INTERNATIONAL STANDARD

ISO/IEC 23271

Second edition
2006-10-01

Information technology — Common Language Infrastructure (CLI) Partitions I to VI

Technologies de l'information — Infrastructure commune de langage (ICL) Partitions I à VI

Reference number
ISO/IEC 23271:2006(E)

© ISO/IEC 2006

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

© ISO/IEC 2006

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Common Language Infrastructure (CLI)
Partitions I–VI

Partition I: Concepts and Architecture
Partition II: Metadata Definition and Semantics
Partition III: CIL Instruction Set
Partition IV: Profiles and Libraries
Partition V: Debug Interchange Format
Partition VI: Annexes

Common Language Infrastructure (CLI)

Partition I:
Concepts and Architecture

Table of Contents

Foreword		**vii**
1	**Scope**	**1**
2	**Conformance**	**2**
3	**Normative references**	**3**
4	**Conventions**	**4**
4.1	Organization	4
4.2	Informative text	4
5	**Terms and definitions**	**5**
6	**Overview of the Common Language Infrastructure**	**9**
6.1	Relationship to type safety	9
6.2	Relationship to managed metadata-driven execution	10
6.2.1	Managed code	10
6.2.2	Managed data	11
6.2.3	Summary	11
7	**Common Language Specification**	**12**
7.1	Introduction	12
7.2	Views of CLS compliance	12
7.2.1	CLS framework	12
7.2.2	CLS consumer	13
7.2.3	CLS extender	13
7.3	CLS compliance	14
7.3.1	Marking items as CLS-compliant	15
8	**Common Type System**	**16**
8.1	Relationship to object-oriented programming	18
8.2	Values and types	18
8.2.1	Value types and reference types	18
8.2.2	Built-in value and reference types	19
8.2.3	Classes, interfaces, and objects	19
8.2.4	Boxing and unboxing of values	20

©ISO/IEC 2006 – All rights reserved

8.2.5	Identity and equality of values	21
8.3	Locations	22
8.3.1	Assignment-compatible locations	22
8.3.2	Coercion	22
8.3.3	Casting	22
8.4	Type members	22
8.4.1	Fields, array elements, and values	23
8.4.2	Methods	23
8.4.3	Static fields and static methods	23
8.4.4	Virtual methods	23
8.5	Naming	24
8.5.1	Valid names	24
8.5.2	Assemblies and scoping	24
8.5.3	Visibility, accessibility, and security	26
8.6	Contracts	28
8.6.1	Signatures	29
8.7	Assignment compatibility	32
8.8	Type safety and verification	33
8.9	Type definers	33
8.9.1	Array types	34
8.9.2	Unmanaged pointer types	35
8.9.3	Delegates	35
8.9.4	Interface type definition	36
8.9.5	Class type definition	37
8.9.6	Object type definitions	38
8.9.7	Value type definition	40
8.9.8	Type inheritance	41
8.9.9	Object type inheritance	41
8.9.10	Value type inheritance	41
8.9.11	Interface type derivation	42
8.10	Member inheritance	42
8.10.1	Field inheritance	42
8.10.2	Method inheritance	42
8.10.3	Property and event inheritance	42
8.10.4	Hiding, overriding, and layout	43
8.11	Member definitions	44
8.11.1	Method definitions	44
8.11.2	Field definitions	44

 ©ISO/IEC 2006 – All rights reserved

8.11.3	Property definitions	45
8.11.4	Event definitions	46
8.11.5	Nested type definitions	46
9	**Metadata**	**47**
9.1	Components and assemblies	47
9.2	Accessing metadata	47
9.2.1	Metadata tokens	47
9.2.2	Member signatures in metadata	48
9.3	Unmanaged code	48
9.4	Method implementation metadata	48
9.5	Class layout	48
9.6	Assemblies: name scopes for types	49
9.7	Metadata extensibility	50
9.8	Globals, imports, and exports	51
9.9	Scoped statics	51
10	**Name and type rules for the Common Language Specification**	**52**
10.1	Identifiers	52
10.2	Overloading	52
10.3	Operator overloading	53
10.3.1	Unary operators	53
10.3.2	Binary operators	54
10.3.3	Conversion operators	55
10.4	Naming patterns	56
10.5	Exceptions	56
10.6	Custom attributes	57
10.7	Generic types and methods	57
10.7.1	Nested type parameter re-declaration	57
10.7.2	Type names and arity encoding	58
10.7.3	Type constraint re-declaration	60
10.7.4	Constraint type restrictions	60
10.7.5	Frameworks and accessibility of nested types	60
10.7.6	Frameworks and abstract or virtual methods	61
11	**Collected Common Language Specification rules**	**62**
12	**Virtual Execution System**	**65**
12.1	Supported data types	65

12.1.1	Native size: native int, native unsigned int, O and &	66
12.1.2	Handling of short integer data types	67
12.1.3	Handling of floating-point data types	67
12.1.4	CIL instructions and numeric types	69
12.1.5	CIL instructions and pointer types	70
12.1.6	Aggregate data	71
12.2	Module information	74
12.3	Machine state	74
12.3.1	The global state	74
12.3.2	Method state	75
12.4	Control flow	78
12.4.1	Method calls	79
12.4.2	Exception handling	82
12.5	Proxies and remoting	92
12.6	Memory model and optimizations	93
12.6.1	The memory store	93
12.6.2	Alignment	93
12.6.3	Byte ordering	93
12.6.4	Optimization	93
12.6.5	Locks and threads	94
12.6.6	Atomic reads and writes	95
12.6.7	Volatile reads and writes	95
12.6.8	Other memory model issues	96
13	**Index**	**97**

 ©ISO/IEC 2006 – All rights reserved

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 23271 was prepared by Ecma (as Ecma-335) and was adopted, under a special “fast-track procedure”, by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, in parallel with its approval by national bodies of ISO and IEC. This second edition cancels and replaces the first edition (ISO/IEC 23271:2003), which has been technically revised.

