0xFB inclusive, are available for experimental purposes. The use of experimental opcodes in any method renders the method invalid and hence unverifiable.

The currently defined encodings are specified in Table 1: Opcode Encodings.

Table 1: Opcode Encodings

0x00	nop
0x01	break
0x02	ldarg.0
0x03	ldarg.1
0x04	ldarg.2
0x05	ldarg.3
0x06	ldloc.0
0x07	ldloc.1
0x08	ldloc.2
0x09	ldloc.3
0x0a	stloc.0
0x0b	stloc.1
0x0c	stloc.2
0x0d	stloc.3
0x0e	ldarg.s
0x0f	ldarga.s
0x10	starg.s
0x11	ldloc.s
0x12	ldloca.s
0x13	stloc.s
0x14	ldnull
0x15	ldc.i4.m1
0x16	ldc.i4.0
0x17	ldc.i4.1
0x18	ldc.i4.2
0x19	ldc.i4.3
0x1a	ldc.i4.4
0x1b	ldc.i4.5
0x1c	ldc.i4.6
0x1d	ldc.i4.7
0x1e	ldc.i4.8
0x1f	ldc.i4.s
0x20	ldc.i4
0x21	ldc.i8
0x22	ldc.r4
0x23	ldc.r8
0x25	dup
0x26	pop
0x27	jmp
0x28	call
0x29	calli
0x2a	ret
0x2b	br.s
0x2c	brfalse.s
0x2d	brtrue.s
0x2e	beq.s
0x2f	bge.s
0x30	bgt.s
0x31	ble.s
0x32	blt.s
0x33	bne.un.s
0x34	bge.un.s
0x35	bgt.un.s
0x36	ble.un.s
0x37	blt.un.s
0x38	br
0x39	brfalse
0x3a	brtrue
0x3b	beq
0x3c	bge
0x3d	bgt
0x3e	ble
0x3f	blt
0x40	bne.un
0x41	bge.un
0x42	bgt.un
0x43	ble.un
0x44	blt.un
0x45	switch
0x46	ldind.i1
0x47	ldind.u1
0x48	ldind.i2

0x49	ldind.u2
0x4a	ldind.i4
0x4b	ldind.u4
0x4c	ldind.i8
0x4d	ldind.i
0x4e	ldind.r4
0x4f	ldind.r8
0x50	ldind.ref
0x51	stind.ref
0x52	stind.i1
0x53	stind.i2
0x54	stind.i4
0x55	stind.i8
0x56	stind.r4
0x57	stind.r8
0x58	add
0x59	sub
0x5a	mul
0x5b	div
0x5c	div.un
0x5d	rem
0x5e	rem.un
0x5f	and
0x60	or
0x61	xor
0x62	shl
0x63	shr
0x64	shr.un
0x65	neg
0x66	not
0x67	conv.i1
0x68	conv.i2
0x69	conv.i4
0x6a	conv.i8
0x6b	conv.r4
0x6c	conv.r8
0x6d	conv.u4

0x6e	conv.u8
0x6f	callvirt
0x70	cpobj
0x71	ldobj
0x72	ldstr
0x73	newobj
0x74	castclass
0x75	isinst
0x76	conv.r.un
0x79	unbox
0x7a	throw
0x7b	ldfld
0x7c	ldflda
0x7d	stfld
0x7e	ldsfld
0x7f	ldsflda
0x80	stsfld
0x81	stobj
0x82	conv.ovf.i1.un
0x83	conv.ovf.i2.un
0x84	conv.ovf.i4.un
0x85	conv.ovf.i8.un
0x86	conv.ovf.u1.un
0x87	conv.ovf.u2.un
0x88	conv.ovf.u4.un
0x89	conv.ovf.u8.un
0x8a	conv.ovf.i.un
0x8b	conv.ovf.u.un
0x8c	box
0x8d	newarr
0x8e	ldlen
0x8f	ldelema
0x90	ldelem.i1
0x91	ldelem.u1
0x92	ldelem.i2
0x93	ldelem.u2
0x94	ldelem.i4

0x95	ldelem.u4
0x96	ldelem.i8
0x97	ldelem.i
0x98	ldelem.r4
0x99	ldelem.r8
0x9a	ldelem.ref
0x9b	stelem.i
0x9c	stelem.i1
0x9d	stelem.i2
0x9e	stelem.i4
0x9f	stelem.i8
0xa0	stelem.r4
0xa1	stelem.r8
0xa2	stelem.ref
0xb3	conv.ovf.i1
0xb4	conv.ovf.u1
0xb5	conv.ovf.i2
0xb6	conv.ovf.u2
0xb7	conv.ovf.i4
0xb8	conv.ovf.u4
0xb9	conv.ovf.i8
0xba	conv.ovf.u8
0xc2	refanyval
0xc3	ckfinite
0xc6	mkrefany
0xd0	ldtoken
0xd1	conv.u2
0xd2	conv.u1
0xd3	conv.i
0xd4	conv.ovf.i
0xd5	conv.ovf.u
0xd6	add.ovf
0xd7	add.ovf.un
0xd8	mul.ovf
0xd9	mul.ovf.un
0xda	sub.ovf
0xdb	sub.ovf.un

0xdc	endfinally
0xdd	leave
0xde	leave.s
0xdf	stind.i
0xe0	conv.u
0xfe 0x00	arglist
0xfe 0x01	ceq
0xfe 0x02	cgt
0xfe 0x03	cgt.un
0xfe 0x04	clt
0xfe 0x05	clt.un
0xfe 0x06	ldftn
0xfe 0x07	ldvirtftn
0xfe 0x09	ldarg
0xfe 0x0a	ldarga
0xfe 0x0b	starg
0xfe 0x0c	ldloc
0xfe 0x0d	ldloca
0xfe 0x0e	stloc
0xfe 0x0f	localloc
0xfe 0x11	endfilter
0xfe 0x12	unaligned.
0xfe 0x13	volatile.
0xfe 0x14	tail.
0xfe 0x15	initobj
0xfe 0x17	cpblk
0xfe 0x18	initblk
0xfe 0x1a	rethrow
0xfe 0x1c	sizeof
0xfe 0x1d	refanytype

1.3 Stack Transition Diagram

The stack transition diagram displays the state of the evaluation stack before and after the instruction is executed. Below is a typical stack transition diagram.

…, value1, value2 → …, result

This diagram indicates that the stack must have at least two elements on it, and in the definition the topmost value ("top of stack" or "most recently pushed") will be called *value2* and the value underneath (pushed prior to *value2*) will be called *value1*. (In diagrams like this, the stack grows to the right, along the page). The instruction removes these values from the stack and replaces them by another value, called *result* in the description.

1.4 English Description

The English description describes any details about the instructions that are not immediately apparent once the format and stack transition have been described.

1.5 Operand Type Table

Many CIL operations take numeric operands on the stack. These operations fall into several categories, depending on how they deal with the types of the operands. The following tables summarize the valid kinds of operand types and the type of the result. Notice that the type referred to here is the type as tracked by the CLI rather than the more detailed types used by tools such as CIL verification. The types tracked by the CLI are: **`int32`**, **`int64`**, **`native int`**, **`F`**, **`O`**, and **`&`**.

A **`op`** B (used for **`add`**, **`div`**, **`mul`**, **`rem`**, and **`sub`**). The table below shows the result type, for each possible combination of operand types. Boxes holding simply a result type, apply to all five instructions. Boxes marked × indicate an invalid CIL instruction. Shaded boxes indicate a CIL instruction that is not verifiable. Boxes with a list of instructions are valid only for those instructions.

Table 2: Binary Numeric Operations

A's Type	**B's Type**					
	int32	**int64**	**native int**	**F**	**&**	**O**
int32	int32	×	native int	×	& (add)	×
int64	×	int64	×	×	×	×
native int	native int	×	native int	×	& (add)	×
F	×	×	×	F	×	×
&	& (add, sub)	×	& (add, sub)	×	native int (sub)	×
O	×	×	×	×	×	×

Used for the **`neg`** instruction. Boxes marked × indicate an invalid CIL instruction. All valid uses of this instruction are verifiable.

Table 3: Unary Numeric Operations

Operand Type	`int32`	`int64`	`native int`	`F`	`&`	`O`
Result Type	`int32`	`int64`	`native int`	`F`	×	×

These return a boolean value or branch based on the top two values on the stack. Used for **`beq`**, **`beq.s`**, **`bge`**, **`bge.s`**, **`bge.un`**, **`bge.un.s`**, **`bgt`**, **`bgt.s`**, **`bgt.un`**, **`bgt.un.s`**, **`ble`**, **`ble.s`**, **`ble.un`**, **`ble.un.s`**, **`blt`**, **`blt.s`**, **`blt.un`**, `blt.un.s`, `bne.un`, `bne.un.s`, `ceq`, `cgt`, `cgt.un`, `clt`, `clt.un`. Boxes marked ✓ indicate that all instructions are valid for that combination of operand types. Boxes marked ✗ indicate invalid CIL sequences. Shaded boxes boxes indicate a CIL instruction that is not verifiable. Boxes with a list of instructions are valid only for those instructions.

Table 4: Binary Comparison or Branch Operations

	int32	**int64**	**native int**	**F**	**&**	**O**
int32	✓	✗	✓	✗	✗	✗
int64	✗	✓	✗	✗	✗	✗
native int	✓	✗	✓	✗	Beq[.s], bne.un[.s], ceq	✗
F	✗	✗	✗	✓	✗	✗
&	✗	✗	beq[.s], bne.un[.s], ceq	✗	[1] ✓	✗
O	✗	✗	✗	✗	✗	beq[.s], bne.un[.s], ceq[2]

1. Except for `beq`, `bne.un` (or short versions) or `ceq` these combinations make sense if both operands are known to be pointers to elements of the same array. However, there is no security issue for a CLI that does not check this constraint

   Note: if the two operands are *not* pointers into the same array, then the result is simply the distance apart in the garbage-collected heap of two unrelated data items. This distance apart will almost certainly change at the next garbage collection. Essentially, the result cannot be used to compute anything useful

2. `cgt.un` is allowed and verifiable on ObjectRefs (O). This is commonly used when comparing an ObjectRef with null (there is no "compare-not-equal" instruction, which would otherwise be a more obvious solution)

These operate only on integer types. Used for `and`, `div.un`, `not`, `or`, `rem.un`, `xor`. The `div.un` and `rem.un` instructions treat their arguments as unsigned integers and produce the bit pattern corresponding to the unsigned result. As described in the CLI Specification, however, the CLI makes no distinction between signed and unsigned integers on the stack. The `not` instruction is unary and returns the same type as the input. The `shl` and `shr` instructions return the same type as their first operand and their second operand must be of type native unsigned int. Boxes marked ✗ indicate invalid CIL sequences. All other boxes denote verifiable combinations of operands.

Table 5: Integer Operations

	int32	**int64**	**native int**	**F**	**&**	**O**
int32	int32	✗	native int	✗	✗	✗
int64	✗	int64	✗	✗	✗	✗
native int	native int	✗	native int	✗	✗	✗
F	✗	✗	✗	✗	✗	✗
&	✗	✗	✗	✗	✗	✗
O	✗	✗	✗	✗	✗	✗

Below are the legal combinations of operands and result for the shift instructions: `shl`, `shr`, `shr_un`. Boxes marked ✗ indicate invalid CIL sequences. All other boxes denote verifiable combinations of operand. If the

"Shift-By" operand is larger than the width of the "To-Be-Shifted" operand, then the results are implementation-defined. (eg shift an int32 integer left by 37 bits)

Table 6: Shift Operations

		Shift-By					
		int32	**int64**	**native int**	**F**	**&**	**O**
To Be Shifted	**int32**	int32	×	int32	×	×	×
	int64	int64	×	int64	×	×	×
	native int	native int	×	native int	×	×	×
	F	×	×	×	×	×	×
	&	×	×	×	×	×	×
	O	×	×	×	×	×	×

These operations generate an exception if the result cannot be represented in the target data type. Used for `add.ovf`, `add.ovf.un`, `mul.ovf`, `mul.ovf.un`, `sub.ovf`, `sub.ovf.un` The shaded uses are not verifiable, while boxes marked × indicate invalid CIL sequences.

Table 7: Overflow Arithmetic Operations

	int32	**int64**	**native int**	**F**	**&**	**O**
int32	int32	×	native int	×	& `add.ovf.un`	×
int64	×	int64	×	×	×	×
native int	native int	×	native int	×	& `add.ovf.un`	×
F	×	×	×	×	×	×
&	& `add.ovf.un`, `sub.ovf.un`	×	& `add.ovf.un`, `sub.ovf.un`	×	native int `sub.ovf.un`	×
O	×	×	×	×	×	×

These operations convert the top item on the evaluation stack from one numeric type to another. The result type is guaranteed to be representable as the data type specified as part of the operation (i.e. the `conv.u2` instruction returns a value that can be stored in a `unsigned int16`). The stack, however, can only store values that are a minimum of 4 bytes wide. Used for the `conv.<`to type`>`, `conv.ovf.<`to type`>`, and `conv.ovf.<`to type`>.un` instructions. The shaded uses are not verifiable, while boxes marked × indicate invalid CIL sequences.

Table 8: Conversion Operations

Convert-To	**Input (from evaluation stack)**					
	int32	**int64**	**native int**	**F**	**&**	**O**
int8 **unsigned int8** **int16** **unsigned int16**	Truncate[1]	Truncate[1]	Truncate[1]	Truncate to zero[2]	×	×
int32 **unsigned int32**	Nop	Truncate[1]	Truncate[1]	Truncate to zero[2]	×	×
int64	Sign extend	Nop	Sign extend	Truncate to zero[2]	Stop GC tracking	Stop GC tracking

unsigned int64	Zero extend	Nop	Zero extend	Truncate to zero[2]	Stop GC tracking	Stop GC tracking
native int	Sign extend	Truncate[1]	Nop	Truncate to zero[2]	Stop GC tracking	Stop GC tracking
native unsigned int	Zero extend	Truncate[1]	Nop	Truncate to zero[2]	Stop GC tracking	Stop GC tracking
All Float Types	To Float	To Float	To Float	Change precision[3]	✗	✗

1. "Truncate" means that the number is truncated to the desired size; ie, the most significant bytes of the input value are simply ignored. If the result is narrower than the minimum stack width of 4 bytes, then this result is zero extended (if the target type is unsigned) or sign-extended (if the target type is signed). Thus, converting the value 0x1234 ABCD from the evaluation stack to an 8-bit datum yields the result 0xCD; if the target type were int8, this is sign-extended to give 0xFFFF FFCD; if, instead, the target type were unsigned int8, this is zero-extended to give 0x0000 00CD.
2. "Trunc to 0" means that the floating-point number will be converted to an integer by truncation toward zero. Thus 1.1 is converted to 1 and –1.1 is converted to –1.
3. Converts from the current precision available on the evaluation stack to the precision specified by the instruction. If the stack has more precision than the output size the conversion is performed using the IEC 60559:1989 "round to nearest" mode to compute the low order bit of the result.
4. "Stop GC Tracking" means that, following the conversion, the item's value will *not* be reported to subsequent garbage-collection operations (and therefore will not be updated by such operations)

1.6 Implicit Argument Coercion

While the CLI operates only on 6 types (int32, native int, int64, F, O, and &) the metadata supplies a much richer model for parameters of methods. When about to call a method, the CLI performs implicit type conversions, detailed in the following table. (Conceptually, it inserts the appropriate `conv.*` instruction into the CIL stream, which may result in an information loss through truncation or rounding) This implicit conversion occurs for boxes marked ✓. Shaded boxes are not verifiable. Boxes marked ✗ indicate invalid CIL sequences. (A compiler is of course free to emit explicit `conv.*` **or** `conv.*.ovf` instructions to achieve any desired effect)

Table 9: Signature Matching

Type In Signature	**Stack Parameter**					
	int32	**native int**	**int64**	**F**	**&**	**O**
int8	✓	✓	✗	✗	✗	✗
unsigned int8, bool	✓	✓	✗	✗	✗	✗
int16	✓	✓	✗	✗	✗	✗
unsigned int16, char	✓	✓	✗	✗	✗	✗
int32	✓	✓	✗	✗	✗	✗
unsigned int32	✓	✓	✗	✗	✗	✗
int64	✗	✗	✓	✗	✗	✗
unsigned	✗	✗	✓	✗	✗	✗

int64						
native int	✓ Sign extend	✓	✗	✗	✗	✗
native unsigned int	✓ Zero extend	✓ Zero extend	✗	✗	✗	✗
float32	✗	✗	✗	Note[4]	✗	✗
float64	✗	✗	✗	Note[4]	✗	✗
Class	✗	✗	✗	✗	✗	✓
Value Type (Note[2])	Note[1]	Note[1]	Note[1]	Note[1]	✗	✗
By-Ref (&)	✗	✓ Start GC tracking	✗	✗	✓	✗
Ref Any (Note[3])	✗	✗	✗	✗	✗	✗

1. Passing a built-in type to a parameter that is required to be a value type is not allowed.
2. The CLI's stack can contain a value type. These may only be passed if the particular value type on the stack exactly matches the class required by the corresponding parameter.
3. There are special instructions to construct and pass a `Ref Any`.
4. The CLI is permitted to pass floating point arguments using its internal F type, see clause 1.1.1. CIL generators may, of course, include an explicit conv.r4, conv.r4.ovf, or similar instruction.

Further notes concerning this table:

- On a 32-bit machine passing a `native int` argument to an `unsigned int32` parameter involves no conversion. On a 64-bit machine it is implicitly converted.
- "Start GC Tracking" means that, following the implicit conversion, the item's value will be reported to any subsequent garbage-collection operations, and perhaps changed as a result of the item pointed-to being relocated in the heap.

1.7 Restrictions on CIL Code Sequences

As well as detailed restrictions on CIL code sequences to ensure:

- Valid CIL
- Verifiable CIL

there are a few further restrictions, imposed to make it easier to construct a simple CIL-to-native-code compiler. This section specifies the general restrictions that apply in addition to this listed for individual instructions.

1.7.1 The Instruction Stream

The implementation of a method is provided by a contiguous block of CIL instructions, encoded as specified below. The address of the instruction block for a method as well as its length is specified in the file format (see Partition II, Common Intermediate Language Physical Layout). The first instruction is at the first byte (lowest address) of the instruction block.

Instructions are variable in size. The size of each instruction can be determined (decoded) from the content of the instruction bytes themselves. The size of and ordering of the bytes within an instruction is specified by each instruction definition. Instructions follow each other without padding in a stream of bytes that is both alignment and byte-order insensitive.

Each instruction occupies an exact number of bytes, and until the end of the instruction block, the next instruction begins immediately at the next byte. It is invalid for the instruction block (as specified by the block's length) to end without forming a complete last instruction.

Instruction prefixes extend the length of an instruction without introducing a new instruction; an instruction having one or more prefixes introduces only one instruction that begins at the first byte of the first instruction prefix.

Note: Until the end of the instruction block, the instruction following any control transfer instruction is decoded as an instruction and thus participates in locating subsequent instructions even if it is not the target of a branch. Only instructions may appear in the instruction stream, even if unreachable. There are no address-relative data addressing modes and raw data cannot be directly embedded within the instruction stream. Certain instructions allow embedding of immediate data as part of the instruction, however that differs from allowing raw data embedded directly in the instruction stream. Unreachable code may appear as the result of machine-generated code and is allowed, but it must always be in the form of properly formed instruction sequences.

The instruction stream can be translated and the associated instruction block discarded prior to execution of the translation. Thus, even instructions that capture and manipulate code addresses, such as `call`, `ret`, etc. can be virtualized to operate on translated addresses instead of addresses in the CIL instruction stream.

1.7.2 Valid Branch Targets

The set of addresses composed of the first byte of each instruction identified in the instruction stream defines the only valid instruction targets. Instruction targets include branch targets as specified in branch instructions, targets specified in exception tables such as protected ranges (see Partition I and Partition II), filter, and handler targets.

Branch instructions specify branch targets as either a 1-byte or 4-byte signed relative offset; the size of the offset is differentiated by the opcode of the instruction. The offset is defined as being relative to the byte following the branch instruction. [**Note**: Thus, an offset value of zero targets the immediately following instruction.]

The value of a 1-byte offset is computed by interpreting that byte as a signed 8-bit integer. The value of a 4-byte offset is can be computed by concatenating the bytes into a signed integer in the following manner: the byte of lowest address forms the least significant byte, and the byte with highest address forms the most significant byte of the integer. [**Note**: This representation is often called "a signed integer in little-endian byte-order".]

1.7.3 Exception Ranges

Exception tables describe ranges of instructions that are protected by catch, fault, or finally handlers (see Partition I and Partition II). The starting address of a protected block, filter clause, or handler shall be a valid branch target as specified in clause 1.7.2. It is invalid for a protected block, filter clause, or handler to end without forming a complete last instruction.

1.7.4 Must Provide Maxstack

Every method specifies a maximum number of items that can be pushed onto the CIL Evaluation. The value is stored in the `IMAGE_COR_ILMETHOD` structure that precedes the CIL body of each method. A method that specifies a maximum number of items less than the amount required by a static analysis of the method (using a traditional control flow graph without analysis of the data) is invalid (hence also unverifiable) and need not be supported by a conforming implementation of the CLI.

Note: Maxstack is related to analysis of the program, not to the size of the stack at runtime. It does not specify the maximum size in bytes of a stack frame, but rather the number of items that must be tracked by an analysis tool.

Rationale: *By analyzing the CIL stream for any method, it is easy to determine how many items will be pushed on the CIL Evaluation stack. However, specifying that maximum number ahead of time helps a CIL-to-native-code compiler (especially a simple one that does only a single pass through the CIL stream) in allocating internal data structures that model the stack and/or verification algorithm.*

1.7.5 Backward Branch Constraints

It must be possible, with a single forward-pass through the CIL instruction stream for any method, to infer the exact state of the evaluation stack at every instruction (where by "state" we mean the number and type of each item on the evaluation stack).

In particular, if that single-pass analysis arrives at an instruction, call it location X, that immediately follows an unconditional branch, and where X is not the target of an earlier branch instruction, then the state of the evaluation stack at X, clearly, cannot be derived from existing information. In this case, the CLI demands that the evaluation stack at X be empty.

Following on from this rule, it would clearly be invalid CIL if a later branch instruction to X were to have a non-empty evaluation stack

Rationale: *This constraint ensures that CIL code can be processed by a simple CIL-to-native-code compiler. It ensures that the state of the evaluation stack at the beginning of each CIL can be inferred from a single, forward-pass analysis of the instruction stream.*

Note: the stack state at location X in the above can be inferred by various means: from a previous forward branch to X; because X marks the start of an exception handler, etc.

See the following sections for further information:

- Exceptions: Partition I
- Verification conditions for branch instructions: Chapter 3
- The **`tail.`** prefix: Section 3.19

1.7.6 Branch Verification Constraints

The *target* of all branch instruction must be a valid branch target (see clause 1.7.2) within the method holding that branch instruction.

1.8 Verifiability

Memory safety is a property that ensures programs running in the same address space are correctly isolated from one another (see Partition I). Thus, it is desirable to test whether programs are memory safe prior to running them. Unfortunately, it is provably impossible to do this with 100% accuracy. Instead, the CLI can test a stronger restriction, called *verifiability*. Every program that is verified is memory safe, but some programs that are not verifiable are still memory safe.

It is perfectly acceptable to generate CIL code that is not verifiable, but which is known to be memory safe by the compiler writer. Thus, conforming CIL may not be verifiable, even though the producing compiler may *know* that it is memory safe. Several important uses of CIL instructions are not verifiable, such as the pointer arithmetic versions of **`add`** that are required for the faithful and efficient compilation of C programs. For non-verifiable code, memory safety is the responsibility of the application programmer.

CIL contains a *verifiable subset*. The Verifiability description gives details of the conditions under which a use of an instruction falls within the verifiable subset of CIL. Verification tracks the types of values in much finer detail than is required for the basic functioning of the CLI, because it is checking that a CIL code sequence respects not only the basic rules of the CLI with respect to the safety of garbage collection, but also the typing rules of the CTS. This helps to guarantee the sound operation of the entire CLI.

The verifiability section of each operation description specifies requirements both for correct CIL generation and for verification. Correct CIL generation always requires guaranteeing that the top items on the stack correspond to the types shown in the stack transition diagram. The verifiability section specifies only requirements for correct CIL generation that are not captured in that diagram. Verification tests both the requirements for correct CIL generation and the specific verification conditions that are described with the instruction. The operation of CIL sequences that do not meet the CIL correctness requirements is unspecified. The operation of CIL sequences that meet the correctness requirements but are not verifiable may violate type safety and hence may violate security or memory access constraints.

1.8.1 Flow Control Restrictions for Verifiable CIL

This section specifies a verification algorithm that, combined with information on individual CIL instructions (see Chapter 3) and metadata validation (see Partition II), guarantees memory integrity.

The algorithm specified here creates a minimum level for all compliant implementations of the CLI in the sense that any program that is considered verifiable by this algorithm shall be considered verifiable and run correctly on all compliant implementations of the CLI.

The CLI provides a security permission (see Partition IV) that controls whether or not the CLI shall run programs that may violate memory safety. Any program that is verifiable according to this specification does not violate memory safety, and a conforming implementation of the CLI shall run such programs. The implementation may also run other programs provided it is able to show they do not violate memory safety (typically because they use a verification algorithm that makes use of specific knowledge about the implementation).

Note: While a compliant implementation is required to accept and run any program this verification algorithm states is verifiable, there may be programs that are accepted as verifiable by a given implementation but which this verification algorithm will fail to consider verifiable. Such programs will run in the given implementation but need not be considered verifiable by other implementations.

For example, an implementation of the CLI may choose to correctly track full signatures on method pointers and permit programs to execute the **calli** instruction even though this is not permitted by the verification algorithm specified here.

Implementers of the CLI are urged to provide a means for testing whether programs generated on their implementation meet this portable verifiability standard. They are also urged to specify where their verification algorithms are more permissive than this standard.

Only valid programs shall be verifiable. For ease of explanation, the verification algorithm described here assumes that the program is valid and does not explicitly call for tests of all validity conditions. Validity conditions are specified on a per-CIL instruction basis (see Chapter 3), and on the overall file format in Partition II.

1.8.1.1 Verification Algorithm

The verification algorithm shall attempt to associate a valid `stack state` with every CIL instruction. The stack state specifies the number of slots on the CIL stack at that point in the code and for each slot a required type that must be present in that slot. The initial stack state is empty (there are no items on the stack).

Verification assumes that the CLI zeroes all memory other than the evaluation stack before it is made visible to programs. A conforming implementation of the CLI shall provide this observable behavior. Furthermore, verifiable methods shall have the “zero initialize” bit set, see Partition II (Flags for Method Headers). If this bit is not set, then a CLI may throw a *Verification* exception at any point where a local variable is accessed, and where the assembly containing that method has not been granted *SecurityPermission.SkipVerification*

Rationale: *This requirement strongly enhances program portability, and a well-known technique (definite assignment analysis) allows a compiler from CIL to native code to minimize its performance impact. Note that a CLI may optionally choose to perform definite-assignment analysis – in such a case, it may confirm that a method, even without the “zero initialize” bit set, may in fact be verifiable (and therefore not throw a Verification exception)*

Note: Definite assignment analysis can be used by the CLI to determine which locations are written before they are read. Such locations needn’t be zeroed, since it isn’t possible to observe the contents of the memory as it was provided by the EE.

Performance measurements on C++ implementations (which does not require definite assignment analysis) indicate that adding this requirement has almost no impact, even in highly optimized code. Furthermore, customers incorrectly attribute bugs to the compiler when this zeroing is not performed, since such code often fails when small, unrelated changes are made to the program.

The verification algorithm shall simulate all possible control flow paths through the code and ensures that a legal stack state exists for every reachable CIL instruction. The verification algorithm does not take advantage of any data values during its simulation (e.g. it does not perform constant propagation), but uses only type assignments. Details of the type system used for verification and the algorithm used to merge stack states are provided in clause 1.8.1.3. The verification algorithm terminates as follows:

1. Successfully, when all control paths have been simulated.
2. Unsuccessfully when it is not possible to compute a valid stack state for a particular CIL instruction.
3. Unsuccessfully when additional tests specified in this clause fail.

There is a control flow path from every instruction to the subsequent instruction, with the exception of the unconditional branch instructions, **`throw`**, **`rethrow`**, and **`ret`**. Finally, there is a control flow path from each branch instruction (conditional or unconditional) to the branch target (targets, plural, for the **`switch`** instruction).

Verification simulates the operation of each CIL instruction to compute the new stack state, and any type mismatch between the specified conditions on the stack state (see Chapter 3) and the simulated stack state shall cause the verification algorithm to fail. (Note that verification simulates only the effect on the stack state: it does not perform the actual computation). The algorithm shall also fail if there is an existing stack state at the next instruction address (for conditional branches or instructions within a **try** block there may be more than one such address) that cannot be merged with the stack state just computed. For rules of this merge operation, see clause 1.8.1.3.

1.8.1.2 Verification Type System

The verification algorithm compresses types that are logically equivalent, since they cannot lead to memory safety violations. The types used by the verification algorithm are specified in clause 1.8.1.2.1, the type compatibility rules are specified in clause 1.8.1.2.2, and the rules for merging stack states are in clause 1.8.1.3.

1.8.1.2.1 Verification Types

The following table specifies the mapping of types used in the CLI and those used in verification. Notice that verification compresses the CLI types to a smaller set that maintains information about the size of those types in memory, but then compresses these again to represent the fact that the CLI stack expands 1, 2 and 4 byte built-in types into 4-byte types on the stack. Similarly, verification treats floating-point numbers on the stack as 64-bit quantities regardless of the actual representation.

Arrays are objects, but with special compatibility rules.

There is a special encoding for **`null`** that represents an object known to be the null value, hence with indeterminate actual type.

In the following table, “CLI Type” is the type as it is described in metadata. The “Verification Type” is a corresponding type used for type compatibility rules in verification (see clause 1.8.1.2.2) when considering the types of local variables, incoming arguments, and formal parameters on methods being called. The column “Verification Type (in stack state)” is used to simulate instructions that load data onto the stack, and shows the types that are actually maintained in the stack state information of the verification algorithm. The column “Managed Pointer to Type” shows the type tracked for managed pointers.

CLI Type	**Verification Type**	**Verification Type (in stack state)**	**Managed Pointer to Type**
`int8, unsigned int8, bool`	`int8`	`int32`	`& int8`
`int16, unsigned int16, char`	`int16`	`int32`	`& int16`
`int32, unsigned int32`	`int32`	`int32`	`& int32`
`int64, unsigned int64`	`int64`	`int64`	`& int64`
`native int, native unsigned int`	`native int`	`native int`	`& native int`
`float32`	`float32`	`float64`	`& float32`
`float64`	`float64`	`float64`	`& float64`

`Any value type`	`Same type`	`Same type`	`& Same type`
`Any object type`	`Same type`	`Same type`	`& Same type`
`Method pointer`	`Same type`	`Same type`	`Not valid`

A method can be defined as returning a managed pointer, but calls upon such methods are not verifiable.

Rationale: *some uses of returning a managed pointer are perfectly verifiable (eg, returning a reference to a field in an object); but some not (eg, returning a pointer to a local variable of the called method). Tracking this in the general case is a burden, and therefore not included in this standard.*

1.8.1.2.2 Verification Type Compatibility

The following rules define type compatibility. We use `S` and `T` to denote verification types, and the notation "`S := T`" to indicate that the verification type `T` can be used wherever the verification type `S` can be used, while "`S !:= T`" indicates that `T` cannot be used where `S` is expected. These are the verification type compatibility (see Partition I) rules. We use `T[]` to denote an array (of any rank) whose elements are of type `T`, and `T&` to denote a managed pointer to type `T`.

1. [`:=` is reflexive] For all verification types `S`, `S := S`
2. [`:=` is transitive] For all verification types `S`, `T`, and `U` if `S := T` and `T := U`, then `S := U`.
3. `S := T` if `S` is the base class of `T` or an interface implemented by `T` and `T` is not a value type.
4. `S := T` if `S` and `T` are both interfaces and the implementation of `T` requires the implementation of `S`
5. `S := null` if `S` is an object type or an interface
6. `S[] := T[]` if `S := T` and the arrays are either both vectors (zero-based, rank one) or neither is a vector and both have the same rank.
7. If `S` and `T` are method pointers, then `S := T` if the signatures (return types, parameter types, calling convention, and any custom attributes or custom modifiers) are the same.
8. Otherwise `S !:= T`

1.8.1.3 Merging Stack States

As the verification algorithm simulates all control flow paths it shall merge the simulated stack state with any existing stack state at the next CIL instruction in the flow. If there is no existing stack state, the simulated stack state is stored for future use. Otherwise the merge shall be computed as follows and stored to replace the existing stack state for the CIL instruction. If the merge fails, the verification algorithm shall fail.

The merge shall be computed by comparing the number of slots in each stack state. If they differ, the merge shall fail. If they match, then the overall merge shall be computed by merging the states slot-by-slot as follows. Let `T` be the type from the slot on the newly computed state and `S` be the type from the corresponding slot on the previously stored state. The merged type, `U`, shall be computed as follows (recall that `S := T` is the compatibility function defined in clause 1.8.1.2.2):

1. if `S := T` then `U=S`
2. Otherwise if `T := S` then `U=T`
3. Otherwise, if `S` and `T` are both object types, then let `V` be the closest common supertype of `S` and `T` then `U=V`.
4. Otherwise, the merge shall fail.

1.8.1.4 Class and Object Initialization Rules

The VES ensures that all statics are initially zeroed (i.e. built-in types are 0 or false, object references are null), hence the verification algorithm does not test for definite assignment to statics.

An object constructor shall not return unless a constructor for the base class or a different construct for the object's class has been called on the newly constructed object. The verification algorithm shall treat the `this` pointer as uninitialized unless the base class constructor has been called. No operations can be performed on an uninitialized **this** except for storing into and loading from the object's fields.

Note: If the constructor generates an exception the **this** pointer in the corresponding catch block is still uninitialized.

1.8.1.5 Delegate Constructors

The verification algorithm shall require that one of the following code sequences is used for constructing delegates; no other code sequence in verifiable code shall contain a **newobj** instruction for a delegate type. There shall be only one instance constructor method for a Delegate (overloading is not allowed)

The verification algorithm shall fail if a branch target is within these instruction sequences (other than at the start of the sequence).

Note: See Partition II for the signature of delegates and a validity requirement regarding the signature of the method used in the constructor and the signature of Invoke and other methods on the delegate class.

1.8.1.5.1 Delegating via Virtual Dispatch

The following CIL instruction sequence shall be used or the verification algorithm shall fail. The sequence begins with an object on the stack.

```
dup
ldvirtftn mthd    ; Method shall be on the class of the object,
          ; or one of its parent classes, or an interface
          ; implemented by the object
newobj delegateclass::.ctor(object, native int)
```

Rationale: *The* ***dup*** *is required to ensure that it is precisely the same object stored in the delegate as was used to compute the virtual method. If another object of a subtype were used the object and the method wouldn't match and could lead to memory violations.*

1.8.1.5.2 Delegating via Instance Dispatch

The following CIL instruction sequence shall be used or the verification algorithm shall fail. The sequence begins with either **null** or an object on the stack.

```
ldftn mthd              ; Method shall either be a static method or
          ; a method on the class of the object on the stack or
          ; one of the object's parent classes
newobj delegateclass::.ctor(object, native int)
```

1.9 Metadata Tokens

Many CIL instructions are followed by a "metadata token". This is a 4-byte value, that specifies a row in a metadata table, or a starting byte offset in the User String heap. The most-significant byte of the token specifies the table or heap. For example, a value of 0x02 specifies the TypeDef table; a value of 0x70 specifies the User String heap. The value corresponds to the number assigned to that metadata table (see Partition II for the full list of tables) or to 0x70 for the User String heap. The least-significant 3 bytes specify the target row within that metadata table, or starting byte offset within the User String heap. The rows within metadata tables are numbered one upwards, whilst offsets in the heap are numbered zero upwards. (So, for example, the metadata token with value 0x02000007 specifies row number 7 in the TypeDef table)

1.10 Exceptions Thrown

A CIL instruction can throw a range of exceptions. The CLI can also throw the general purpose exception called `ExecutionEngineException`. See Partition I for details.

2 Prefixes to Instructions

These special values are reserved to precede specific instructions. They do not constitute full instructions in their own right. It is not valid CIL to branch to the instruction following the prefix, but the prefix itself is a valid branch target. It is not valid CIL to have a prefix without immediately following it by one of the instructions it is permitted to precede.

2.1 tail. (prefix) – call terminates current method

Format	Assembly Format	Description
FE 14	tail.	Subsequent call terminates current method

Description:

The `tail.` instruction must immediately precede a `call`, `calli`, or `callvirt` instruction. It indicates that the current method's stack frame is no longer required and thus can be removed before the call instruction is executed. Because the value returned by the call will be the value returned by this method, the call can be converted into a cross-method jump.

The evaluation stack must be empty except for the arguments being transferred by the following call. The instruction following the call instruction must be a `ret`. Thus the only legal code sequence is

`tail. call` (or `calli` or `callvirt`) *somewhere*
`ret`

Correct CIL must not branch to the `call` instruction, but it is permitted to branch to the `ret`. The only values on the stack must be the arguments for the method being called.

The `tail.call` (or `calli` or `callvirt`) instruction cannot be used to transfer control out of a try, filter, catch, or finally block. See Partition I.

The current frame cannot be discarded when control is transferred from untrusted code to trusted code, since this would jeopardize code identity security. Security checks may therefore cause the `tail.` to be ignored, leaving a standard call instruction.

Similarly, in order to allow the exit of a synchronized region to occur after the call returns, the `tail.` prefix is ignored when used to exit a method that is marked synchronized.

There may also be implementation-specific restrictions that prevent the `tail.` prefix from being obeyed in certain cases. While an implementation is free to ignore the `tail.` prefix under these circumstances, they should be clearly documented as they can affect the behavior of programs.