Information technology — Common Language Infrastructure (CLI) Partition I to VI

1 Scope

This International Standard defines the Common Language Infrastructure (CLI) in which applications written in multiple high-level languages can be executed in different system environments without the need to rewrite those 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 an informative description of the metadata.
- 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 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 file, CLILibrary.xml, 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: Debug Interchange Format.
- Partition VI: 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 can be used to derive parts of the grammar used by this assembler, as well as other tools that manipulate CIL, a set of guidelines used in the design of the libraries of Partition IV and portability considerations.

©ISO/IEC 2006 – All rights reserved

2 Conformance

A system claiming conformance to this International Standard shall implement all the normative requirements of this standard, and shall specify the profile (see Partition IV) that it implements. The minimal implementation is the Kernel Profile. A conforming implementation can also include additional functionality provided that functionality does not prevent running code written to rely solely on the profile as specified in this standard. For example, a conforming implementation can 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 correct CIL as specified in this standard. Such a compiler can also claim that it generates *verifiable* code, in which case, the CIL it generates shall be verifiable as specified in this standard.

 ©ISO/IEC 2006 – All rights reserved

3 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

Extensible Markup Language (XML) 1.0 (Third Edition), 2004 February 4, http://www.w3.org/TR/2004/REC-xml-2004

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 (all parts), *Codes for the representation of names of languages*

ISO 3166 (all parts), *Codes for the representation of names of countries and their subdivision*

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

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

ISO/IEC 10646, *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*

ISO/IEC 11578:1996, *Information technology — Open Systems Interconnection — Remote Procedure Call (RPC)*

ISO/IEC 14882:2003, *Programming languages — C++*

ISO/IEC 23270:2006, *Information technology — Programming languages — C#*

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

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

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

RFC-793, *Transmission Control Protocol*, *Darpa Internet Program Protocol Specification,* 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 and 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,* 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

©ISO/IEC 2006 – All rights reserved

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

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 4.0, defined by: *The Unicode Standard, Version 4.0* (Addison-Wesley, Boston, MA, 2003. ISBN 0-321-18578-1)

4 Conventions

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 *clause*, followed by *subclause*. Divisions within a subclause are also referred to as 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 Subclause 2 in Subclause 3 in Subclause 4 in Clause 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 clauses or subclauses. Notes are provided to give advice or guidance to implementers or programmers. Annexes provide additional information.

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

- The beginning and end of a block of informative text is marked using rectangular boxes.
- As some informative passages span pages, informative text also contains a bold set of vertical black stripes in the right margin.
- By the use of the following pairs of markers: [*Example*: … *end example*], [*Note*: … *end note*], and [*Rationale*: … *end rationale*].

Unless text is identified as being informative, it is normative.

 ©ISO/IEC 2006 – All rights reserved

5 Terms and definitions

For the purposes of this International Standard, the following terms and definitions apply. Other terms are defined where they appear in *italic* type.

5.1
ANSI character
character from an implementation-defined 8-bit character set whose first 128 code points correspond exactly to those of ISO/IEC 10646

5.2
ANSI string
string of ANSI characters, of which the final character has the value all-bits-zero

5.3
assembly
configured set of loadable code modules and other resources that together implement a unit of functionality

5.4
attribute
characteristic of a type and/or its members that contains descriptive information

NOTE While the most common attributes are predefined, and have a specific encoding in the metadata associated with them, user-defined attributes can also be added to the metadata.

5.5
behavior, implementation-specific
unspecified behavior, for which each implementation is required to document the choice it makes

5.6
behavior, unspecified
behavior, for a well-formed program construct and correct data, that depends on the implementation

NOTE The implementation is not required to document which behavior occurs.

5.7
behavior, undefined
behavior, such as might arise upon use of an erroneous program construct or erroneous data, for which this International Standard imposes no requirements

NOTE Undefined behavior can also be expected in cases when this International Standard omits the description of any explicit definition of behavior.

5.8
boxing
conversion of a value having some value type, to a newly allocated instance of the reference type `System.Object`

5.9
Common Intermediate Language
CIL
instruction set understood by the VES

5.10
Common Language Infrastructure
CLI
specification for the format of executable code, and the run-time environment that can execute that code

5.11
Common Language Specification
CLS
agreement between language designers and framework (class library) designers

NOTE It specifies a subset of the CTS and a set of usage conventions.

5.12
Common Type System
CTS
unified type system that is shared by compilers, tools and the CLI itself

NOTE 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.

5.13
delegate
reference type such that an instance of it can encapsulate one or more methods in an invocation list

NOTE Given a delegate instance and an appropriate set of arguments, one can invoke all of the methods in a delegate's invocation list with that set of arguments.

5.14
event
member that enables an object or class to provide notifications

5.15
Execution Engine
(See **Virtual Execution System**)

5.16
field
member that designates a typed memory location that stores some data in a program

5.17
garbage collection
process by which memory for managed data is allocated and released

5.18
generic argument
actual type used to instantiate a particular generic type or generic method

NOTE For example, in `List<string>`, `string` is the generic argument corresponding to the generic parameter `T` in the generic type definition `List<T>`.

5.19
generic parameter
parameter within the definition of a generic type or generic method that acts as a place holder for a generic argument

NOTE For example, in the generic type definition `List<T>`.`T` is a generic parameter.