CLI implementations are required to honor `tail. call` requests where caller and callee methods can be statically determined to lie in the same assembly; and where the caller is not in a synchronized region; and where caller and callee satisfy all conditions listed in the "Verifiability" rules below. (To "honor" the `tail.` prefix means to remove the caller's frame, rather than revert to a regular call sequence). Consequently, a CLI implementation need not honor `tail. calli` or `tail. callvirt` sequences.

Rationale: *tail. calls allow some linear space algorithms to be converted to constant space algorithms and are required by some languages. In the presence of* `ldloca` *and* `ldarga` *instructions it isn't always possible for a compiler from CIL to native code to optimally determine when a* `tail.` *can be automatically inserted.*

Exceptions:

None.

Verifiability:

Correct CIL obeys the control transfer constraints listed above. In addition, no managed pointers can be passed to the method being called if they point into the stack frame that is about to be removed. The return type of the method being called must be compatible with the return type of the current method. Verification requires that no managed pointers are passed to the method being called, since it does not track pointers into the current frame.

2.2 unaligned. (prefix) – pointer instruction may be unaligned

Format	Assembly Format	Description
FE 12 <**unsigned int8**>	unaligned. *alignment*	Subsequent pointer instruction may be unaligned

Stack Transition:

`..., addr` → `..., addr`

Description:

`Unaligned.` specifies that *address* (an unmanaged pointer (**&**), or **`native int)`** on the stack may not be aligned to the natural size of the immediately following **`ldind`**, **`stind`**, **`ldfld`**, **`stfld`**, **`ldobj`**, **`stobj`**, **`initblk`**, or **`cpblk`** instruction. That is, for a **`ldind.i4`** instruction the alignment of *addr* may not be to a 4-byte boundary. For **`initblk`** and **`cpblk`** the default alignment is architecture dependent (4-byte on 32-bit CPUs, 8-byte on 64-bit CPUs). Code generators that do not restrict their output to a 32-bit word size (see Partition I and Partition II) must use **`unaligned.`** if the alignment is not known at compile time to be 8-byte.

The value of *alignment* shall be 1, 2, or 4 and means that the generated code should assume that *addr* is byte, double byte, or quad byte aligned, respectively.

Rationale: *While the alignment for a **`cpblk`** instruction would logically require two numbers (one for the source and one for the destination), there is no noticeable impact on performance if only the lower number is specified.*

The **`unaligned.`** and **`volatile.`** prefixes may be combined in either order. They must immediately precede a **`ldind`**, **`stind`**, **`ldfld`**, **`stfld`**, **`ldobj`**, **`stobj`**, **`initblk`**, or **`cpblk`** instruction. Only the **`volatile.`** prefix is allowed for the **`ldsfld`** and **`stsfld`** instructions.

Note: See Partition I, 12.7 for information about atomicity and data alignment.

Exceptions:

None.

Verifiability:

An **`unaligned.`** prefix shall be immediately followed by one of the instructions listed above.

2.3 volatile. (prefix) - pointer reference is volatile

Format	Assembly Format	Description
FE 13	volatile.	Subsequent pointer reference is volatile

Stack Transition:

`..., addr` → `..., addr`

Description:

`volatile.` specifies that *addr* is a volatile address (i.e. it may be referenced externally to the current thread of execution) and the results of reading that location cannot be cached or that multiple stores to that location cannot be suppressed. Marking an access as **`volatile.`** affects only that single access; other accesses to the same location must be marked separately. Access to volatile locations need not be performed atomically. [see Partition I]

The **`unaligned.`** and **`volatile.`** prefixes may be combined in either order. They must immediately precede a **`ldind`**, **`stind`**, **`ldfld`**, **`stfld`**, **`ldobj`**, **`stobj`**, **`initblk`**, or **`cpblk`** instruction. Only the **`volatile.`** prefix is allowed for the **`ldsfld`** and **`stsfld`** instructions.

Exceptions:

None.

Verifiability:

A **`volatile.`** prefix should be immediately followed by one of the instructions listed above.

3 Base Instructions

These instructions form a “Turing Complete” set of basic operations. They are independent of the object model that may be employed. Operations that are specifically related to the CTS’s object model are contained in the Object Model Instructions section.

3.1 add - add numeric values

Format	Assembly Format	Description
58	add	Add two values, returning a new value

Stack Transition:

…, value1, value2 → …, result

Description:

The **add** instruction adds *value2* to *value1* and pushes the result on the stack. Overflow is not detected for integral operations (but see **add.ovf**); floating-point overflow returns **+inf** or **-inf**.

The acceptable operand types and their corresponding result data type is encapsulated in Table 2: Binary Numeric Operations.

Exceptions:

None.

Verifiability:

See Table 2: Binary Numeric Operations.

3.2 add.ovf.<signed> - add integer values with overflow check

Format	Assembly Format	Description
D6	add.ovf	Add signed integer values with overflow check.
D7	add.ovf.un	Add unsigned integer values with overflow check.

Stack Transition:

`…, value1, value2` → `…, result`

Description:

The **`add.ovf`** instruction adds *value1* and *value2* and pushes the result on the stack. The acceptable operand types and their corresponding result data type is encapsulated in Table 7: Overflow Arithmetic Operations.

Exceptions:

`OverflowException` is thrown if the result cannot be represented in the result type.

Verifiability:

See Table 7: Overflow Arithmetic Operations.

3.3 and - bitwise AND

Format	Instruction	Description
5F	And	Bitwise AND of two integral values, returns an integral value

Stack Transition:

`…, value1, value2 → …, result`

Description:

The **and** instruction computes the bitwise AND of *value1* and *value2* and pushes the result on the stack. The acceptable operand types and their corresponding result data type is encapsulated in
Table 5: Integer Operations.

Exceptions:

None.

Verifiability:

See Table 5: Integer Operations.

3.4 arglist - get argument list

Format	Assembly Format	Description
FE 00	arglist	Return argument list handle for the current method

Stack Transition:

`… → …, argListHandle`

Description:

The **`arglist`** instruction returns an opaque handle (an unmanaged pointer, type **`native int`**) representing the argument list of the current method. This handle is valid only during the lifetime of the current method. The handle can, however, be passed to other methods as long as the current method is on the thread of control. The **`arglist`** instruction may only be executed within a method that takes a variable number of arguments.

Rationale: *This instruction is needed to implement the C 'va_*' macros used to implement procedures like 'printf'. It is intended for use with the class library implementation of* `System.ArgIterator`.

Exceptions:

None.

Verifiability:

It is incorrect CIL generation to emit this instruction except in the body of a method whose signature indicates it accepts a variable number of arguments. Within such a method its use is verifiable, but verification requires that the result is an instance of the `System.RuntimeArgumentHandle` class.

3.5 beq.<length> – branch on equal

Format	Assembly Format	Description
3B <**int32**>	beq *target*	Branch to *target* if equal
2E <**int8**>	beq.s *target*	Branch to *target* if equal, short form

Stack Transition:

…, value1, value2 → …

Description:

The **beq** instruction transfers control to *target* if *value1* is equal to *value2*. The effect is identical to performing a **ceq** instruction followed by a **brtrue** ***target***. *target* is represented as a signed offset (4 bytes for **beq**, 1 byte for **beq.s**) from the beginning of the instruction following the current instruction.

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

If the target instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **leave** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL must observe all of the control transfer rules specified above and must guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See Section 1.5 for more details.

3.6 bge.<length> – branch on greater than or equal to

Format	Assembly Format	Description
3C <**int32**>	bge *target*	Branch to *target* if greater than or equal to
2F <**int8**>	bge.s *target*	Branch to *target* if greater than or equal to, short form

Stack Transition:

…, value1, value2 → …

Description:

The **bge** instruction transfers control to *target* if *value1* is greater than or equal to *value2*. The effect is identical to performing a **clt.un** instruction followed by a **brfalse** ***target***. *target* is represented as a signed offset (4 bytes for **bge**, 1 byte for **bge.s**) from the beginning of the instruction following the current instruction.

The effect of a "**bge** *target*" instruction is identical to:

- If stack operands are integers, then : **clt** followed by a **brfalse** ***target***
- If stack operands are floating-point, then : **clt.un** followed by a **brfalse** ***target***

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

If the target instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **leave** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL must observe all of the control transfer rules specified above and must guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See Section 1.5 for more details.

3.7 bge.un.<length> – branch on greater/equal, unsigned or unordered

Format	Assembly Format	Description
41 <**int32**>	bge.un *target*	Branch to *target* if greater than or equal to (unsigned or unordered)
34 <**int8**>	bge.un.s *target*	Branch to *target* if greater than or equal to (unsigned or unordered), short form

Stack Transition:

…, value1, value2 → …

Description:

The **bge.un** instruction transfers control to *target* if *value1* is greater than or equal to *value2,* when compared unsigned (for integer values) or unordered (for float point values). The effect is identical to performing a **clt** instruction followed by a **brfalse** ***target***. *target* is represented as a signed offset (4 bytes for **bge.un**, 1 byte for **bge.un.s**) from the beginning of the instruction following the current instruction.

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

If the target instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **leave** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL must observe all of the control transfer rules specified above and must guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See Section 1.5 for more details.

3.8 bgt.<length> – branch on greater than

Format	Assembly Format	Description
3D <**int32**>	bgt *target*	Branch to *target* if greater than
30 <**int8**>	bgt.s *target*	Branch to *target* if greater than, short form

Stack Transition:

…, value1, value2 → …

Description:

The **`bgt`** instruction transfers control to *target* if *value1* is greater than *value2*. The effect is identical to performing a **`cgt`** instruction followed by a **`brtrue`** ***target***. *target* is represented as a signed offset (4 bytes for **`bgt`**, 1 byte for **`bgt.s`**) from the beginning of the instruction following the current instruction.

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

If the target instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Control transfers into and out of **`try`**, **`catch`**, **`filter`**, and **`finally`** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **`leave`** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL must observe all of the control transfer rules specified above and must guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See Section 1.5 for more details.

3.9 bgt.un.<length> – branch on greater than, unsigned or unordered

Format	Assembly Format	Description
42 <**int32**>	bgt.un *target*	Branch to *target* if greater than (unsigned or unordered)
35 <**int8**>	bgt.un.s *target*	Branch to *target* if greater than (unsigned or unordered), short form

Stack Transition:

…, value1, value2 → …

Description:

The **`bgt.un`** instruction transfers control to *target* if *value1* is greater than *value2,* when compared unsigned (for integer values) or unordered (for float point values). The effect is identical to performing a **`cgt.un`** instruction followed by a **`brtrue`** ***target***. *target* is represented as a signed offset (4 bytes for **`bgt.un`**, 1 byte for **`bgt.un.s`**) from the beginning of the instruction following the current instruction.

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

If the target instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Control transfers into and out of **`try`**, **`catch`**, **`filter`**, and **`finally`** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **`leave`** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL must observe all of the control transfer rules specified above and must guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See Section 1.5 for more details.

3.10 ble.<length> – branch on less than or equal to

Format	Assembly Format	Description
3E <**int32**>	ble *target*	Branch to *target* if less than or equal to
31 <**int8**>	ble.s *target*	Branch to *target* if less than or equal to, short form

Stack Transition:

…, value1, value2 → …

Description:

The **ble** instruction transfers control to *target* if *value1* is less than or equal to *value2*. *target* is represented as a signed offset (4 bytes for **ble**, 1 byte for **ble.s**) from the beginning of the instruction following the current instruction.

The effect of a "**ble** *target*" instruction is identical to:

- If stack operands are integers, then : **cgt** followed by a **brfalse** ***target***
- If stack operands are floating-point, then : **cgt.un** followed by a **brfalse** ***target***

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

If the target instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **leave** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL must observe all of the control transfer rules specified above and must guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See Section 1.5 for more details.

3.11 ble.un.<length> – branch on less/equal, unsigned or unordered

Format	Assembly Format	Description
43 <**int32**>	ble.un *target*	Branch to *target* if less than or equal to (unsigned or unordered)
36 <**int8**>	ble.un.s *target*	Branch to *target* if less than or equal to (unsigned or unordered), short form

Stack Transition:

…, value1, value2 → …

Description:

The **ble.un** instruction transfers control to *target* if *value1* is less than or equal to *value2,* when compared unsigned (for integer values) or unordered (for float point values). *target* is represented as a signed offset (4 bytes for **ble.un**, 1 byte for **ble.un.s**) from the beginning of the instruction following the current instruction.

The effect of a "**ble.un** *target*" instruction is identical to:

- If stack operands are integers, then : **cgt.un** followed by a **brfalse** ***target***
- If stack operands are floating-point, then : **cgt** followed by a **brfalse** ***target***

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

If the target instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **leave** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL must observe all of the control transfer rules specified above and must guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See Section 1.5 for more details.

3.12 blt.<length> – branch on less than

Format	Assembly Format	Description
3F <**int32**>	blt *target*	Branch to *target* if less than
32 <**int8**>	blt.s *target*	Branch to *target* if less than, short form

Stack Transition:

…, value1, value2 → …

Description:

The **`blt`** instruction transfers control to *target* if *value1* is less than *value2*. The effect is identical to performing a **`clt`** instruction followed by a **`brtrue`** ***target***. *target* is represented as a signed offset (4 bytes for **`blt`**, 1 byte for **`blt.s`**) from the beginning of the instruction following the current instruction.

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

If the target instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Control transfers into and out of **`try`**, **`catch`**, **`filter`**, and **`finally`** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **`leave`** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL must observe all of the control transfer rules specified above and must guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See Section 1.5 for more details.

3.13 blt.un.<length> – branch on less than, unsigned or unordered

Format	Assembly Format	Description
44 <**int32**>	blt.un *target*	Branch to *target* if less than (unsigned or unordered)
37 <**int8**>	blt.un.s *target*	Branch to *target* if less than (unsigned or unordered), short form

Stack Transition:

…, value1, value2 → …

Description:

The **`blt.un`** instruction transfers control to *target* if *value1* is less than *value2,* when compared unsigned (for integer values) or unordered (for float point values). The effect is identical to performing a **`clt.un`** instruction followed by a **`brtrue`** ***target***. *target* is represented as a signed offset (4 bytes for **`blt.un`**, 1 byte for **`blt.un.s`**) from the beginning of the instruction following the current instruction.

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

If the target instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Control transfers into and out of **`try`**, **`catch`**, **`filter`**, and **`finally`** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **`leave`** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL must observe all of the control transfer rules specified above and must guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See Section 1.5 for more details.

3.14 bne.un<length> – branch on not equal or unordered

Format	Assembly Format	Description
40 <**int32**>	bne.un *target*	Branch to *target* if unequal or unordered
33 <**int8**>	bne.un.s *target*	Branch to *target* if unequal or unordered, short form

Stack Transition:

…, value1, value2 → …

Description:

The `bne.un` instruction transfers control to *target* if *value1* is not equal to *value2,* when compared unsigned (for integer values) or unordered (for float point values). The effect is identical to performing a `ceq` instruction followed by a `brfalse` ***target***. *target* is represented as a signed offset (4 bytes for `bne.un`, 1 byte for `bne.un.s`) from the beginning of the instruction following the current instruction.

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

If the target instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Control transfers into and out of `try`, `catch`, `filter`, and `finally` blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the `leave` instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL must observe all of the control transfer rules specified above and must guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See Section 1.5 for more details.

3.15 br.<length> – unconditional branch

Format	Assembly Format	Description
38 <**int32**>	br *target*	Branch to *target*
2B <**int8**>	br.s *target*	Branch to *target*, short form

Stack Transition:

…, → …

Description:

The **br** instruction unconditionally transfers control to *target*. *target* is represented as a signed offset (4 bytes for **br**, 1 byte for **br.s**) from the beginning of the instruction following the current instruction.

If the target instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **leave** instruction instead; see Partition I for details).

Rationale: *While a* ***leave*** *instruction can be used instead of a* ***br*** *instruction when the evaluation stack is empty, doing so may increase the resources required to compile from CIL to native code and/or lead to inferior native code. Therefore CIL generators should use a* ***br*** *instruction in preference to a* ***leave*** *instruction when both are legal.*

Exceptions:

None.

Verifiability:

Correct CIL must observe all of the control transfer rules specified above.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See Section 1.5 for more details.

3.16 break – breakpoint instruction

Format	Assembly Format	Description
01	break	Inform a debugger that a breakpoint has been reached.

Stack Transition:

…, $\rightarrow$ …

Description:

The **break** instruction is for debugging support. It signals the CLI to inform the debugger that a break point has been tripped. It has no other effect on the interpreter state.

The **break** instruction has the smallest possible instruction size so that code can be patched with a breakpoint with minimal disturbance to the surrounding code.

The **break** instruction may trap to a debugger, do nothing, or raise a security exception: the exact behavior is implementation-defined

Exceptions:

None.

Verifiability:

The **break** instruction is always verifiable.

3.17 brfalse.<length> - branch on false, null, or zero

Format	Assembly Format	Description
39 <**int32**>	brfalse *target*	Branch to *target* if *value* is zero (false)
2C <**int8**>	brfalse.s *target*	Branch to *target* if *value* is zero (false), short form
39 <**int32**>	brnull *target*	Branch to *target* if *value* is null (*alias for* **`brfalse`**)
2C <**int8**>	brnull.s *target*	Branch to *target* if *value* is null (*alias for* **`brfalse.s`**), short form
39 <**int32**>	brzero *target*	Branch to *target* if *value* is zero (*alias for* **`brfalse`**)
2C <**int8**>	brzero.s *target*	Branch to *target* if *value* is zero (*alias for* **`brfalse.s`**), short form

Stack Transition:

…, value → …

Description:

The **`brfalse`** instruction transfers control to *target* if *value* (of type **`int32, int64, object reference, managed pointer, unmanaged pointer`** `or` **`native int`**) is zero (false). If *value* is non-zero (true) execution continues at the next instruction.

Target is represented as a signed offset (4 bytes for **`brfalse`**, 1 byte for **`brfalse.s`**) from the beginning of the instruction following the current instruction.

If the target instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Control transfers into and out of **`try`**, **`catch`**, **`filter`**, and **`finally`** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **`leave`** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL must observe all of the control transfer rules specified above and must guarantee there is a minimum of one item on the stack.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See Section 1.5 for more details.

3.18 brtrue.<length> - branch on non-false or non-null

Format	Assembly Format	Description
3A <**int32**>	brtrue *target*	Branch to *target* if *value* is non-zero (true)
2D <**int8**>	brtrue.s *target*	Branch to *target* if *value* is non-zero (true), short form
3A <**int32**>	brinst *target*	Branch to *target* if *value* is a non-null object reference (alias for **`brtrue`**)
2D <**int8**>	brinst.s *target*	Branch to *target* if *value* is a non-null object reference, short form (*alias for* **`brtrue.s`**)

Stack Transition:

`…, value` → `…`

Description:

The **`brtrue`** instruction transfers control to *target* if *value* (of type **`native int`**) is nonzero (true). If *value* is zero (false) execution continues at the next instruction.

If the *value* is an object reference (type **`O`**) then **`brinst`** (an alias for **`brtrue`**) transfers control if it represents an instance of an object (i.e. isn't the null object reference, see **`ldnull`**).

Target is represented as a signed offset (4 bytes for **`brtrue`**, 1 byte for **`brtrue.s`**) from the beginning of the instruction following the current instruction.

If the target instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Control transfers into and out of **`try`**, **`catch`**, **`filter`**, and **`finally`** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **`leave`** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL must observe all of the control transfer rules specified above and must guarantee there is a minimum of one item on the stack.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See Section 1.5 for more details.

3.19 call – call a method

Format	Assembly Format	Description
28 <T>	call *method*	Call method described by *method*

Stack Transition:

…, arg1, arg2 … argn → …, retVal (not always returned)

Description:

The **call** instruction calls the method indicated by the descriptor *method*. *Method* is a metadata token (either a **methodref** or **methoddef** See Partition II) that indicates the method to call and the number, type, and order of the arguments that have been placed on the stack to be passed to that method as well as the calling convention to be used. See Partition I for a detailed description of the CIL calling sequence. The **call** instruction may be immediately preceded by a **tail.** prefix to specify that the current method state should be released before transferring control (see Section 2.1).

The metadata token carries sufficient information to determine whether the call is to a static method, an instance method, a virtual method, or a global function. In all of these cases the destination address is determined entirely from the metadata token (Contrast with the **callvirt** instruction for calling virtual methods, where the destination address also depends upon the runtime type of the instance reference pushed before the **callvirt**; see below).

If the method does not exist in the class specified by the metadata token, the base classes are searched to find the most derived class which defines the method and that method is called.

Rationale: *This implements "call superclass" behavior.*

The arguments are placed on the stack in left-to-right order. That is, the first argument is computed and placed on the stack, then the second argument, etc. There are three important special cases:

1. Calls to an instance (or virtual, see below) method must push that instance reference (the **this** pointer) before any of the user-visible arguments. The signature carried in the metadata does not contain an entry in the parameter list for the **this** pointer but uses a bit (called HASTHIS) to indicate whether the method requires passing the **this** pointer (see Partition II)
2. It is legal to call a virtual method using **call** (rather than **callvirt**); this indicates that the method is to be resolved using the class specified by *method* rather than as specified dynamically from the object being invoked. This is used, for example, to compile calls to "methods on **super**" (i.e. the statically known parent class).
3. Note that a delegate's **Invoke** method may be called with either the **call** or **callvirt** instruction.

Exceptions:

`SecurityException` may be thrown if system security does not grant the caller access to the called method. The security check may occur when the CIL is converted to native code rather than at runtime.

Verifiability:

Correct CIL ensures that the stack contains the correct number and type of arguments for the method being called.

For a typical use of the **call** instruction, verification checks that (a) *method* refers to a valid **methodref** or **methoddef** token; (b) the types of the objects on the stack are consistent with the types expected by the method call, and (c) the method is accessible from the callsite, and (d) the method is not abstract (ie, it has an implementation)

The **call** instruction may also be used to call an object's superclass constructor, or to initialize a value type location by calling an appropriate constructor, both of which are treated as special cases by verification. A **call** annotated by **tail.** is also a special case.

If the target method is global (defined outside of any type), then the method must be static.

3.20 calli– indirect method call

Format	Assembly Format	Description
29 <T>	calli *callsitedescr*	Call method indicated on the stack with arguments described by *callsitedescr*.

Stack Transition:

`…, arg1, arg2 … argn, ftn` → `…, retVal (not always returned)`

Description:

The **`calli`** instruction calls *ftn* (a pointer to a method entry point) with the arguments **`arg1`** … **`argn`**. The types of these arguments are described by the signature **`callsitedescr`**. See Partition I for a description of the CIL calling sequence. The **`calli`** instruction may be immediately preceded by a **`tail.`** prefix to specify that the current method state should be released before transferring control. If the call would transfer control to a method of higher trust than the origin method the stack frame will not be released; instead, the execution will continue silently as if the **`tail.`** prefix had not been supplied.

[A callee of "higher trust" is defined as one whose permission grant-set is a strict superset of the grant-set of the caller.]

The *ftn* argument is assumed to be a pointer to native code (of the target machine) that can be legitimately called with the arguments described by *callsitedescr* (a metadata token for a stand-alone signature). Such a pointer can be created using the **`ldftn`** or **`ldvirtftn`** instructions, or have been passed in from native code.

The standalone signature specifies the number and type of parameters being passed, as well as the calling convention (See Partition II) The calling convention is not checked dynamically, so code that uses a **`calli`** instruction will not work correctly if the destination does not actually use the specified calling convention.

The arguments are placed on the stack in left-to-right order. That is, the first argument is computed and placed on the stack, then the second argument, etc. The argument-building code sequence for an instance or virtual method must push that instance reference (the **`this`** pointer, which must not be null) before any of the user-visible arguments.

Exceptions:

`SecurityException` may be thrown if the system security does not grant the caller access to the called method. The security check may occur when the CIL is converted to native code rather than at runtime.

Verifiability:

Correct CIL requires that the function pointer contains the address of a method whose signature matches that specified by *callsitedescr* and that the arguments correctly correspond to the types of the destination function's parameters.

Verification checks that *ftn* is a pointer to a function generated by **`ldftn`** or **`ldvirtfn`**.

3.21 ceq - compare equal

Format	Assembly Format	Description
FE 01	ceq	Push 1 (of type **int32**) if *value*1 equals *value*2, else 0

Stack Transition:

`…, value1, value2` → `…, result`

Description:

The **`ceq`** instruction compares *value*1 and *value*2. If *value*1 is equal to *value*2, then 1 (of type **`int32`**) is pushed on the stack. Otherwise 0 (of type **`int32`**) is pushed on the stack.

For floating-point numbers, **`ceq`** will return 0 if the numbers are unordered (either or both are NaN). The infinite values are equal to themselves.

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

Exceptions:

None.

Verifiability:

Correct CIL provides two values on the stack whose types match those specified in Table 4: Binary Comparison or Branch Operations. There are no additional verification requirements.

3.22 cgt - compare greater than

Format	Assembly Format	Description
FE 02	cgt	Push 1 (of type **`int32`**) if *value*1 > *value*2, else 0

Stack Transition:

…, value1, value2 → …, result

Description:

The **`cgt`** instruction compares *value1* and *value2*. If *value1* is strictly greater than *value2*, then 1 (of type **`int32`**) is pushed on the stack. Otherwise 0 (of type **`int32`**) is pushed on the stack

For floating-point numbers, **`cgt`** returns 0 if the numbers are unordered (that is, if one or both of the arguments are NaN).

As per IEC 60559:1989, infinite values are ordered with respect to normal numbers (e.g. +infinity > 5.0 > -infinity).

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

Exceptions:

None.

Verifiability:

Correct CIL provides two values on the stack whose types match those specified in Table 4: Binary Comparison or Branch Operations. There are no additional verification requirements.

3.23 cgt.un - compare greater than, unsigned or unordered

Format	Assembly Format	Description
FE 03	cgt.un	Push 1 (of type **int32**) if *value*1 > *value*2, unsigned or unordered, else 0

Stack Transition:

…, value1, value2 → …, result

Description:

The **cgt.un** instruction compares *value1* and *value2*. A value of 1 (of type **int32**) is pushed on the stack if

- for floating-point numbers, either *value1* is strictly greater than *value2*, or *value1* is not ordered with respect to *value2*
- for integer values, *value1* is strictly greater than *value2* when considered as unsigned numbers

Otherwise 0 (of type **int32**) is pushed on the stack.

As per IEC 60559:1989, infinite values are ordered with respect to normal numbers (e.g. +infinity > 5.0 > -infinity).

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

Exceptions:

None.

Verifiability:

Correct CIL provides two values on the stack whose types match those specified in Table 4: Binary Comparison or Branch Operations. There are no additional verification requirements.

3.24 ckfinite – check for a finite real number

Format	Assembly Format	Description
C3	ckfinite	Throw `ArithmeticException` if value is not a finite number

Stack Transition:

`…, value` → `…, value`

Description:

The **`ckfinite`** instruction throws `ArithmeticException` if *value* (a floating-point number) is either a “not a number” value (NaN) or +/- infinity value. **`ckfinite`** leaves the value on the stack if no exception is thrown. Execution is unspecified if *value* is not a floating-point number.

Exceptions:

`ArithmeticException` is thrown if *value* is not a ‘normal’ number.

Verifiability:

Correct CIL guarantees that *value* is a floating-point number. There are no additional verification requirements.

3.25 clt - compare less than

Format	Assembly Format	Description
FE 04	clt	Push 1 (of type `int32`) if *value*1 < *value*2, else 0

Stack Transition:

`…, value1, value2` → `…, result`

Description:

The `clt` instruction compares *value1* and *value2*. If *value1* is strictly less than *value2*, then 1 (of type `int32`) is pushed on the stack. Otherwise 0 (of type `int32`) is pushed on the stack

For floating-point numbers, `clt` will return 0 if the numbers are unordered (that is, one or both of the arguments are NaN).

As per IEC 60559:1989, infinite values are ordered with respect to normal numbers (e.g. +infinity > 5.0 > -infinity).

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

Exceptions:

None.

Verifiability:

Correct CIL provides two values on the stack whose types match those specified in Table 4: Binary Comparison or Branch Operations. There are no additional verification requirements.

3.26 clt.un - compare less than, unsigned or unordered

Format	Assembly Format	Description
FE 05	clt.un	Push 1 (of type `int32`) if *value*1 < *value2*, unsigned or unordered, else 0

Stack Transition:

…, value1, value2 → …, result

Description:

The `clt.un` instruction compares *value1* and *value2*. A value of 1 (of type `int32`) is pushed on the stack if

- for floating-point numbers, either *value1* is strictly less than *value2*, or *value1* is not ordered with respect to *value2*
- for integer values, *value1* is strictly less than *value2* when considered as unsigned numbers

Otherwise 0 (of type `int32`) is pushed on the stack.

As per IEC 60559:1989, infinite values are ordered with respect to normal numbers (e.g. +infinity > 5.0 > -infinity).

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

Exceptions:

None.

Verifiability:

Correct CIL provides two values on the stack whose types match those specified in Table 4: Binary Comparison or Branch Operations. There are no additional verification requirements.

3.27 conv.<to type> - data conversion

Format	Assembly Format	Description
67	conv.i1	Convert to **int8**, pushing **int32** on stack
68	conv.i2	Convert to **int16**, pushing **int32** on stack
69	conv.i4	Convert to **int32**, pushing **int32** on stack
6A	conv.i8	Convert to **int64**, pushing **int64** on stack
6B	conv.r4	Convert to **float32**, pushing **F** on stack
6C	conv.r8	Convert to **float64**, pushing **F** on stack
D2	conv.u1	Convert to **unsigned int8**, pushing **int32** on stack
D1	conv.u2	Convert to **unsigned int16**, pushing **int32** on stack
6D	conv.u4	Convert to **unsigned int32**, pushing **int32** on stack
6E	conv.u8	Convert to **unsigned int64**, pushing **int64** on stack
D3	conv.i	Convert to **native int**, pushing **native int** on stack
E0	conv.u	Convert to **native unsigned int**, pushing **native int** on stack
76	conv.r.un	Convert unsigned integer to floating-point, pushing **F** on stack

Stack Transition:

…, value → …, result

Description:

Convert the value on top of the stack to the type specified in the opcode, and leave that converted value on the top of the stack. Note that integer values of less than 4 bytes are extended to **int32** (not **native int**) when they are loaded onto the evaluation stack, and floating-point values are converted to the **F** type.

Conversion from floating-point numbers to integral values truncates the number toward zero. When converting from a **float64** to a **float32**, precision may be lost. If *value* is too large to fit in a **float32**, the IEC 60559:1989 positive infinity (if *value* is positive) or IEC 60559:1989 negative infinity (if *value* is negative) is returned. If overflow occurs converting one integer type to another the high-order bits are silently truncated. If the result is smaller than an **int32**, then the value is sign-extended to fill the slot.

If overflow occurs converting a floating-point type to an integer the value returned is unspecified. The **conv.r.un** operation takes an integer off the stack, interprets it as unsigned, and replaces it with a floating-point number to represent the integer; either a **float32**, if this is wide enough to represent the integer without loss of precision, else a **float64**.

No exceptions are ever thrown. See **conv.ovf** for instructions that will throw an exception when the result type cannot properly represent the result value.

The acceptable operand types and their corresponding result data type is encapsulated in Table 8: Conversion Operations.

Exceptions:

None.

Verifiability:

Correct CIL has at least one value, of a type specified in Table 8: Conversion Operations, on the stack. The same table specifies a restricted set of types that are acceptable in verified code.

3.28 conv.ovf.<to type> - data conversion with overflow detection

Format	Assembly Format	Description
B3	conv.ovf.i1	Convert to an `int8` (on the stack as `int32`) and throw an exception on overflow
B5	conv.ovf.i2	Convert to an `int16` (on the stack as `int32`) and throw an exception on overflow
B7	conv.ovf.i4	Convert to an `int32` (on the stack as `int32`) and throw an exception on overflow
B9	conv.ovf.i8	Convert to an `int64` (on the stack as `int64`) and throw an exception on overflow
B4	conv.ovf.u1	Convert to a `unsigned int8` (on the stack as `int32`) and throw an exception on overflow
B6	conv.ovf.u2	Convert to a `unsigned int16` (on the stack as `int32`) and throw an exception on overflow
B8	conv.ovf.u4	Convert to a `unsigned int32` (on the stack as `int32`) and throw an exception on overflow
BA	conv.ovf.u8	Convert to a `unsigned int64` (on the stack as `int64`) and throw an exception on overflow
D4	conv.ovf.i	Convert to an `native int` (on the stack as `native int`) and throw an exception on overflow
D5	conv.ovf.u	Convert to a `native unsigned int` (on the stack as `native int`) and throw an exception on overflow

Stack Transition:

`…, value` → `…, result`

Description:

Convert the value on top of the stack to the type specified in the opcode, and leave that converted value on the top of the stack. If the value is too large or too small to be represented by the target type, an exception is thrown.

Conversions from floating-point numbers to integral values truncate the number toward zero. Note that integer values of less than 4 bytes are extended to `int32` (not `native int`) on the evaluation stack.

The acceptable operand types and their corresponding result data type is encapsulated in Table 8: Conversion Operations.

Exceptions:

`OverflowException` is thrown if the result can not be represented in the result type

Verifiability:

Correct CIL has at least one value, of a type specified in Table 8: Conversion Operations, on the stack. The same table specifies a restricted set of types that are acceptable in verified code.

3.29 conv.ovf.<to type>.un – unsigned data conversion with overflow detection

Format	Assembly Format	Description
82	conv.ovf.i1.un	Convert unsigned to an `int8` (on the stack as `int32`) and throw an exception on overflow
83	conv.ovf.i2.un	Convert unsigned to an `int16` (on the stack as `int32`) and throw an exception on overflow
84	conv.ovf.i4.un	Convert unsigned to an `int32` (on the stack as `int32`) and throw an exception on overflow
85	conv.ovf.i8.un	Convert unsigned to an `int64` (on the stack as `int64`) and throw an exception on overflow
86	conv.ovf.u1.un	Convert unsigned to an `unsigned int8` (on the stack as `int32`) and throw an exception on overflow
87	conv.ovf.u2.un	Convert unsigned to an `unsigned int16` (on the stack as `int32`) and throw an exception on overflow
88	conv.ovf.u4.un	Convert unsigned to an `unsigned int32` (on the stack as `int32`) and throw an exception on overflow
89	conv.ovf.u8.un	Convert unsigned to an `unsigned int64` (on the stack as `int64`) and throw an exception on overflow
8A	conv.ovf.i.un	Convert unsigned to a `native int` (on the stack as `native int`) and throw an exception on overflow
8B	conv.ovf.u.un	Convert unsigned to a `native unsigned int` (on the stack as `native int`) and throw an exception on overflow

Stack Transition:

…, value → …, result

Description:

Convert the value on top of the stack to the type specified in the opcode, and leave that converted value on the top of the stack. If the value cannot be represented, an exception is thrown. The item at the top of the stack is treated as an unsigned value.

Conversions from floating-point numbers to integral values truncate the number toward zero. Note that integer values of less than 4 bytes are extended to `int32` (not `native int`) on the evaluation stack.

The acceptable operand types and their corresponding result data type is encapsulated in Table 8: Conversion Operations.

Exceptions:

`OverflowException` is thrown if the result cannot be represented in the result type

Verifiability:

Correct CIL has at least one value, of a type specified in Table 8: Conversion Operations, on the stack. The same table specifies a restricted set of types that are acceptable in verified code.

3.30 cpblk - copy data from memory to memory

Format	Instruction	Description
FE 17	cpblk	Copy data from memory to memory

Stack Transition:

…, destaddr, srcaddr, size → …

Description:

The **`cpblk`** instruction copies *size* (of type **`unsigned int32`**) bytes from address *srcaddr* (of type **`native int`**, or **`&`**) to address *destaddr* (of type **`native int`**, or **`&`**). The behavior of **`cpblk`** is unspecified if the source and destination areas overlap.

`cpblk` assumes that both *destaddr* and *srcaddr* are aligned to the natural size of the machine (but see the **`unaligned.`** prefix instruction). The **`cpblk`** instruction may be immediately preceded by the **`unaligned.`** prefix instruction to indicate that either the source or the destination is unaligned.

Rationale: ***`cpblk`*** *is intended for copying structures (rather than arbitrary byte-runs). All such structures, allocated by the CLI, are naturally aligned for the current platform. Therefore, there is no need for the compiler that generates* ***`cpblk`*** *instructions to be aware of whether the code will eventually execute on a 32-bit or 64-bit platform.*

The operation of the **`cpblk`** instruction may be altered by an immediately preceding **`volatile.`** or **`unaligned.`** prefix instruction.

Exceptions:

`NullReferenceException` may be thrown if an invalid address is detected.

Verifiability:

The **`cpblk`** instruction is never verifiable. Correct CIL ensures the conditions specified above.

3.31 div - divide values

Format	Assembly Format	Description
5B	div	Divide two values to return a quotient or floating-point result

Stack Transition:

`…, value1, value2` → `…, result`

Description:

result = value1 **div** *value2* satisfies the following conditions:

|*result*| = |*value1*| / |*value2*|, and

sign(result) = +, if sign(value1) = sign(value2), or
–, if sign(value1) ~= sign(value2)

The **`div`** instruction computes *result* and pushes it on the stack.

Integer division truncates towards zero.

Floating-point division is per IEC 60559:1989. In particular division of a finite number by 0 produces the correctly signed infinite value and

`0 / 0 =` **`NaN`**

`infinity` `/` **`infinity`** `=` **`NaN`**`.`

`X /` **`infinity`** `= 0`

The acceptable operand types and their corresponding result data type is encapsulated in Table 2: Binary Numeric Operations.

Exceptions:

Integral operations throw `ArithmeticException` if the result cannot be represented in the result type. This can happen if *value1* is the smallest representable integer value, and *value2* is -1.

Integral operations throw `DivideByZeroException` if *value2* is zero.

Floating-point operations never throw an exception (they produce NaNs or infinities instead, see Partition I).

Example:

`+14` **`div`** `+3 is 4`

`+14` **`div`** `-3 is -4`

`-14` **`div`** `+3 is -4`

`-14` **`div`** `-3 is 4`

Verifiability:

See Table 2: Binary Numeric Operations.