5.20
generics
feature that allows types and methods to be defined such that they are parameterized with one or more generic parameters

5.21
library
repository for a set of types, which are grouped into one or more assemblies

NOTE A library can also contain modifications to types defined in other libraries. For example, a library can include additional methods, interfaces and exceptions for types defined in other libraries.

 ©ISO/IEC 2006 – All rights reserved

5.22
managed code
code that contains enough information to allow the CLI to provide a set of core services

NOTE For example, given an address for a method inside the code, the CLI will be able to locate the metadata describing that method. It will also be able to walk the stack, handle exceptions and store and retrieve security information.

5.23
managed data
data that is allocated and released automatically by the CLI, through a process called garbage collection

5.24
manifest
that part of an assembly that specifies the following information about that assembly; its version, name, culture and security requirements; which other files, if any, belong to that assembly, along with a cryptographic hash of each file; which of the types defined in other files of that assembly are to be exported from that assembly; and, optionally, a digital signature for the manifest itself, and the public key used to compute it

5.25
member
any of the fields, array elements, methods, properties and events of a type

5.26
metadata
data that describes and references the types defined by the CTS

NOTE Metadata is stored 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 (such as compilers and debuggers) as well as between these tools and the VES.

5.27
method
member that describes an operation that can be performed on values of an exact type

5.28
method, generic
method (be it static, instance or virtual), defined within a type, whose signature includes one or more generic parameters, not present in the type definition itself

NOTE The enclosing type itself might, or might not, be generic. For example, within the generic type `List<T>`, the generic method `S ConvertTo<S>()` is generic.

5.29
method, non-generic
method that is not generic

5.30
module
single file containing content that can be executed by the VES

5.31
object
instance of a reference type

NOTE An object has more to it than a value. An object is self-typing; 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 can be either objects or values). While the contents of its slots can be changed, the identity of an object never changes.

5.32
profile
set of libraries, grouped together to form a consistent whole that provides a fixed level of functionality

5.33
property
member that defines a named value and the methods that access that value

NOTE A property definition defines the accessing contracts on that value. Hence, the property definition specifies which accessing methods exist and their respective method contracts.

5.34
signature
the part of a contract that can be checked and automatically enforced

NOTE 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.

5.35
type, generic
type whose definition is parameterized by one or more other types; for example, `List<T>`, where `T` is a generic parameter

NOTE The CLI supports the creation and use of instances of generic types. For example, `List<int32>` or `List<string>`.

5.36
type, reference
type such that an instance of it contains a reference to its data

5.37
type, value
type such that an instance of it directly contains all its data

5.38
unboxing
conversion of a value having type `System.Object`, whose run-time type is a value type, to a value type instance

5.39
unmanaged code
code that is not managed

5.40
unmanaged data
data that is not managed

5.41
value
simple bit pattern for something like an integer or a float

NOTE 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 can be performed on that representation. Values are intended for representing the simple types and non-objects in programming languages.

5.42
verification
checking of both CIL and its related metadata to ensure that the CIL code sequences do not permit any access to memory outside the program's logical address space

NOTE In conjunction with the validation tests, verification ensures that the program cannot access memory or other resources to which it is not granted access.

5.43
Virtual Execution System
VES
system that implements and enforces the CTS model

NOTE 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. The VES is also known as the **Execution Engine**.

 ©ISO/IEC 2006 – All rights reserved

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) 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 clause and its subclauses contain only informative text

At the center of the CLI is a unified type system, the Common Type System 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 clause describes the architecture of the CLI by describing the CTS.

The following four areas are covered in this clause:

- **The Common Type System (CTS)**—The CTS provides a rich type system that supports the types and operations found in many programming languages. The CTS is intended to support the complete implementation of a wide range of programming languages. See §8
- **Metadata**—The CLI uses metadata to describe and reference the types defined by the CTS. Metadata is stored (that is, persisted) in a way that is independent of any particular programming language. Thus, metadata provides a common interchange mechanism for use between tools (such as compilers and debuggers) that manipulate programs, as well as between these tools and the VES. See §9.
- **The Common Language Specification (CLS)**—The CLS is an agreement between language designers and framework (that is, class library) designers. It specifies a subset of the CTS 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 exported aspects (e.g., classes, interfaces, methods, and fields) use only types that are part of the CLS and that adhere to the CLS conventions. See §10.
- **The Virtual Execution System (VES)**—The 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). See §12.

Together, these aspects of the CLI form a unifying infrastructure for designing, developing, deploying, and executing distributed components and applications. The appropriate subset of the CTS is available from each programming language that targets the CLI. Language-based tools communicate with each other and with the VES using metadata to define and reference the types used to construct the application. The VES 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, and security).

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 CTS, type safety guarantees that:

- **References are what they say they are** – Every reference is typed, the object or value referenced also has a type, and these types are assignment compatible (see §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. Access to an object is through accessible functions and fields. An object can 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.

©ISO/IEC 2006 – All rights reserved

- **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 derived classes).

The CTS promotes type safety (e.g., everything is typed). Type safety can optionally be enforced. The hard problem is determining if an implementation conforms to a type-safe 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 §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 CTS assumes that the execution environment is metadata-driven. Using metadata allows the CLI to support:

- **Multiple execution models** – The metadata allows the execution environment to deal with a mixture of interpreted, JIT-compiled, native, and legacy code, and still present uniform services to tools like debuggers and 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: requirements and context are made available via metadata; where to look, how to find an implementation, and how to decide the best match all come from application packaging and deployment.

6.2.1 Managed code

Managed code is 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

 ©ISO/IEC 2006 – All rights reserved

This standard specifies a particular instruction set, the 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 CTS 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 in any language. It does this by defining a standard set of types, by making all components fully self-describing, and by 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 CTS 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 how types are instantiated, interact, and die.