3.32 div.un - divide integer values, unsigned

Format	Assembly Format	Description
5C	div.un	Divide two values, unsigned, returning a quotient

Stack Transition:

…, value1, value2 → …, result

Description:

The **div.un** instruction computes *value1* divided by *value2,* both taken as unsigned integers, and pushes the result on the stack.

The acceptable operand types and their corresponding result data type are encapsulated in Table 5: Integer Operations.

Exceptions:

DivideByZeroException is thrown if *value2* is zero.

Example:

```
+5 div.un +3      is 1
+5 div.un -3      is 0
-5 div.un +3      is 14316557630 or 0x55555553
-5 div.un -3      is 0
```

Verifiability:

See Table 5: Integer Operations.

3.33 dup – duplicate the top value of the stack

Format	Assembly Format	Description
25	dup	Duplicate value on the top of the stack

Stack Transition:

`…, value` $\rightarrow$ `…, value, value`

Description:

The **dup** instruction duplicates the top element of the stack.

Exceptions:

None.

Verifiability:

No additional requirements.

3.34 endfilter – end filter clause of SEH

Format	Assembly Format	Description
FE 11	Endfilter	End filter clause of SEH exception handling

Stack Transition:

…, value → …

Description:

Return from **filter** clause of an exception (see the Exception Handling section of Partition I for a discussion of exceptions). *Value* (which must be of type **int32** and is one of a specific set of values) is returned from the **filter** clause. It should be one of:

- **exception_continue_search** (0) to continue searching for an exception handler
- **exception_execute_handler** (1) to start the second phase of exception handling where finally blocks are run until the handler associated with this filter clause is located. Then the handler is executed.

Other integer values will produce unspecified results.

The entry point of a filter, as shown in the method's exception table, must be the (lexically) first instruction in the filter's code block. The **endfilter** must be the (lexically) last instruction in the filter's code block (hence there can only be one **endfilter** for any single filter block). After executing the **endfilter** instruction, control logically flows back to the CLI exception handling mechanism.

Control cannot be transferred into a **filter** block except through the exception mechanism. Control cannot be transferred out of a **filter** block except through the use of a **throw** instruction or executing the final **endfilter** instruction. In particular, it is not legal to execute a **ret** or **leave** instruction within a **filter** block. It is not legal to embed a **try** block within a **filter** block. If an exception is thrown inside the **filter** block, it is intercepted and a value of **exception_continue_search** is returned.

Exceptions:

None.

Verifiability:

Correct CIL guarantees the control transfer restrictions specified above. Also, the stack must contain exactly one item (of type **int32**).

3.35 endfinally – end the finally or fault clause of an exception block

Format	Assembly Format	Description
DC	Endfault	End fault clause of an exception block
DC	Endfinally	End finally clause of an exception block

Stack Transition:

… → …

Description:

Return from the **finally** or **fault** clause of an exception block; see the Exception Handling section of Partition I for details.

Signals the end of the **finally** or **fault** clause so that stack unwinding can continue until the exception handler is invoked. The **endfinally** or **endfault** instruction transfers control back to the CLI exception mechanism. This then searches for the next **finally** clause in the chain, if the protected block was exited with a **leave** instruction. If the protected block was exited with an exception, the CLI will search for the next **finally** or **fault**, or enter the exception handler chosen during the first pass of exception handling.

An **endfinally** instruction may only appear lexically within a **finally** block. Unlike the **endfilter** instruction, there is no requirement that the block end with an **endfinally** instruction, and there can be as many **endfinally** instructions within the block as required. These same restrictions apply to the **endfault** instruction and the **fault** block, *mutatis mutandis*.

Control cannot be transferred into a **finally** (or **fault** block) except through the exception mechanism. Control cannot be transferred out of a **finally** (or **fault)** block except through the use of a **throw** instruction or executing the **endfinally** (or **endfault**) instruction. In particular, it is not legal to "fall out" of a **finally** (or **fault**) block or to execute a **ret** or **leave** instruction within a **finally** (or **fault**) block.

Note that the **endfault** and **endfinally** instructions are aliases – they correspond to the same opcode.

Exceptions:

None.

Verifiability:

Correct CIL guarantees the control transfer restrictions specified above. There are no additional verification requirements.

3.36 initblk - initialize a block of memory to a value

Format	Assembly Format	Description
FE 18	initblk	Set a block of memory to a given byte

Stack Transition:

`…, addr, value, size` → `…`

Description:

The **`initblk`** instruction sets *size* (of type **`unsigned int32`**) bytes starting at *addr* (of type **`native int`**, or **`&`**) to *value* (of type **`unsigned int8`**). **`initblk`** assumes that *addr* is aligned to the natural size of the machine (but see the **`unaligned.`** prefix instruction).

Rationale: ***`initblk`*** *is intended for initializing structures (rather than arbitrary byte-runs). All such structures, allocated by the CLI, are naturally aligned for the current platform. Therefore, there is no need for the compiler that generates* ***`initblk`*** *instructions to be aware of whether the code will eventually execute on a 32-bit or 64-bit platform.*

The operation of the **`initblk`** instructions may be altered by an immediately preceding **`volatile.`** or **`unaligned.`** prefix instruction.

Exceptions:

`NullReferenceException` may be thrown if an invalid address is detected.

Verifiability:

The **`initblk`** instruction is never verifiable. Correct CIL code ensures the restrictions specified above.

3.37 jmp – jump to method

Format	Assembly Format	Description
27 <T>	jmp *method*	Exit current method and jump to specified method

Stack Transition:

… $\rightarrow$ …

Description:

Transfer control to the method specified by *method*, which is a metadata token (either a **`methodref`** or **`methoddef`** (See Partition II). The current arguments are transferred to the destination method.

The evaluation stack must be empty when this instruction is executed. The calling convention, number and type of arguments at the destination address must match that of the current method.

The jmp instruction cannot be used to transferred control out of a try, filter, catch, fault or finally block; or out of a synchronized region. If this is done, results are undefined. See Partition I.

Exceptions:

None.

Verifiability:

The **`jmp`** instruction is never verifiable. Correct CIL code obeys the control flow restrictions specified above.

3.38 ldarg.<length> - load argument onto the stack

Format	Assembly Format	Description
FE 09 <*unsigned int16*>	ldarg *num*	Load argument numbered *num* onto stack.
0E <*unsigned int8*>	ldarg.s *num*	Load argument numbered *num* onto stack, short form.
02	ldarg.0	Load argument 0 onto stack
03	ldarg.1	Load argument 1 onto stack
04	ldarg.2	Load argument 2 onto stack
05	ldarg.3	Load argument 3 onto stack

Stack Transition:

… $\rightarrow$ …, value

Description:

The **ldarg** *num* instruction pushes the *num*'th incoming argument, where arguments are numbered 0 onwards (see Partition I) onto the evaluation stack. The **ldarg** instruction can be used to load a value type or a built-in value onto the stack by copying it from an incoming argument. The type of the value is the same as the type of the argument, as specified by the current method's signature.

The **ldarg.0**, **ldarg.1**, **ldarg.2**, and **ldarg.3** instructions are efficient encodings for loading any of the first 4 arguments. The **ldarg.s** instruction is an efficient encoding for loading argument numbers 4 through 255.

For procedures that take a variable-length argument list, the **ldarg** instructions can be used only for the initial fixed arguments, not those in the variable part of the signature. (See the **arglist** instruction)

Arguments that hold an integer value smaller than 4 bytes long are expanded to type **int32** when they are loaded onto the stack. Floating-point values are expanded to their native size (type **F**).

Exceptions:

None.

Verifiability:

Correct CIL guarantees that *num* is a valid argument index. See Section 1.5 for more details on how verification determines the type of the value loaded onto the stack.

3.39 ldarga.<length> - load an argument address

Format	Assembly Format	Description
FE 0A <*unsigned int16*>	ldarga *argNum*	fetch the address of argument *argNum*.
0F <*unsigned int8*>	ldarga.s *argNum*	fetch the address of argument *argNum*, short form

Stack Transition:

…, → …, address of argument number *argNum*

Description:

The **`ldarga`** instruction fetches the address (of type &, i.e. managed pointer) of the *argNum*'th argument, where arguments are numbered 0 onwards. The address will always be aligned to a natural boundary on the target machine (cf. **`cpblk`** and **`initblk`**). The short form (**`ldarga.s`**) should be used for argument numbers 0 through 255.

For procedures that take a variable-length argument list, the **`ldarga`** instructions can be used only for the initial fixed arguments, not those in the variable part of the signature.

Rationale: ***`ldarga`*** *is used for by-ref parameter passing (see Partition I). In other cases,* **`ldarg`** *and* **`starg`** *should be used.*

Exceptions:

None.

Verifiability:

Correct CIL ensures that *argNum* is a valid argument index. See Section 1.5 for more details on how verification determines the type of the value loaded onto the stack.

3.40 ldc.<type> - load numeric constant

Format	Assembly Format	Description
20 <`int32`>	ldc.i4 *num*	Push *num* of type `int32` onto the stack as `int32`.
21 <`int64`>	ldc.i8 *num*	Push *num* of type `int64` onto the stack as `int64`.
22 <`float32`>	ldc.r4 *num*	Push *num* of type `float32` onto the stack as `F`.
23 <`float64`>	ldc.r8 *num*	Push *num* of type `float64` onto the stack as `F`.
16	ldc.i4.0	Push 0 onto the stack as `int32`.
17	ldc.i4.1	Push 1 onto the stack as `int32`.
18	ldc.i4.2	Push 2 onto the stack as `int32`.
19	ldc.i4.3	Push 3 onto the stack as `int32`.
1A	ldc.i4.4	Push 4 onto the stack as `int32`.
1B	ldc.i4.5	Push 5 onto the stack as `int32`.
1C	ldc.i4.6	Push 6 onto the stack as `int32`.
1D	ldc.i4.7	Push 7 onto the stack as `int32`.
1E	ldc.i4.8	Push 8 onto the stack as `int32`.
15	ldc.i4.m1	Push -1 onto the stack as `int32`.
15	ldc.i4.M1	Push -1 of type `int32` onto the stack as `int32` (alias for `ldc.i4.m1`).
1F <`int8`>	ldc.i4.s *num*	Push *num* onto the stack as `int32`, short form.

Stack Transition:

… → …, num

Description:

The `ldc` *num* instruction pushes number *num* onto the stack. There are special short encodings for the integers –128 through 127 (with especially short encodings for –1 through 8). All short encodings push 4 byte integers on the stack. Longer encodings are used for 8 byte integers and 4- and 8-byte floating-point numbers, as well as 4-byte values that do not fit in the short forms.

There are three ways to push an 8-byte integer constant onto the stack

1. use the `ldc.i8` instruction for constants that must be expressed in more than 32 bits
2. use the `ldc.i4` instruction followed by a `conv.i8` for constants that require 9 to 32 bits
3. use a short form instruction followed by a `conv.i8` for constants that can be expressed in 8 or fewer bits

There is no way to express a floating-point constant that has a larger range or greater precision than a 64 bit IEC 60559:1989 number, since these representations are not portable across architectures.

Exceptions:

None.

Verifiability:

The `ldc` instruction is always verifiable.

3.41 ldftn - load method pointer

Format	Assembly Format	Description
FE 06 <*T*>	ldftn *method*	Push a pointer to a method referenced by *method* on the stack

Stack Transition:

… → …, ftn

Description:

The **`ldftn`** instruction pushes an unmanaged pointer (type **`native int`**) to the native code implementing the method described by *method* (a metadata token, either a **`methoddef`** or **`methodref`**; see Partition II) onto the stack. The value pushed can be called using the **`calli`** instruction if it references a managed method (or a stub that transitions from managed to unmanaged code).

The value returned points to native code using the calling convention specified by *method*. Thus a method pointer can be passed to unmanaged native code (e.g. as a callback routine). Note that the address computed by this instruction may be to a thunk produced specially for this purpose (for example, to re-enter the CIL interpreter when a native version of the method isn't available).

Exceptions:

None.

Verifiability:

Correct CIL requires that *method* is a valid **`methoddef`** or **`methodref`** token. Verification tracks the type of the value pushed in more detail than the "**`native int`**" type, remembering that it is a method pointer. Such a method pointer can then be used with **`calli`** or to construct a delegate.

3.42 ldind.<type> - load value indirect onto the stack

Format	Assembly Format	Description
46	ldind.i1	Indirect load value of type **`int8`** as **`int32`** on the stack.
48	ldind.i2	Indirect load value of type **`int16`** as **`int32`** on the stack.
4A	ldind.i4	Indirect load value of type **`int32`** as **`int32`** on the stack.
4C	ldind.i8	Indirect load value of type **`int64`** as **`int64`** on the stack.
47	ldind.u1	Indirect load value of type **`unsigned int8`** as **`int32`** on the stack.
49	ldind.u2	Indirect load value of type **`unsigned int16`** as **`int32`** on the stack.
4B	ldind.u4	Indirect load value of type **`unsigned int32`** as **`int32`** on the stack.
4E	ldind.r4	Indirect load value of type **`float32`** as **`F`** on the stack.
4C	ldind.u8	Indirect load value of type **`unsigned int64`** as **`int64`** on the stack (alias for **`ldind.i8`**).
4F	ldind.r8	Indirect load value of type **`float64`** as **`F`** on the stack.
4D	ldind.i	Indirect load value of type **`native int`** as **`native int`** on the stack
50	ldind.ref	Indirect load value of type **`object ref`** as **`O`** on the stack.

Stack Transition:

`…, addr` → `…, value`

Description:

The **`ldind`** instruction indirectly loads a value from address *addr* (an unmanaged pointer, **`native int`**, or managed pointer, **`&`**) onto the stack. The source value is indicated by the instruction suffix. All of the **`ldind`** instructions are shortcuts for a **`ldobj`** instruction that specifies the corresponding built-in value class.

Note that integer values of less than 4 bytes are extended to **`int32`** (not **`native int`**) when they are loaded onto the evaluation stack. Floating-point values are converted to **`F`** type when loaded onto the evaluation stack.

Correct CIL ensures that the **`ldind`** instructions are used in a manner consistent with the type of the pointer.

The address specified by *addr* must be aligned to the natural size of objects on the machine or a `NullReferenceException` may occur (but see the **`unaligned.`** prefix instruction). The results of all CIL instructions that return addresses (e.g. **`ldloca`** and **`ldarga`**) are safely aligned. For datatypes larger than 1 byte, the byte ordering is dependent on the target CPU. Code that depends on byte ordering may not run on all platforms.

The operation of the **`ldind`** instructions may be altered by an immediately preceding **`volatile.`** or **`unaligned.`** prefix instruction.

Rationale: *Signed and unsigned forms for the small integer types are needed so that the CLI can know whether to sign extend or zero extend. The* **`ldind.u8`** *and* **`ldind.u4`** *variants are provided for convenience;* **`ldind.u8`** *is an alias for* **`ldind.i8`***;* **`ldind.u4`** *and* **`ldind.i4`** *have different opcodes, but their effect is identical*

Exceptions:

`NullReferenceException` may be thrown if an invalid address is detected.

Verifiability:

Correct CIL only uses an **`ldind`** instruction in a manner consistent with the type of the pointer.

3.43 ldloc - load local variable onto the stack

Format	Assembly Format	Description
FE 0C<*unsigned int16*>	ldloc *indx*	Load local variable of index *indx* onto stack.
11 <*unsigned int8*>	ldloc.s *indx*	Load local variable of index *indx* onto stack, short form.
06	ldloc.0	Load local variable 0 onto stack.
07	ldloc.1	Load local variable 1 onto stack.
08	ldloc.2	Load local variable 2 onto stack.
09	ldloc.3	Load local variable 3 onto stack.

Stack Transition:

`… → …, value`

Description:

The **`ldloc`** *indx* instruction pushes the contents of the local variable number *indx* onto the evaluation stack, where local variables are numbered 0 onwards. Local variables are initialized to 0 before entering the method only if the initialize flag on the method is true (see Partition I). The **`ldloc.0`**, **`ldloc.1`**, **`ldloc.2`**, and **`ldloc.3`** instructions provide an efficient encoding for accessing the first four local variables. The **`ldloc.s`** instruction provides an efficient encoding for accessing local variables 4 through 255.

The type of the value is the same as the type of the local variable, which is specified in the method header. See Partition I.

Local variables that are smaller than 4 bytes long are expanded to type **`int32`** when they are loaded onto the stack. Floating-point values are expanded to their native size (type **`F`**).

Exceptions:

`VerificationException` is thrown if the the "zero initialize" bit for this method has not been set, and the assembly containing this method has not been granted SecurityPermission.SkipVerification (and the CIL does not perform automatic definite-assignment analysis)

Verifiability:

Correct CIL ensures that *indx* is a valid local index. See Section 1.5 for more details on how verification determines the type of a local variable. For the *ldloca indx* instruction, i*ndx* must lie in the range 0 to 65534 inclusive (specifically, 65535 is not valid)

Rationale: *The reason for excluding 65535 is pragmatic: likely implementations will use a 2-byte integer to track both a local's index, as well as the total number of locals for a given method. If an index of 65535 had been made legal, it would require a wider integer to track the number of locals in such a method.*

Also, for verifiable code, this instruction must guarantee that it is not loading an uninitialized value – whether that initialization is done explicitly by having set the "zero initialize" bit for the method, or by previous instructions (where the CLI performs definite-assignment analysis)

3.44 ldloca.<length> - load local variable address

Format	Assembly Format	Description
FE 0D <*unsigned int16*>	ldloca *index*	Load address of local variable with index *indx*
12 <*unsigned int8*>	ldloca.s *index*	Load address of local variable with index *indx, short form*

Stack Transition:

… $\rightarrow$ …, address

Description:

The **`ldloca`** instruction pushes the address of the local variable number *index* onto the stack, where local variables are numbered 0 onwards. The value pushed on the stack is already aligned correctly for use with instructions like **`ldind`** and **`stind`**. The result is a managed pointer (type **`&`**). The **`ldloca.s`** instruction provides an efficient encoding for use with the local variables 0 through 255.

Exceptions:

`VerificationException` is thrown if the the "zero initialize" bit for this method has not been set, and the assembly containing this method has not been granted SecurityPermission.SkipVerification (and the CIL does not perform automatic definite-assignment analysis)

Verifiability:

Correct CIL ensures that *indx* is a valid local index. See Section 1.5 for more details on how verification determines the type of a local variable. For the *ldloca indx* instruction, i*ndx* must lie in the range 0 to 65534 inclusive (specifically, 65535 is not valid)

Rationale: *The reason for excluding 65535 is pragmatic: likely implementations will use a 2-byte integer to track both a local's index, as well as the total number of locals for a given method. If an index of 65535 had been made legal, it would require a wider integer to track the number of locals in such a method.*

Also, for verifiable code, this instruction must guarantee that it is not loading an uninitialized value – whether that initialization is done explicitly by having set the "zero initialize" bit for the method, or by previous instructions (where the CLI performs definite-assignment analysis)

3.45 ldnull – load a null pointer

Format	Assembly Format	Description
14	ldnull	Push null reference on the stack

Stack Transition:

… $\rightarrow$ `…, null value`

Description:

The `ldnull` pushes a null reference (type `O`) on the stack. This is used to initialize locations before they become live or when they become dead.

Rationale: *It might be thought that* `ldnull` *is redundant: why not use* `ldc.i4.0` *or* `ldc.i8.0` *instead? The answer is that* `ldnull` *provides a size-agnostic null – analogous to a* `ldc.i` *instruction, which does not exist. However, even if CIL were to include a* `ldc.i` *instruction it would still benefit verification algorithms to retain the* `ldnull` *instruction because it makes type tracking easier.*

Exceptions:

None.

Verifiability:

The `ldnull` instruction is always verifiable, and produces a value that verification considers compatible with any other reference type.

3.46 leave.<length> – exit a protected region of code

Format	Assembly Format	Description
DD <**int32**>	leave *target*	Exit a protected region of code.
DE <**int8**>	leave.s *target*	Exit a protected region of code, *short form*

Stack Transition:

…, $\rightarrow$

Description:

The **leave** instruction unconditionally transfers control to *target*. *Target* is represented as a signed offset (4 bytes for **leave**, 1 byte for **leave.s**) from the beginning of the instruction following the current instruction.

The **leave** instruction is similar to the **br** instruction, but it can be used to exit a **try**, **filter**, or **catch** block whereas the ordinary branch instructions can only be used in such a block to transfer control within it. The **leave** instruction empties the evaluation stack and ensures that the appropriate surrounding **finally** blocks are executed.

It is not legal to use a **leave** instruction to exit a **finally** block. To ease code generation for exception handlers it is legal from within a **catch** block to use a **leave** instruction to transfer control to any instruction within the associated **try** block.

If an instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Exceptions:

None.

Verifiability:

Correct CIL requires the computed destination lie within the current method. See Section 1.5 for more details.

3.47 localloc – allocate space in the local dynamic memory pool

Format	Assembly Format	Description
FE 0F	localloc	Allocate space from the local memory pool.

Stack Transition:

size → address

Description:

The **`localloc`** instruction allocates *size* (type **`native unsigned int`**) bytes from the local dynamic memory pool and returns the address (a managed pointer, type **`&`**) of the first allocated byte. The block of memory returned is initialized to 0 only if the initialize flag on the method is true (see Partition I). The area of memory is newly allocated. When the current method returns, the local memory pool is available for reuse.

Address is aligned so that any built-in data type can be stored there using the **`stind`** instructions and loaded using the **`ldind`** instructions.

The **`localloc`** instruction cannot occur within an exception block: **`filter`**, **`catch`**, **`finally`**, or **`fault`**.

Rationale: ***`Localloc`*** *is used to create local aggregates whose size must be computed at runtime. It can be used for C's intrinsic* ***`alloca`*** *method.*

Exceptions:

`StackOverflowException` is thrown if there is insufficient memory to service the request.

Verifiability:

Correct CIL requires that the evaluation stack be empty, apart from the *size* item. This instruction is never verifiable.

3.48 mul - multiply values

Format	Assembly Format	Description
5A	mul	Multiply values

Stack Transition:

`…, value1, value2` → `…, result`

Description:

The **`mul`** instruction multiplies *value1* by *value2* and pushes the result on the stack. Integral operations silently truncate the upper bits on overflow (see **`mul.ovf`**).

For floating-point types, 0 × **`infinity`** = **`NaN`**.

The acceptable operand types and their corresponding result data types are encapsulated in Table 2: Binary Numeric Operations.

Exceptions:

None.

Verifiability:

See Table 2: Binary Numeric Operations.

3.49 mul.ovf.<type> - multiply integer values with overflow check

Format	Assembly Format	Description
D8	mul.ovf	Multiply signed integer values. Signed result must fit in same size
D9	mul.ovf.un	Multiply unsigned integer values. Unsigned result must fit in same size

Stack Transition:

…, value1, value2 → …, result

Description:

The **`mul.ovf`** instruction multiplies integers, *value1* and *value2,* and pushes the result on the stack. An exception is thrown if the result will not fit in the result type.

The acceptable operand types and their corresponding result data types are encapsulated in Table 7: Overflow Arithmetic Operations.

Exceptions:

`OverflowException` is thrown if the result can not be represented in the result type.

Verifiability:

See Table 8: Conversion Operations.

3.50 neg - negate

Format	Assembly Format	Description
65	neg	Negate value

Stack Transition:

`…, value → …, result`

Description:

The **`neg`** instruction negates *value* and pushes the result on top of the stack. The return type is the same as the operand type.

Negation of integral values is standard twos complement negation. In particular, negating the most negative number (which does not have a positive counterpart) yields the most negative number. To detect this overflow use the **`sub.ovf`** instruction instead (i.e. subtract from 0).

Negating a floating-point number cannot overflow; negating **`NaN`** returns **`NaN`**.

The acceptable operand types and their corresponding result data types are encapsulated in Table 3: Unary Numeric Operations.

Exceptions:

None.

Verifiability:

See Table 3: Unary Numeric Operations.

3.51 nop – no operation

Format	Assembly Format	Description
00	nop	Do nothing

Stack Transition:

…, → …,

Description:

The **nop** operation does nothing. It is intended to fill in space if bytecodes are patched.

Exceptions:

None.

Verifiability:

The **nop** instruction is always verifiable.

3.52 not - bitwise complement

Format	Assembly Format	Description
66	not	Bitwise complement

Stack Transition:

…, value → …, result

Description:

Compute the bitwise complement of the integer value on top of the stack and leave the result on top of the stack. The return type is the same as the operand type.

The acceptable operand types and their corresponding result data type is encapsulated in Table 5: Integer Operations.

Exceptions:

None.

Verifiability:

See Table 5: Integer Operations.

3.53 or - bitwise OR

Format	Instruction	Description
60	or	Bitwise OR of two integer values, returns an integer.

Stack Transition:

…, value1, value2 → …, result

Description:

The `or` instruction computes the bitwise OR of the top two values on the stack and leaves the result on the stack.

The acceptable operand types and their corresponding result data type is encapsulated in Table 5: Integer Operations.

Exceptions:

None.

Verifiability:

See Table 5: Integer Operations.

3.54 pop – remove the top element of the stack

Format	Assembly Format	Description
26	pop	Pop a value from the stack

Stack Transition:

`…, value` $\rightarrow$ `…`

Description:

The **pop** instruction removes the top element from the stack.

Exceptions:

None.

Verifiability:

No additional requirements.

3.55 rem - compute remainder

Format	Assembly Format	Description
5D	rem	Remainder of dividing value1 by value2

Stack Transition:

`…, value1, value2` → `…, result`

Description:

The acceptable operand types and their corresponding result data type are encapsulated in Table 2: Binary Numeric Operations.

For integer operands

result = *value1* **rem** *value2* satisfies the following conditions:

result = *value1* – *value2*×(*value1* **div** *value2*), and

$0 \le |result| < |value2|$, and

sign(*result*) = *sign*(*value1*),

where **`div`** is the division instruction, which truncates towards zero.

The **`rem`** instruction computes *result* and pushes it on the stack.

For floating-point operands

`rem` is defined similarly, except that, if *value2* is zero or *value1* is infinity the result is NaN. If *value2* is **`infinity`**, the result is *value1* (negated for –**`infinity`**). This definition is different from the one for floating-point remainder in the IEC 60559:1989 Standard. That Standard specifies that *value1* ***div*** *value2* is the nearest integer instead of truncating towards zero. `System.Math.IEEERemainder` (see Partition IV) provides the IEC 60559:1989 behavior.

Exceptions:

Integral operations throw `DivideByZeroException` if *value2* is zero.

Integral operations may throw `ArithmeticException` if *value1* is the smallest representable integer value and *value2* is -1.

Example:

```
+10 rem +6 is 4    (+10 div +6 = 1)
+10 rem -6 is 4    (+10 div -6 = -1)
-10 rem +6 is -4   (-10 div +6 = -1)
-10 rem -6 is -4   (-10 div -6 = 1)
```

For the various floating-point values of 10.0 and 6.0, ***rem*** gives the same values; `System.Math.IEEERemainder`, however, gives the following values.

```
System.Math.IEEERemainder(+10.0,+6.0) is -2    (+10.0 div +6.0 =  1.666…7)
System.Math.IEEERemainder(+10.0,-6.0) is -2    (+10.0 div -6.0 = -1.666…7)
System.Math.IEEERemainder(-10.0,+6.0) is  2    (-10.0 div +6.0 = -1.666…7)
System.Math.IEEERemainder(-10.0,-6.0) is  2    (-10.0 div -6.0 =  1.666…7)
```

Verifiability:

See Table 2: Binary Numeric Operations.

3.56 rem.un - compute integer remainder, unsigned

Format	Assembly Format	Description
5E	rem.un	Remainder of unsigned dividing value1 by value2

Stack Transition:

…, value1, value2 $\rightarrow$ …, result

Description:

result = *value1* **rem.un** *value2* satisfies the following conditions:

result = *value1* – *value2*×(*value1* **div.un** *value2*), and

$0 \le result < value2$,

where `div.un` is the unsigned division instruction. The `rem.un` instruction computes *result* and pushes it on the stack. `Rem.un` treats its arguments as unsigned integers, while `rem` treats them as signed integers. `rem.un` is unspecified for floating-point numbers.

The acceptable operand types and their corresponding result data type are encapsulated in Table 5: Integer Operations.

Exceptions:

Integral operations throw `DivideByZeroException` if *value2* is zero.

Example:

```
+5 rem.un +3     is 2                  (+5 div.un +3 = 1)
+5 rem.un -3     is 5                  (+5 div.un -3 = 0)
-5 rem.un +3     is 2                  ( -5 div.un +3 = 1431655763 or 0x55555553)
-5 rem.un -3     is -5 or 0xfffffffb   ( -5 div.un -3 = 0)
```

Verifiability:

See Table 5: Integer Operations.

3.57 ret – return from method

Format	Assembly Format	Description
2A	ret	Return from method, possibly returning a value

Stack Transition:

```
 retVal on callee evaluation stack (not always present) →
…, retVal on caller evaluation stack (not always present)
```

Description:

Return from the current method. The return type, if any, of the current method determines the type of value to be fetched from the top of the stack and copied onto the stack of the method that called the current method. The evaluation stack for the current method must be empty except for the value to be returned.

The **`ret`** instruction cannot be used to transfer control out of a **`try`**, **`filter`**, **`catch`**, or **`finally`** block. From within a **`try`** or **`catch`**, use the **`leave`** instruction with a destination of a **`ret`** instruction that is outside all enclosing exception blocks. Because the **`filter`** and **`finally`** blocks are logically part of exception handling, not the method in which their code is embedded, correctly generated CIL does not perform a method return from within a **`filter`** or **`finally`**. See Partition I.

Exceptions:

None.

Verifiability:

Correct CIL obeys the control constraints describe above. Verification requires that the type of *retVal* is compatible with the declared return type of the current method.

3.58 shl - shift integer left

Format	Assembly Format	Description
62	shl	Shift an integer left (shifting in zeros), return an integer

Stack Transition:

`…, value, shiftAmount` → `…, result`

Description:

The **`shl`** instruction shifts *value* (**`int32`**, **`int64`** or **`native int`**) left by the number of bits specified by *shiftAmount*. *shiftAmount* is of type **`int32`**, **`int64`** or **`native int`**. The return value is unspecified if *shiftAmount* is greater than or equal to the width of *value*. See Table 6: Shift Operations for details of which operand types are allowed, and their corresponding result type.

Exceptions:

None.

Verifiability:

See Table 5: Integer Operations.

3.59 shr - shift integer right

Format	Assembly Format	Description
63	shr	Shift an integer right (shift in sign), return an integer

Stack Transition:

`…, value, shiftAmount` $\rightarrow$ `…, result`

Description:

The **`shr`** instruction shifts *value* (**`int32`**, **`int64`** or **`native int`**) right by the number of bits specified by *shiftAmount*. *shiftAmount* is of type **`int32`**, **`int64`** or **`native int`**. The return value is unspecified if *shiftAmount* is greater than or equal to the width of *value*. **`shr`** replicates the high order bit on each shift, preserving the sign of the original value in the result. See Table 6: Shift Operations for details of which operand types are allowed, and their corresponding result type.

Exceptions:

None.

Verifiability:

See Table 5: Integer Operations.

3.60 shr.un - shift integer right, unsigned

Format	Assembly Format	Description
64	shr.un	Shift an integer right (shift in zero), return an integer

Stack Transition:

`…, value, shiftAmount` → `…, result`

Description:

The **`shr.un`** instruction shifts *value* (**`int32`**, **`int 64`** or **`native int`**) right by the number of bits specified by *shiftAmount*. *shiftAmount* is of type **`int32`** or **`native int`**. The return value is unspecified if *shiftAmount* is greater than or equal to the width of *value*. **`shr.un`** inserts a zero bit on each shift. See Table 6: Shift Operations for details of which operand types are allowed, and their corresponding result type.

Exceptions:

None.

Verifiability:

See Table 5: Integer Operations.

3.61 starg.<length> - store a value in an argument slot

Format	Assembly Format	Description
FE 0B <**unsigned int16**>	starg *num*	Store a value to the argument numbered *num*
10 <**unsigned int8**>	starg.s *num*	Store a value to the argument numbered *num*, short form

Stack Transition:

`…, value` → `…,`

Description:

The **`starg`** *num* instruction pops a value from the stack and places it in argument slot *num* (see Partition I). The type of the value must match the type of the argument, as specified in the current method's signature. The **`starg.s`** instruction provides an efficient encoding for use with the first 256 arguments.

For procedures that take a variable argument list, the **`starg`** instructions can be used only for the initial fixed arguments, not those in the variable part of the signature.

Storing into arguments that hold an integer value smaller than 4 bytes long truncates the value as it moves from the stack to the argument. Floating-point values are rounded from their native size (type **`F`**) to the size associated with the argument.

Exceptions:

None.

Verifiability:

Correct CIL requires that *num* is a valid argument slot.

Verification also checks that the verification type of *value* matches the type of the argument, as specified in the current method's signature (verification types are less detailed than CLI types).

3.62 stind.<type> - store value indirect from stack

Format	Assembly Format	Description
52	stind.i1	Store value of type `int8` into memory at address
53	stind.i2	Store value of type `int16` into memory at address
54	stind.i4	Store value of type `int32` into memory at address
55	stind.i8	Store value of type `int64` into memory at address
56	stind.r4	Store value of type `float32` into memory at address
57	stind.r8	Store value of type `float64` into memory at address
DF	stind.i	Store value of type `native int` into memory at address
51	stind.ref	Store value of type `object ref` (type `O`) into memory at address

Stack Transition:

…, addr, val → …

Description:

The **`stind`** instruction stores a value *val* at address *addr* (an unmanaged pointer, type **`native int`**, or managed pointer, type **`&`**). The address specified by *addr* must be aligned to the natural size of *val* or a `NullReferenceException` may occur (but see the **`unaligned.`** prefix instruction). The results of all CIL instructions that return addresses (e.g. **`ldloca`** and **`ldarga`**) are safely aligned. For datatypes larger than 1 byte, the byte ordering is dependent on the target CPU. Code that depends on byte ordering may not run on all platforms.

Type safe operation requires that the **`stind`** instruction be used in a manner consistent with the type of the pointer.

The operation of the **`stind`** instruction may be altered by an immediately preceding **`volatile.`** or **`unaligned.`** prefix instruction.

Exceptions:

`NullReferenceException` is thrown if *addr* is not naturally aligned for the argument type implied by the instruction suffix

Verifiability:

Correct CIL ensures that *addr* be a pointer whose type is known and is assignment compatible with that of *val*.

3.63 stloc - pop value from stack to local variable

Format	Assembly Format	Description
FE 0E <*unsigned int16*>	stloc *indx*	Pop value from stack into local variable *indx*.
13 <*unsigned int8*>	stloc.s *indx*	Pop value from stack into local variable *indx*, short form.
0A	stloc.0	Pop value from stack into local variable 0.
0B	stloc.1	Pop value from stack into local variable 1.
0C	stloc.2	Pop value from stack into local variable 2.
0D	stloc.3	Pop value from stack into local variable 3.

Stack Transition:

…, value → …

Description:

The **stloc** *indx* instruction pops the top value off the evalution stack and moves it into local variable number *indx* (see Partition I), where local variables are numbered 0 onwards. The type of *value* must match the type of the local variable as specified in the current method's locals signature. The **stloc.0**, **stloc.1**, **stloc.2**, and **stloc.3** instructions provide an efficient encoding for the first four local variables; the **stloc.s** instruction provides an efficient encoding for local variables 4 through 255.

Storing into locals that hold an integer value smaller than 4 bytes long truncates the value as it moves from the stack to the local variable. Floating-point values are rounded from their native size (type **F**) to the size associated with the argument.

Exceptions:

None.

Verifiability:

Correct CIL requires that *indx* is a valid local index. For the *stloc indx* instruction, i*ndx* must lie in the range 0 to 65534 inclusive (specifically, 65535 is not valid)

Rationale: *The reason for excluding 65535 is pragmatic: likely implementations will use a 2-byte integer to track both a local's index, as well as the total number of locals for a given method. If an index of 65535 had been made legal, it would require a wider integer to track the number of locals in such a method.*

Verification also checks that the verification type of *value* matches the type of the local, as specified in the current method's locals signature.

3.64 sub - subtract numeric values

Format	Assembly Format	Description
59	sub	Subtract *value2* from *value1*, returning a new value

Stack Transition:

`…, value1, value2 → …, result`

Description:

The **`sub`** instruction subtracts *value2* from *value1* and pushes the result on the stack. Overflow is not detected for the integral operations (see **`sub.ovf`**); for floating-point operands, **`sub`** returns **`+inf`** on positive overflow, **`-inf`** on negative overflow, and zero on floating-point underflow.

The acceptable operand types and their corresponding result data type is encapsulated in Table 2: Binary Numeric Operations.

Exceptions:

None.

Verifiability:

See Table2: Binary Numeric Operations.

3.65 sub.ovf.<type> - subtract integer values, checking for overflow

Format	Assembly Format	Description
DA	sub.ovf	Subtract `native int` from a `native int`. Signed result must fit in same size
DB	sub.ovf.un	Subtract native `unsigned int` from a `native unsigned int`. Unsigned result must fit in same size

Stack Transition:

…, value1, value2 → …, result

Description:

The `sub.ovf` instruction subtracts *value2* from *value1* and pushes the result on the stack. The type of the values and the return type is specified by the instruction. An exception is thrown if the result does not fit in the result type.

The acceptable operand types and their corresponding result data type is encapsulated in Table 7: Overflow Arithmetic Operations.

Exceptions:

`OverflowException` is thrown if the result can not be represented in the result type.

Verifiability:

See Table 7: Overflow Arithmetic Operations.

3.66 switch – table switch on value

Format	Assembly Format	Description
45 *<unsigned int32> <int32>… <int32>*	switch *(t1, t2 … tn)*	jump to one of n values

Stack Transition:

…, value → …,

Description:

The `switch` instruction implements a jump table. The format of the instruction is an `unsigned int32` representing the number of targets *N*, followed by *N* `int32` values specifying jump targets: these targets are represented as offsets (positive or negative) from the beginning of the instruction following this switch instruction.