End informative text

7 Common Language Specification

7.1 Introduction

The 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 clauses and are summarized in §11. CLS conformance is a characteristic of types that are generated for execution on a CLI implementation. Such types must conform to the CLI standard, 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 that are accessible outside the assembly; that is, those that have an accessibility of **public**, **family** (but not on sealed types), or **family-or-assembly** (but not on sealed types).

[*Note:* A library consisting of CLS-compliant code is herein referred to as a *framework*. Compilers that generate code for the CLI can 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. *end note*]

7.2 Views of CLS compliance

This block contains 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 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.
- Not expect users of the framework to be able to author nested types.
- Assume that implementations of methods of the same name and signature on different interfaces are independent.
- Not rely on initialization of value types to be performed automatically based on specified initializer values.
- Assume users can instantiate and use generic types and methods, but do not require them to define new generic types or methods, or deal with partially constructed generic types.

Frameworks shall not:

- Require users to define new generic types/methods, override existing generic methods, or deal with partially constructed generics in any way.

A CLS Rule applies to this topic; see the normative text at the end of §7.2.

 ©ISO/IEC 2006 – All rights reserved

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, 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 whose names 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.
- Have a mechanism to import, instantiate, and use generic types and methods.

[*Note*: Consumers should consider supporting:

- Type inferencing over generic methods with language-defined rules for matching.
- Casting syntax to clarify ambiguous casts to a common supertype.

end note]

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 can choose to use initialization metadata, but can 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 whose names 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.
- Define new CLS-compliant (non-generic) types that extend any (non-sealed) CLS-compliant base type. Valid base types include normal (non-generic) types and also fully constructed generic types.

[*Note*: Extenders should consider supporting:

- Type inferencing over generic methods with language-defined rules for matching.
- Casting syntax to clarify ambiguous casts to a common supertype.

end note]

Extenders need not support the following:

- Definition of new CLS-compliant interfaces.
- Definition of nested types.
- Definition of generic types and methods.
- Overriding existing virtual generic methods.

The CLS is designed to be large enough that it is properly expressive yet small enough that all languages can reasonably accommodate it.

End informative text

CLS Rule 48: If two or more CLS-compliant methods declared in a type have the same name and, for a specific set of type instantiations, they have the same parameter and return types, then all these methods shall be semantically equivalent at those type instantiations.

[*Note*:

CLS (consumer): May select any one of the methods.

CLS (extender): Same as consumer.

CLS (framework): Shall not expose methods that violate this rule. *end note*]

[*Note*: To avoid confusion, the CLS rules follow historical numbering from the previous version of this Standard, despite removal/addition of rules in this version. As such, the first rule shown in this partition is Rule 48. *end note*]

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 can be used. Thus, the CLS rules apply only to items that are visible (see §8.5.3) outside of their defining assembly and have **public**, **family**, or **family-or-assembly** accessibility (see §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 exported 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. *end note*]

 ©ISO/IEC 2006 – All rights reserved

Any construct that would make it impossible to rapidly verify code is excluded from the CLS. This allows all CLS-compliant language translators 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. (Implementers are discouraged from marking extensions to this standard as CLS-compliant.) This is done using the custom attribute mechanism (see §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 level of compliance for all top-level types that assembly contains.

The constructor for `System.CLSCompliantAttribute` takes a Boolean argument indicating whether the item with which it is associated is 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 level of compliance attached to its assembly (for top-level types). A type can 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 can 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): Can ignore any member that is not CLS-compliant using the above rules.

CLS (extender): Should encourage correct labeling of newly authored assemblies and publicly exported types and members. Compile-time enforcement of the CLS rules is strongly encouraged.

CLS (framework): Shall correctly label all publicly exported members as to their CLS compliance. The rules specified here can 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). *end note*]

8 Common Type System

Types describe values and specify a contract (see §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 can 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 can be either objects or values). While the contents of its slots can be changed, the identity of an object never changes.

There are several kinds of objects and values, as shown in the (informative) diagram below.

The generics feature allows a whole family of types and methods to be defined using a pattern, which includes placeholders called *generic parameters*. These generic parameters are replaced, as required, by specific types, to instantiate whichever member of the family is actually required. The design of generics meets the following goals:

- Orthogonality: Where possible, generic types can occur in any context where existing CLI types can occur.
- Language independence: No assumptions about the source language are made. But CLI-generics attempts to support existing generics-like features of as many languages as possible. Furthermore, the design permits clean extensions of languages currently lacking generics.
- Implementation independence: An implementation of the CLI is allowed to specialize representations and code on a case-by-case basis, or to share all representations and code, perhaps boxing and unboxing values to achieve this.
- Implementation efficiency: Performance of generics is no worse than the use of `Object` to simulate generics; a good implementation can do much better, avoiding casts on reference type instantiations, and producing specialized code for value type instantiations.
- Statically checkable at point of definition: A generic type definition can be validated and verified independently of its instantiations. Thus, a generic type is statically verifiable, and its methods are guaranteed to JIT-compile for all valid instantiations.
- Uniform behavior with respect to generic parameters: In general, the behavior of parameterized types and generic methods is “the same” at all type instantiations.