The switch instruction pops *value* off the stack and compares it, as an unsigned integer, to *N*. If *value* is less than *N*, execution is transferred to the *value*'th target, where targets are numbered from 0 (i.e., a *value* of 0 takes the first target, a *value* of 1 takes the second target, etc). If *value* is not less than *N*, execution continues at the next instruction (fall through).

If the target instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Control transfers into and out of `try`, `catch`, `filter`, and `finally` blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the `leave` instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL obeys the control transfer constraints listed above. In addition, verification requires the type-consistency of the stack, locals and arguments for every possible way of reaching all destination instructions. See Section 1.5 for more details.

3.67 xor - bitwise XOR

Format	Assembly Format	Description
61	xor	Bitwise XOR of integer values, returns an integer

Stack Transition:

`..., value1, value2` → `..., result`

Description:

The **`xor`** instruction computes the bitwise XOR of *value1* and *value2* and leaves the result on the stack.

The acceptable operand types and their corresponding result data type is encapsulated in Table 5: Integer Operations.

Exceptions:

None.

Verifiability:

See Table 5: Integer Operations.

4 Object Model Instructions

The instructions described in the base instruction set are independent of the object model being executed. Those instructions correspond closely to what would be found on a real CPU. The object model instructions are less built-in than the base instructions in the sense that they could be built out of the base instructions and calls to the underlying operating system.

Rationale: *The object model instructions provide a common, efficient implementation of a set of services used by many (but by no means all) higher-level languages. They embed in their operation a set of conventions defined by the common type system. This include (among other things):*

Field layout within an object

Layout for late bound method calls (vtables)

Memory allocation and reclamation

Exception handling

Boxing and unboxing to convert between reference-based Objects and Value Types

For more details, see Partition I.

4.1 box – convert value type to object reference

Format	Assembly Format	Description
8C <*T*>	box *valTypeTok*	Convert *valueType* to a true object reference

Stack Transition:

`…, valueType` → `…, obj`

Description:

A value type has two separate representations (see Partition I) within the CLI:

- A ‘raw’ form used when a value type is embedded within another object or on the stack.
- A ‘boxed’ form, where the data in the value type is wrapped (boxed) into an object so it can exist as an independent entity.

The **`box`** instruction converts the ‘raw’ *valueType* (an unboxed value type) into an instance of type Object (of type **`O`**). This is accomplished by creating a new object and copying the data from *valueType* into the newly allocated object. *ValTypeTok* is a metadata token (a **`typeref`** or **`typedef`**) indicating the type of *valueType* (See Partition II)

Exceptions:

`OutOfMemoryException` is thrown if there is insufficient memory to satisfy the request.

`TypeLoadException` is thrown if *class* cannot be found. This is typically detected when CIL is converted to native code rather than at runtime.

Verifiability:

Correct CIL ensures that *valueType* is of the correct value type, and that *valTypeTok* is a **`typeref`** or **`typedef`** metadata token for that value type.

4.2 callvirt – call a method associated, at runtime, with an object

Format	Assembly Format	Description
6F <*T*>	callvirt *method*	Call a method associated with *obj*

Stack Transition:

…, obj, arg1, … argN → …, returnVal (not always returned)

Description:

The **callvirt** instruction calls a late-bound method on an object. That is, the method is chosen based on the runtime type of *obj* rather than the compile-time class visible in the *method* metadata token. **Callvirt** can be used to call both virtual and instance methods. See Partition I for a detailed description of the CIL calling sequence. The **callvirt** instruction may be immediately preceded by a **tail.** prefix to specify that the current stack frame should be released before transferring control. If the call would transfer control to a method of higher trust than the original method the stack frame will not be released.

[A callee of "higher trust" is defined as one whose permission grant-set is a strict superset of the grant-set of the caller]

method is a metadata token (a **methoddef** or **methodref**; see Partition II) that provides the name, class and signature of the method to call. In more detail, **callvirt** can be thought of as follows. Associated with *obj* is the class of which it is an instance. If *obj*'s class defines a non-static method that matches the indicated method name and signature, this method is called. Otherwise all classes in the superclass chain of obj's class are checked in order. It is an error if no method is found.

Callvirt pops the object and the arguments off the evaluation stack before calling the method. If the method has a return value, it is pushed on the stack upon method completion. On the callee side, the *obj* parameter is accessed as argument 0, *arg1* as argument 1 etc.

The arguments are placed on the stack in left-to-right order. That is, the first argument is computed and placed on the stack, then the second argument, etc. The **this** pointer (always required for **callvirt**) must be pushed before any of the user-visible arguments. The signature carried in the metadata does not contain an entry in the parameter list for the **this** pointer, but uses a bit (called HASTHIS) to indiciate whether the method requires passing the this pointer (see Partition II)

Note that a virtual method may also be called using the **call** instruction.

Exceptions:

`MissingMethodException` is thrown if a non-static method with the indicated name and signature could not be found in *obj's* class or any of its superclasses. This is typically detected when CIL is converted to native code, rather than at runtime.

`NullReferenceException` is thrown if *obj* is null.

`SecurityException` is thrown if system security does not grant the caller access to the called method. The security check may occur when the CIL is converted to native code rather than at runtime.

Verifiability:

Correct CIL ensures that the destination method exists and the values on the stack correspond to the types of the parameters of the method being called.

In its typical use, **callvirt** is verifiable if (a) the above restrictions are met, (b) the verification type of *obj* is consistent with the method being called, (c) the verification types of the arguments on the stack are consistent with the types expected by the method call, and (d) the method is accessible from the callsite. A **callvirt** annotated by **tail.** has additional considerations – see Section 1.5.

4.3 castclass – cast an object to a class

Format	Assembly Format	Description
74 <*T*>	castclass *class*	Cast *obj* to *class*

Stack Transition:

…, obj → …, obj2

Description:

The **`castclass`** instruction attempts to cast *obj* (an **`O`**) to the *class*. *Class* is a metadata token (a **`typeref`** or **`typedef`**), indicating the desired class. If the class of the object on the top of the stack does not implement *class* (if *class* is an interface), and is not a subclass of *class* (if *class* is a regular class), then an `InvalidCastException` is thrown.

Note that:

1. Arrays inherit from `System.Array`
2. If Foo can be cast to Bar, then Foo[] can be cast to Bar[]
3. For the purposes of 2., enums are treated as their undertlying type: thus E1[] can cast to E2[] if E1 and E2 share an underlying type

If *obj* is null, **`castclass`** succeeds and returns null. This behavior differs from **`isInst`**.

Exceptions:

`InvalidCastException` is thrown if *obj* cannot be cast to *class*.

`TypeLoadException` is thrown if *class* cannot be found. This is typically detected when CIL is converted to native code rather than at runtime.

Verifiability:

Correct CIL ensures that *class* is a valid **`typeRef`** or **`typeDef`** token, and that *obj* is always either null or an object reference.

4.4 cpobj - copy a value type

Format	Assembly Format	Description
70 <T>	cpobj *classTok*	Copy a value type from *srcValObj* to *destValObj*

Stack Transition:

…, *destValObj*, *srcValObj* → …,

Description:

The `cpobj` instruction copies the value type located at the address specified by *srcValObj* (an unmanaged pointer, `native int`, or a managed pointer, `&`) to the address specified by *destValObj* (also a pointer). Behavior is unspecified if *srcValObj* and *dstValObj* are not pointers to instances of the class represented by *classTok* (a `typeref` or `typedef`), or if *classTok* does not represent a value type.

Exceptions:

`NullReferenceException` may be thrown if an invalid address is detected.

Verifiability:

Correct CIL ensures that *classTok* is a valid `typeRef` or `typeDef` token for a value type, as well as that *srcValObj* and *destValObj* are both pointers to locations of that type.

Verification requires, in addition, that *srcValObj* and *destValObj* are both managed pointers (not unmanaged pointers).

4.5 initobj - initialize a value type

Format	Assembly Format	Description
FE 15 <**T**>	initobj *classTok*	Initialize a value type

Stack Transition:

…, addrOfValObj → …,

Description:

The **initobj** instruction initializes all the fields of the object represented by the address *addrOfValObj* (of type **native int,** or **&**) to **null** or a 0 of the appropriate built-in type. After this method is called, the instance is ready for the constructor method to be called. Behavior is unspecified if either *addrOfValObj* is not a pointer to an instance of the class represented by *classTok* (a **typeref** or **typedef;** see Partition II), or *classTok* does not represent a value type.

Notice that, unlike **newobj**, the constructor method is not called by **initobj**. **Initobj** is intended for initializing value types, while **newobj** is used to allocate and initialize objects.

Exceptions:

None.

Verifiability:

Correct CIL ensures that *classTok* is a valid **typeref** or **typedef** token specifying a value type, and that *valObj* is a managed pointer to an instance of that value type.

4.6 isinst – test if an object is an instance of a class or interface

Format	Assembly Format	Description
75 <*T*>	isinst *class*	test if *obj* is an instance of *class*, returning null or an instance of that class or interface

Stack Transition:

`…, obj` → `…, result`

Description:

The **`isinst`** instruction tests whether *obj* (type **`O`**) is an instance of *class*. *Class* is a metadata token (a **`typeref`** or **`typedef`** see Partition II) indicating the desired class. If the class of the object on the top of the stack implements *class* (if *class* is an interface) or is a subclass of *class* (if *class* is a regular class), then it is cast to the type *class* and the result is pushed on the stack, exactly as though **`castclass`** had been called. Otherwise null is pushed on the stack. If *obj* is null, **`isinst`** returns null.

Note that:

1. Arrays inherit from `System.Array`
2. If Foo can be cast to Bar, then Foo[] can be cast to Bar[]
3. For the purposes of 2., enums are treated as their undertlying type: thus E1[] can cast to E2[] if E1 and E2 share an underlying type

Exceptions:

`TypeLoadException` is thrown if *class* cannot be found. This is typically detected when CIL is converted to native code rather than at runtime.

Verifiability:

Correct CIL ensures that *class* is a valid **`typeref`** or **`typedef`** token indicating a class, and that *obj* is always either null or an object reference

4.7 ldelem.<type> – load an element of an array

Format	Assembly Format	Description
90	ldelem.i1	Load the element with type **int8** at *index* onto the top of the stack as an **int32**
92	ldelem.i2	Load the element with type **int16** at *index* onto the top of the stack as an **int32**
94	ldelem.i4	Load the element with type **int32** at *index* onto the top of the stack as an **int32**
96	ldelem.i8	Load the element with type **int64** at *index* onto the top of the stack as an **int64**
91	ldelem.u1	Load the element with type **unsigned int8** at *index* onto the top of the stack as an **int32**
93	ldelem.u2	Load the element with type **unsigned int16** at *index* onto the top of the stack as an **int32**
95	ldelem.u4	Load the element with type **unsigned int32** at *index* onto the top of the stack as an **int32**
96	ldelem.u8	Load the element with type **unsigned int64** at *index* onto the top of the stack as an **int64** (alias for **ldelem.i8**)
98	ldelem.r4	Load the element with type **float32** at *index* onto the top of the stack as an **F**
99	ldelem.r8	Load the element with type **float64** at *index* onto the top of the stack as an **F**
97	ldelem.i	Load the element with type **native int** at *index* onto the top of the stack as an **native int**
9A	ldelem.ref	Load the element of type object, at *index* onto the top of the stack as an **O**

Stack Transition:

…, array, index → …, value

Description:

The **ldelem** instruction loads the value of the element with index *index* (of type **int32** or **native int**) in the zero-based one-dimensional array *array* and places it on the top of the stack. Arrays are objects and hence represented by a value of type **O**. The return value is indicated by the instruction.

For one-dimensional arrays that aren't zero-based and for multidimensional arrays, the array class provides a **Get** method.

Note that integer values of less than 4 bytes are extended to **int32** (not **native int**) when they are loaded onto the evaluation stack. Floating-point values are converted to **F** type when loaded onto the evaluation stack.

Exceptions:

`NullReferenceException` is thrown if *array* is null.

`IndexOutOfRangeException` is thrown if *index* is negative, or larger than the bound of *array*.

`ArrayTypeMismatchException` is thrown if *array* doesn't hold elements of the required type.

Verifiability:

Correct CIL code requires that *array* is either null or a zero-based, one-dimensional array whose declared element type matches exactly the type for this particular instruction suffix (e.g. **`ldelem.r4`** can only be applied to a zero-based, one dimensional array of **`float32`**s)

4.8 ldelema – load address of an element of an array

Format	Assembly Format	Description
8F <*T*>	ldelema *class*	Load the address of element at *index* onto the top of the stack

Stack Transition:

…, array, index $\rightarrow$ …, address

Description:

The `ldelema` instruction loads the address of the element with index *index* (of type `int32` or `native int`) in the zero-based one-dimensional array *array* (of element type *class)* and places it on the top of the stack. Arrays are objects and hence represented by a value of type `O`. The return address is a managed pointer (type `&`).

For one-dimensional arrays that aren't zero-based and for multidimensional arrays, the array class provides a `Address` method.

Exceptions:

`NullReferenceException` is thrown if *array* is null.

`IndexOutOfRangeException` is thrown if *index* is negative, or larger than the bound of *array*.

`ArrayTypeMismatchException` is thrown if *array* doesn't hold elements of the required type.

Verifiability:

Correct CIL ensures that *class* is a `typeref` or `typedef` token to a class, and that *array* is indeed always either null or a zero-based, one-dimensional array whose declared element type matches *class* exactly.

4.9 ldfld – load field of an object

Format	Assembly Format	Description
7B <*T*>	ldfld *field*	Push the value of *field* of object, or value type, *obj,* onto the stack

Stack Transition:

…, obj → …, value

Description:

The `ldfld` instruction pushes onto the stack the value of a field of *obj*. *obj* must be an object (type `O`), a managed pointer (type `&`), an unmanaged pointer (type `native int`), or an instance of a value type. The use of an unmanaged pointer is not permitted in verifiable code. *field* is a metadata token (a `fieldref` or `fielddef` see Partition II) that must refer to a field member. The return type is that associated with *field*. `ldfld` pops the object reference off the stack and pushes the value for the field in its place. The field may be either an instance field (in which case *obj* must not be null) or a static field.

The `ldfld` instruction may be preceded by either or both of the `unaligned.` and `volatile.` prefixes.

Exceptions:

`NullReferenceException` is thrown if *obj* is null and the field is not static.

`MissingFieldException` is thrown if *field* is not found in the metadata. This is typically checked when CIL is converted to native code, not at runtime.

Verifiability:

Correct CIL ensures that *field* is a valid token referring to a field, and that *obj* will always have a type compatible with that required for the lookup being performed. For verifiable code, *obj* may not be an unmanaged pointer.

4.10 ldflda – load field address

Format	Assembly Format	Description
7C <*T*>	ldflda *field*	Push the address of *field* of object *obj* on the stack

Stack Transition:

`…, obj → …, address`

Description:

The **`ldflda`** instruction pushes the address of a field of *obj*. *obj* is either an object, type **`O`**, a managed pointer, type **`&`**, or an unmanaged pointer, type **`native int`**. The use of an unmanaged pointer is not allowed in verifiable code. The value returned by **`ldflda`** is a managed pointer (type **`&`**) unless *obj* is an unmanaged pointer, in which case it is an unmanaged pointer (type **`native int`**).

field is a metadata token (a **`fieldref`** or **`fielddef;`** see Partition II) that must refer to a field member. The field may be either an instance field (in which case *obj* must not be null) or a static field.

Exceptions:

`InvalidOperationException` is thrown if the *obj* is not within the application domain from which it is being accessed. The address of a field that is not inside the accessing application domain cannot be loaded.

`MissingFieldException` is thrown if *field* is not found in the metadata. This is typically checked when CIL is converted to native code, not at runtime.

`NullReferenceException` is thrown if *obj* is null and the field isn't static.

Verifiability:

Correct CIL ensures that *field* is a valid **`fieldref`** token and that *obj* will always have a type compatible with that required for the lookup being performed.

Note: Using **`ldflda`** to compute the address of a static, init-only field and then using the resulting pointer to modify that value outside the body of the class initializer may lead to unpredictable behavior. It cannot, however, compromise memory integrity or type safety so it is not tested by verification.

4.11 ldlen – load the length of an array

Format	Assembly Format	Description
8E	ldlen	Push the length (of type `native unsigned int`) of *array* on the stack

Stack Transition:

…, array → …, length

Description:

The `ldlen` instruction pushes the number of elements of *array* (a zero-based, one-dimensional array) on the stack.

Arrays are objects and hence represented by a value of type `O`. The return value is a `native unsigned  int`.

Exceptions:

`NullReferenceException` is thrown if *array* is null.

Verifiability:

Correct CIL ensures that *array* is indeed always either null or a zero-based, one dimensional array.

4.12 ldobj - copy value type to the stack

Format	Assembly Format	Description
71 <T>	ldobj *classTok*	Copy instance of value type *classTok* to the stack.

Stack Transition:

…, addrOfValObj → …, valObj

Description:

The **`ldobj`** instruction copies the value pointed to by *addrOfValObj* (of type managed pointer, **`&`**, or unmanaged pointer, **`native unsigned  int`**) to the top of the stack. The number of bytes copied depends on the size of the class represented by *classTok*. *ClassTok* is a metadata token (a **`typeref`** or **`typedef;`** see Partition II) representing a value type.

Rationale: *The* **`ldobj`** *instruction is used to pass a value type as a parameter. See Partition I.*

It is unspecified what happens if *addrOfValObj* is not an instance of the class represented by *ClassTok* or if *ClassTok* does not represent a value type.

The operation of the **`ldobj`** instruction may be altered by an immediately preceding **`volatile.`** or **`unaligned.`** prefix instruction.

Exceptions:

`TypeLoadException` is thrown if *class* cannot be found. This is typically detected when CIL is converted to native code rather than at runtime.

Verifiability:

Correct CIL ensures that *classTok* is a metadata token representing a value type and that *addrOfValObj* is a pointer to a location containing a value of the type specified by *classTok*. Verifiable code additionally requires that *addrOfValObj* is a managed pointer of a matching type.

4.13 ldsfld – load static field of a class

Format	Assembly Format	Description
7E <*T*>	ldsfld *field*	Push the value of *field* on the stack

Stack Transition:

…, → …, value

Description:

The **`ldsfld`** instruction pushes the value of a static (shared among all instances of a class) field on the stack. *field* is a metadata token (a **`fieldref`** or **`fielddef;`** see Partition II) referring to a static field member. The return type is that associated with *field*.

The **`ldsfld`** instruction may have a **`volatile.`** prefix.

Exceptions:

None.

Verifiability:

Correct CIL ensures that *field* is a valid metadata token referring to a static field member.

4.14 ldsflda – load static field address

Format	Assembly Format	Description
7F <*T*>	ldsflda *field*	Push the address of the static field, *field,* on the stack

Stack Transition:

…, → …, address

Description:

The **ldsflda** instruction pushes the address (a managed pointer, type **&**, if *field* refers to a type whose memory is managed; otherwise an unmanaged pointer, type **native int**) of a static field on the stack. *field* is a metadata token (a **fieldref** or **fielddef;** see Partition II) referring to a static field member. (Note that *field* may be a static global with assigned RVA, in which case its memory is *un*managed; where RVA stands for Relative Virtual Address, the offset of the field from the base address at which its containing PE file is loaded into memory)

Exceptions:

MissingFieldException is thrown if *field* is not found in the metadata. This is typically checked when CIL is converted to native code, not at runtime.

Verifiability:

Correct CIL ensures that *field* is a valid metadata token referring to a static field member if *field* refers to a type whose memory is managed.

Note: Using **ldsflda** to compute the address of a static, init-only field and then using the resulting pointer to modify that value outside the body of the class initializer may lead to unpredictable behavior. It cannot, however, compromise memory integrity or type safety so it is not tested by verification.

4.15 ldstr – load a literal string

Format	Assembly Format	Description
72 <*T*>	ldstr *string*	push a string object for the literal *string*

Stack Transition:

…, → …, string

Description:

The **`ldstr`** instruction pushes a new string object representing the literal stored in the metadata as *string* (that must be a string literal).

The **`ldstr`** instruction allocates memory and performs any format conversion required to convert from the form used in the file to the string format required at runtime. The CLI guarantees that the result of two **`ldstr`** instructions referring to two metadata tokens that have the same sequence of characters return precisely the same string object (a process known as "string interning").

Exceptions:

None.

Verifiability:

Correct CIL requires that *string* is a valid string literal metadata token.

4.16 ldtoken - load the runtime representation of a metadata token

Format	Assembly Format	Description
D0 <*T*>	ldtoken *token*	Convert metadata *token* to its runtime representation

Stack Transition:

… → …, `RuntimeHandle`

Description:

The `ldtoken` instruction pushes a RuntimeHandle for the specified metadata token. The token must be one of:

A `methoddef` or `methodref` : pushes a `RuntimeMethodHandle`

A `typedef` or `typeref` : pushes a `RuntimeTypeHandle`

A `fielddef` or `fieldref` : pushes a `RuntimeFieldHandle`

The value pushed on the stack can be used in calls to Reflection methods in the system class library

Exceptions:

None.

Verifiability:

Correct CIL requires that *token* describes a valid metadata token.

4.17 ldvirtftn - load a virtual method pointer

Format	Assembly Format	Description
FE 07 <*T*>	ldvirtftn *mthd*	Push address of virtual method *mthd* on the stack

Stack Transition:

… object → …, ftn

Description:

The **`ldvirtftn`** instruction pushes an unmanaged pointer (type **`native int`**) to the native code implementing the virtual method associated with *object* and described by the method reference *mthd* (a metadata token, either a **`methoddef`** or **`methodref;`** see Partition II) onto the stack. The value pushed can be called using the **`calli`** instruction if it references a managed method (or a stub that transitions from managed to unmanaged code).

The value returned points to native code using the calling convention specified by *mthd*. Thus a method pointer can be passed to unmanaged native code (e.g. as a callback routine) if that routine expects the corresponding calling convention. Note that the address computed by this instruction may be to a thunk produced specially for this purpose (for example, to re-enter the CLI when a native version of the method isn't available)

Exceptions:

None.

Verifiability:

Correct CIL ensures that *mthd* is a valid **`methoddef`** or **`methodref`** token. Also that *mthd* references a non-static method that is defined for *object.* Verification tracks the type of the value pushed in more detail than the "**`native int`**" type, remembering that it is a method pointer. Such a method pointer can then be used in verified code with **`calli`** or to construct a delegate.

4.18 mkrefany – push a typed reference on the stack

Format	Assembly Format	Description
C6 <*T*>	mkrefany *class*	Push a typed reference to *ptr* of type *class* onto the stack

Stack Transition:

…, ptr → …, typedRef

Description:

The `mkrefany` instruction supports the passing of dynamically typed references. *ptr* must be a pointer (type `&`, or `native int`) that holds the address of a piece of data. *Class* is the class token (a `typeref` or `typedef;` see Partition II) describing the type of *ptr*. `Mkrefany` pushes a typed reference on the stack, that is an opaque descriptor of *ptr* and *class*. The only legal operation on a typed reference on the stack is to pass it to a method that requires a typed reference as a parameter. The callee can then use the `refanytype` and `refanyval` instructions to retrieve the type (*class*) and address (*ptr*) respectively.

Exceptions:

`TypeLoadException` is thrown if *class* cannot be found. This is typically detected when CIL is converted to native code rather than at runtime.

Verifiability:

Correct CIL ensures that *class* is a valid `typeref` or `typedef` token describing some type and that *ptr* is a pointer to exactly that type. Verification additionally requires that *ptr* be a managed pointer. Verification will fail if it cannot deduce that *ptr* is a pointer to an instance of *class*.

4.19 newarr – create a zero-based, one-dimensional array

Format	Assembly Format	Description
8D <*T*>	newarr *etype*	Create a new array with elements of type *etype*

Stack Transition:

…, *numElems* → …, *array*

Description:

The **`newarr`** instruction pushes a reference to a new zero-based, one-dimensional array whose elements are of type *elemtype***,** a metadata token (a **`typeref`** or **`typedef;`** see Partition II). *numElems* (of type native int) specifies the number of elements in the array. Valid array indexes are $0 \le \text{index} < numElems$. The elements of an array can be any type, including value types.

Zero-based, one-dimensional arrays of numbers are created using a metadata token referencing the appropriate value type (`System.Int32`, etc.). Elements of the array are initialized to 0 of the appropriate type.

One-dimensional arrays that aren't zero-based and multidimensional arrays are created using **`newobj`** rather than **`newarr`**. More commonly, they are created using the methods of `System.Array` class in the Base Framework.

Exceptions:

`OutOfMemoryException` is thrown if there is insufficient memory to satisfy the request.

`OverflowException` is thrown if *numElems* is < 0

Verifiability:

Correct CIL ensures that *etype* is a valid **`typeref`** or **`typedef`** token.

4.20 newobj – create a new object

Format	Assembly Format	Description
73 <*T*>	newobj *ctor*	Allocate an uninitialized object or value type and call *ctor*

Stack Transition:

…, arg1, … argN → *…, obj*

Description:

The **newobj** instruction creates a new object or a new instance of a value type. *ctor* is a metadata token (a **methodref** or **methodef** that must be marked as a constructor; see Partition II) that indicates the name, class and signature of the constructor to call. If a constructor exactly matching the indicated name, class and signature cannot be found, MissingMethodException is thrown.

The **newobj** instruction allocates a new instance of the class associated with *constructor* and initializes all the fields in the new instance to 0 (of the proper type) or **null** as appropriate. It then calls the constructor with the given arguments along with the newly created instance. After the constructor has been called, the now initialized object reference is pushed on the stack.

From the constructor's point of view, the uninitialized object is argument 0 and the other arguments passed to **newobj** follow in order.

All zero-based, one-dimensional arrays are created using **newarr**, not **newobj**. On the other hand, all other arrays (more than one dimension, or one-dimensional but not zero-based) are created using **newobj**.

Value types are not usually created using **newobj**. They are usually allocated either as arguments or local variables, using **newarr** (for zero-based, one-dimensional arrays), or as fields of objects. Once allocated, they are initialized using **initobj**. However, the **newobj** instruction can be used to create a new instance of a value type on the stack, that can then be passed as an argument, stored in a local, etc.

Exceptions:

OutOfMemoryException is thrown if there is insufficient memory to satisfy the request.

MissingMethodException is thrown if a constructor method with the indicated name, class and signature could not be found. This is typically detected when CIL is converted to native code, rather than at runtime.

Verifiability:

Correct CIL ensures that **constructor** is a valid **methodref** or **methoddef** token, and that the arguments on the stack are compatible with those expected by the constructor. Verification considers a delegate constructor as a special case, checking that the method pointer passed in as the second argument, of type **native int,** does indeed refer to a method of the correct type.

4.21 refanytype – load the type out of a typed reference

Format	Assembly Format	Description
FE 1D	refanytype	Push the type token stored in a typed reference

Stack Transition:

…, TypedRef → …, type

Description:

Retrieves the type token embedded in **TypedRef**. See the **mkrefany** instruction.

Exceptions:

None.

Verifiability:

Correct CIL ensures that *TypedRef* is a valid typed reference (created by a previous call to **mkrefany**). The **refanytype** instruction is always verifiable.

4.22 refanyval – load the address out of a typed reference

Format	Assembly Format	Description
C2 <*T*>	refanyval *type*	Push the address stored in a typed reference

Stack Transition:

…, TypedRef → …, address

Description:

Retrieves the address (of type **&**) embedded in **TypedRef**. The type of reference in **TypedRef** must match the type specified by **type** (a metadata token, either a **typedef** or a **typeref;** see Partition II). See the **mkrefany** instruction.

Exceptions:

InvalidCastException is thrown if *type* is not identical to the type stored in the TypedRef (ie, the *class* supplied to the **mkrefany** instruction that constructed that TypedRef)

TypeLoadException is thrown if *type* cannot be found.

Verifiability:

Correct CIL ensures that *TypedRef* is a valid typed reference (created by a previous call to **mkrefany**). The **refanyval** instruction is always verifiable.

4.23 rethrow – rethrow the current exception

Format	Assembly Format	Description
FE 1A	rethrow	Rethrow the current exception

Stack Transition:

…, → …,

Description:

The **rethrow** instruction is only permitted within the body of a **catch** handler (see Partition I). It throws the same exception that was caught by this handler.

Exceptions:

The original exception is thrown.

Verifiability:

Correct CIL uses this instruction only within the body of a **catch** handler (not of any exception handlers embedded within that **catch** handler). If a rethrow occurs elsewhere, then an exception will be thrown, but precisely which exception is undefined

4.24 sizeof – load the size in bytes of a value type

Format	Assembly Format	Description
FE 1C <**T**>	sizeof *valueType*	Push the size, in bytes, of a value type as a **`unsigned int32`**

Stack Transition:

…, → …, size (4 bytes, unsigned)

Description:

Returns the size, in bytes, of a value type. *valueType* must be a metadata token (a **`typeref`** or **`typedef;`** see Partition II) that specifies a value type.

Rationale: *The definition of a value type can change between the time the CIL is generated and the time that it is loaded for execution. Thus, the size of the type is not always known when the CIL is generated. The* ***`sizeof`*** *instruction allows CIL code to determine the size at runtime without the need to call into the Framework class library. The computation can occur entirely at runtime or at CIL-to-native-code compilation time.* ***`sizeof`*** *returns the total size that would be occupied by each element in an array of this value type – including any padding the implementation chooses to add. Specifically, array elements lie* ***`sizeof`*** *bytes apart*

Exceptions:

None.

Verifiability:

Correct CIL ensures that **`valueType`** is a **`typeref`** or **`typedef`** referring to a value type. It is always verificable.

4.25 stelem.<type> – store an element of an array

Format	Assembly Format	Description
9C	stelem.i1	Replace array element at *index* with the `int8` value on the stack
9D	stelem.i2	Replace array element at *index* with the `int16` value on the stack
9E	stelem.i4	Replace array element at *index* with the `int32` value on the stack
9F	stelem.i8	Replace array element at *index* with the `int64` value on the stack
A0	stelem.r4	Replace array element at *index* with the `float32` value on the stack
A1	stelem.r8	Replace array element at *index* with the `float64` value on the stack
9B	stelem.i	Replace array element at *index* with the `i` value on the stack
A2	stelem.ref	Replace array element at *index* with the `ref` value on the stack

Stack Transition:

…, array, index, value → *…,*

Description:

The `stelem` instruction replaces the value of the element with zero-based index *index* (of type `int32` or `native int`) in the one-dimensional array *array* with *value*. Arrays are objects and hence represented by a value of type `O`.

Note that `stelem.ref` implicitly casts *value* to the element type of *array* before assigning the value to the array element. This cast can fail, even for verified code. Thus the `stelem.ref` instruction may throw the `ArrayTypeMismatchException`.

For one-dimensional arrays that aren't zero-based and for multidimensional arrays, the array class provides a **StoreElement** method.

Exceptions:

`NullReferenceException` is thrown if *array* is null.

`IndexOutOfRangeException` is thrown if *index* is negative, or larger than the bound of *array*.

`ArrayTypeMismatchException` is thrown if *array* doesn't hold elements of the required type.

Verifiability:

Correct CIL requires that *array* be a zero-based, one-dimensional array whose declared element type matches exactly the type for this particular instruction suffix (eg `stelem.r4` can only be applied to a zero-based, one dimensional array of `float32`'s); also that *index* lies within the bounds of *array*

4.26 stfld – store into a field of an object

Format	Assembly Format	Description
7D <*T*>	stfld *field*	Replace the value of *field* of the object *obj* with *val*

Stack Transition:

…, obj, value → *…,*

Description:

The `stfld` instruction replaces the value of a field of an *obj* (an **O**) or via a pointer (type `native int`, or `&`) with `value`. `field` is a metadata token (a `fieldref` or `fielddef;` see Partition II) that refers to a field member reference. `stfld` pops the value and the object reference off the stack and updates the object.

The `stfld` instruction may have a prefix of either or both of `unaligned.` and `volatile.`.

Exceptions:

`NullReferenceException` is thrown if *obj* is null and the field isn't static.

`MissingFieldException` is thrown if *field* is not found in the metadata. This is typically checked when CIL is converted to native code, not at runtime.

Verifiability:

Correct CIL ensures that *field* is a valid token referring to a field, and that *obj* and *value* will always have types appropriate for the assignment being performed. For verifiable code, *obj* may not be an unmanaged pointer.

Note: Using `stfld` to change the value of a static, init-only field outside the body of the class initializer may lead to unpredictable behavior. It cannot, however, compromise memory integrity or type safety so it is not tested by verification .

4.27 stobj - store a value type from the stack into memory

Format	Assembly Format	Description
81 <T>	stobj *classTok*	Store a value of type *classTok* from the stack into memory

Stack Transition:

`…, addr, valObj` → `…,`

Description:

The **`stobj`** instruction copies the value type *valObj* into the address specified by *addr* (a pointer of type **`native int`**, or **`&`**). The number of bytes copied depends on the size of the class represented by **`classTok`**. **`classTok`** is a metadata token (a **`typeref`** or **`typedef;`** see Partition II) representing a value type.

It is unspecified what happens if **`valObj`** is not an instance of the class represented by **`classTok`** or if **`classTok`** does not represent a value type.

The operation of the **`stobj`** instruction may be altered by an immediately preceding **`volatile.`** or **`unaligned.`** prefix instruction.

Exceptions:

`TypeLoadException` is thrown if *class* cannot be found. This is typically detected when CIL is converted to native code rather than at runtime.

Verifiability:

Correct CIL ensures that **`classTok`** is a metadata token representing a value type and that **`valObj`** is a pointer to a location containing an initialized value of the type specified by **`classTok`**. In addition, verifiable code requires that **`valObj`** be a managed pointer.

4.28 stsfld – store a static field of a class

Format	Assembly Format	Description
80 <*T*>	stsfld *field*	Replace the value of *field* with *val*

Stack Transition:

…, *val* → …,

Description:

The **stsfld** instruction replaces the value of a static field with a value from the stack. *field* is a metadata token (a **fieldref** or **fielddef;** see Partition II) that must refer to a static field member. **Stsfld** pops the value off the stack and updates the static field with that value.

The **stsfld** instruction may be prefixed by **volatile.**.

Exceptions:

MissingFieldException is thrown if *field* is not found in the metadata. This is typically checked when CIL is converted to native code, not at runtime.

Verifiability:

Correct CIL ensures that *field* is a valid token referring to a static field, and that *value* will always have a type appropriate for the assignment being performed.

Note: Using **stsfld** to change the value of a static, init-only field outside the body of the class initializer may lead to unpredictable behavior. It cannot, however, compromise memory integrity or type safety so it is not tested by verification.

4.29 throw – throw an exception

Format	Assembly Format	Description
7A	throw	Throw an exception

Stack Transition:

…, object → …,

Description:

The **`throw`** instruction throws the exception *object* (type **`O`**) on the stack. For details of the exception mechanism, see Partition I.

Note: While the CLI permits any object to be thrown, the common language specification (CLS) describes a specific exception class that must be used for language interoperability.

Exceptions:

`NullReferenceException` is thrown if *obj* is null.

Verifiability:

Correct CIL ensures that *class* a valid **`typeRef`** token indicating a class, and that *obj* is always either null or an object reference, i.e. of type **`O`**.

4.30 unbox – Convert boxed value type to its raw form

Format	Assembly Format	Description
79 <*T*>	unbox *valuetype*	Extract the value type data from *obj*, its boxed representation

Stack Transition:

…, obj → …, valueTypePtr

Description:

A value type has two separate representations (see Partition I) within the CLI:

- A 'raw' form used when a value type is embedded within another object.
- A 'boxed' form, where the data in the value type is wrapped (boxed) into an object so it can exist as an independent entity.

The **unbox** instruction converts *obj* (of type **O**), the boxed representation of a value type, to *valueTypePtr* (a managed pointer, type **&**), its unboxed form. *valuetype* is a metadata token (a **typeref** or **typedef**) indicating the type of value type contained within *obj*. If *obj* is not a boxed instance of *valuetype*, or, if *obj* is a boxed enum and *valuetype* is not its underlying type, then this instruction will throw an `InvalidCastException`

Unlike **box**, which is required to make a copy of a value type for use in the object, **unbox** is *not* required to copy the value type from the object. Typically it simply computes the address of the value type that is already present inside of the boxed object.

Exceptions:

`InvalidCastException` is thrown if *obj* is not a boxed *valuetype* (or if *obj* is a boxed enum and *valuetype* is not its underlying type)

`NullReferenceException` is thrown if obj is null.

`TypeLoadException` is thrown if *class* cannot be found. This is typically detected when CIL is converted to native code rather than at runtime.

Verifiability:

Correct CIL ensures that *valueType* is a **typeref** or **typedef** metadata token for some value type, and that *obj* is always an object reference, i.e. of type **O**, and represents a boxed instance of a *valuetype* value type.

Common Language Infrastructure (CLI)
Partition IV:
Profiles and Libraries

Table of contents

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 (method signatures, etc.) 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 are designed with the following goals in mind:

- Wide reach across programming languages
- Consistent design patterns throughout
- Features on parity with the ISO/IEC C Standard library of 1990
- Features for more recent programming paradigms, notably networking, XML, runtime type inspection, instance creation, and dynamic method dispatch
- Factoring into self-consistent libraries with minimal interdependence

This document provides an overview of the CLI Libraries and a specification of their factoring into Profiles and Libraries. A companion document, considered to be part of this Partition but distributed in XML format, provides details of each class, value type, and interface 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

Partition V contains an informative annex describing programming conventions used in defining the CLI Libraries. These conventions, while not normative, can significantly simplify the use of libraries. Implementers are encouraged to follow them when creating additional (non-Standard) Libraries.

2 Libraries and Profiles

Libraries and Profiles, defined below, are constructs created for the purpose of standards conformance/compliance. They specify a set of features that shall be present in an implementation of the Common Language Infrastructure (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 may 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").

There is, in general, 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 methods and types.

2.1 Libraries

A Library specifies three things:

1. A set of types that shall be available, including their grouping into assemblies.
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 Chapter 5 has text that defines what CLI features are required for implementations that support the Library.

3. Modifications to types defined in *other* Libraries. These modifications are typically the addition of methods and interfaces to types belonging to the other Library, and additional exceptions that may be thrown by methods of the 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 (informative): Consider the Extended Numerics Library. Since it provides a new 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, `System.NotFiniteNumberException`, and specifies existing methods in other Libraries methods that throw it (as it happens, there are no such methods).

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 (Extended Numerics) but whose type is in another Library (BCL), the XML specifies the Library that defines the method. See Chapter 7.

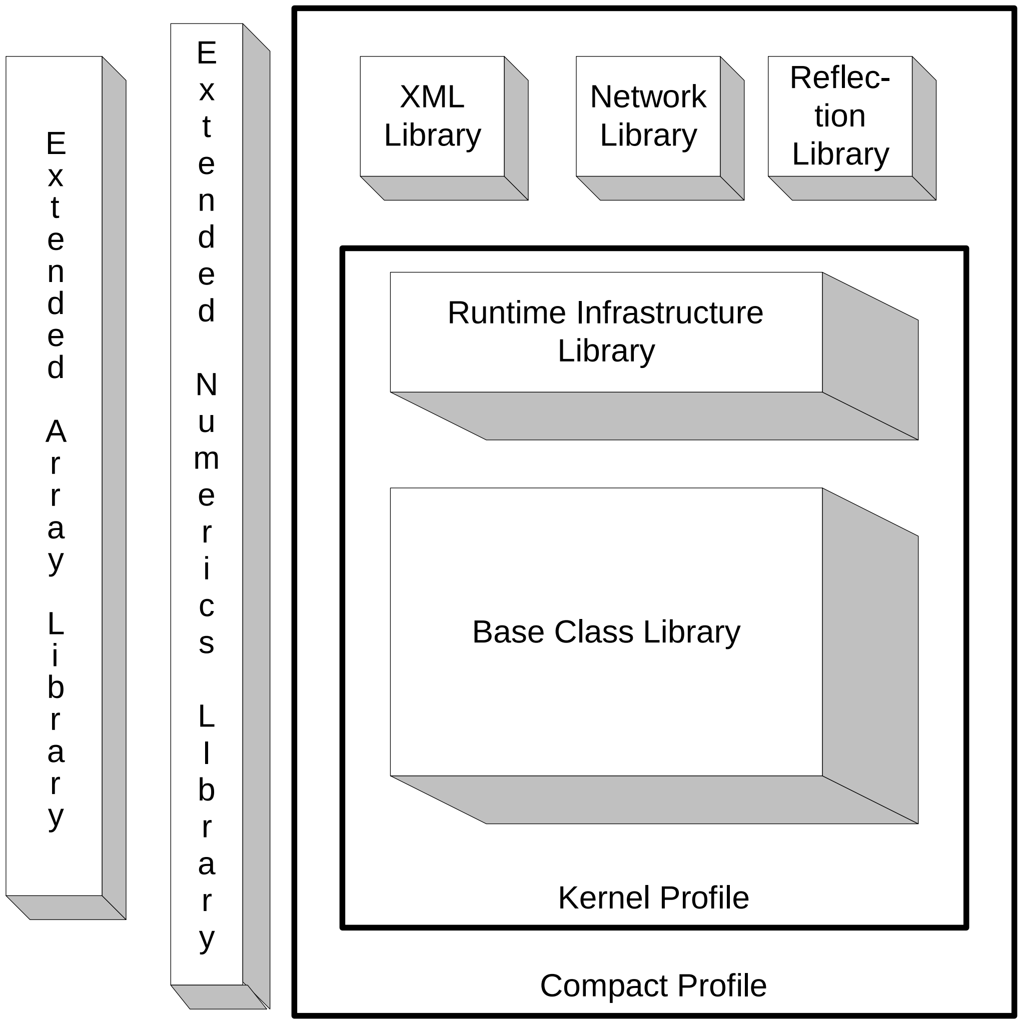
2.2 Profiles

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 a Profile it implements, as well as any additional Libraries that it provides. The Kernel Profile (see Section 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 Chapter 4.

Rationale: *The rules for combining Libraries together are complex, since each Library may add members to types defined in other libraries. By standardizing a small number of Profiles we specify completely the interaction of the Libraries that are part of each Profile. 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 we increase the market for each Profile, making each a desirable target for a class of applications across a wide range of implementations and tool sets.*

2.3 Structure of the Standard

This standard specifies two Standard Profiles (see Chapter 3) and 7 Standard Libraries (see Chapter 5). The following diagram shows the relationship between the Libraries and the Profiles:



The Extended Array Library and the Extended Numerics Library are not part of either Profile, but may 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.

Note: Vendors of compliant CLI implementations should specify exactly which configurations of Standard Libraries and Standard Profiles they support.

Note: "Features" may 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.

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 classes 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 and consequently all CLI features required by the Kernel Profile must be implemented by all conforming implementations. This section defines that minimal feature set by enumerating the set of features that are not required, i.e., a minimal conforming implementation must implement all CLI features except those specified in the remainder of this section. The feature requirements of individual Libraries as specified in Chapter 5 are defined by reference to restricted items described in this section. For ease of reference, each feature has a name indicated by the name of the section 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 Section are required.

4.1 Features Excluded from 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 (add, etc.) need not handle those types.

`Part of Library``: Extended Numerics (see` Section 5.6`)`

4.1.2 Non-vector Arrays

The **non-vector arrays feature set** includes the 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 generic 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 class are vectors.

`Part of Library``: Extended Arrays (see` Section 5.7`).`

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.

`Part of Library``: Reflection (see` Section 5.4`).`

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 Section 5.2) 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 Varargs

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

If omitted: Any attempt to reference a method with the `varargs` calling convention or the signature encodings associated with varargs 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``: (none)`

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 section 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 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

5.2 Base Class Library

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, and so forth.

Name used in XML: BCL

CLI Feature Requirement: None

5.3 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

5.4 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 Section 5.1.

5.5 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 Section 5.3).

Name used in XML: XML

CLI Feature Requirement: None

5.6 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.

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.*

CLI Feature Requirement: Floating Point, see clause 4.1.1.

5.7 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, and arrays that have non-zero lower bounds.

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

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 may 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 may be thrown, but where possible these should be subclasses 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 the class provides an implementation.

  Rationale: *An interface or virtual method may be added only where it carries an implementation. This allows programs written when the interface or method was not present to continue to work.*

- 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.*

- The accessibility of fields and non-virtual methods may 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 may implement interfaces not defined in this standard.
- Adding fields (values) to enumerations.
- An implementation may 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.

Rationale: *An implementation may wish to split functionality across several types in order to provide non-standard extension mechanisms, or may 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.*

7 Semantics of the XML Specification

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.

The text associated with an element or an attribute (e.g. #PCDATA, #CDATA) is, unless explicitly stated otherwise, 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.]

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. These correspond to sections of the metadata of an assembly as described in Partition II and include information from the AssemblyName, AssemblyPublicKey, AssemblyVersion, AssemblyCulture and Attributes elements.

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

(Normative) Specifies the name of the assembly of which a given type is a member. 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 1.0.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?, example?)>
```

(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 the signature is written in. 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 NMTOKEN #REQUIRED
```

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

```
  FullName NMTOKEN #REQUIRED
```

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

```
  FullNameSP NMTOKEN #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 specification.]

`>`

`<!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 specification.]

`>`

`<!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 may 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 can be thrown by a member of a type. 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 specification.]

>

<!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 specification.]

```
>
<!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 specification.]

```
  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 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.

Common Language Infrastructure (CLI)

Partition V:
Annexes

Annex A Scope

Annex A this annex.

Annex B contains a number of sample programs written in CIL Assembly Language (ILAsm).

Annex C contains information about a particular implementation of an assembler, which provides a superset of the functionality of the syntax described in Partition II. It also provides a machine-readable description of the CIL instruction set which may be used to derive parts of the grammar used by this assembler as well as other tools that manipulate CIL.

Annex D contains a set of guidelines used in the design of the libraries of Partition IV. The rules are provided here since they have proven themselves effective in designing cross-language APIs. They also serve as guidelines for those intending to supply additional functionality in a way that meshes seamlessly with the standardized libraries.

Annex E contains information of interest to implementers with respect to the latitude they have in implementing the CLI.

Annex B Sample Programs

This chapter contains only informative text

This Annex shows several complete examples written using ilasm.

B.1. Mutually Recursive Program (with tail calls)

The following is an example of a mutually recursive program that uses tail calls. The methods below determine whether a number is even or odd.

```
.assembly extern mscorlib { }
.assembly test.exe { }
.class EvenOdd
{ .method private static bool IsEven(int32 N) cil managed
  { .maxstack   2
    ldarg.0             // N
    ldc.i4.0
    bne.un      NonZero
    ldc.i4.1
    ret
NonZero:
    ldarg.0
    ldc.i4.1
    sub
    tail.
    call bool EvenOdd::IsOdd(int32)
    ret
  } // end of method ‘EvenOdd::IsEven’

  .method private static bool IsOdd(int32 N) cil managed
  { .maxstack   2
    // Demonstrates use of argument names and labels
    // Notice that the assembler does not convert these
    // automatically to their short versions
    ldarg       N
    ldc.i4.0
    bne.un      NonZero
    ldc.i4.0
    ret
NonZero:
    ldarg       N
    ldc.i4.1
    sub
    tail.
    call bool EvenOdd::IsEven(int32)
    ret
  } // end of method ‘EvenOdd::IsOdd’

  .method public static void Test(int32 N) cil managed
  { .maxstack   1
    ldarg       N
    call        void [mscorlib]System.Console::Write(int32)
    ldstr       " is "
    call        void [mscorlib]System.Console::Write(string)
    ldarg       N
    call        bool EvenOdd::IsEven(int32)
    brfalse     LoadOdd
    ldstr       "even"
Print:
    call        void [mscorlib]System.Console::WriteLine(string)
    ret
LoadOdd:
    ldstr       "odd"
    br          Print
  } // end of method ‘EvenOdd::Test’
} // end of class ‘EvenOdd’

//Global method

.method public static void main() cil managed
{ .entrypoint
  .maxstack     1
  ldc.i4.5
  call          void EvenOdd::Test(int32)
  ldc.i4.2
  call          void EvenOdd::Test(int32)
  ldc.i4        100
  call          void EvenOdd::Test(int32)
  ldc.i4        1000001
  call          void EvenOdd::Test(int32)
  ret
} // end of global method ‘main’
```

B.2. Using Value Types

The following program shows how rational numbers can be implemented using value types.

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

```
.assembly rational.exe { }
.class private sealed Rational extends [mscorlib]System.ValueType
                             implements [mscorlib]System.IComparable
{ .field public int32 Numerator
  .field public int32 Denominator

  .method virtual public int32 CompareTo(object o)
  // Implements IComparable::CompareTo(Object)
  { ldarg.0     // ‘this’ as a managed pointer
    ldfld int32 value class Rational::Numerator
    ldarg.1     // ‘o’ as an object
    unbox value class Rational
    ldfld int32 value class Rational::Numerator
    beq.s TryDenom
    ldc.i4.0
    ret
TryDenom:
    ldarg.0     // ‘this’ as a managed pointer
    ldfld int32 value class Rational::Denominator
    ldarg.1     // ‘o’ as an object
    unbox value class Rational
    ldfld int32 class Rational::Denominator
    ceq
    ret
  }

  .method virtual public string ToString()
  // Implements Object::ToString
  { .locals init (class [mscorlib]System.Text.StringBuilder SB,
                  string S, object N, object D)
    newobj void [mscorlib]System.Text.StringBuilder::.ctor()
    stloc.s SB
    ldstr "The value is: {0}/{1}"
    stloc.s S
    ldarg.0     // Managed pointer to self
    dup
    ldfld int32 value class Rational::Numerator
    box [mscorlib]System.Int32
    stloc.s N
    ldfld int32 value class Rational::Denominator
    box [mscorlib]System.Int32
    stloc.s D
    ldloc.s SB
    ldloc.s S
    ldloc.s N
    ldloc.s D
    call instance class [mscorlib]System.Text.StringBuilder
      [mscorlib]System.Text.StringBuilder::AppendFormat(string,
                 object, object)
    callvirt instance string [mscorlib]System.Object::ToString()
    ret
  }
  .method public value class Rational Mul(value class Rational)
  {
    .locals init (value class Rational Result)
    ldloca.s Result
    dup
    ldarg.0     // ‘this’
    ldfld int32 value class Rational::Numerator
    ldarga.s    1     // arg
    ldfld int32 value class Rational::Numerator
    mul
    stfld int32 value class Rational::Numerator
    ldarg.0     // this
    ldfld int32 value class Rational::Denominator
    ldarga.s    1     // arg
    ldfld int32 value class Rational::Denominator
    mul
    stfld int32 value class Rational::Denominator
    ldloc.s Result
    ret
  }
}
.method static void main()
{
  .entrypoint
  .locals init (value class Rational Half,
                value class Rational Third,
                value class Rational Temporary,
                object H, object T)
  // Initialize Half, Third, H, and T
  ldloca.s Half
  dup
  ldc.i4.1
  stfld int32 value class Rational::Numerator
  ldc.i4.2
```

```
stfld  int32 value class Rational::Denominator
ldloca.s Third
dup
ldc.i4.1
stfld int32 value class Rational::Numerator
ldc.i4.3
stfld int32 value class Rational::Denominator
ldloc.s Half
box value class Rational
stloc.s H
ldloc.s Third
box value class Rational
stloc.s T
// WriteLine(H.IComparable::CompareTo(H))
// Call CompareTo via interface using boxed instance
ldloc H
dup
callvirt int32 [mscorlib]System.IComparable::CompareTo(object)
call void [mscorlib]System.Console::WriteLine(bool)
// WriteLine(Half.CompareTo(T))
// Call CompareTo via value type directly
ldloca.s Half
ldloc T
call instance int32
value class Rational::CompareTo(object)
call void [mscorlib]System.Console::WriteLine(bool)
// WriteLine(Half.ToString())
// Call virtual method via value type directly
ldloca.s Half
call instance string class Rational::ToString()
call void [mscorlib]System.Console::WriteLine(string)
// WriteLine(T.ToString)
// Call virtual method inherited from Object, via boxed instance
ldloc T
callvirt string [mscorlib]System.Object::ToString()
call void [mscorlib]System.Console::WriteLine(string)
// WriteLine((Half.Mul(T)).ToString())
// Mul is called on two value types, returning a value type
// ToString is then called directly on that value type
// Note that we are required to introduce a temporary variable
//   since the call to ToString requires
//   a managed pointer (address)
ldloca.s Half
```

```
  ldloc.s Third
  call instance value class Rational
         Rational::Mul(value class Rational)
  stloc.s Temporary
  ldloca.s Temporary
  call instance string Rational::ToString()
  call void [mscorlib]System.Console::WriteLine(string)
  ret
}
```

Annex C CIL Assembler Implementation

This chapter contains only informative text

This section provides information about a particular assembler for CIL, called ILASM. It supports a superset of the syntax defined normatively in Partition II, and provides a concrete syntax for the CIL instructions specified in Partition III.

Even for those who have no interest in this particular assembler, Section C.1 and Section 0 may prove of interest. The former is a machine-readable file (ready for input to a C or C++ preprocessor) that partially describes the CIL instructions. It can be used to generate tables for use by a wide variety of tools that deal with CIL. The latter contains a concrete syntax for CIL instructions, which is not described elsewhere.

C.1. ILAsm Keywords

This Section provides a complete list of the keywords used by ILASM. If users wish to use any of these as simple identifiers within programs they just make use of the appropriate escape notation (single or double quotation marks as specified in the grammar). This assembler is case-sensitive.

#line	.locale	.subsystem	at
.addon	.localized	.try	auto
.assembly	.locals	.ver	autochar
.cctor	.manifestres	.vtable	beforefieldinit
.class	.maxstack	.vtentry	beq
.corflags	.method	.vtfixup	beq.s
.ctor	.module	.zeroinit	bge
.custom	.mresource	^THE_END^	bge.s
.data	.namespace	abstract	bge.un
.emitbyte	.other	add	bge.un.s
.entrypoint	.override	add.ovf	bgt
.event	.pack	add.ovf.un	bgt.s
.export	.param	algorithm	bgt.un
.field	.pdirect	alignment	bgt.un.s
.file	.permission	and	ble
.fire	.permissionset	ansi	ble.s
.get	.property	any	ble.un
.hash	.publickey	arglist	ble.un.s
.imagebase	.publickeytoken	array	blob
.import	.removeon	as	blob_object
.language	.set	assembly	blt
.line	.size	assert	blt.s

blt.un	cil	conv.u	filetime
blt.un.s	ckfinite	conv.u1	filter
bne.un	class	conv.u2	final
bne.un.s	clsid	conv.u4	finally
bool	clt	conv.u8	fixed
box	clt.un	cpblk	float
br	const	cpobj	float32
br.s	conv.i	currency	float64
break	conv.i1	custom	forwardref
brfalse	conv.i2	date	fromunmanaged
brfalse.s	conv.i4	decimal	handler
brinst	conv.i8	default	hidebysig
brinst.s	conv.ovf.i	default	hresult
brnull	conv.ovf.i.un	demand	idispatch
brnull.s	conv.ovf.i1	deny	il
brtrue	conv.ovf.i1.un	div	illegal
brtrue.s	conv.ovf.i2	div.un	implements
brzero	conv.ovf.i2.un	dup	implicitcom
brzero.s	conv.ovf.i4	endfault	implicitres
bstr	conv.ovf.i4.un	endfilter	import
bytearray	conv.ovf.i8	endfinally	in
byvalstr	conv.ovf.i8.un	endmac	inheritcheck
call	conv.ovf.u	enum	init
calli	conv.ovf.u.un	error	initblk
callmostderived	conv.ovf.u1	explicit	initobj
callvirt	conv.ovf.u1.un	extends	initonly
carray	conv.ovf.u2	extern	instance
castclass	conv.ovf.u2.un	false	int
catch	conv.ovf.u4	famandassem	int16
cdecl	conv.ovf.u4.un	family	int32
ceq	conv.ovf.u8	famorassem	int64
cf	conv.ovf.u8.un	fastcall	int8
cgt	conv.r.un	fastcall	interface
cgt.un	conv.r4	fault	internalcall
char	conv.r8	field	isinst

iunknown	ldelem.ref	ldtoken	nop
jmp	ldelem.u1	ldvirtftn	noprocess
lasterr	ldelem.u2	leave	not
lcid	ldelem.u4	leave.s	not_in_gc_heap
ldarg	ldelem.u8	linkcheck	notremotable
ldarg.0	ldelema	literal	notserialized
ldarg.1	ldfld	localloc	null
ldarg.2	ldflda	lpstr	nullref
ldarg.3	ldftn	lpstruct	object
ldarg.s	ldind.i	lptstr	objectref
ldarga	ldind.i1	lpvoid	opt
ldarga.s	ldind.i2	lpwstr	optil
ldc.i4	ldind.i4	managed	or
ldc.i4.0	ldind.i8	marshal	out
ldc.i4.1	ldind.r4	method	permitonly
ldc.i4.2	ldind.r8	mkrefany	pinned
ldc.i4.3	ldind.ref	modopt	pinvokeimpl
ldc.i4.4	ldind.u1	modreq	pop
ldc.i4.5	ldind.u2	mul	prefix1
ldc.i4.6	ldind.u4	mul.ovf	prefix2
ldc.i4.7	ldind.u8	mul.ovf.un	prefix3
ldc.i4.8	ldlen	native	prefix4
ldc.i4.M1	ldloc	neg	prefix5
ldc.i4.m1	ldloc.0	nested	prefix6
ldc.i4.s	ldloc.1	newarr	prefix7
ldc.i8	ldloc.2	newobj	prefixref
ldc.r4	ldloc.3	newslot	prejitdeny
ldc.r8	ldloc.s	noappdomain	prejitgrant
ldelem.i	ldloca	noinlining	preservesig
ldelem.i1	ldloca.s	nomachine	private
ldelem.i2	ldnull	nomangle	privatescope
ldelem.i4	ldobj	nometadata	protected
ldelem.i8	ldsfld	noncasdemand	public
ldelem.r4	ldsflda	noncasinheritance	readonly
ldelem.r8	ldstr	noncaslinkdemand	record

refany	starg	stloc.2	true
refanytype	starg.s	stloc.3	typedref
refanyval	static	stloc.s	unaligned.
rem	stdcall	stobj	unbox
rem.un	stdcall	storage	unicode
reqmin	stelem.i	stored_object	unmanaged
reqopt	stelem.i1	stream	unmanagedexp
reqrefuse	stelem.i2	streamed_object	unsigned
reqsecobj	stelem.i4	string	unused
request	stelem.i8	struct	userdefined
ret	stelem.r4	stsfld	value
rethrow	stelem.r8	sub	valuetype
retval	stelem.ref	sub.ovf	vararg
rtspecialname	stfld	sub.ovf.un	variant
runtime	stind.i	switch	vector
safearray	stind.i1	synchronized	virtual
sealed	stind.i2	syschar	void
sequential	stind.i4	sysstring	volatile.
serializable	stind.i8	tail.	wchar
shl	stind.r4	tbstr	winapi
shr	stind.r8	thiscall	with
shr.un	stind.ref	thiscall	wrapper
sizeof	stloc	throw	xor
special	stloc.0	tls	
specialname	stloc.1	to	

C.2. CIL Opcode Descriptions

This Section contains text, which is intended for use with the C or C++ preprocessor. By appropriately defining the macros `OPDEF` and `OPALIAS` before including this text, it is possible to use this to produce tables or code for handling CIL instructions.

The `OPDEF` macro is passed 10 arguments, in the following order:

1. A symbolic name for the opcode, beginning with CEE_
2. A string that constitutes the name of the opcode and corresponds to the names given in Partition III.
3. Data removed from the stack to compute this operations result. The possible values here are the following:

   a. Pop0 - no inputs

   b. Pop1 - one value type specified by data flow
   c. Pop1+Pop1 - two input values, types specified by data flow
   d. PopI - one machine-sized integer
   e. PopI+Pop1 - Top of stack is described by data flow, next item is a native pointer
   f. PopI+PopI - Top two items on stack are integers (size may vary by instruction)
   g. PopI+PopI+PopI - Top three items on stack are machine-sized integers
   h. PopI8+Pop8 - Top of stack is an 8-byte integer, next is a native pointer
   i. PopI+PopR4 - Top of stack is a 4-byte floating point number, next is a native pointer
   j. PopI+PopR8 - Top of stack is an 8-byte floating point number, next is a native pointer
   k. PopRef - Top of stack is an object reference
   l. PopRef+PopI - Top of stack is an integer (size may vary by instruction), next is an object reference
   m. PopRef+PopI+PopI - Top of stack has two integers (size may vary by instruction), next is an object reference
   n. PopRef+PopI+PopI8 - Top of stack is an 8-byte integer, then a native-sized integer, then an object reference
   o. PopRef+PopI+PopR4 - Top of stack is an 4-byte floating point number, then a native-sized integer, then an object reference
   p. PopRef+PopI+PopR8 - Top of stack is an 8-byte floating point number, then a native-sized integer, then an object reference
   q. VarPop - variable number of items used, see Partition III for details

4. Amount and type of data pushed as a result of the instruction. The possible values here are the following:
   a. Push0 - no output value
   b. Push1 - one output value, type defined by data flow.
   c. Push1+Push1 - two output values, type defined by data flow
   d. PushI - push one native integer or pointer
   e. PushI8 - push one 8-byte integer
   f. PushR4 - push one 4-byte floating point number
   g. PushR8 - push one 8-byte floating point number
   h. PushRef - push one object reference
   i. VarPush - variable number of items pushed, see Partition III for details

5. Type of in-line argument to instruction. The in-line argument is stored with least significant byte first (“little endian”). The possible values here are the following:
   a. InlineBrTarget - Branch target, represented as a 4-byte signed integer from the beginning of the instruction following the current instruction.
   b. InlineField - Metadata token (4 bytes) representing a FieldRef (i.e. a MemberRef to a field) or FieldDef

c. InlineI - 4-byte integer
d. InlineI8 - 8-byte integer
e. InlineMethod - Metadata token (4 bytes) representing a MethodRef (i.e. a MemberRef to a method) or MethodDef
f. InlineNone - No in-line argument
g. InlineR - 8-byte floating point number
h. InlineSig - Metadata token (4 bytes) representing a standalone signature
i. InlineString - Metadata token (4 bytes) representing a UserString
j. InlineSwitch - Special for the switch instructions, see Partition III for details
k. InlineTok - Arbitrary metadata token (4 bytes) , used for ldtoken instruction, see Partition III for details
l. InlineType - Metadata token (4 bytes) representing a TypeDef, TypeRef, or TypeSpec
m. InlineVar - 2-byte integer representing an argument or local variable
n. ShortInlineBrTarget - Short branch target, represented as 1 signed byte from the beginning of the instruction following the current instruction.
o. ShortInlineI - 1-byte integer, signed or unsigned depending on instruction
p. ShortInlineR - 4-byte floating point number
q. ShortInlineVar - 1-byte integer representing an argument or local variable

6. Type of opcode. The current classification is of no current value, but is retained for historical reasons.
7. Number of bytes for the opcode. Currently 1 or 2, can be 4 in future
8. First byte of two byte encoding, or 0xFF if single byte instruction.
9. One byte encoding, or second byte of two-byte encoding.
10. Control flow implications of instruction. The possible values here are the following:
    a. BRANCH - unconditional branch
    b. CALL - method call
    c. COND_BRANCH - conditional branch
    d. META - unused operation or prefix code
    e. NEXT - control flow unaltered (“fall through”)
    f. RETURN - return from method
    g. THROW - throw or rethrow an exception

The `OPALIAS` macro takes three arguments:

1. A symbolic name for a “new instruction” which is simply an alias (renaming for the assembler) of an existing instruction.
2. A string name for the “new instruction.”
3. The symbolic name for an instruction introduced using the `OPDEF` macro. The “new instruction” is really just an alternative name for this instruction.

```
#ifndef __OPCODE_DEF_
```

```
#define __OPCODE_DEF_


#define MOOT    0x00    // Marks unused second byte when encoding single

#define STP1    0xFE    // Prefix code 1 for Standard Map

#define REFPRE  0xFF    // Prefix for Reference Code Encoding

#define RESERVED_PREFIX_START 0xF7


#endif


// If the first byte of the standard encoding is 0xFF, then

// the second byte can be used as 1 byte encoding.  Otherwise
l   b         b

// the encoding is two bytes.
e   y         y

//
n   t         t

//
g   e         e

//
(unused)       t

//  Canonical Name                    String Name              Stack Behaviour
Operand Params    Opcode Kind      h   1         2    Control Flow

// ------------------------------------------------------------------------------------
-----------------------------------------------------------------------

OPDEF(CEE_NOP,                        "nop",              Pop0,               Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0x00,    NEXT)

OPDEF(CEE_BREAK,                      "break",            Pop0,               Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0x01,    BREAK)

OPDEF(CEE_LDARG_0,                    "ldarg.0",          Pop0,               Push1,
InlineNone,         IMacro,      1,  0xFF,    0x02,    NEXT)

OPDEF(CEE_LDARG_1,                    "ldarg.1",          Pop0,               Push1,
InlineNone,         IMacro,      1,  0xFF,    0x03,    NEXT)

OPDEF(CEE_LDARG_2,                    "ldarg.2",          Pop0,               Push1,
InlineNone,         IMacro,      1,  0xFF,    0x04,    NEXT)

OPDEF(CEE_LDARG_3,                    "ldarg.3",          Pop0,               Push1,
InlineNone,         IMacro,      1,  0xFF,    0x05,    NEXT)

OPDEF(CEE_LDLOC_0,                    "ldloc.0",          Pop0,               Push1,
InlineNone,         IMacro,      1,  0xFF,    0x06,    NEXT)

OPDEF(CEE_LDLOC_1,                    "ldloc.1",          Pop0,               Push1,
InlineNone,         IMacro,      1,  0xFF,    0x07,    NEXT)

OPDEF(CEE_LDLOC_2,                    "ldloc.2",          Pop0,               Push1,
InlineNone,         IMacro,      1,  0xFF,    0x08,    NEXT)

OPDEF(CEE_LDLOC_3,                    "ldloc.3",          Pop0,               Push1,
InlineNone,         IMacro,      1,  0xFF,    0x09,    NEXT)

OPDEF(CEE_STLOC_0,                    "stloc.0",          Pop1,               Push0,
InlineNone,         IMacro,      1,  0xFF,    0x0A,    NEXT)

OPDEF(CEE_STLOC_1,                    "stloc.1",          Pop1,               Push0,
InlineNone,         IMacro,      1,  0xFF,    0x0B,    NEXT)

OPDEF(CEE_STLOC_2,                    "stloc.2",          Pop1,               Push0,
InlineNone,         IMacro,      1,  0xFF,    0x0C,    NEXT)
```

```
OPDEF(CEE_STLOC_3,                    "stloc.3",          Pop1,               Push0,
InlineNone,         IMacro,      1,  0xFF,    0x0D,    NEXT)

OPDEF(CEE_LDARG_S,                    "ldarg.s",          Pop0,               Push1,
ShortInlineVar,     IMacro,      1,  0xFF,    0x0E,    NEXT)

OPDEF(CEE_LDARGA_S,                   "ldarga.s",         Pop0,               PushI,
ShortInlineVar,     IMacro,      1,  0xFF,    0x0F,    NEXT)

OPDEF(CEE_STARG_S,                    "starg.s",          Pop1,               Push0,
ShortInlineVar,     IMacro,      1,  0xFF,    0x10,    NEXT)

OPDEF(CEE_LDLOC_S,                    "ldloc.s",          Pop0,               Push1,
ShortInlineVar,     IMacro,      1,  0xFF,    0x11,    NEXT)

OPDEF(CEE_LDLOCA_S,                   "ldloca.s",         Pop0,               PushI,
ShortInlineVar,     IMacro,      1,  0xFF,    0x12,    NEXT)

OPDEF(CEE_STLOC_S,                    "stloc.s",          Pop1,               Push0,
ShortInlineVar,     IMacro,      1,  0xFF,    0x13,    NEXT)

OPDEF(CEE_LDNULL,                     "ldnull",           Pop0,
PushRef,     InlineNone,         IPrimitive,  1,  0xFF,    0x14,    NEXT)

OPDEF(CEE_LDC_I4_M1,                  "ldc.i4.m1",        Pop0,               PushI,
InlineNone,         IMacro,      1,  0xFF,    0x15,    NEXT)

OPDEF(CEE_LDC_I4_0,                   "ldc.i4.0",         Pop0,               PushI,
InlineNone,         IMacro,      1,  0xFF,    0x16,    NEXT)

OPDEF(CEE_LDC_I4_1,                   "ldc.i4.1",         Pop0,               PushI,
InlineNone,         IMacro,      1,  0xFF,    0x17,    NEXT)

OPDEF(CEE_LDC_I4_2,                   "ldc.i4.2",         Pop0,               PushI,
InlineNone,         IMacro,      1,  0xFF,    0x18,    NEXT)

OPDEF(CEE_LDC_I4_3,                   "ldc.i4.3",         Pop0,               PushI,
InlineNone,         IMacro,      1,  0xFF,    0x19,    NEXT)

OPDEF(CEE_LDC_I4_4,                   "ldc.i4.4",         Pop0,               PushI,
InlineNone,         IMacro,      1,  0xFF,    0x1A,    NEXT)

OPDEF(CEE_LDC_I4_5,                   "ldc.i4.5",         Pop0,               PushI,
InlineNone,         IMacro,      1,  0xFF,    0x1B,    NEXT)

OPDEF(CEE_LDC_I4_6,                   "ldc.i4.6",         Pop0,               PushI,
InlineNone,         IMacro,      1,  0xFF,    0x1C,    NEXT)

OPDEF(CEE_LDC_I4_7,                   "ldc.i4.7",         Pop0,               PushI,
InlineNone,         IMacro,      1,  0xFF,    0x1D,    NEXT)

OPDEF(CEE_LDC_I4_8,                   "ldc.i4.8",         Pop0,               PushI,
InlineNone,         IMacro,      1,  0xFF,    0x1E,    NEXT)

OPDEF(CEE_LDC_I4_S,                   "ldc.i4.s",         Pop0,               PushI,
ShortInlineI,       IMacro,      1,  0xFF,    0x1F,    NEXT)

OPDEF(CEE_LDC_I4,                     "ldc.i4",           Pop0,               PushI,
InlineI,            IPrimitive,  1,  0xFF,    0x20,    NEXT)

OPDEF(CEE_LDC_I8,                     "ldc.i8",           Pop0,
PushI8,      InlineI8,           IPrimitive,  1,  0xFF,    0x21,    NEXT)

OPDEF(CEE_LDC_R4,                     "ldc.r4",           Pop0,
PushR4,      ShortInlineR,       IPrimitive,  1,  0xFF,    0x22,    NEXT)

OPDEF(CEE_LDC_R8,                     "ldc.r8",           Pop0,
PushR8,      InlineR,            IPrimitive,  1,  0xFF,    0x23,    NEXT)

OPDEF(CEE_UNUSED49,                   "unused",                 Pop0,
Push0,       InlineNone,         IPrimitive,  1,  0xFF,    0x24,    NEXT)

OPDEF(CEE_DUP,                        "dup",              Pop1,
Push1+Push1, InlineNone,         IPrimitive,  1,  0xFF,    0x25,    NEXT)

OPDEF(CEE_POP,                        "pop",              Pop1,               Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0x26,    NEXT)
```

```
OPDEF(CEE_JMP,                      "jmp",              Pop0,               Push0,
InlineMethod,       IPrimitive,  1,  0xFF,    0x27,    CALL)

OPDEF(CEE_CALL,                     "call",             VarPop,
VarPush,     InlineMethod,       IPrimitive,  1,  0xFF,    0x28,    CALL)

OPDEF(CEE_CALLI,                    "calli",            VarPop,
VarPush,     InlineSig,          IPrimitive,  1,  0xFF,    0x29,    CALL)

OPDEF(CEE_RET,                      "ret",              VarPop,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0x2A,    RETURN)

OPDEF(CEE_BR_S,                     "br.s",             Pop0,               Push0,
ShortInlineBrTarget,IMacro,      1,  0xFF,    0x2B,    BRANCH)

OPDEF(CEE_BRFALSE_S,                "brfalse.s",        PopI,               Push0,
ShortInlineBrTarget,IMacro,      1,  0xFF,    0x2C,    COND_BRANCH)

OPDEF(CEE_BRTRUE_S,                 "brtrue.s",         PopI,               Push0,
ShortInlineBrTarget,IMacro,      1,  0xFF,    0x2D,    COND_BRANCH)

OPDEF(CEE_BEQ_S,                    "beq.s",            Pop1+Pop1,          Push0,
ShortInlineBrTarget,IMacro,      1,  0xFF,    0x2E,    COND_BRANCH)

OPDEF(CEE_BGE_S,                    "bge.s",            Pop1+Pop1,          Push0,
ShortInlineBrTarget,IMacro,      1,  0xFF,    0x2F,    COND_BRANCH)

OPDEF(CEE_BGT_S,                    "bgt.s",            Pop1+Pop1,          Push0,
ShortInlineBrTarget,IMacro,      1,  0xFF,    0x30,    COND_BRANCH)

OPDEF(CEE_BLE_S,                    "ble.s",            Pop1+Pop1,          Push0,
ShortInlineBrTarget,IMacro,      1,  0xFF,    0x31,    COND_BRANCH)

OPDEF(CEE_BLT_S,                    "blt.s",            Pop1+Pop1,          Push0,
ShortInlineBrTarget,IMacro,      1,  0xFF,    0x32,    COND_BRANCH)

OPDEF(CEE_BNE_UN_S,                 "bne.un.s",         Pop1+Pop1,          Push0,
ShortInlineBrTarget,IMacro,      1,  0xFF,    0x33,    COND_BRANCH)

OPDEF(CEE_BGE_UN_S,                 "bge.un.s",         Pop1+Pop1,          Push0,
ShortInlineBrTarget,IMacro,      1,  0xFF,    0x34,    COND_BRANCH)

OPDEF(CEE_BGT_UN_S,                 "bgt.un.s",         Pop1+Pop1,          Push0,
ShortInlineBrTarget,IMacro,      1,  0xFF,    0x35,    COND_BRANCH)

OPDEF(CEE_BLE_UN_S,                 "ble.un.s",         Pop1+Pop1,          Push0,
ShortInlineBrTarget,IMacro,      1,  0xFF,    0x36,    COND_BRANCH)

OPDEF(CEE_BLT_UN_S,                 "blt.un.s",         Pop1+Pop1,          Push0,
ShortInlineBrTarget,IMacro,      1,  0xFF,    0x37,    COND_BRANCH)

OPDEF(CEE_BR,                       "br",               Pop0,               Push0,
InlineBrTarget,     IPrimitive,  1,  0xFF,    0x38,    BRANCH)

OPDEF(CEE_BRFALSE,                  "brfalse",          PopI,               Push0,
InlineBrTarget,     IPrimitive,  1,  0xFF,    0x39,    COND_BRANCH)

OPDEF(CEE_BRTRUE,                   "brtrue",           PopI,               Push0,
InlineBrTarget,     IPrimitive,  1,  0xFF,    0x3A,    COND_BRANCH)

OPDEF(CEE_BEQ,                      "beq",              Pop1+Pop1,          Push0,
InlineBrTarget,     IMacro,      1,  0xFF,    0x3B,    COND_BRANCH)

OPDEF(CEE_BGE,                      "bge",              Pop1+Pop1,          Push0,
InlineBrTarget,     IMacro,      1,  0xFF,    0x3C,    COND_BRANCH)

OPDEF(CEE_BGT,                      "bgt",              Pop1+Pop1,          Push0,
InlineBrTarget,     IMacro,      1,  0xFF,    0x3D,    COND_BRANCH)

OPDEF(CEE_BLE,                      "ble",              Pop1+Pop1,          Push0,
InlineBrTarget,     IMacro,      1,  0xFF,    0x3E,    COND_BRANCH)