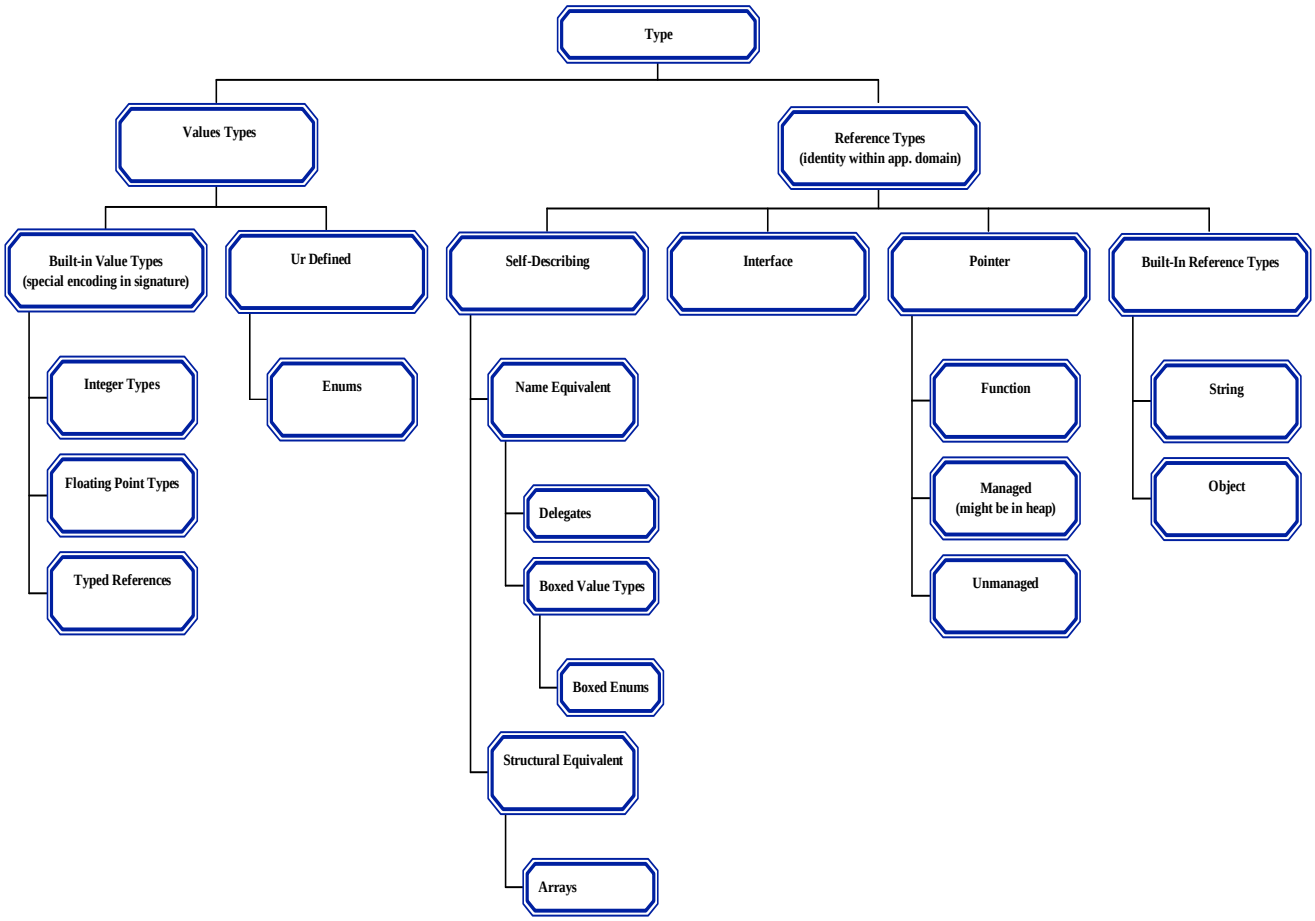
In addition, CLI supports covariant and contravariant generic parameters, with the following characteristics:

- It is type-safe (based on purely static checking)
- Simplicity: in particular, variance is only permitted on generic interfaces and generic delegates (not classes or value-types)
- Languages not wishing to support variance can ignore the feature, and treat all generic types as non-variant.
- Enable implementation of more complex covariance scheme as used in some languages, e.g. Eiffel.

 ©ISO/IEC 2006 – All rights reserved

This figure is informative

Figure 1: Type System



Note: A managed pointer might point into the heap.

End informative figure

8.1 Relationship to object-oriented programming

This subclause 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 compatible types if and only if they have compatible representations and compatible behaviors. Thus, in the CTS, if one type is derived from a base type, then instances of the derived type can be substituted for instances of the base type because both the representation and the behavior are compatible.

Unlike in some OOP languages, in the CTS, 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 §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*. Any value described by a type is called an *instance* of that type. Any use of a value—storing it, passing it as an argument, operating on it—requires a type. This applies in particular to all variables, arguments, 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.

Every value has an *exact type* that fully describes its type properties.

Every value is an instance of its exact type, and can be an instance of other types as well. In particular, if a value is an instance of a type that inherits from another type, it is also an instance of that other type.

8.2.1 Value types and reference types

There are two kinds of types: **value types** and **reference types**.

- Value types – The values described by a value type are self-contained (each can be understood without reference to other values).
- Reference types –A value described by a reference type denotes the location of another value. There are four kinds of reference type:
  - An **object type** is a reference type of a self-describing value (see §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 reference types.

 ©ISO/IEC 2006 – All rights reserved

8.2.2 Built-in value and reference types

The following data types are an integral part of the CTS and are supported directly by the 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`[1]	Yes	`System.Boolean`	True/false value
`char`[1]	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 exact 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

[1] `bool` and `char` are integer types in the categorization shown in the figure above.

8.2.3 Classes, interfaces, and objects

A type fully describes a value if it unambiguously defines the value's representation and the operations defined on that 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 can 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; for example, **interface types**. These 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 can 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: In general, reference types do not 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 can be stored. A boxed type is an object type and a boxed value is an object.