OPDEF(CEE_BLT,                      "blt",              Pop1+Pop1,          Push0,
InlineBrTarget,     IMacro,      1,  0xFF,    0x3F,    COND_BRANCH)

OPDEF(CEE_BNE_UN,                   "bne.un",           Pop1+Pop1,          Push0,
InlineBrTarget,     IMacro,      1,  0xFF,    0x40,    COND_BRANCH)
```

```
OPDEF(CEE_BGE_UN,                     "bge.un",          Pop1+Pop1,          Push0,
InlineBrTarget,     IMacro,      1,  0xFF,    0x41,    COND_BRANCH)

OPDEF(CEE_BGT_UN,                     "bgt.un",          Pop1+Pop1,          Push0,
InlineBrTarget,     IMacro,      1,  0xFF,    0x42,    COND_BRANCH)

OPDEF(CEE_BLE_UN,                     "ble.un",          Pop1+Pop1,          Push0,
InlineBrTarget,     IMacro,      1,  0xFF,    0x43,    COND_BRANCH)

OPDEF(CEE_BLT_UN,                     "blt.un",          Pop1+Pop1,          Push0,
InlineBrTarget,     IMacro,      1,  0xFF,    0x44,    COND_BRANCH)

OPDEF(CEE_SWITCH,                     "switch",          PopI,               Push0,
InlineSwitch,       IPrimitive,  1,  0xFF,    0x45,    COND_BRANCH)

OPDEF(CEE_LDIND_I1,                   "ldind.i1",        PopI,               PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0x46,    NEXT)

OPDEF(CEE_LDIND_U1,                   "ldind.u1",        PopI,               PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0x47,    NEXT)

OPDEF(CEE_LDIND_I2,                   "ldind.i2",        PopI,               PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0x48,    NEXT)

OPDEF(CEE_LDIND_U2,                   "ldind.u2",        PopI,               PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0x49,    NEXT)

OPDEF(CEE_LDIND_I4,                   "ldind.i4",        PopI,               PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0x4A,    NEXT)

OPDEF(CEE_LDIND_U4,                   "ldind.u4",        PopI,               PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0x4B,    NEXT)

OPDEF(CEE_LDIND_I8,                   "ldind.i8",        PopI,
PushI8,      InlineNone,         IPrimitive,  1,  0xFF,    0x4C,    NEXT)

OPDEF(CEE_LDIND_I,                    "ldind.i",         PopI,               PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0x4D,    NEXT)

OPDEF(CEE_LDIND_R4,                   "ldind.r4",        PopI,
PushR4,      InlineNone,         IPrimitive,  1,  0xFF,    0x4E,    NEXT)

OPDEF(CEE_LDIND_R8,                   "ldind.r8",        PopI,
PushR8,      InlineNone,         IPrimitive,  1,  0xFF,    0x4F,    NEXT)

OPDEF(CEE_LDIND_REF,                  "ldind.ref",       PopI,
PushRef,     InlineNone,         IPrimitive,  1,  0xFF,    0x50,    NEXT)

OPDEF(CEE_STIND_REF,                  "stind.ref",       PopI+PopI,          Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0x51,    NEXT)

OPDEF(CEE_STIND_I1,                   "stind.i1",        PopI+PopI,          Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0x52,    NEXT)

OPDEF(CEE_STIND_I2,                   "stind.i2",        PopI+PopI,          Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0x53,    NEXT)

OPDEF(CEE_STIND_I4,                   "stind.i4",        PopI+PopI,          Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0x54,    NEXT)

OPDEF(CEE_STIND_I8,                   "stind.i8",        PopI+PopI8,         Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0x55,    NEXT)

OPDEF(CEE_STIND_R4,                   "stind.r4",        PopI+PopR4,         Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0x56,    NEXT)

OPDEF(CEE_STIND_R8,                   "stind.r8",        PopI+PopR8,         Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0x57,    NEXT)

OPDEF(CEE_ADD,                        "add",             Pop1+Pop1,          Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0x58,    NEXT)

OPDEF(CEE_SUB,                        "sub",             Pop1+Pop1,          Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0x59,    NEXT)

OPDEF(CEE_MUL,                        "mul",             Pop1+Pop1,          Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0x5A,    NEXT)
```

```
OPDEF(CEE_DIV,                     "div",           Pop1+Pop1,          Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0x5B,    NEXT)

OPDEF(CEE_DIV_UN,                  "div.un",        Pop1+Pop1,          Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0x5C,    NEXT)

OPDEF(CEE_REM,                     "rem",           Pop1+Pop1,          Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0x5D,    NEXT)

OPDEF(CEE_REM_UN,                  "rem.un",        Pop1+Pop1,          Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0x5E,    NEXT)

OPDEF(CEE_AND,                     "and",           Pop1+Pop1,          Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0x5F,    NEXT)

OPDEF(CEE_OR,                      "or",            Pop1+Pop1,          Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0x60,    NEXT)

OPDEF(CEE_XOR,                     "xor",           Pop1+Pop1,          Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0x61,    NEXT)

OPDEF(CEE_SHL,                     "shl",           Pop1+Pop1,          Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0x62,    NEXT)

OPDEF(CEE_SHR,                     "shr",           Pop1+Pop1,          Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0x63,    NEXT)

OPDEF(CEE_SHR_UN,                  "shr.un",        Pop1+Pop1,          Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0x64,    NEXT)

OPDEF(CEE_NEG,                     "neg",           Pop1,               Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0x65,    NEXT)

OPDEF(CEE_NOT,                     "not",           Pop1,               Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0x66,    NEXT)

OPDEF(CEE_CONV_I1,                 "conv.i1",       Pop1,               PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0x67,    NEXT)

OPDEF(CEE_CONV_I2,                 "conv.i2",       Pop1,               PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0x68,    NEXT)

OPDEF(CEE_CONV_I4,                 "conv.i4",       Pop1,               PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0x69,    NEXT)

OPDEF(CEE_CONV_I8,                 "conv.i8",       Pop1,
PushI8,      InlineNone,         IPrimitive,  1,  0xFF,    0x6A,    NEXT)

OPDEF(CEE_CONV_R4,                 "conv.r4",       Pop1,
PushR4,      InlineNone,         IPrimitive,  1,  0xFF,    0x6B,    NEXT)

OPDEF(CEE_CONV_R8,                 "conv.r8",       Pop1,
PushR8,      InlineNone,         IPrimitive,  1,  0xFF,    0x6C,    NEXT)

OPDEF(CEE_CONV_U4,                 "conv.u4",       Pop1,               PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0x6D,    NEXT)

OPDEF(CEE_CONV_U8,                 "conv.u8",       Pop1,
PushI8,      InlineNone,         IPrimitive,  1,  0xFF,    0x6E,    NEXT)

OPDEF(CEE_CALLVIRT,                "callvirt",      VarPop,
VarPush,     InlineMethod,       IObjModel,   1,  0xFF,    0x6F,    CALL)

OPDEF(CEE_CPOBJ,                   "cpobj",         PopI+PopI,          Push0,
InlineType,         IObjModel,   1,  0xFF,    0x70,    NEXT)

OPDEF(CEE_LDOBJ,                   "ldobj",         PopI,               Push1,
InlineType,         IObjModel,   1,  0xFF,    0x71,    NEXT)

OPDEF(CEE_LDSTR,                   "ldstr",         Pop0,
PushRef,     InlineString,       IObjModel,   1,  0xFF,    0x72,    NEXT)

OPDEF(CEE_NEWOBJ,                  "newobj",        VarPop,
PushRef,     InlineMethod,       IObjModel,   1,  0xFF,    0x73,    CALL)

OPDEF(CEE_CASTCLASS,               "castclass",     PopRef,
PushRef,     InlineType,         IObjModel,   1,  0xFF,    0x74,    NEXT)
```

```
OPDEF(CEE_ISINST,                    "isinst",          PopRef,             PushI,
InlineType,        IObjModel,   1,  0xFF,    0x75,    NEXT)

OPDEF(CEE_CONV_R_UN,                 "conv.r.un",       Pop1,
PushR8,      InlineNone,         IPrimitive,  1,  0xFF,    0x76,    NEXT)

OPDEF(CEE_UNUSED58,                  "unused",          Pop0,               Push0,
InlineNone,        IPrimitive,  1,  0xFF,    0x77,    NEXT)

OPDEF(CEE_UNUSED1,                   "unused",          Pop0,               Push0,
InlineNone,        IPrimitive,  1,  0xFF,    0x78,    NEXT)

OPDEF(CEE_UNBOX,                     "unbox",           PopRef,             PushI,
InlineType,        IPrimitive,  1,  0xFF,    0x79,    NEXT)

OPDEF(CEE_THROW,                     "throw",           PopRef,             Push0,
InlineNone,        IObjModel,   1,  0xFF,    0x7A,    THROW)

OPDEF(CEE_LDFLD,                     "ldfld",           PopRef,             Push1,
InlineField,       IObjModel,   1,  0xFF,    0x7B,    NEXT)

OPDEF(CEE_LDFLDA,                    "ldflda",          PopRef,             PushI,
InlineField,       IObjModel,   1,  0xFF,    0x7C,    NEXT)

OPDEF(CEE_STFLD,                     "stfld",           PopRef+Pop1,        Push0,
InlineField,       IObjModel,   1,  0xFF,    0x7D,    NEXT)

OPDEF(CEE_LDSFLD,                    "ldsfld",          Pop0,               Push1,
InlineField,       IObjModel,   1,  0xFF,    0x7E,    NEXT)

OPDEF(CEE_LDSFLDA,                   "ldsflda",         Pop0,               PushI,
InlineField,       IObjModel,   1,  0xFF,    0x7F,    NEXT)

OPDEF(CEE_STSFLD,                    "stsfld",          Pop1,               Push0,
InlineField,       IObjModel,   1,  0xFF,    0x80,    NEXT)

OPDEF(CEE_STOBJ,                     "stobj",           PopI+Pop1,          Push0,
InlineType,        IPrimitive,  1,  0xFF,    0x81,    NEXT)

OPDEF(CEE_CONV_OVF_I1_UN,            "conv.ovf.i1.un",  Pop1,               PushI,
InlineNone,        IPrimitive,  1,  0xFF,    0x82,    NEXT)

OPDEF(CEE_CONV_OVF_I2_UN,            "conv.ovf.i2.un",  Pop1,               PushI,
InlineNone,        IPrimitive,  1,  0xFF,    0x83,    NEXT)

OPDEF(CEE_CONV_OVF_I4_UN,            "conv.ovf.i4.un",  Pop1,               PushI,
InlineNone,        IPrimitive,  1,  0xFF,    0x84,    NEXT)

OPDEF(CEE_CONV_OVF_I8_UN,            "conv.ovf.i8.un",  Pop1,
PushI8,      InlineNone,         IPrimitive,  1,  0xFF,    0x85,    NEXT)

OPDEF(CEE_CONV_OVF_U1_UN,            "conv.ovf.u1.un",  Pop1,               PushI,
InlineNone,        IPrimitive,  1,  0xFF,    0x86,    NEXT)

OPDEF(CEE_CONV_OVF_U2_UN,            "conv.ovf.u2.un",  Pop1,               PushI,
InlineNone,        IPrimitive,  1,  0xFF,    0x87,    NEXT)

OPDEF(CEE_CONV_OVF_U4_UN,            "conv.ovf.u4.un",  Pop1,               PushI,
InlineNone,        IPrimitive,  1,  0xFF,    0x88,    NEXT)

OPDEF(CEE_CONV_OVF_U8_UN,            "conv.ovf.u8.un",  Pop1,
PushI8,      InlineNone,         IPrimitive,  1,  0xFF,    0x89,    NEXT)

OPDEF(CEE_CONV_OVF_I_UN,             "conv.ovf.i.un",   Pop1,               PushI,
InlineNone,        IPrimitive,  1,  0xFF,    0x8A,    NEXT)

OPDEF(CEE_CONV_OVF_U_UN,             "conv.ovf.u.un",   Pop1,               PushI,
InlineNone,        IPrimitive,  1,  0xFF,    0x8B,    NEXT)

OPDEF(CEE_BOX,                       "box",             Pop1,
PushRef,     InlineType,         IPrimitive,  1,  0xFF,    0x8C,    NEXT)

OPDEF(CEE_NEWARR,                    "newarr",          PopI,
PushRef,     InlineType,         IObjModel,   1,  0xFF,    0x8D,    NEXT)

OPDEF(CEE_LDLEN,                     "ldlen",           PopRef,             PushI,
InlineNone,        IObjModel,   1,  0xFF,    0x8E,    NEXT)
```

```
OPDEF(CEE_LDELEMA,                    "ldelema",          PopRef+PopI,        PushI,
InlineType,         IObjModel,   1,  0xFF,    0x8F,    NEXT)

OPDEF(CEE_LDELEM_I1,                  "ldelem.i1",        PopRef+PopI,        PushI,
InlineNone,         IObjModel,   1,  0xFF,    0x90,    NEXT)

OPDEF(CEE_LDELEM_U1,                  "ldelem.u1",        PopRef+PopI,        PushI,
InlineNone,         IObjModel,   1,  0xFF,    0x91,    NEXT)

OPDEF(CEE_LDELEM_I2,                  "ldelem.i2",        PopRef+PopI,        PushI,
InlineNone,         IObjModel,   1,  0xFF,    0x92,    NEXT)

OPDEF(CEE_LDELEM_U2,                  "ldelem.u2",        PopRef+PopI,        PushI,
InlineNone,         IObjModel,   1,  0xFF,    0x93,    NEXT)

OPDEF(CEE_LDELEM_I4,                  "ldelem.i4",        PopRef+PopI,        PushI,
InlineNone,         IObjModel,   1,  0xFF,    0x94,    NEXT)

OPDEF(CEE_LDELEM_U4,                  "ldelem.u4",        PopRef+PopI,        PushI,
InlineNone,         IObjModel,   1,  0xFF,    0x95,    NEXT)

OPDEF(CEE_LDELEM_I8,                  "ldelem.i8",        PopRef+PopI,
PushI8,      InlineNone,         IObjModel,   1,  0xFF,    0x96,    NEXT)

OPDEF(CEE_LDELEM_I,                   "ldelem.i",         PopRef+PopI,        PushI,
InlineNone,         IObjModel,   1,  0xFF,    0x97,    NEXT)

OPDEF(CEE_LDELEM_R4,                  "ldelem.r4",        PopRef+PopI,
PushR4,      InlineNone,         IObjModel,   1,  0xFF,    0x98,    NEXT)

OPDEF(CEE_LDELEM_R8,                  "ldelem.r8",        PopRef+PopI,
PushR8,      InlineNone,         IObjModel,   1,  0xFF,    0x99,    NEXT)

OPDEF(CEE_LDELEM_REF,                 "ldelem.ref",       PopRef+PopI,
PushRef,     InlineNone,         IObjModel,   1,  0xFF,    0x9A,    NEXT)

OPDEF(CEE_STELEM_I,                   "stelem.i",         PopRef+PopI+PopI,   Push0,
InlineNone,         IObjModel,   1,  0xFF,    0x9B,    NEXT)

OPDEF(CEE_STELEM_I1,                  "stelem.i1",        PopRef+PopI+PopI,   Push0,
InlineNone,         IObjModel,   1,  0xFF,    0x9C,    NEXT)

OPDEF(CEE_STELEM_I2,                  "stelem.i2",        PopRef+PopI+PopI,   Push0,
InlineNone,         IObjModel,   1,  0xFF,    0x9D,    NEXT)

OPDEF(CEE_STELEM_I4,                  "stelem.i4",        PopRef+PopI+PopI,   Push0,
InlineNone,         IObjModel,   1,  0xFF,    0x9E,    NEXT)

OPDEF(CEE_STELEM_I8,                  "stelem.i8",        PopRef+PopI+PopI8,  Push0,
InlineNone,         IObjModel,   1,  0xFF,    0x9F,    NEXT)

OPDEF(CEE_STELEM_R4,                  "stelem.r4",        PopRef+PopI+PopR4,  Push0,
InlineNone,         IObjModel,   1,  0xFF,    0xA0,    NEXT)

OPDEF(CEE_STELEM_R8,                  "stelem.r8",        PopRef+PopI+PopR8,  Push0,
InlineNone,         IObjModel,   1,  0xFF,    0xA1,    NEXT)

OPDEF(CEE_STELEM_REF,                 "stelem.ref",       PopRef+PopI+PopRef, Push0,
InlineNone,         IObjModel,   1,  0xFF,    0xA2,    NEXT)

OPDEF(CEE_UNUSED2,                    "unused",           Pop0,               Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xA3,    NEXT)

OPDEF(CEE_UNUSED3,                    "unused",           Pop0,               Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xA4,    NEXT)

OPDEF(CEE_UNUSED4,                    "unused",           Pop0,               Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xA5,    NEXT)

OPDEF(CEE_UNUSED5,                    "unused",           Pop0,               Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xA6,    NEXT)

OPDEF(CEE_UNUSED6,                    "unused",           Pop0,               Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xA7,    NEXT)

OPDEF(CEE_UNUSED7,                    "unused",           Pop0,               Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xA8,    NEXT)
```

```
OPDEF(CEE_UNUSED8,                  "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xA9,    NEXT)

OPDEF(CEE_UNUSED9,                  "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xAA,    NEXT)

OPDEF(CEE_UNUSED10,                 "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xAB,    NEXT)

OPDEF(CEE_UNUSED11,                 "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xAC,    NEXT)

OPDEF(CEE_UNUSED12,                 "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xAD,    NEXT)

OPDEF(CEE_UNUSED13,                 "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xAE,    NEXT)

OPDEF(CEE_UNUSED14,                 "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xAF,    NEXT)

OPDEF(CEE_UNUSED15,                 "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xB0,    NEXT)

OPDEF(CEE_UNUSED16,                 "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xB1,    NEXT)

OPDEF(CEE_UNUSED17,                 "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xB2,    NEXT)

OPDEF(CEE_CONV_OVF_I1,              "conv.ovf.i1",     Pop1,              PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0xB3,    NEXT)

OPDEF(CEE_CONV_OVF_U1,              "conv.ovf.u1",     Pop1,              PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0xB4,    NEXT)

OPDEF(CEE_CONV_OVF_I2,              "conv.ovf.i2",     Pop1,              PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0xB5,    NEXT)

OPDEF(CEE_CONV_OVF_U2,              "conv.ovf.u2",     Pop1,              PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0xB6,    NEXT)

OPDEF(CEE_CONV_OVF_I4,              "conv.ovf.i4",     Pop1,              PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0xB7,    NEXT)

OPDEF(CEE_CONV_OVF_U4,              "conv.ovf.u4",     Pop1,              PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0xB8,    NEXT)

OPDEF(CEE_CONV_OVF_I8,              "conv.ovf.i8",     Pop1,
PushI8,      InlineNone,         IPrimitive,  1,  0xFF,    0xB9,    NEXT)

OPDEF(CEE_CONV_OVF_U8,              "conv.ovf.u8",     Pop1,
PushI8,      InlineNone,         IPrimitive,  1,  0xFF,    0xBA,    NEXT)

OPDEF(CEE_UNUSED50,                 "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xBB,    NEXT)

OPDEF(CEE_UNUSED18,                 "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xBC,    NEXT)

OPDEF(CEE_UNUSED19,                 "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xBD,    NEXT)

OPDEF(CEE_UNUSED20,                 "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xBE,    NEXT)

OPDEF(CEE_UNUSED21,                 "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xBF,    NEXT)

OPDEF(CEE_UNUSED22,                 "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xC0,    NEXT)

OPDEF(CEE_UNUSED23,                 "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xC1,    NEXT)

OPDEF(CEE_REFANYVAL,                "refanyval",       Pop1,              PushI,
InlineType,         IPrimitive,  1,  0xFF,    0xC2,    NEXT)
```

```
OPDEF(CEE_CKFINITE,                    "ckfinite",        Pop1,
PushR8,      InlineNone,         IPrimitive,  1,  0xFF,    0xC3,    NEXT)

OPDEF(CEE_UNUSED24,                    "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xC4,    NEXT)

OPDEF(CEE_UNUSED25,                    "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xC5,    NEXT)

OPDEF(CEE_MKREFANY,                    "mkrefany",        PopI,              Push1,
InlineType,         IPrimitive,  1,  0xFF,    0xC6,    NEXT)

OPDEF(CEE_UNUSED59,                    "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xC7,    NEXT)

OPDEF(CEE_UNUSED60,                    "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xC8,    NEXT)

OPDEF(CEE_UNUSED61,                    "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xC9,    NEXT)

OPDEF(CEE_UNUSED62,                    "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xCA,    NEXT)

OPDEF(CEE_UNUSED63,                    "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xCB,    NEXT)

OPDEF(CEE_UNUSED64,                    "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xCC,    NEXT)

OPDEF(CEE_UNUSED65,                    "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xCD,    NEXT)

OPDEF(CEE_UNUSED66,                    "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xCE,    NEXT)

OPDEF(CEE_UNUSED67,                    "unused",          Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xCF,    NEXT)

OPDEF(CEE_LDTOKEN,                     "ldtoken",         Pop0,              PushI,
InlineTok,          IPrimitive,  1,  0xFF,    0xD0,    NEXT)

OPDEF(CEE_CONV_U2,                     "conv.u2",         Pop1,              PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0xD1,    NEXT)

OPDEF(CEE_CONV_U1,                     "conv.u1",         Pop1,              PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0xD2,    NEXT)

OPDEF(CEE_CONV_I,                      "conv.i",          Pop1,              PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0xD3,    NEXT)

OPDEF(CEE_CONV_OVF_I,                  "conv.ovf.i",      Pop1,              PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0xD4,    NEXT)

OPDEF(CEE_CONV_OVF_U,                  "conv.ovf.u",      Pop1,              PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0xD5,    NEXT)

OPDEF(CEE_ADD_OVF,                     "add.ovf",         Pop1+Pop1,         Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0xD6,    NEXT)

OPDEF(CEE_ADD_OVF_UN,                  "add.ovf.un",      Pop1+Pop1,         Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0xD7,    NEXT)

OPDEF(CEE_MUL_OVF,                     "mul.ovf",         Pop1+Pop1,         Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0xD8,    NEXT)

OPDEF(CEE_MUL_OVF_UN,                  "mul.ovf.un",      Pop1+Pop1,         Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0xD9,    NEXT)

OPDEF(CEE_SUB_OVF,                     "sub.ovf",         Pop1+Pop1,         Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0xDA,    NEXT)

OPDEF(CEE_SUB_OVF_UN,                  "sub.ovf.un",      Pop1+Pop1,         Push1,
InlineNone,         IPrimitive,  1,  0xFF,    0xDB,    NEXT)

OPDEF(CEE_ENDFINALLY,                  "endfinally",      Pop0,              Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xDC,    RETURN)
```

```
OPDEF(CEE_LEAVE,                      "leave",          Pop0,             Push0,
InlineBrTarget,     IPrimitive,  1,  0xFF,    0xDD,    BRANCH)

OPDEF(CEE_LEAVE_S,                    "leave.s",        Pop0,             Push0,
ShortInlineBrTarget,IPrimitive,  1,  0xFF,    0xDE,    BRANCH)

OPDEF(CEE_STIND_I,                    "stind.i",        PopI+PopI,        Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xDF,    NEXT)

OPDEF(CEE_CONV_U,                     "conv.u",         Pop1,             PushI,
InlineNone,         IPrimitive,  1,  0xFF,    0xE0,    NEXT)

OPDEF(CEE_UNUSED26,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xE1,    NEXT)

OPDEF(CEE_UNUSED27,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xE2,    NEXT)

OPDEF(CEE_UNUSED28,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xE3,    NEXT)

OPDEF(CEE_UNUSED29,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xE4,    NEXT)

OPDEF(CEE_UNUSED30,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xE5,    NEXT)

OPDEF(CEE_UNUSED31,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xE6,    NEXT)

OPDEF(CEE_UNUSED32,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xE7,    NEXT)

OPDEF(CEE_UNUSED33,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xE8,    NEXT)

OPDEF(CEE_UNUSED34,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xE9,    NEXT)

OPDEF(CEE_UNUSED35,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xEA,    NEXT)

OPDEF(CEE_UNUSED36,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xEB,    NEXT)

OPDEF(CEE_UNUSED37,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xEC,    NEXT)

OPDEF(CEE_UNUSED38,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xED,    NEXT)

OPDEF(CEE_UNUSED39,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xEE,    NEXT)

OPDEF(CEE_UNUSED40,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xEF,    NEXT)

OPDEF(CEE_UNUSED41,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xF0,    NEXT)

OPDEF(CEE_UNUSED42,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xF1,    NEXT)

OPDEF(CEE_UNUSED43,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xF2,    NEXT)

OPDEF(CEE_UNUSED44,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xF3,    NEXT)

OPDEF(CEE_UNUSED45,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xF4,    NEXT)

OPDEF(CEE_UNUSED46,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xF5,    NEXT)

OPDEF(CEE_UNUSED47,                   "unused",         Pop0,             Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xF6,    NEXT)
```

```
OPDEF(CEE_UNUSED48,                  "unused",           Pop0,               Push0,
InlineNone,         IPrimitive,  1,  0xFF,    0xF7,    NEXT)

OPDEF(CEE_PREFIX7,                   "prefix7",          Pop0,               Push0,
InlineNone,         IInternal,   1,  0xFF,    0xF8,    META)

OPDEF(CEE_PREFIX6,                   "prefix6",          Pop0,               Push0,
InlineNone,         IInternal,   1,  0xFF,    0xF9,    META)

OPDEF(CEE_PREFIX5,                   "prefix5",          Pop0,               Push0,
InlineNone,         IInternal,   1,  0xFF,    0xFA,    META)

OPDEF(CEE_PREFIX4,                   "prefix4",          Pop0,               Push0,
InlineNone,         IInternal,   1,  0xFF,    0xFB,    META)

OPDEF(CEE_PREFIX3,                   "prefix3",          Pop0,               Push0,
InlineNone,         IInternal,   1,  0xFF,    0xFC,    META)

OPDEF(CEE_PREFIX2,                   "prefix2",          Pop0,               Push0,
InlineNone,         IInternal,   1,  0xFF,    0xFD,    META)

OPDEF(CEE_PREFIX1,                   "prefix1",          Pop0,               Push0,
InlineNone,         IInternal,   1,  0xFF,    0xFE,    META)

OPDEF(CEE_PREFIXREF,                 "prefixref",        Pop0,               Push0,
InlineNone,         IInternal,   1,  0xFF,    0xFF,    META)


OPDEF(CEE_ARGLIST,                   "arglist",          Pop0,               PushI,
InlineNone,         IPrimitive,  2,  0xFE,    0x00,    NEXT)

OPDEF(CEE_CEQ,                       "ceq",              Pop1+Pop1,          PushI,
InlineNone,         IPrimitive,  2,  0xFE,    0x01,    NEXT)

OPDEF(CEE_CGT,                       "cgt",              Pop1+Pop1,          PushI,
InlineNone,         IPrimitive,  2,  0xFE,    0x02,    NEXT)

OPDEF(CEE_CGT_UN,                    "cgt.un",           Pop1+Pop1,          PushI,
InlineNone,         IPrimitive,  2,  0xFE,    0x03,    NEXT)

OPDEF(CEE_CLT,                       "clt",              Pop1+Pop1,          PushI,
InlineNone,         IPrimitive,  2,  0xFE,    0x04,    NEXT)

OPDEF(CEE_CLT_UN,                    "clt.un",           Pop1+Pop1,          PushI,
InlineNone,         IPrimitive,  2,  0xFE,    0x05,    NEXT)

OPDEF(CEE_LDFTN,                     "ldftn",            Pop0,               PushI,
InlineMethod,       IPrimitive,  2,  0xFE,    0x06,    NEXT)

OPDEF(CEE_LDVIRTFTN,                 "ldvirtftn",        PopRef,             PushI,
InlineMethod,       IPrimitive,  2,  0xFE,    0x07,    NEXT)

OPDEF(CEE_UNUSED56,                  "unused",           Pop0,               Push0,
InlineNone,         IPrimitive,  2,  0xFE,    0x08,    NEXT)

OPDEF(CEE_LDARG,                     "ldarg",            Pop0,               Push1,
InlineVar,          IPrimitive,  2,  0xFE,    0x09,    NEXT)

OPDEF(CEE_LDARGA,                    "ldarga",           Pop0,               PushI,
InlineVar,          IPrimitive,  2,  0xFE,    0x0A,    NEXT)

OPDEF(CEE_STARG,                     "starg",            Pop1,               Push0,
InlineVar,          IPrimitive,  2,  0xFE,    0x0B,    NEXT)

OPDEF(CEE_LDLOC,                     "ldloc",            Pop0,               Push1,
InlineVar,          IPrimitive,  2,  0xFE,    0x0C,    NEXT)

OPDEF(CEE_LDLOCA,                    "ldloca",           Pop0,               PushI,
InlineVar,          IPrimitive,  2,  0xFE,    0x0D,    NEXT)

OPDEF(CEE_STLOC,                     "stloc",            Pop1,               Push0,
InlineVar,          IPrimitive,  2,  0xFE,    0x0E,    NEXT)

OPDEF(CEE_LOCALLOC,                  "localloc",         PopI,               PushI,
InlineNone,         IPrimitive,  2,  0xFE,    0x0F,    NEXT)

OPDEF(CEE_UNUSED57,                  "unused",           Pop0,               Push0,
InlineNone,         IPrimitive,  2,  0xFE,    0x10,    NEXT)
```

```
OPDEF(CEE_ENDFILTER,                  "endfilter",        PopI,               Push0,
InlineNone,         IPrimitive,  2,  0xFE,    0x11,    RETURN)

OPDEF(CEE_UNALIGNED,                  "unaligned.",       Pop0,               Push0,
ShortInlineI,       IPrefix,     2,  0xFE,    0x12,    META)

OPDEF(CEE_VOLATILE,                   "volatile.",        Pop0,               Push0,
InlineNone,         IPrefix,     2,  0xFE,    0x13,    META)

OPDEF(CEE_TAILCALL,                   "tail.",            Pop0,               Push0,
InlineNone,         IPrefix,     2,  0xFE,    0x14,    META)

OPDEF(CEE_INITOBJ,                    "initobj",          PopI,               Push0,
InlineType,         IObjModel,   2,  0xFE,    0x15,    NEXT)

OPDEF(CEE_UNUSED68,                   "unused",           Pop0,               Push0,
InlineNone,         IPrimitive,  2,  0xFE,    0x16,    NEXT)

OPDEF(CEE_CPBLK,                      "cpblk",            PopI+PopI+PopI,     Push0,
InlineNone,         IPrimitive,  2,  0xFE,    0x17,    NEXT)

OPDEF(CEE_INITBLK,                    "initblk",          PopI+PopI+PopI,     Push0,
InlineNone,         IPrimitive,  2,  0xFE,    0x18,    NEXT)

OPDEF(CEE_UNUSED69,                   "unused",           Pop0,               Push0,
InlineNone,         IPrimitive,  2,  0xFE,    0x19,    NEXT)

OPDEF(CEE_RETHROW,                    "rethrow",          Pop0,               Push0,
InlineNone,         IObjModel,   2,  0xFE,    0x1A,    THROW)

OPDEF(CEE_UNUSED51,                   "unused",           Pop0,               Push0,
InlineNone,         IPrimitive,  2,  0xFE,    0x1B,    NEXT)

OPDEF(CEE_SIZEOF,                     "sizeof",           Pop0,               PushI,
InlineType,         IPrimitive,  2,  0xFE,    0x1C,    NEXT)

OPDEF(CEE_REFANYTYPE,                 "refanytype",       Pop1,               PushI,
InlineNone,         IPrimitive,  2,  0xFE,    0x1D,    NEXT)

OPDEF(CEE_UNUSED52,                   "unused",           Pop0,               Push0,
InlineNone,         IPrimitive,  2,  0xFE,    0x1E,    NEXT)

OPDEF(CEE_UNUSED53,                   "unused",           Pop0,               Push0,
InlineNone,         IPrimitive,  2,  0xFE,    0x1F,    NEXT)

OPDEF(CEE_UNUSED54,                   "unused",           Pop0,               Push0,
InlineNone,         IPrimitive,  2,  0xFE,    0x20,    NEXT)

OPDEF(CEE_UNUSED55,                   "unused",           Pop0,               Push0,
InlineNone,         IPrimitive,  2,  0xFE,    0x21,    NEXT)

OPDEF(CEE_UNUSED70,                   "unused",           Pop0,               Push0,
InlineNone,         IPrimitive,  2,  0xFE,    0x22,    NEXT)


// These are not real opcodes, but they are handy internally in the EE


OPDEF(CEE_ILLEGAL,                    "illegal",          Pop0,               Push0,
InlineNone,         IInternal,   0,  MOOT,    MOOT,    META)

OPDEF(CEE_MACRO_END,                  "endmac",           Pop0,               Push0,
InlineNone,         IInternal,   0,  MOOT,    MOOT,    META)




#ifndef OPALIAS

#define _OPALIAS_DEFINED_

#define OPALIAS(canonicalName, stringName, realOpcode)

#endif
```

```
OPALIAS(CEE_BRNULL,          "brnull",           CEE_BRFALSE)
OPALIAS(CEE_BRNULL_S,        "brnull.s",         CEE_BRFALSE_S)
OPALIAS(CEE_BRZERO,          "brzero",           CEE_BRFALSE)
OPALIAS(CEE_BRZERO_S,        "brzero.s",         CEE_BRFALSE_S)
OPALIAS(CEE_BRINST,          "brinst",           CEE_BRTRUE)
OPALIAS(CEE_BRINST_S,        "brinst.s",         CEE_BRTRUE_S)
OPALIAS(CEE_LDIND_U8,        "ldind.u8",         CEE_LDIND_I8)
OPALIAS(CEE_LDELEM_U8,       "ldelem.u8",        CEE_LDELEM_I8)
OPALIAS(CEE_LDC_I4_M1x,      "ldc.i4.M1",        CEE_LDC_I4_M1)
OPALIAS(CEE_ENDFAULT,        "endfault",         CEE_ENDFINALLY)

#ifdef _OPALIAS_DEFINED_
#undef OPALIAS
#undef _OPALIAS_DEFINED_
#endif
```

C.3. Complete Grammar

This grammar provides a number of ease-of-use features not provided in the grammar of Partition II, as well as supporting some features which are not portable across implementations and hence are not part of this standard. Unlike the grammar of Partition II, this one is designed for ease of programming rather than ease of reading; it can be converted directly into a YACC grammar.

```
Lexical tokens
    ID - C style alphaNumeric identifier (e.g. Hello_There2)
    QSTRING  - C style quoted string (e.g.  "hi\n")
    SQSTRING - C style singlely quoted string(e.g.  'hi')
    INT32    - C style 32 bit integer (e.g.  235,  03423, 0x34FFF)
    INT64    - C style 64 bit integer (e.g.  -2353453636235234,  0x34FFFFFFFFFF)
    FLOAT64  - C style floating point number (e.g.  -0.2323, 354.3423, 3435.34E-5)
    INSTR_*  - IL instructions of a particular class (see opcode.def).
----------------------------------------------------------------------------------
START           : decls
                ;

decls                   : /* EMPTY */
                        | decls decl
                        ;

decl                    : classHead '{' classDecls '}'
                        | nameSpaceHead '{' decls '}'
                        | methodHead  methodDecls '}'
                        | fieldDecl
                        | dataDecl
                        | vtableDecl
                        | vtfixupDecl
                        | extSourceSpec
                        | fileDecl
                        | assemblyHead '{' assemblyDecls '}'
                        | assemblyRefHead '{' assemblyRefDecls '}'
                        | comtypeHead '{' comtypeDecls '}'
                        | manifestResHead '{' manifestResDecls '}'
                        | moduleHead
                        | secDecl
                        | customAttrDecl
                                | '.subsystem' int32
                                | '.corflags' int32
                                | '.file' 'alignment' int32
                                | '.imagebase' int64
                                | languageDecl
                        ;

compQstring             : QSTRING
                        | compQstring '+' QSTRING
                                ;

languageDecl                    : '.language' SQSTRING
                        | '.language' SQSTRING ',' SQSTRING
                        | '.language' SQSTRING ',' SQSTRING ',' SQSTRING
                                ;

customAttrDecl          : '.custom' customType
                        | '.custom' customType '=' compQstring
                        | customHead bytes ')'
                        | '.custom' '(' ownerType ')' customType
                        | '.custom' '(' ownerType ')' customType '=' compQstring
                        | customHeadWithOwner bytes ')'
                        ;

moduleHead              : '.module'
                        | '.module' name1
                                | '.module' 'extern' name1
                        ;


vtfixupDecl             : '.vtfixup' '[' int32 ']' vtfixupAttr 'at' id
                        ;
```

```
vtfixupAttr             : /* EMPTY */
                        | vtfixupAttr 'int32'
                        | vtfixupAttr 'int64'
                        | vtfixupAttr 'fromunmanaged'
                        | vtfixupAttr 'callmostderived'
                        ;

vtableDecl              : vtableHead bytes ')'
                        ;

vtableHead              : '.vtable' '=' '('
                        ;

nameSpaceHead           : '.namespace' name1
                        ;

classHead               : '.class' classAttr id extendsClause implClause
                        ;

classAttr               : /* EMPTY */
                        | classAttr 'public'
                        | classAttr 'private'
                        | classAttr 'value'
                        | classAttr 'enum'
                        | classAttr 'interface'
                        | classAttr 'sealed'
                        | classAttr 'abstract'
                        | classAttr 'auto'
                        | classAttr 'sequential'
                        | classAttr 'explicit'
                        | classAttr 'ansi'
                        | classAttr 'unicode'
                        | classAttr 'autochar'
                        | classAttr 'import'
                        | classAttr 'serializable'
                        | classAttr 'nested' 'public'
                        | classAttr 'nested' 'private'
                        | classAttr 'nested' 'family'
                        | classAttr 'nested' 'assembly'
                        | classAttr 'nested' 'famandassem'
                        | classAttr 'nested' 'famorassem'
                        | classAttr 'beforefieldinit'
                        | classAttr 'specialname'
                        | classAttr 'rtspecialname'
                        ;

extendsClause           : /* EMPTY */
                        | 'extends' className
                        ;

implClause              : /* EMPTY */
                        | 'implements' classNames
                                                ;

classNames              : classNames ',' className
                        | className
                        ;

classDecls              : /* EMPTY */
                        | classDecls classDecl
                        ;

classDecl               : methodHead  methodDecls '}'
                        | classHead '{' classDecls '}'
                        | eventHead '{' eventDecls '}'
                        | propHead '{' propDecls '}'
                        | fieldDecl
                        | dataDecl
                        | secDecl
                        | extSourceSpec
                        | customAttrDecl
                        | '.size' int32
                        | '.pack' int32
                        | exportHead '{' comtypeDecls '}'
                        | '.override' typeSpec '::' methodName 'with' callConv type
typeSpec '::' methodName '(' sigArgs0 ')'
                                | languageDecl
                        ;

fieldDecl               : '.field' repeatOpt fieldAttr type id atOpt initOpt
                        ;


atOpt                   : /* EMPTY */
                        | 'at' id
                        ;

initOpt                 : /* EMPTY */
                        | '=' fieldInit
                                        ;

repeatOpt                       : /* EMPTY */
                        | '[' int32 ']'
                                        ;

customHead              : '.custom' customType '=' '('
                        ;

customHeadWithOwner     : '.custom' '(' ownerType ')' customType '=' '('
                        ;

memberRef                       : methodSpec callConv type typeSpec '::' methodName
'(' sigArgs0 ')'
                        | methodSpec callConv type methodName '(' sigArgs0 ')'
                        | 'field' type typeSpec '::' id
                        | 'field' type id
                        ;

customType              : callConv type typeSpec '::' '.ctor' '(' sigArgs0 ')'
                        | callConv type '.ctor' '(' sigArgs0 ')'
                        ;

ownerType               : typeSpec
                        | memberRef
                        ;

eventHead               : '.event' eventAttr typeSpec id
                        | '.event' eventAttr id
                        ;


eventAttr               : /* EMPTY */
                        | eventAttr 'rtspecialname' /**/
                        | eventAttr 'specialname'
                        ;

eventDecls              : /* EMPTY */
                        | eventDecls eventDecl
                        ;

eventDecl               : '.addon' callConv type typeSpec '::' methodName '('
sigArgs0 ')'
                        | '.addon' callConv type methodName '(' sigArgs0 ')'
                        | '.removeon' callConv type typeSpec '::' methodName '('
sigArgs0 ')'
                        | '.removeon' callConv type methodName '(' sigArgs0 ')'
                        | '.fire' callConv type typeSpec '::' methodName '('
sigArgs0 ')'
                        | '.fire' callConv type methodName '(' sigArgs0 ')'
                        | '.other' callConv type typeSpec '::' methodName '('
sigArgs0 ')'
                        | '.other' callConv type methodName '(' sigArgs0 ')'
                        | extSourceSpec
                        | customAttrDecl
                                | languageDecl
                        ;

propHead                : '.property' propAttr callConv type id '(' sigArgs0 ')'
initOpt
                        ;

propAttr                : /* EMPTY */
                        | propAttr 'rtspecialname' /**/
                        | propAttr 'specialname'
                        ;

propDecls               : /* EMPTY */
                        | propDecls propDecl
                        ;


propDecl                : '.set' callConv type typeSpec '::' methodName '(' sigArgs0
')'
                        | '.set' callConv type methodName '(' sigArgs0 ')'
                        | '.get' callConv type typeSpec '::' methodName '(' sigArgs0
')'
                        | '.get' callConv type methodName '(' sigArgs0 ')'
                        | '.other' callConv type typeSpec '::' methodName '('
sigArgs0 ')'
                        | '.other' callConv type methodName '(' sigArgs0 ')'
                        | customAttrDecl
                        | extSourceSpec
                                | languageDecl
                        ;
```

```
methodHeadPart1         : '.method'
                        ;

methodHead              : methodHeadPart1 methAttr callConv paramAttr type
methodName '(' sigArgs0 ')' implAttr '{'
                        | methodHeadPart1 methAttr callConv paramAttr type 'marshal'
'(' nativeType ')' methodName '(' sigArgs0 ')' implAttr '{'
                        ;


methAttr                : /* EMPTY */

                        | methAttr 'static'
                        | methAttr 'public'
                        | methAttr 'private'
                        | methAttr 'family'
                        | methAttr 'final'
                        | methAttr 'specialname'
                        | methAttr 'virtual'
                        | methAttr 'abstract'
                        | methAttr 'assembly'
                        | methAttr 'famandassem'
                        | methAttr 'famorassem'
                        | methAttr 'privatescope'
                        | methAttr 'hidebysig'
                        | methAttr 'newslot'
                        | methAttr 'rtspecialname' /**/
                        | methAttr 'unmanagedexp'
                        | methAttr 'reqsecobj'

                        | methAttr 'pinvokeimpl' '(' compQstring 'as' compQstring
pinvAttr ')'
                        | methAttr 'pinvokeimpl' '(' compQstring  pinvAttr ')'
                        | methAttr 'pinvokeimpl' '(' pinvAttr ')'
                        ;

pinvAttr                : /* EMPTY */
                        | pinvAttr 'nomangle'
                        | pinvAttr 'ansi'
                        | pinvAttr 'unicode'
                        | pinvAttr 'autochar'
                        | pinvAttr 'lasterr'
                        | pinvAttr 'winapi'
                        | pinvAttr 'cdecl'
                        | pinvAttr 'stdcall'
                        | pinvAttr 'thiscall'
                        | pinvAttr 'fastcall'
                        ;

methodName              : '.ctor'
                        | '.cctor'
                        | name1
                        ;

paramAttr               : /* EMPTY */
                        | paramAttr '[' 'in' ']'
                        | paramAttr '[' 'out' ']'
                        | paramAttr '[' 'opt' ']'
                        | paramAttr '[' int32 ']'
                        ;

fieldAttr               : /* EMPTY */
                        | fieldAttr 'static'
                        | fieldAttr 'public'
                        | fieldAttr 'private'
                        | fieldAttr 'family'
                        | fieldAttr 'initonly'
                        | fieldAttr 'rtspecialname'  /**/
                        | fieldAttr 'specialname'
                                /* commented out because PInvoke for fields
is not supported by EE
                        | fieldAttr 'pinvokeimpl' '(' compQstring 'as' compQstring
pinvAttr ')'
                        | fieldAttr 'pinvokeimpl' '(' compQstring  pinvAttr ')'
                        | fieldAttr 'pinvokeimpl' '(' pinvAttr ')'
                                */
                        | fieldAttr 'marshal' '(' nativeType ')'
                        | fieldAttr 'assembly'
                        | fieldAttr 'famandassem'
                        | fieldAttr 'famorassem'
                        | fieldAttr 'privatescope'
                        | fieldAttr 'literal'
                        | fieldAttr 'notserialized'
                        ;
```

```
implAttr                : /* EMPTY */
                        | implAttr 'native'
                        | implAttr 'cil'
                        | implAttr 'optil'
                        | implAttr 'managed'
                        | implAttr 'unmanaged'
                        | implAttr 'forwardref'
                        | implAttr 'preservesig'
                        | implAttr 'runtime'
                        | implAttr 'internalcall'
                        | implAttr 'synchronized'
                        | implAttr 'noinlining'
                        ;

localsHead              : '.locals'
                        ;


methodDecl              : '.emitbyte' int32
                        | sehBlock
                        | '.maxstack' int32
                        | localsHead '(' sigArgs0 ')'
                        | localsHead 'init' '(' sigArgs0 ')'
                        | '.entrypoint'
                        | '.zeroinit'
                        | dataDecl
                        | instr
                        | id ':'
                        | secDecl
                        | extSourceSpec
                                | languageDecl
                        | customAttrDecl
                                | '.export' '[' int32 ']'
                                | '.export' '[' int32 ']'     'as' id
                        | '.vtentry' int32 ':' int32
                        | '.override' typeSpec '::' methodName
                        | scopeBlock
                        | '.param' '[' int32 ']' initOpt
                        ;

scopeBlock              : scopeOpen methodDecls '}'
                        ;
```

```
scopeOpen               : '{'
                        ;

sehBlock                : tryBlock sehClauses
                        ;

sehClauses              : sehClause sehClauses
                        | sehClause
                        ;

tryBlock                : tryHead scopeBlock
                        | tryHead id 'to' id
                        | tryHead int32 'to' int32
                        ;

tryHead                 : '.try'
                        ;


sehClause               : catchClause handlerBlock
                        | filterClause handlerBlock
                        | finallyClause handlerBlock
                        | faultClause handlerBlock
                        ;


filterClause            : filterHead scopeBlock
                        | filterHead id
                        | filterHead int32
                        ;

filterHead              : 'filter'
                        ;

catchClause             : 'catch' className
                        ;

finallyClause           : 'finally'
                        ;

faultClause             : 'fault'
                        ;
handlerBlock            : scopeBlock
                        | 'handler' id 'to' id
                        | 'handler' int32 'to' int32
                        ;


methodDecls             : /* EMPTY */
                        | methodDecls methodDecl
                        ;

dataDecl                : ddHead ddBody
                        ;

ddHead                  : '.data' tls id '='
                        | '.data' tls
                        ;

tls                     : /* EMPTY */
                        | 'tls'
                        ;

ddBody                  : '{' ddItemList '}'
                        | ddItem
                        ;

ddItemList              : ddItem ',' ddItemList
                        | ddItem
                        ;

ddItemCount             : /* EMPTY */
                        | '[' int32 ']'
                        ;

ddItem                  : 'char' '*' '(' compQstring ')'
                        | '&' '(' id ')'
                        | bytearrayhead bytes ')'
                        | 'float32' '(' float64 ')' ddItemCount
                        | 'float64' '(' float64 ')' ddItemCount
                        | 'int64' '(' int64 ')' ddItemCount
                        | 'int32' '(' int32 ')' ddItemCount
                        | 'int16' '(' int32 ')' ddItemCount
                        | 'int8' '(' int32 ')' ddItemCount
                        | 'float32' ddItemCount
                        | 'float64' ddItemCount
                        | 'int64' ddItemCount
                        | 'int32' ddItemCount
                        | 'int16' ddItemCount
                        | 'int8' ddItemCount
                        ;

fieldInit               : 'float32' '(' float64 ')'
                        | 'float64' '(' float64 ')'
                        | 'float32' '(' int64 ')'
                        | 'float64' '(' int64 ')'
                        | 'int64' '(' int64 ')'
                        | 'int32' '(' int64 ')'
                        | 'int16' '(' int64 ')'
                        | 'char' '(' int64 ')'
                        | 'int8' '(' int64 ')'
                        | 'bool' '(' truefalse ')'
                        | compQstring
                        | bytearrayhead bytes ')'
                                        | 'nullref'
                        ;

bytearrayhead           : 'bytearray' '('
                        ;

bytes                                   : /* EMPTY */
                                        | hexbytes
                                        ;

hexbytes                : HEXBYTE
                        | hexbytes HEXBYTE
                        ;

instr_r_head            : INSTR_R '('
                        ;

instr_tok_head          : INSTR_TOK
                        ;

methodSpec              : 'method'
                        ;

instr                   : INSTR_NONE
```

```
                        | INSTR_VAR int32
                        | INSTR_VAR id
                        | INSTR_I int32
                        | INSTR_I8 int64
                        | INSTR_R float64
                        | INSTR_R int64
                        | instr_r_head bytes ')'
                        | INSTR_BRTARGET int32
                        | INSTR_BRTARGET id
                        | INSTR_METHOD callConv type typeSpec '::' methodName '('
sigArgs0 ')'
                        | INSTR_METHOD callConv type methodName '(' sigArgs0 ')'
                        | INSTR_FIELD type typeSpec '::' id
                        | INSTR_FIELD type id
                        | INSTR_TYPE typeSpec
                        | INSTR_STRING compQstring
                        | INSTR_STRING bytearrayhead bytes ')'
                        | INSTR_SIG callConv type '(' sigArgs0 ')'
                        | INSTR_RVA id
                        | INSTR_RVA int32
                        | instr_tok_head ownerType /* ownerType ::= memberRef |
typeSpec */
                        | INSTR_SWITCH '(' labels ')'
                        | INSTR_PHI int16s
                        ;

sigArgs0                : /* EMPTY */
                        | sigArgs1
                        ;

sigArgs1                : sigArg
                        | sigArgs1 ',' sigArg
                        ;

sigArg                  : '...'
                        | paramAttr type
                        | paramAttr type id
                        | paramAttr type 'marshal' '(' nativeType ')'
                        | paramAttr type 'marshal' '(' nativeType ')' id
                        ;

name1                   : id
                        | DOTTEDNAME
                        | name1 '.' name1
                        ;

className               : '[' name1 ']' slashedName
                        | '[' '.module' name1 ']' slashedName
                        | slashedName
                        ;

slashedName             : name1
                        | slashedName '/' name1
                        ;

typeSpec                : className
                        | '[' name1 ']'
                        | '[' '.module' name1 ']'
                        | type
                        ;

callConv                : 'instance' callConv
                        | 'explicit' callConv
                        | callKind
                        ;

callKind                : /* EMPTY */
                        | 'default'
                        | 'vararg'
                        | 'unmanaged' 'cdecl'
                        | 'unmanaged' 'stdcall'
                        | 'unmanaged' 'thiscall'
                        | 'unmanaged' 'fastcall'
                        ;

nativeType              : /* EMPTY */
                        | 'custom' '(' compQstring ',' compQstring ',' compQstring
',' compQstring ')'
                        | 'custom' '(' compQstring ',' compQstring ')'
                        | 'fixed' 'sysstring' '[' int32 ']'
                        | 'fixed' 'array' '[' int32 ']'
                        | 'variant'
                        | 'currency'
                        | 'syschar'
                        | 'void'
                        | 'bool'
```

```
                         | 'int8'
                         | 'int16'
                         | 'int32'
                         | 'int64'
                         | 'float32'
                         | 'float64'
                         | 'error'
                         | 'unsigned' 'int8'
                         | 'unsigned' 'int16'
                         | 'unsigned' 'int32'
                         | 'unsigned' 'int64'
                         | nativeType '*'
                         | nativeType '[' ']'
                         | nativeType '[' int32 ']'
                         | nativeType '[' int32 '+' int32 ']'
                         | nativeType '[' '+' int32 ']'
                                         | 'decimal'
                         | 'date'
                         | 'bstr'
                         | 'lpstr'
                         | 'lpwstr'
                         | 'lptstr'
                         | 'objectref'
                         | 'iunknown'
                         | 'idispatch'
                         | 'struct'
                         | 'interface'
                         | 'safearray' variantType
                         | 'safearray' variantType ',' compQstring

                         | 'int'
                         | 'unsigned' 'int'
                         | 'nested' 'struct'
                         | 'byvalstr'
                         | 'ansi' 'bstr'
                         | 'tbstr'
                         | 'variant' 'bool'
                         | methodSpec
                         | 'as' 'any'
                         | 'lpstruct'
                         ;

variantType              : /* EMPTY */
```

```
| 'null'
| 'variant'
| 'currency'
| 'void'
| 'bool'
| 'int8'
| 'int16'
| 'int32'
| 'int64'
| 'float32'
| 'float64'
| 'unsigned' 'int8'
| 'unsigned' 'int16'
| 'unsigned' 'int32'
| 'unsigned' 'int64'
| '*'
| variantType '[' ']'
| variantType 'vector'
| variantType '&'
| 'decimal'
| 'date'
| 'bstr'
| 'lpstr'
| 'lpwstr'
| 'iunknown'
| 'idispatch'
| 'safearray'
| 'int'
| 'unsigned' 'int'
| 'error'
| 'hresult'
| 'carray'
| 'userdefined'
| 'record'
| 'filetime'
| 'blob'
| 'stream'
| 'storage'
| 'streamed_object'
| 'stored_object'
| 'blob_object'
| 'cf'
| 'clsid'
```

```
                        ;

type                    : 'class' className
                                | 'object'
                                | 'string'
                        | 'value' 'class' className
                        | 'valuetype' className
                        | type '[' ']'
                        | type '[' bounds1 ']'
                                /* uncomment when and if this type is
supported by the Runtime
                        | type 'value' '[' int32 ']'
                        */
                                | type '&'
                        | type '*'
                        | type 'pinned'
                        | type 'modreq' '(' className ')'
                        | type 'modopt' '(' className ')'
                        | '!' int32
                        | methodSpec callConv type '*' '(' sigArgs0 ')'
                        | 'typedref'
                        | 'char'
                        | 'void'
                        | 'bool'
                        | 'int8'
                        | 'int16'
                        | 'int32'
                        | 'int64'
                        | 'float32'
                        | 'float64'
                        | 'unsigned' 'int8'
                        | 'unsigned' 'int16'
                        | 'unsigned' 'int32'
                        | 'unsigned' 'int64'
                        | 'native' 'int'
                        | 'native' 'unsigned' 'int'
                        | 'native' 'float'
                        ;

bounds1                 : bound
                        | bounds1 ',' bound
                        ;
```

```
bound                    : /* EMPTY */
                         | '...'
                         | int32
                         | int32 '...' int32
                         | int32 '...'
                         ;

labels                   : /* empty */
                         | id ',' labels
                         | int32 ',' labels
                         | id
                         | int32
                         ;


id                       : ID
                         | SQSTRING
                         ;

int16s                   : /* EMPTY */
                         | int16s int32
                         ;

int32                    : INT64
                         ;

int64                    : INT64
                         ;

float64                  : FLOAT64
                         | 'float32' '(' int32 ')'
                         | 'float64' '(' int64 ')'
                         ;

secDecl                  : '.permission' secAction typeSpec '(' nameValPairs ')'
                         | '.permission' secAction typeSpec
                         | psetHead bytes ')'
                         ;

psetHead                 : '.permissionset' secAction '=' '('
                         ;

nameValPairs             : nameValPair
```

```
                        | nameValPair ',' nameValPairs
                        ;

nameValPair             : compQstring '=' caValue
                        ;

truefalse                       : 'true'
                                | 'false'
                                ;

caValue                 : truefalse
                        | int32
                        | 'int32' '(' int32 ')'
                        | compQstring
                        | className '(' 'int8' ':' int32 ')'
                        | className '(' 'int16' ':' int32 ')'
                        | className '(' 'int32' ':' int32 ')'
                        | className '(' int32 ')'
                        ;

secAction               : 'request'
                        | 'demand'
                        | 'assert'
                        | 'deny'
                        | 'permitonly'
                        | 'linkcheck'
                        | 'inheritcheck'
                        | 'reqmin'
                        | 'reqopt'
                        | 'reqrefuse'
                        | 'prejitgrant'
                        | 'prejitdeny'
                        | 'noncasdemand'
                        | 'noncaslinkdemand'
                        | 'noncasinheritance'
                        ;

extSourceSpec           : '.line' int32 SQSTRING
                        | '.line' int32
                        | '.line' int32 ':' int32 SQSTRING
                        | '.line' int32 ':' int32
                        | P_LINE int32 QSTRING
                        ;
```

```
fileDecl                : '.file' fileAttr name1 fileEntry hashHead bytes ')'
fileEntry
                        | '.file' fileAttr name1 fileEntry
                        ;

fileAttr                : /* EMPTY */
                        | fileAttr 'nometadata'
                        ;

fileEntry               : /* EMPTY */
                        | '.entrypoint'
                        ;

hashHead                : '.hash' '=' '('
                        ;

assemblyHead            : '.assembly' asmAttr name1
                        ;

asmAttr                 : /* EMPTY */
                        | asmAttr 'noappdomain'
                        | asmAttr 'noprocess'
                        | asmAttr 'nomachine'
                        ;

assemblyDecls           : /* EMPTY */
                        | assemblyDecls assemblyDecl
                        ;

assemblyDecl            : '.hash' 'algorithm' int32
                        | secDecl
                        | asmOrRefDecl

                        ;

asmOrRefDecl            : publicKeyHead bytes ')'
                        | '.ver' int32 ':' int32 ':' int32 ':' int32
                        | '.locale' compQstring
                        | localeHead bytes ')'
                        | customAttrDecl
                        ;
```

```
publicKeyHead           : '.publickey' '=' '('
                        ;

publicKeyTokenHead      : '.publickeytoken' '=' '('
                        ;

localeHead              : '.locale' '=' '('
                        ;

assemblyRefHead         : '.assembly' 'extern' name1
                        | '.assembly' 'extern' name1 'as' name1
                        ;

assemblyRefDecls        : /* EMPTY */
                        | assemblyRefDecls assemblyRefDecl
                        ;

assemblyRefDecl         : hashHead bytes ')'
                        | asmOrRefDecl
                        | publicKeyTokenHead bytes ')'
                        ;

comtypeHead             : '.class' 'extern' comtAttr name1
                        ;

exportHead              : '.export' comtAttr name1
                        ;

comtAttr                : /* EMPTY */
                        | comtAttr 'private'
                        | comtAttr 'public'
                        | comtAttr 'nested' 'public'
                        | comtAttr 'nested' 'private'
                        | comtAttr 'nested' 'family'
                        | comtAttr 'nested' 'assembly'
                        | comtAttr 'nested' 'famandassem'
                        | comtAttr 'nested' 'famorassem'
                        ;

comtypeDecls            : /* EMPTY */
                        | comtypeDecls comtypeDecl
                        ;
```

```
comtypeDecl              : '.file' name1
                         | '.class' 'extern' name1
                         | '.class'  int32
                         | customAttrDecl
                         ;

manifestResHead          : '.mresource' manresAttr name1
                         ;

manresAttr               : /* EMPTY */
                         | manresAttr 'public'
                         | manresAttr 'private'
                         ;

manifestResDecls         : /* EMPTY */
                         | manifestResDecls manifestResDecl
                         ;

manifestResDecl          : '.file' name1 'at' int32
                         | '.assembly' 'extern' name1
                         | customAttrDecl
                         ;
```

C.4. Instruction Syntax

While each section specifies the exact list of instructions that are included in a grammar class, this information is subject to change over time. The precise format of an instruction can be found by combining the information in Section C.1 with the information in the following table:

Table 1: Instruction Syntax classes

Grammar Class	Format(s) Specified in Section C.1
`<instr_brtarget>`	`InlineBrTarget, ShortInlineBrTarget`
`<instr_field>`	`InlineField`
`<instr_i>`	`InlineI, ShortInlineI`
`<instr_i8>`	`InlineI8`
`<instr_method>`	`InlineMethod`
`<instr_none>`	`InlineNone`
`<instr_phi>`	`InlinePhi`
`<instr_r>`	`InlineR, ShortInlineR`
`<instr_rva>`	`InlineRVA`
`<instr_sig>`	`InlineSig`
`<instr_string>`	`InlineString`

```
<instr_switch>                  InlineSwitch
<instr_tok>                     InlineTok
<instr_type>                    InlineType
<instr_var>                     InlineVar, ShortInlineVar
```

C.4.1. Top-level Instruction Syntax

```
<instr> ::=
    <instr_brtarget> <int32>
  | <instr_brtarget> <label>
  | <instr_field> <type> [ <typeSpec> :: ] <id>
  | <instr_i> <int32>
  | <instr_i8> <int64>
  | <instr_method>
      <callConv> <type> [ <typeSpec> :: ]
          <methodName> ( <parameters> )
  | <instr_none>
  | <instr_phi> <int16>*
  | <instr_r> ( <bytes> )       // <bytes> represent the binary image of
                                // float or double (4 or 8 bytes,
                                // respectively)
  | <instr_r> <float64>
  | <instr_r> <int64>   // integer is converted to float
                                // with possible
                                // loss of precision
  | <instr_sig> <callConv> <type> ( <parameters> )
  | <instr_string> bytearray ( <bytes> )
  | <instr_string> <QSTRING>
  | <instr_switch> ( <labels> )
  | <instr_tok> field <type> [ <typeSpec> :: ] <id>
  | <instr_tok> b
      <callConv> <type> [ <typeSpec> :: ]
          <methodName> ( <parameters> )
  | <instr_tok> <typeSpec>
  | <instr_type> <typeSpec>
  | <instr_var> <int32>
  | <instr_var> <localname>
```

C.4.2. Instructions with no operand

These instructions require no operands, so they simply appear by themselves.

```
<instr> ::= <instr_none>
<instr_none> ::= // Derived from opcode.def
        add             | add.ovf     | add.ovf.un     | and        |
        arglist        | break       | ceq            | cgt         |
        cgt.un         | ckfinite    | clt            | clt.un      |
        conv.i         | conv.i1     | conv.i2        | conv.i4     |
        conv.i8        | conv.ovf.i  | conv.ovf.i.un  | conv.ovf.i1|
        conv.ovf.i1.un | conv.ovf.i2 | conv.ovf.i2.un | conv.ovf.i4|
        conv.ovf.i4.un | conv.ovf.i8 | conv.ovf.i8.un | conv.ovf.u |
        conv.ovf.u.un  | conv.ovf.u1 | conv.ovf.u1.un | conv.ovf.u2|
        conv.ovf.u2.un | conv.ovf.u4 | conv.ovf.u4.un | conv.ovf.u8|
        conv.ovf.u8.un | conv.r.un   | conv.r4        | conv.r8     |
        conv.u         | conv.u1     | conv.u2        | conv.u4     |
        conv.u8        | cpblk       | div            | div.un      |
        dup            | endfault    | endfilter      | endfinally |
        initblk        |             | ldarg.0        | ldarg.1     |
        ldarg.2        | ldarg.3     | ldc.i4.0       | ldc.i4.1    |
        ldc.i4.2       | ldc.i4.3    | ldc.i4.4       | ldc.i4.5    |
        ldc.i4.6       | ldc.i4.7    | ldc.i4.8       | ldc.i4.M1   |
        ldelem.i       | ldelem.i1   | ldelem.i2      | ldelem.i4   |
        ldelem.i8      | ldelem.r4   | ldelem.r8      | ldelem.ref |
        ldelem.u1      | ldelem.u2   | ldelem.u4      | ldind.i     |
        ldind.i1       | ldind.i2    | ldind.i4       | ldind.i8    |
        ldind.r4       | ldind.r8    | ldind.ref      | ldind.u1    |
        ldind.u2       | ldind.u4    | ldlen          | ldloc.0     |
        ldloc.1        | ldloc.2     | ldloc.3        | ldnull      |
        localloc       | mul         | mul.ovf        | mul.ovf.un |
        neg            | nop         | not            | or          |
        pop            | refanytype  | rem            | rem.un      |
        ret            | rethrow     | shl            | shr         |
        shr.un         | stelem.i    | stelem.i1      | stelem.i2   |
        stelem.i4      | stelem.i8   | stelem.r4      | stelem.r8   |
        stelem.ref     | stind.i     | stind.i1       | stind.i2    |
        stind.i4       | stind.i8    | stind.r4       | stind.r8    |
        stind.ref      | stloc.0     | stloc.1        | stloc.2     |
        stloc.3        | sub         | sub.ovf        | sub.ovf.un |
        tail.          | throw       | volatile.      | xor
```

Examples:

```
ldlen
not
```

C.4.3. Instructions that Refer to Parameters or Local Variables

These instructions take one operand, which references a parameter or local variable of the current method. The variable can be referenced by its number (starting with variable 0) or by name (if the names are supplied as part of a signature using the form that supplies both a type and a name).

```
<instr> ::= <instr_var> <int32> |
            <instr_var> <localname>
<instr_var> ::= // Derived from opcode.def
               | ldarg    | ldarg.s  | ldarga
     ldarga.s  | ldloc    | ldloc.s  | ldloca
     ldloca.s  | starg    | starg.s  | stloc
     stloc.s
```

Examples:

```
     stloc 0          // store into 0th local
     ldarg X3      // load from argument named X3
```

C.4.4. Instructions that Take a Single 32-bit Integer Argument

These instructions take one operand, which must be a 32-bit integer.

```
<instr> ::= <instr_i> <int32>
<instr_i> ::= // Derived from opcode.def
     ldc.i4 | ldc.i4.s | unaligned.
```

Examples:

```
     ldc.i4 123456  // Load the number 123456
     ldc.i4.s 10    // Load the number 10
```

C.4.5. Instructions that Take a Single 64-bit Integer Argument

These instructions take one operand, which must be a 64-bit integer.

```
<instr> ::= <instr_i8> <int64>
<instr_i8> ::= // Derived from opcode.def
     ldc.i8
```

Examples:

```
     ldc.i8 0x123456789AB
     ldc.i8 12
```

C.4.6. Instructions that Take a Single Floating Point Argument

These instructions take one operand, which must be a floating point number.

```
     <instr> ::= <instr_r> <float64> |
                 <instr_r> <int64>   |
                               <instr_r> ( <bytes> )  // <bytes> is binary image
<instr_r> ::= // Derived from opcode.def
```

```
ldc.r4 | ldc.r8
```

Examples:

```
ldc.r4 10.2
ldc.r4 10
ldc.r4 0x123456789ABCDEF
ldc.r8 (00 00 00 00 00 00 F8 FF)
```

C.4.7. Branch instructions

The assembler does not optimize branches. The branch must be specified explicitly as using either the short or long form of the instruction. If the displacement is too large for the short form, then the assembler will display an error.

```
<instr> ::=
     <instr_brtarget> <int32> |
     <instr_brtarget> <label>
<instr_brtarget> ::= // Derived from opcode.def
                           | beq    | beq.s    | bge    | bge.s    |
     bge.un    | bge.un.s   | bgt    | bgt.s    | bgt.un | bgt.un.s |
     ble       | ble.s      | ble.un | ble.un.s | blt    | blt.s    |
     blt.un    | blt.un.s   | bne.un | bne.un.s | br     | br.s     |
     brfalse   | brfalse.s  | brtrue | brtrue.s | leave  | leave.s
```

Example:

```
br.s 22
br foo
```

C.4.8. Instructions that Take a Method as an Argument

These instructions reference a method, either in another class (first instruction format) or in the current class (second instruction format).

```
<instr> ::=
   <instr_method>
     <callConv> <type> [ <typeSpec> :: ] <methodName> ( <parameters> )
<instr_method> ::= // Derived from opcode.def
     call  | callvirt | jmp | ldftn    | ldvirtftn       | newobj
```

Examples:

```
call instance int32 C.D.E::X(class W, native int)
ldftn vararg char F(...)   // Global Function F
```

C.4.9. Instructions that Take a Field of a Class as an Argument

These instructions reference a field of a class.

```
<instr> ::=
     <instr_field> <type> <typeSpec> :: <id>
<instr_field> ::= // Derived from opcode.def
     ldfld | ldflda | ldsfld | ldsflda | stfld | stsfld
```

Examples:

```
     ldfld native int X::IntField
     stsfld int32 Y::AnotherField
```

C.4.10. Instructions that Take a Type as an Argument

These instructions reference a type.

```
<instr> ::= <instr_type> <typeSpec>
<instr_type> ::= // Derived from opcode.def
     box     | castclass | cpobj    | initobj | isinst    |
     ldelema | ldobj     | mkrefany | newarr  | refanyval |
     sizeof  | stobj     | unbox
```

Examples:

```
     initobj [mscorlib]System.Console
     sizeof class X
```

C.4.11. Instructions that Take a String as an Argument

These instructions take a string as an argument.

```
<instr> ::= <instr_string> <QSTRING>
<instr_string> ::= // Derived from opcode.def
     ldstr
```

Examples:

```
     ldstr “This is a string”
     ldstr “This has a\nnewline in it”
```

C.4.12. Instructions that Take a Signature as an Argument

These instructions take a stand-alone signature as an argument.

```
<instr> ::= <instr_sig> <callConv> <type> ( <parameters> )
<instr_sig> ::= // Derived from opcode.def
     calli
```

Examples:

```
     calli class A.B(wchar *)
     calli vararg bool(int32[,] X, ...)
     // Returns a boolean, takes at least one argument. The first
```

```
// argument, named X, must be a two-dimensional array of
// 32-bit ints
```

C.4.13. Instructions that Take a Metadata Token as an Argument

This instruction takes a metadata token as an argument. The token can reference a type, a method, or a field of a class.

```
<instr> ::= <instr_tok> <typeSpec> |
            <instr_tok> method
               <callConv> <type> <typeSpec> :: <methodName>
                          ( <parameters> ) |
            <instr_tok> method
               <callConv> <type> <methodName>
                          ( <parameters> ) |
            <instr_tok> field <type> <typeSpec> :: <id>
<instr_tok> ::= // Derived from opcode.def
     ldtoken
```

Examples:

```
ldtoken class [mscorlib]System.Console
ldtoken method int32 X::Fn()
ldtoken method bool GlobalFn(int32 &)
ldtoken field class X.Y Class::Field
```

C.4.14. Switch instruction

The switch instruction takes a set of labels or decimal relative values.

```
<instr> ::= <instr_switch> ( <labels> )
<instr_switch> ::= // Derived from opcode.def
     switch
```

Examples:

```
switch (0x3, -14, Label1)
switch (5, Label2)
```

Annex D Class Library Design Guidelines

This chapter contains only informative text

This chapter describes the guidelines that were used in the design of the class libraries, including naming conventions and coding patterns. They are intended to give guidance to anyone who is extending the libraries, including:

- Implementers of the CLI who wish to extend the libraries beyond those specified in this Standard
- Implementers of libraries that will run on top of the CLI and wish their libraries to be consistent with the standard libraries
- Future standards efforts aimed at refining the existing libraries or defining additional libraries.

As with any set of guidelines, they should be applied with an eye toward the end goal of consistency but understanding that for functionality, performance, or external compatibility reasons they may require modification or simply prove inappropriate in particular cases. The guidelines should not be applied blindly, and they should be revisited periodically to ensure that they remain viable.

Throughout this chapter, we use the following convention:

- Do means that the described practice should be followed where possible
- Do not means that the described practice should be avoided where possible
- Consider means that the described practice is often helpful but there are common cases where it is impractical or inadvisable; thus, the practice should be carefully considered but may not be appropriate.

D.1. Naming Guidelines

One of the most important elements of predictability and discoverability in a managed class library is the use of a consistent naming pattern. Many of the most common user questions should not arise once these conventions are understood and widely used.

There are three elements of naming guidelines.

- **Case:** Use the correct capitalization style.
- **Mechanics:** Use nouns for classes, verbs for methods, etc.
- **Word Choice:** Use terms consistently across libraries.

The following section describes rules for case and mechanics, and some philosophy regarding word choice.

D.1.1. Capitalization Styles

The following section describes different ways of capitalizing identifiers. These terms will be referred to throughout the rest of this document.

D.1.1.1. Pascal Casing

This convention capitalizes the first character of each word as in the following example.

```
BackColor
```

D.1.1.2. Camel Casing

This convention capitalizes the first character of each word except the first word as in the following example. `backColor`

D.1.1.3. Upper Case

Only use all upper case letters for identifiers if it contains an abbreviation that is two characters long or less. Identifiers of three or more characters should us Pascal Casing.

```
System.IO
System.Web.UI
System.CodeDom
```

D.1.1.4. Capitalization summary

The following table describes the capitalization rules for different types of identifiers.

Type	Case	Notes
Class	PascalCase	
Enum values	PascalCase	
Enum type	PascalCase	
Events	PascalCase	
Exception class	PascalCase	Ends with the suffix Exception.
Final Static field	PascalCase	
Interface	PascalCase	Begins with the prefix **I.**
Method	PascalCase	
Namespace	PascalCase	
Property	PascalCase	
Public Instance Field	PascalCase	Rarely used, prefer properties.
Protected Instance Field	camelCase	Rarely used, prefer properties.
Parameter	camelCase	

D.1.2. Word Choice

- Do avoid using class names duplicated in heavily used namespaces. For example, do not use any of the following for a class name.

  ```
  System
  Collections
  Forms
  UI
  ```

- Do avoid using identifiers that conflict with the following keywords.

alias	and	ansi	as	assembly
auto	base	bool	boolean	byte
call	case	catch	char	class
const	current	date	decimal	declare

default	delegate	dim	do	double
each	else	elseif	end	enum
erase	error	eval	event	exit
extends	finalize	finally	float	for
friend	function	get	goto	handles
if	implements	import	imports	in
inherit	inherits	instanceof	int	integer
interface	is	let	lib	like
lock	long	loop	me	mod
module	namespace	new	next	not
nothing	null	object	on	or
overloads	override	overrides	package	private
property	protected	public	raise	readonly
redim	rem	resume	return	select
self	set	shared	short	single
static	step	stop	string	structure
sub	synchronize	synchronized	then	this
throw	to	try	typeof	unlock
until	use	uses	using	var
void	volatile	when	while	with
xor	FALSE	TRUE		

- Do not use abbreviations in identifiers (including parameter names).
- If you must use abbreviations, do use camelCasing for any abbreviation over two characters long, even if this is not the standard abbreviation.

D.1.3. Case Sensitivity

Do not use names that require case sensitivity. Components must be fully usable from both case-sensitive and case-insensitive languages. Since case-insensitive languages cannot distinguish between two names within the same context that differ only by case, components must avoid this situation.

- Do not have two namespaces whose names differ only by case

```
namespace ee.cummings;
namespace Ee.Cummings;
```

- Do not have a function with two parameters whose names differ only by case.

```
void foo(string a, string A)
```

- Do not have a namespace with two types whose names differ only by case.

```
System.Drawing.Point p;
System.Drawing.POINT pp;
```

- Do not have a type with two properties whose names differ only by case.

```
int Foo {get, set};
int FOO {get, set}
```

- Do not have a type with two methods whose names differ only by case.

  ```
  void foo();
  void Foo();
  ```

D.1.4. Avoiding Type Name Confusion

Different languages use different terms to identify the fundamental managed types. Designers must avoid using language-specific terminology. Follow the rules described in this section to avoid type name confusion.

- Do use semantically interesting names rather than type names.
- In the rare case that a parameter has no semantic meaning beyond its type, use a generic name. For example, a class that supports writing a variety of data types into a stream might have the following methods.

  ```
  void Write(double value);
  void Write(float value);
  void Write(long value);
  void Write(int value);
  void Write(short value);
  ```

  The above example is preferred to the following language-specific alternative.