A boxed type cannot be directly referred to by name, therefore no field or local variable can be given a boxed type. The closest named base class to a boxed enumerated value type is `System.Enum`; for all other value types it is `System.ValueType`. Fields typed `System.ValueType` can only contain the null value or an instance of a boxed value type. Locals typed `System.Enum` can only contain the null value or an instance of a boxed enumeration type.

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 bitwise copy of the original value. If the value type is a nullable type—defined as an instantiation of the value type System.Nullable<*T*>—the result is a null reference or bitwise copy of its `Value` property of type *T*, depending on its `HasValue` property (false and true, respectively). All boxed types have an operation called unbox, which results in a managed pointer to the bit representation of the value.

The box instruction can be applied to more than just value types; such types are called *boxable* types. A type is boxable if it is one of the following:

- A value type (including instantiations of generic value types) that does not contain fields that can point into the CIL evaluation stack

  [*Rationale*: A value type that does contain such fields cannot be boxed, else those embedded pointers could outlive the entries in the CIL evaluation stack to which they point; e.g., `System.RuntimeArgumentHandle`, `System.TypedReference`. Value types that contain such pointers are informally described as “byref-like” value types. *end rationale*]

- A reference type (including classes, arrays, delegates, and instantiations of generic classes)
- An unmanaged pointer type
- A generic parameter (to a generic type definition, or a generic method definition) [*Note*: Boxing and unboxing of generic arguments adds performance overhead to a CLI implementation. The constrained. prefix can improve performance during virtual dispatch to a method defined by a value type, by avoiding boxing the value type. *end note*]

The type `System.Void` is never boxable.

Interfaces and inheritance are defined only on reference types. Thus, while a value type definition (see §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: Boxed value types are not CLS-compliant.

[*Note*:

In lieu of boxed types, use `System.Object`, `System.ValueType`, or `System.Enum`, as appropriate.

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

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

 ©ISO/IEC 2006 – All rights reserved

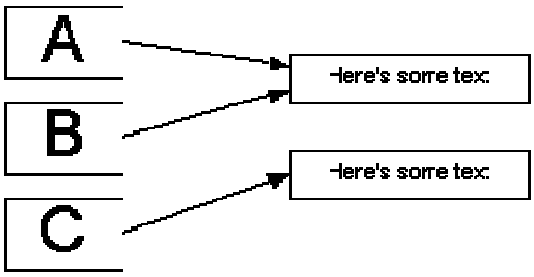
CLS (framework): Shall not use boxed value types in its publicly exported aspects. *end note*]

8.2.5 Identity and equality of values

There are two binary operators defined on all pairs of values: **identity** and **equality**. They return a Boolean result, and are mathematical **equivalence** operators; that is, 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, while identity always implies equality, the reverse is not true. To understand the difference between these operations, consider three variables, A, B, and C, 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 IEC standard. *end note*]

8.3 Locations

Values are stored in **locations**. A location can hold only one 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 that 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 can 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 can 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 §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 might 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 §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 might be needed when casting a value read from a location that is typed as holding a value 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.

 ©ISO/IEC 2006 – All rights reserved

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 can be an array type and an array element can be a compound type.

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

8.4.2 Methods

A type can associate operations with that type or with each instance of that type. Such operations are called methods. A method is named, and has a signature (see §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 §8.4.3).

A method that is associated with an instance of the type is either an instance method or a virtual method (see §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** can be **null** (a special value indicating that no instance is being specified) or an instance of any type that inherits (see §8.9.8) from the class that defines the method. A virtual method can 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 base 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. *end rationale*]

A virtual or instance method can also be called by a different mechanism, a **virtual call**. Any type that inherits from a type that defines a virtual method can provide its own implementation of that method (this is known as **overriding**, see §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 can 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 §12.5), but they can also be allocated on a per-thread basis.

Similarly, types can 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 can declare any of its methods as **virtual**. Unlike other methods, each exact type that implements the type can provide its own implementation of a virtual method. A virtual method can 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 can be **null**. In addition, however, a virtual method can 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 can be marked **final** (see §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 name 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 (e.g., 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 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, one-to-one 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 VI Annex D) *end note*]

8.5.2 Assemblies and scoping

Generally, names are not unique. Names are collected into groupings called **scopes**. Within a scope, a name can 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. *end note*]

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;

 ©ISO/IEC 2006 – All rights reserved

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. A type name is 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 the 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 §8.3.3) or coercion (see §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, as follows:

- 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 §8.6.1.2)

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 §8.9.9).

CLS Rule 7: The underlying type of an enum shall be a built-in CLS integer type, the name of the field shall be "value__", and that field shall be marked `RTSpecialName`.

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 represents 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 §8.6.1) of an enum shall have the type of the enum itself.

[*Note*:

CLS (consumer): Shall accept the 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). *end note*]

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 §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)
  { 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 §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 and, in the case of a nested type, accessible.
2. The member is accessible.
3. All relevant security demands (see §8.5.3.3) have been granted.

An instantiated generic type is visible from some assembly if and only if the generic type itself and each of its component parts (generic type definition and generic arguments) are visible. For example, if *List* is exported from assembly A (i.e., declared "public") and *MyClass* is defined in assembly B but not exported, then *List<MyClass>* is visible only from within assembly B.

Accessibility of members of instantiated generic types is independent of instantiation.

Access to a member *C<T1, ... Tn>.m* is therefore permitted if the following conditions are met:

- *C<T1, ... Tn>* is visible.
- Member *m* within generic type *C* (i.e., *C.m*) is accessible.
- Security permissions 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 can 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 §8.5.3.4.

A top-level named type is *exported* if and only if it has public visibility. A type generated by a type definer is exported if and only if it is made from exported types.

A type generated by a type definer is visible if all types from which it was generated are visible.

 ©ISO/IEC 2006 – All rights reserved

8.5.3.2 Accessibility of members and nested types

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 the 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 §8.8), there is an additional requirement that can 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.

A member or nested type is exported if and only if it has public, family-or-assembly, or family accessibility, and its defining type (in the case of members) or its enclosing type (in the case of nested types) is exported.

The accessibility of a type definer is the same as that for the type from which it was generated.

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

1. Members (other than nested types) 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.
3. A member defined by a nested type, or a nested type enclosed by a nested type, shall not have greater accessibility than the nested type that defines it (in the case of a member) or the nested type that encloses it (in the case of a nested type).

[*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 §8.10.2) rather than relying on limited accessibility. *end rationale*]

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 exported portion of the framework or by users of the framework. *end note*]

8.5.3.3 Security permissions

Access to members is also controlled by security demands that can be attached to an assembly, type, method, property, or event. Security demands are not part of a type contract (see §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 can 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 can be a member of an enclosing type, in which case, it is a nested 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 can 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 §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 §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 the same interface contract. Like class contracts, interface contracts specify which other contracts the interface supports (e.g., which interfaces, methods, properties, and events shall be implemented).

 ©ISO/IEC 2006 – All rights reserved

[*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. *end note*]

- **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 can be overwritten with a different value or whether a value can ever be changed.

All locations have signatures, as do all values. Assignment compatibility requires that the signature of the value, including constraints, be compatible with the signature of the location, including constraints. There are four fundamental kinds of signatures: type signatures (see §8.6.1.1), location signatures (see §8.6.1.2), parameter signatures (see §8.6.1.4), and method signatures (see §8.6.1.5). (A fifth kind, a local signature (see §8.6.1.3) is really a version of a location signature.)

CLS Rule 11: All types appearing in a signature shall be CLS-compliant. All types composing an instantiated generic type 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. The visibility and accessibility of types composing an instantiated generic type used in the signature of any member shall be visible and accessible whenever the member itself is visible and accessible. For example, an instantiated generic type present in the signature of a member that is visible outside its assembly shall not have a generic 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 exported types and their members. *end note*]

The following subclauses 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 or from the operation that computes it. 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 made because certain constraints, such as "constant," are constraints on values not locations. Future versions of this standard, or non-standard extensions, can introduce type constraints, thus making the distinction meaningful. *end rationale*]

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 can additionally contain the constant constraint. The location signature can 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 can be stored in the location. The contents are always identical to the initialized value (see §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 can 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 can 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. *end note*]

[*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 can be encoded using a custom attribute (see §9.7), although this standard does not specify such an attribute. *end note*]

 ©ISO/IEC 2006 – All rights reserved

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 can specify one additional constraint:

The **byref** constraint states that the content of the corresponding location is a **managed pointer**. A managed pointer can 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 §12.5) a conforming implementation can 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 can 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 statically as part of the type description), a typed reference provides the type information dynamically. A typed reference is a full signature in itself and cannot 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, or return value.

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 exported members. *end note*]

8.6.1.4 Parameter signatures

A **parameter signature**, defines 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

A **method signatures** is composed of

- a calling convention,
- the number of generic parameters, if the method is generic,
- 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 **vararg** constraint can 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: as part of a method definition and as a description of a calling site when calling through a function pointer. In the latter case, the method signature indicates

- the calling convention (which can include platform-specific calling conventions),
- the types of all the argument values that are being passed, and

- if needed, a vararg marker indicating where the fixed parameter list ends and the variable parameter list begins.

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

CLS Rule 15: The vararg 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 vararg methods or unmanaged calling conventions.

CLS (framework): Neither vararg methods nor methods with unmanaged calling conventions shall be exported externally. *end note*]

8.7 Assignment compatibility

Assignment compatibility refers to the ability to store a value V (statically described by a type signature) into a location L (described by a location signature), and is abbreviated L := V. Because the type signature for V is described statically, V might not actually be a value of the type described by the signature, but rather something compatible with that type. No location or value shall have type `System.Void`.

The formal description of assignment compatibility is captured in Partition III, where it is referred to as *verification type compatibility*. In essence, a value V is assignment compatible with a location L if it meets one of the following conditions:

- The exact static type referred to by the type signature of V matches the exact type of the location.
- V, described by the generic type signature G<U1,...,Un>, is assignment compatible with L, described by the generic type signature H<T1,…,Tn>, if and only if G=H, and for each generic parameter of G with a variance annotation of var_i we have:
  - var_i = none or Ti is a value type or Ti is a generic parameter: Ui is the same exact type as Ti
  - var_i = +: Ti := Ui (covariant: an instance of Ui can be stored in a location of type Ti)
  - var_i = -: Ui := Ti (contravariant: an instance of Ti can be stored in a location of type Ui).
- V, described by type signature U[] is assignment compatible with L, described by location signature T[], if and only if T and U are either reference types or interfaces, T := U, and the array types are either both vectors (zero-based, rank one) or neither is a vector and both have the same rank. [*Note*: This means that array types are covariant. *end note*]
- Vector V, described by type signature U[], is assignment compatible with System.Collections.Generic.IList<U>, and also System.Collections.Generic.IList<B> where B is the set of types supported by U.
- V's type signature is an enumeration type and L's location signature is the underlying type of that enumeration.
- The type signature of V is a byref (§8.6.1.3) to a signed integer and the location signature of L is a byref to an unsigned integer of the same size, and vice versa; e.g., int32& := uint32&.
- The type signature of V is an int32 and the location signature of L is a bool. [*Note*: The CLI stack only traces int32 values even if a short integer is pushed. See Partition III. *end note*]
- One of the types supported by the exact type of V is assignment compatible to the type of L. This allows, for example, an instance of a class that inherits from a base class (hence supports

 ©ISO/IEC 2006 – All rights reserved

the base class's type contract) to be stored into a location whose type is that of the base class. [*Note*: Recall that a location constraint is just a type constraint plus two additional possible constraints (literal and constant), and thus a location constraint can be converted into a type constraint in a natural way. *end note*] Under this set of rules, transitivity of assignment compatibility holds: if L := V and M := L, then M := V.