  ```
  void Write(double doubleValue);
  void Write(float floatValue);
  void Write(long longValue);
  void Write(int intValue);
  void Write(short shortValue);
  ```

In the extremely rare case that it is necessary to have a uniquely-named method for each fundamental data type, do use the following **universal type** names.

C# type name	**ILAsm representation**	**Universal type name**
sbyte	int8	**SByte**
byte	unsigned int8	**Byte**
short	int16	**Int16**
ushort	unsigned int16	**UInt16**
int	int32	**Int32**
uint	unsigned int32	**UInt32**
long	int64	**Int64**
ulong	unsigned int64	**UInt64**
float	float32	**Single**
double	float64	**Double**
bool	int32	**Boolean**
char	unsigned int16	**Char**
string	System.String	**String**

object	System.Object	**Object**

A class that supports reading a variety of data types from a stream might have the following methods.

```
double ReadDouble();
float ReadSingle();
long ReadInt64();
int ReadInt32();
short ReadInt16();
```

The above example is preferred to the following language-specific alternative.

```
double ReadDouble();
float ReadFloat();
long ReadLong();
int ReadInt();
short ReadShort();
```

D.1.5. Namespaces

The following example illustrates the general rule for naming namespaces.

```
CompanyName.TechnologyName
```

Therefore, we should expect to see namespaces like the following.

```
Microsoft.Office
PowerSoft.PowerBuilder
```

- Do avoid the possibility of two published namespaces having the same name, by prefixing namespace names with a company name or other well-established brand. For example, `Microsoft.Office` for the Office Automation Classes provided by Microsoft.
- Do use PascalCasing, and separate logical components with periods (For example, `Microsoft.Office.PowerPoint)`. If your brand employs non-traditional casing, do follow the casing defined by your brand, even if it deviates from normal namespace casing (For example, `NeXT.WebObjects`, and `ee.cummings`).
- Do use plural namespace names where appropriate. For example, use `System.Collections` not `System.Collection`. Exceptions to this rule are brand names and abbreviations. For example, use `System.IO` not `System.IOs`.
- Do not specify the same name for namespaces and classes. For example, do not use `Debug` for a namespace name and also provide a class named `Debug`.

D.1.6. Classes

- Do name classes with nouns or noun phrases.
- Do use PascalCasing.
- Do use abbreviations in class names sparingly.
- Do not use any type of class prefix (such as **C**).
- Do not use the underscore character.

- Occasionally, it is necessary to have a class name that begins with **I,** that is not an interface. This is acceptable as long as the character that follows **I** is lower case (For example, IdentityStore).

The following are examples of correctly named classes.

```
public class FileStream

{

}

public class Button

{

}

public class String

{

}
```

D.1.7. Interfaces

- Do name interfaces with nouns or noun phrases, or adjectives describing behaviour. For example, `IComponent` (descriptive noun), `ICustomAttributeProvider` (noun phrase), and `IPersistable` (adjective) are appropriate interface names.
- Do use **`PascalCasing`**.
- Do use abbreviations in interface names sparingly.
- Do not use the underscore character.
- Do prefix interface names with the letter **I**, to indicate that the type is an interface.
- Do use similar names when defining a class/interface pair where the class is a standard implementation of the interface. The names should differ only by the letter **I** prefix on the interface name.

The following example illustrates these guidelines for the interface `IComponent` and its standard implementation, the class `Component`.

```
public interface IComponent

{

}

public class Component : IComponent

{

}

public interface IServiceProvider

{

}

public interface IFormattable

{

}
```

D.1.8. Attributes

- Do add the **Attribute** suffix to custom attribute classes as in the following example.

  ```
  public class ObsoleteAttribute
  ```

```
{
}
```

D.1.9. Enums

- Do use PascalCasing for an `enum` type.
- Do use PascalCasing for an `enum` value name.
- Do use abbreviations in `enum` names sparingly.
- Do not use a prefix on `enum` names (For example, adXXX for ADO enums, rtfXXX for rich text enums, etc.).
- Do not use an `Enum` suffix on `enum` types.
- Do use a singular name for an `enum`.
- Do use a plural name for bit fields.

D.1.10. Fields

- Do use camelCasing (except for static fields, see clause D.1.10.1).
- Do not abbreviate field names.

  Spell out all the words used in a field name. Only use abbreviations if developers generally understand them. Do not use uppercase letters for field names. For example:

  ```
  class Foo
  {
     string url;
     string destinationUrl;
  }
  ```
- Do not use Hungarian notation for field names. Good names describe semantics, not type.
- Do not use a prefix for field names.
- Do not include a prefix on a field name, for example 'g_' or 's_' to distinguish static vs. non-static fields.

D.1.10.1. Static Fields

- Do name static fields with nouns, noun phrases, or abbreviations for nouns.
- Do not use a prefix for static field names.
- Do name static fields with PascalCasing.
- Do not prefix static field names with Hungarian type notation.

D.1.11. Parameter Names

- Do use descriptive parameter names. Parameter names should be descriptive enough that in most scenarios the name of the parameter and its type can be used to determine its meaning.
- Do name parameters with camelCasing.
- Do use names based on a parameter’s meaning rather than names based on the parameter’s type. We expect development tools to provide the information about type in

a useful manner, so the parameter name can be put to better use describing semantics rather than type. Occasional use of type-based parameter names is entirely appropriate.

- Do not use **reserved** parameters. If more data is needed in the next version, a new overload can be added.
- Do not prefix parameter names with Hungarian type notation.

```
Type GetType (string typeName)
string Format (string format, object [] args)
```

D.1.12. Method Names

- Do name methods with PascalCasing as in the following examples.

```
RemoveAll()
GetCharArray()
Invoke()
```

- Do not use Hungarian notation.
- Do name methods with verbs or verb phrases.

D.1.13. Property Names

- Do name properties using a noun or noun phrase.
- Do name properties with PascalCasing.
- Do not use Hungarian notation.

D.1.14. Event Names

- Do name events using PascalCasing.
- Do not use Hungarian notation.
- Do name event handlers (delegate types) with the **`EventHandler`** suffix as in the following example.

```
public delegate void MouseEventHandler(object sender, MouseEvent e);
```

- Consider using two parameters named **`sender`** and **`e`**.

  The sender parameter represents the object that raised the event. The sender parameter is always of type **`object`**, even if it is possible to employ a more specific type.

  The state associated with the event is encapsulated in an instance of an event class named **`e`**. Use an appropriate and specific event class for its type.

```
public delegate void MouseEventHandler(object sender, MouseEvent e);
```

- Do name event argument classes with the **`EventArgs`** suffix as in the following example.

```
public class MouseEventArgs : EventArgs
{
   int x;
   int y;
   public MouseEventArgs(int x, int y)
       { this.x = x; this.y = y; }
   public int X { get { return x; } }
   public int Y { get { return y; } }
}
```

- Do name event names that have a concept of pre and post using the present and past tense (do not use the BeforeXxx\AfterXxx pattern). For example, a close event that can be canceled would have a Closing and Closed event.
- Consider naming events with a verb.

D.2. Type Member Usage Guidelines

D.2.1. Property Usage Guidelines

- Do see clause D.2.1.1 on choosing between properties and methods.
- Do not use properties and types with the same name.

  Defining a property with the same name as a type can cause ambiguity in some programming languages. It is best to avoid this ambiguity unless there is a clear justification for not doing so.
- Do preserve the previous value if a property set throws an exception.
- Do allow properties to be set in any order. Properties should be stateless with respect to other properties.

  It is often the case that a particular feature of an object will not take effect until the developer specifies a particular set of properties, or until an object has a particular state. Until the object is in the correct state, the feature is not active. When the object is in the correct state, the feature automatically activates itself without requiring an explicit call. The semantics are the same regardless of the order in which the developer sets the property values or how the developer gets the object into the active state.

D.2.1.1. Properties vs. Methods

Library designers sometimes face a decision between a property and a method. Use the following guidelines to help you choose between these options. The philosophy here is that users will think of properties as though they were fields, hence methods are preferred where the intuitive semantics or performance differ from those of fields.

- Do use a property if the member has a logical backing store.
- Do use a method in the following situations.
  - The operation is a conversion (such as `Object.ToString()`)
  - The operation is expensive (orders of magnitude slower than a field set would be).
  - Obtaining a property value using the `Get` accessor has an observable side effect.
  - Calling the member twice in succession results in different results.
  - The order of execution relative to other properties is important.
  - The member is static but returns a mutable value.
  - The member returns an array.

    Properties that return arrays can be very misleading. Usually it is necessary to return a copy of the internal array so that the user cannot change internal state. This, coupled with the fact that a user could easily assume it is an indexed property, leads to inefficient code. In the following example, each call to the Methods property creates a copy of the array. That would be 2n+1 copies for this loop.

    ```
    Type type = //get a type somehow
    for (int i = 0; i < type.Methods.Length; i++)
    ```

```
{
   if (type.Methods[i].Name.Equals ("foo"))
   {...}
}
```

D.2.1.2. Read-Only and Write-Only Properties

- Do use Read-only properties when the user cannot change the logical backing data field.
- Do not use Write-only properties.

D.2.1.3. Indexed Property Usage

- Do use only one indexed property per class and make it the default indexed property for that class.
- Do not use non-default indexed properties.
- Do use the name **`Item`** for indexed properties unless there is an obviously better name (for example, a **`Chars`** property on **`string`** is better than **`Item`**).
- Do use indexed properties when the logical backing store is an array.
- Do not provide both indexed properties and methods that are semantically equivalent to two or more overloaded methods.

```
MethodInfo Type.Method[string name]       ;; Should be method
MethodInfo Type.GetMethod (string name, boolean ignoreCase)
```

D.2.2. Event Usage Guidelines

- Do use the "raise" terminology for events rather than "fire" or “trigger” terminology.
- Do use a return type of void for event handlers.
- Do make Event classes extend the class `System.EventArgs`
- Do implement `AddOn<EventName>` and `RemoveOn<EventName>` for each event.
- Do use a **`family virtual`** method to raise each event.

  This is not appropriate for sealed classes, because classes cannot be derived from them. The purpose of the method is to provide a way for a derived class to handle the event using an override. This is more natural than using delegates in the case where the developer is creating a derived class.

  The derived class can choose not to call the base during the processing of **`On<EventName>`**. Be prepared for this by not including any processing in the **`On<EventName>`** method that is required for the base class to work correctly.
- Do assume that anything can go in an event handler.

  Classes are ready for the handler of the event to do almost anything, and in all cases the object is left in a good state after the event has been raised. Consider using a try/finally block at the point where the event is raised. Since the developer can call back on the object to perform other actions, do not assume anything about the object state when control returns to the point at which the event was raised

D.2.3. Method Usage Guidelines

- Do use non-virtual methods unless overriding is intended by the design. Providing the ability to override a method (i.e. making the method virtual) implies that the design of the type is independent of details of the method's implementation; this is rarely true without careful design of the type.

- Do use method overloading when you provide different methods that do semantically the same thing.

- Do favor method overloading to default arguments. Default arguments are not allowed in the common language specification (CLS).

```
int String.IndexOf (String name);

int String.IndexOf (String name, int startIndex);
```

- Do use default values correctly.
  In a family of overloaded methods the complex method should use parameter names that indicate a change from the default state assumed in the simple method.
  For example, in the code below, the first method assumes the look-up will not be case sensitive. In method two we use the name *ignoreCase* rather than *caseSensitive* because the former indicates how we are changing the default behavior.

```
MethodInfo Type.GetMethod(String name);  //ignoreCase = false

MethodInfo Type.GetMethod (String name, boolean ignoreCase);
```

  It is very common to use a zero'ed state for the default value (such as: 0, 0.0, false, "", etc).

- Do be consistent in the ordering and naming of method parameters.

  It is common to have a set of overloaded methods with an increasing number of parameters to allow the developer to specify a desired level of information. The more parameters specified, the more detail that is specified. All the related methods have a consistent parameter order and naming pattern. Each of the method variations has the same semantics for their shared set of parameters.

  This consistency is useful even if the parameters have different types.

  The only method in such a group that should be virtual is the one that has the most parameters.

- Do use method overloading for variable numbers of parameters.

  Where it is appropriate to have variable numbers of parameters to a method, use the convention of declaring N methods with increasing numbers of parameters, and also provide a method which takes an array of values for numbers greater than N. N=3 or N=4 is appropriate for most cases. Only the method that takes the array should be virtual.

- Do make only the most complete overload virtual (if extensibility is needed) and define the other operations in terms of it.

```
public int IndexOf (string s)

{ return IndexOf (s, 0); }

public int IndexOf (string s, int start)

{ return IndexOf (s, startIndex, s.Length); }

public virtual int IndexOf (string s, int start, int count)

{ //do real work }
```

- Do use the `ParamsAttribute` pattern for defining methods with a variable number of arguments.

```
void Format (string formatString, params object [] args)
```

- Consider using the varargs ("…") calling convention to provide variable number of arguments, but do not use this without providing an alternate mechanism to accomplish the same thing since it is not CLS compliant.

- Consider providing special-case code for a small number of arguments to a method that takes a variable number of arguments, but only where the performance gained is significant. When this approach is taken it becomes difficult to allow the method to be overridden because all the special cases must be overridden as well.

D.2.4. Constructor Usage Guidelines

- Do have only a default **`private`** constructor (or no constructor at all) if there are only static methods and properties on a class.
- Do minimal work in the constructor.
- Do provide a **`family`** constructor that can be used by types in a derived class.
- Do not provide an empty default constructor for value types.
- Do use parameters in constructors as shortcuts for setting properties.

  There should be no difference in semantics between using the empty constructor followed by some calls to property setters, and using a constructor with multiple arguments.
- Do be consistent in the ordering and naming of constructor parameters.

  A common pattern for constructor parameters is to provide an increasing number of parameters to allow the developer to specify a desired level of information. The more parameters that are specified, the more detail that is specified. For all of the following constructors, there is a consistent order and naming of the parameters.

D.2.5. Field Usage Guidelines

- Do not use instance fields that are **`public`** or **`family`**.
- Consider providing `get` and `set` property accessors for fields instead of making them `public`.
- Do use a `family` property that returns the value of a `private` field to expose a field to a derived class. By not exposing fields directly to the developer, the class can be versioned more easily for the following reasons:

  a. A field cannot be changed to a property while maintaining binary compatibility.

  b. The presence of executable code in `get` and `set` property accessors allows later improvements, such as demand-creation of an object upon usage of the property, or a property change notification.
- Do use **`readonly static`** fields instead of properties where the value is a global constant.
- Do not use **`literal static`** fields if the value can change between versions.
- Do use **`public static readonly`** fields for predefined object instances.

D.2.6. Parameter Usage Guidelines

- Do check arguments for validity.

  Perform argument validation for every **`public`** or **`family`** method and property **`set`** accessor, and throw meaningful exceptions to the developer. The `System.ArgumentException` exception, or one of its subclasses, is used in these cases.

  Note that the actual checking does not necessarily have to happen in the **`public`**/**`family`** method itself. It could happen at a lower level in some **`private`** routines. The main point is that the entire surface area that is exposed to developers checks for valid arguments.

Parameter validation should occur before any side-effects are performed.

D.3. Type Usage Guidelines

D.3.1. Class Usage Guidelines

- Do favor using classes over any other type (i.e. interfaces or value types)

D.3.1.1. Base Class Usage Guidelines

Base classes are a useful way to group objects that share a common set of functionality. Base classes can provide a default set of functionality, while allowing customization though extension.

Add extensibility or polymorphism to your design only if you have a clear customer scenario for it.

- Do use base classes rather than interfaces.

  From a versioning perspective, interfaces are less flexible than classes. With a class, you can ship Version 1.0 and then in Version 2.0 decide to add another method. As long as the method is not abstract (that is, as long as you provide a default implementation of the method), any existing derived classes continue to function unchanged.

  Because interfaces do not support implementation inheritance, the pattern that applies to classes does not apply to interfaces. Adding a method to an interface is like adding an abstract method to a base class:any class that implements the interface will break because the class does not implement the interface's new method.

  Interfaces are appropriate in the following situations:

  o Several unrelated classes want to support the protocol.

  o These classes already have established base classes.

  o Aggregation is not appropriate or practical.

  For all other cases, class inheritance is a better model. For example, make **`IByteStream`** an interface so a class can implement multiple stream types. Make **`ValueEditor`** an abstract class because classes derived from **`ValueEditor`** have no other purpose than to edit values.

- Do provide customization through `family` methods.

  The public interface of a base class should provide a rich set of functionality for the consumer of that class. However, customizers of that class often want to implement the fewest methods possible to provide that rich set of functionality to the consumer. To meet this goal, provide a set of non-virtual or final public methods that call through to a single family method with the **`Impl`** suffix that provides implementations for such a method. This pattern is also known as the "Template Method".

```
Public Control
{ public void SetBounds(int x, int y, int width, int height)
  { . . .
    SetBoundsImpl (…);
   }

   public void SetBounds(int x, int y,
                         int width, int height,
                         BoundsSpecified specified)
   { . . .
     SetBoundsImpl (…);
   }
protected virtual void SetBoundsImpl
                       (int x, int y,
                        int width, int height,
                        BoundsSpecified specified)
 { // Do the real work here.
  }
}
```

- Do define a `family` constructor on all `abstract` classes. Many compilers will insert a `public` constructor if you do not. This can be very misleading to users as it can only be called from derived classes.

D.3.1.2. Sealed Class Usage Guidelines

- Do use **`sealed`** classes if creating derived classes will not be required.
- Do use **`sealed`** classes if there are only **`static`** methods and properties on a class.

D.3.2. Value Type Usage Guidelines

- Do use a value type for types that meet all of the following criteria.
  - Act like built-in types.
  - Have an instance size under 16 bytes.
  - Value semantics are desirable.
- Do not provide a default constructor.
- Do program assuming a state where all instance data is set to zero, false, or null (as appropriate) is valid, since this will be the state if no constructor is run and there is no guarantee that a constructor will be run (unlike for classes).

D.3.2.1. Enum Usage Guidelines

- Do use an `Enum` to strongly type parameters, property and return type. This allows development tools to know the possible values for a property or parameter.
- Do use the `System.Flags` custom attribute for an **`enum`** if a bitwise OR operation is to be performed on the numeric values.
- Do use **`int32`** as the underlying type of an **`enum`**.
  An exception to this rule is if the **`enum`** represents flags and there are many flags (>32) or the **`enum`** may grow to many flags in the future or the type needs to be different than type **`int32`** for backwards compatibility.
- Do use an **`enum`** with flags attribute only if the value can be completely expressed as a set of bitflags. Do not use an **`enum`** for open sets (eg., a version number).
- Do not assume **`enum`** arguments will be in the defined range Do argument validation.
- Do favor using an **`enum`** over static final constants.
- Do use **`int32`** as the underlying type of an **`enum`** unless either of the following is true.
  a. The **`enum`** represents flags, and there are currently many flags (>32), or the **`enum`** may grow to many flags in the future.
  b. The type needs to be different than **`int`** for backward compatibility.
- Do not use a non-integral **`enum`** type. Only use **`int8`**, **`int16`**, **`int32`**, or **`int64`**.

- Do not define methods, properties or events on an `enum`.
- Do not use any suffix on `enum` types.

D.3.3. Interface Usage Guidelines

See introductory paragraph of clause D.3.1.

- Do use a class or abstract class in preference to an interface, where possible.
- Do use interfaces to provide extensibility and the ability to customize.
- Do provide a default implementation of an interface where it is appropriate. For example, `System.Collections.DictionaryBase` is the default implementation of the `System.Collections.IDictionary` interface.
- Do see clause D.3.1.1 on the versioning issues with interfaces and abstract classes.
- Do not use interfaces as empty markers. Use Custom Attributes instead.

  If you need to mark a class as having a specific attribute (such as immutable or serializable) use a custom attribute rather than an interface.
- Do implement interfaces using “method impls” (see Partition II) and **`private virtual`** methods if you only want the interface methods available when cast to that interface. This is particularly useful when a class or value type implements an internal interface that is of no interest to a consumer of the class or value type.

D.3.4. Delegate Usage Guidelines

Delegates are a powerful tool that allow the managed code object model designer to encapsulate method calls. They are used in two basic areas.

Event notifications

See clause D.2.2 on event usage guidelines.

Callbacks

Passed to a method so that user code can be called multiple times during execution to provide customization. The classic example of this is passing a Compare callback to a sort routine. These methods should use the Callback conventions

- Do use an Event design pattern for events (even if it is not user interface related).

D.3.5. Attribute Classes

The CLI enables developers to invent new kinds of declarative information, to specify declarative information for various program entities, and to retrieve attribute information in a runtime environment. New kinds of declarative information are defined through the declaration of attribute classes, which may have positional and named parameters.

- Do specify an **`AttributeUsage`** on your attributes to define their usage precisely.
- Do seal attribute classes if possible.
- Do provide a single constructor for the attribute.
- Do use a parameter to the attribute’s constructor when the value of that parameter is always required to make the attribute.
- Do use a field on an attribute when the value of that property can be optionally specified to make the attribute.

- Do not name a parameter to the constructor with the same name as a field or property of the attribute.
- Do provide a read-only property with the same name (different casing) as each parameter to the constructor.
- Do provide a read-write property with the same name (different casing) as each field of the attribute.

D.3.6. Nested Types

A nested type is a type defined within the scope of another type. They are very useful for encapsulating implementation details of a type, such as an enumerator over a collection, because they can have access to private state. Public nested types are rarely used.

Do not use public nested types unless all of the following are true.

- The nested type logically belongs to the containing type.
- The nested type is not used very often, or at least not directly.

D.4. Error Raising and Handling

- Do end Exception class names with the *Exception* suffix.
- Do use these common constructors.

```
public class XxxException : Exception

{

   XxxException() { }

   XxxException(string message) { }

   XxxException(string message, Exception inner) { }

}
```

- Do use the predefined exception types. Only define new exception types for programmatic scenarios, meaning you expect users of you library to catch exceptions of this new type and perform a programmatic action based on the exception type..
- Do not derive new exceptions directly from the base class `Exception`. Use one of its predefined subclasses instead.
- Do use a localized description string. When the user sees an error message, it will be derived from the description string of the exception that was thrown, and never from the exception class. Include a description string in every exception.
- Do use grammatically correct error messages including ending punctuation.

  Each sentence in a description string of an exception should end in a period. This way code that generically displays an exception message to the user does not have to handle the case where a developer forgot the final period, which is relatively cumbersome and expensive.
- Do provide exception properties for programmatic access. Include extra information (besides the description string) in an exception only when there is a programmatic scenario where that additional information is useful.
- Do throw exceptions only in exceptional cases.
  - Do not use exceptions for normal or expected errors.
  - Do not use exceptions for normal flow of control.

- Do return null for extremely common error cases. For example, `File.Open` returns a null if the file is not found, but throws an exception if the file is locked.
- Do design classes such that in the normal course of use there will never be an exception thrown. For example, a `FileStream` class might expose a way of determining if the end of the file has been reached to avoid the exception that will be thrown if the developer reads past the end of the file.
- Do throw an `InvalidOperationException` if in an inappropriate state.

  The `System.InvalidOperationException` exception should be thrown if the property set or method call is not appropriate given the object's current state.
- Do throw an `ArgumentException` or create an exception derived from this class if bad parameters are passed or detected.
- Do realize that the stack trace starts at the point where an exception is thrown, not where it is created with the **`new`** operator. You should consider this when deciding where to throw an exception.
- Do throw Exceptions rather than return an error code.
- Do throw the most specific exception possible.
- Do set all the fields on the exception you use.
- Do use Inner exceptions (chained exceptions).
- Do cleanup side effects when throwing an exception. Clearly document cases where an exception may occur after a side-effect has already taken place and cannot be retracted.
- Do not assume that side-effects do not occur before an exception is thrown, but rather that the state is restored if one is thrown. That is, another thread may see the side-effect, but will then see an addition one to restore the state.

D.4.1. Standard Exception Types

The following table breaks down the standard exceptions and the conditions for which you should create a derived class.

Exception Type	**Base Type**	**Description**	**Example**
Exception	Object	Base class for all Exceptions.	None (use a derived class of this exception).
SystemException	Exception	Base class for all runtime generated errors.	None (use a derived class of this exception).
IndexOutOfRangeException	SystemException	Thrown only by the runtime when an array is indexed improperly.	Indexing an array outside of its valid range: arr[arr.Length+1]
NullReferenceException	SystemException	Thrown only by the runtime when a null object is referenced.	object o = null; o.ToString();
InvalidOperationException	SystemException	Thrown by methods when in an invalid state.	Calling Enumerator.GetNext() after

			removing an Item from the underlying collection.
ArgumentException	SystemException	Base class for all Argument Exceptions. Derived classes of this exception should be thrown where applicable.	None (use a derived class of this exception).
ArgumentNullException	ArgumentException	Thrown by methods that do not allow an argument to be null.	String s = null; ”foo”.IndexOf (s);
ArgumentOutOfRangeException	ArgumentException	Thrown by methods that verify that arguments are in a given range.	String s = “string”; s.Chars[9];
InteropException	SystemException	Base class for exceptions that occur or are targeted at environments outside of the runtime.	None (use a derived class of this exception).

D.5. Array Usage Guidelines

- Do use a collection when **`Add`**, **`Remove`** or other methods for manipulating the collection are supported. This scopes all related methods to the collection.
- Do use collections to add read-only wrappers around internal arrays.
- Do use collections to avoid the inefficiencies in the following code.

```
for (int i = 0; i < obj.myObj.Count; i++)
      DoSomething(obj.myObj[i])
```

  Also see clause D.2.1.1.

- Do return an Empty array instead of a null.

  Users assume that the following code will work:

```
public void DoSomething(…)
{ int a[] = SomeOtherFunc();
  if (a.Length > 0)             // Don’t expect NULL here!
  { // do something
  }
}
```

D.6. Operator Overloading Usage Guidelines

- Do define operators on Value types that are logically a built-in language type.
- Do provide operator-overloading methods only involving the class in which the methods are defined.
- Do use the names and signature conventions described in the common language specification.

- Do not be cute.

  Operator overloading is useful in cases where it is immediately obvious what the result of the operation will be. For example, it makes sense to be able to subtract one **`Time`** value from another **`Time`** value and get a **`TimeSpan`**. However, it is not appropriate to use **`shift`** to write to a stream.

- Do overload operators in a symmetric fashion. For example, if you overload the **`Equal`** operator (**`==`**), you should also overload not equals (**`!=`**) operator.

- Do provide alternate signatures.

  Most languages do not support operator overloading. For this reason it is a CLS requirement that you include a method with an appropriate domain-specific name that has the equivalent functionality as in the following example.

  ```
  class Time {
     TimeSpan operator -(Time t1, Time t2) { }
     TimeSpan Difference(Time t1, Time t2) { }
  }
  ```

  See Partition I (Operator Overloading)

D.6.1. Implementing Equals and Operator==

Do see the section on implementing the **`Equals`** method in Section D.7.

Do implement `GetHashCode()` whenever you implement `Equals()`. This keeps `Equals()` and `GetHashCode()` synchronized.

Do override `Equals` whenever you implement `operator==` and make them do the same thing. This allows infrastructure code such as `Hashtable` and `ArrayList` which use `Equals()` to behave the same way as user code written using `operator==`.

Do override `Equals` anytime you implement `IComparable`.

Consider implementing operator overloading for `==`, `!=` , `<`, and `>` when you implement `IComparable`.

Do not throw exceptions from `Equals()`, `GetHashCode()`, or `operator==` methods.

D.6.1.1. Implementing operator== on Value Types

Do overload `operator==` anytime equality is meaningful, because in most programming languages there is no default implementation of `operator==` for value types.

Consider implementing `Equals()` on ValueTypes because the default implementation on `System.ValueType` will not perform as well as your custom implementation.

Do implement `operator==` anytime you override `Equals()`

D.6.1.2. Implementing operator== on Reference Types

Do use care when implementing `operator==` on reference types. Most languages do provide a default implementation of `operator==` for reference types, therefore overriding the default implementation should be done with care. Most reference types, even those that implement `Equals()` should not override `operator==`.

Do override `operator==` if your type has value semantics (that is, if it looks like a base type such as a Point, String, BigNumber, etc.). Anytime you are tempted to overload `+` and `-` you also should consider overloading `operator==`.

D.6.2. Cast Operations (op_Explicit and op_Implicit)

- Do not lose precision in implicit casts.

  For example, there should not be an implicit cast from **`Double`** to **`Int32`**, but there may be one from **`Int32`** to **`Int64`**.

- Do not throw exceptions from implicit casts because it is very difficult for the developer to understand what is happening.

- Do provide cast operations that operate on the whole object. The value that is cast represents the whole value being cast, not one sub part. For example, it is not appropriate for a Button to cast to a string by returning its caption.

- Do not generate a semantically different value.

  For example, it is appropriate to convert a **`Time`** or **`TimeSpan`** into an **`Int`**. The **`Int`** still represents the time or duration. It does not make sense to convert a file name string such as, "c:\mybitmap.gif" into a **Bitmap** object.

- Do not provide cast operations for values between different semantic domains. For example, it makes sense that an **`Int32`** can cast to a **`Double`**. It does not make sense for an **`Int`** to cast to a **`String`**, because they are in different domains.

D.7. Equals

Do see clause D.6.1 on implementing operator==.

Do override `GetHashCode()` in order for the type to behave correctly in a hashtable.

Do not throw an exception in your `Equals` implementation. Return false for a null argument, etc.

Do follow the contract defined on `Object.Equals`.

- x.Equals(x) returns true.
- x.Equals(y) returns the same value as y.Equals(x).
- If (x.Equals(y) && y.Equals(z)) returns true, then x.Equals(z) returns true.
- Successive invocations of x.Equals(y) return the same value as long as the objects referenced by x and y are not modified.
- x.Equals(null) returns false for non-null x.

For some kinds of objects, it is desirable to have `Equals` test for *value equality* instead of referential equality. Such implementations of `Equals` return true if the two objects have the same value, even if they are not the same instance. The definition of what constitutes an object's value is up to the implementer of the type, but it is typically some or all of the data stored in the instance variables of the object. For example, the value of a string is based on the characters of the string; the `Equals` method of the `String` class returns true for any two string instances that contain exactly the same characters in the same order.

When the `Equals` method of a base class provides value equality, an override of `Equals` in a class derived from that base class should invoke the inherited implementation of `Equals`.

If you choose to overload the equality operator for a given type, that type should override the `Equals` method. Such implementations of the `Equals` method should return the same results as the equality operator. Following this guideline will help ensure that class library code using `Equals` (such as `ArrayList` and `Hashtable`) behaves in a manner that is consistent with the way the equality operator is used by application code.

If you are implementing a value type, you should follow these guidelines.

- Consider overriding `Equals` to gain increased performance over that provided by the default implementation of `Equals` on `System.ValueType`.
- If you override `Equals` and the language supports operator overloading, you should overload the equality operator for your value type.

If you are implementing reference types, you should follow these guidelines.

- Consider overriding `Equals` on a reference type if the semantics of the type are based on the fact that the type represents some value(s). For example, reference types such as Point and BigNumber should override `Equals`.
- Most reference types should not overload the equality operator, even if they override `Equals`. However, if you are implementing a reference type that is intended to have value semantics, such as a complex number type, you should override the equality operator.

If you implement `IComparable` on a given type, you should override `Equals` on that type.

D.8. Callbacks

Delegates, Interfaces and Events can each be used to provide callback functionality. Each has its own specific usage characteristics that make it better suited to particular situations.

Use Events if the following are true.

- One signs up for the callback up front (typically through separate `Add` and `Remove` methods).
- Typically more than one object will care.

Use a Delegate if the following are true.

- You want a C style function pointer.
- Single callback.
- Registered in the call or at construction time (not through separate Add method)

Use an Interface if the following is true.

- The callback entails complex behavior.

D.9. Security in Class Libraries

Class library authors need to consider two perspectives with respect to security.Whether these perspectives are applicable will depend upon the class itself. Some classes, such as `System.IO.FileStream` represent objects that need protection with permissions; the implementation of these classes is responsible for checking the appropriate permissions of the caller required for each action and only allowing authorized callers to perform the actions for which they have permission. The `System.Security` namespace contains some classes to help make these checks easier. Additionally, class library code often is fully-trusted or at least highly-trusted code. Any flaws in the code represent a serious threat to the integrity of the entire security system. Therefore, extra care is required when writing class library code as detailed below.

- Do access protected resources only after checking the permissions of your callers, either through a declarative security attribute or an explicit call to `Demand` on an appropriate security permission object.
- Do assert a permission only when necessary, and always precede it by the necessary checks.
- Do not assume that code will only be called by callers with certain permissions.

- Do not define non-type-safe interfaces that might be used to bypass security.
- Do not expose functionality that allows a semi-trusted caller to take advantage of higher trust of the class.

D.10. Threading Design Guidelines

- Do not provide static methods that mutate static state.

  In common server scenarios, static state is shared across requests, which means multiple threads can execute that code at the same time. This opens up the possibility for threading bugs. Consider using a design pattern that encapsulates data into instances that are not shared.

- Do not normally provide thread safe instance state.

  By default, the library is not thread safe. Adding locks to create thread safe code decreases performance and increases lock contention (as well as opening up deadlock bugs). In common application models, only one thread at a time executes user code, which minimizes the need for thread safety. In cases where it is interesting to provide a thread safe version a `GetSynchronized()` method can be used to return a thread safe instance of that type. (See `System.Collections` for examples).

- Do make all static state thread safe.

  If you must use static state, make it thread safe. In common server scenarios, static data is shared across requests, which means multiple threads can execute that code at the same time. For this reason it is necessary to protect static state.

- Do be aware of non-atomic operations.

  Value types whose underlying representations are greater than 32 bits may have non-atomic operations. Specifically, because value types are copied bitwise (by value as opposed to by reference), race conditions can occur in what appears to be straightforward assignments within code.

  For example, consider the following code (executing on two separate threads) where the variable `x` has been declared as type `Int64`.

  ```
  // Code executing on Thread "A".
  x = 54343343433;

  // Code executing on Thread "B".
  x = 934343434343;
  ```

  At first glance it seems to indicate that there is no possibility of race conditions (since each line looks like a straight assignment operation). However, because the underlying variable is a 64-bit value type, the actual code is not doing an atomic assignment operation. Instead, it is doing a bitwise copy of two 32 bit halves. In the event of a context switch, halfway during the value type assignment operation on one of the threads, the resulting `x` variable can have corrupt data (for example, the resulting value will be composed of 32 bits of the first number, and 32 bits of the second number).

- Do be aware of method calls in locked sections.

  Deadlocks can result when a static method in class A calls static methods in class B and vice versa. If A and B both synchronize their static methods, this will cause a deadlock. You might only discover this deadlock under heavy threading stress.

  Performance issues can result when a static method in class A calls a static method in class A. If these methods are not factored correctly, performance will suffer because there will be a large amount of redundant synchronization. Excessive use of fine-grained synchronization might negatively impact performance. In addition, it might have a significant negative impact on scalability.

- Do be aware of issues with the `lock` statement and consider using `System.Threading.Interlocked` instead.

  It's tempting to use the `lock` statement in C# to solve all threading problems. But the `System.Threading.Interlocked` class is superior for updates that must be made automically

- Do avoid the need for synchronization if possible.

  Obviously for high traffic pathways it is nice to avoid synchronization. Sometimes the algorithm can be adjusted to tolerate races rather than eliminating them.

Annex E Portability Considerations

This chapter contains only informative text

This Chapter gathers together information about areas where this Standard deliberately leaves leeway to implementations. This leeway is intended to allow compliant implementations to make choices that provide better performance or add value in other ways. But this leeway inherently makes programs non-portable. This chapter describes the techniques that can be used to ensure that programs operate the same way independent of the particular implementation of the CLI.

Note that code may be portable even though the data is not, both due to size of integer type and direction of bytes in words. Read/write invariance holds provided the read method corresponds to the write method (i.e. write as int read as int works, but write as string read as int might not).

E.1. Uncontrollable Behavior

The following aspects of program behavior are implementation dependent. Many of these items will be familiar to programmers used to writing code designed for portability (for example, the fact that the CLI does not impose a minimum size for heap or stack).

1. Size of heap and stack aren't required to have minimum sizes
2. Behavior relative to asynchronous exceptions (see `System.Thread.Abort`)
3. Globalization is not supported, so every implementation specifies its culture information including such user-visible features as sort order for strings.
4. Threads cannot be assumed to be either pre-emptively or non-pre-emptively scheduled. This decision is implementation specific.
5. Locating assemblies is an implementation-specific mechanism.
6. Security policy is an implemenation-specific mechanism.
7. File names are implementation-specific.
8. Timer resolution (granularity) is implementation-specific, although the unit is specified.

E.2. Language- and Compiler-Controllable Behavior

The following aspects of program behavior can be controlled through language design or careful generation of CIL by a language-specific compiler. The CLI provides all the support necessary to control the behavior, but the default is to allow implementation-specific optimizations.

1. Unverifiable code can access arbitrary memory and cannot be guaranteed to be portable
2. Floating point – compiler can force all intermediate values to known precision
3. Integer overflow – compiler can force overflow checking
4. Native integer type need not be exposed, or can be exposed for opaque handles only, or can reliably recast with overflow check to known size values before use. Note that "free conversion" between native integer and fixed-size integer without overflow checks will not be portable.
5. Deterministic initialization of types is portable, but "before first reference to static variable" is not. Language design can either force all initialization to be deterministic (cf. Java) or can restrict initialization to deterministic cases (i.e. simple static assignments).

E.3. Programmer-Controllable Behavior

The following aspects of program behavior can be controlled directly by the programmer.

1. Code that is not thread-safe may operate differently even on a single implementation. In particular, the atomicity guarantees around 64-bit must be adhered to and testing on 64-bit implementations may not be sufficient to find all such problems. The key is never to use both normal read/write and interlocked access to the same 64-bit datum.
2. Calls to unmanaged code or calls to non-standardized extensions to libraries
3. Do not depend on the relative order of finalization of objects.
4. Do not use explicit layout of data.
5. Do not rely on the relative order of exceptions within a single CIL instruction or a given library method call.