- Signed and unsigned integral primitive types can be assigned to each other; e.g., int8 := uint8 is valid. For this purpose, bool shall be considered compatible with uint8 and vice versa, which makes bool := uint8 valid, and vice versa. This is also true for arrays of signed and unsigned integral primitive types of the same size; e.g., int32[] := uint32[] is valid.

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 **type-safe**. 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.

Type-safe implementations only store values described by a type signature in a location that is assignment-compatible (§8.7) with the location signature of the location (see §8.6.1). Type-safe implementations never apply an operation to a value that is not defined by the exact type of the value. Type-safe implementations only access locations that are both visible and accessible to them. In a type-safe implementation, the exact type of a value cannot change.

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

There are many different processes that can be the basis of verification. The simplest possible process simply says that all implementations are not type-safe. While correct and efficient this is clearly not particularly useful. By spending more resources (time and space) a process can correctly identify more type-safe implementations. It has been proven, however, that no mechanical process can, in finite time and with no errors, correctly identify all implementations as either type-safe or not type-safe. 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 type safety 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 class types, which can be either of the following:
  - object types (including delegates)
  - 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 §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 §8.2.3. *end note*]

Class, interface, and value type definitions can be parameterized, a feature known as *generic type definitions*. That is, the definition of a class, interface, or value type can include generic parameters. When used, a specific instantiation of the generic class, interface, or value type is made, at which point the generic parameters are bound to specific generic arguments. The generic parameters can be constrained, so that only generic arguments that match these constraints can be used for instantiations.

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 only at runtime, the signature can specify the information that is known at compile time (e.g., no bounds, a lower bound, or both an upper and a 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 can include platform-specific padding. (The size of this storage, in bytes, is returned by the sizeof instruction when it is applied to the type of that array's elements.)

Values of an array type are objects; hence an array type is a kind of object type (see §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.

Additionally, a created vector with element type `T`, implements the interface `System.Collections.Generic.IList<U>` (§8.7), where `U := T`.

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 can 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 exported 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. *end note*]

Array types form a hierarchy, with all array types inheriting from the type `System.Array`. This is an abstract class (see §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. However, the VES treats single

 ©ISO/IEC 2006 – All rights reserved

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 §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 type-safe 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 to 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 exported. *end note*]

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 one or more compatible static or instance methods on particular objects. The objects and methods to which it delegates are chosen when the delegate instance is created.

In addition to an instance constructor and an **Invoke** method, delegates can 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 can 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 can 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. *end note*]

Interface 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 can declare static fields (see §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 can—and usually does—define method contracts (method name and method signature) that shall be implemented by supporting types. An interface type definition can define and implement static methods (see §8.4.3) since static methods are associated with the interface type itself rather than with any value of the type.

Interfaces can 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 can 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 can be defined that provides them. *end note*]

Interface types can 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 (§8.6), the above rules for method definitions apply. For more information, see §8.11.4 and §8.11.3.

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

An interface type is given a visibility attribute, as described in §8.5.3, that controls from where the interface type can 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 can be attached to members or to the interface itself.

 ©ISO/IEC 2006 – All rights reserved

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. Unless it specifies that the class is an **abstract object type**, a class definition 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 §8.2.3. *end note*]

An explicit class definition is used to define:

- An object type (see §8.2.3).
- A value type and its associated boxed type (see §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 §8.5.2).
- Defines the class contract of the same name (see §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 §8.11).
- Defines the static members of the class type (see §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 §8.5.3).
- Can optionally specify a method (called `.cctor`) 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 can have a type-initializer method, or not.
2. A type can 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):
   - first access to any static field of that type, or
   - first invocation of any static method of that type or
   - first invocation of any constructor for 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

For reference types, a constructor has to be called to create a non-null instance. Thus, for reference types, the .cctor will be called before instance fields can be accessed and methods can be called on non-null instances. For value types, an "all-zero" instance can be created without a constructor (but only this value can be created

without a constructor). Thus for value types, the .cctor is only guaranteed to be called for instances of the value type that are not "all-zero". [*Note*: This changes the semantics slightly in the reference class case from the first edition of this standard, in that the .cctor might not be called before an instance method is invoked if the 'this' argument is null. The added performance of avoiding class constructors warrants this change. *end note*]

[*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 base class's 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 base class and/or interfaces it implements, or
- by making explicit calls to `System.Runtime.CompilerServices.RuntimeHelpers.RunClassConstructor` (see Partition IV).

end note]

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 §8.5.2. Similarly, object type definitions shall also explicitly state the visibility attribute of the object type (either **public**or **assembly**); see §8.5.3.

8.9.6.2 Concreteness

An object type can be marked as **abstract** by the object type definition. An object type that is not marked **abstract** is, by definition, **concrete**. Only object types can 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 §8.11). 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 can be concrete if it provides implementations for all abstract methods in the base object type and is not itself marked as abstract. Instances can be made of such a concrete derived class. Locations can have an abstract type, and instances of a concrete type that derives from the abstract type can 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 can be invoked, properties that are available, and events that can be raised. Each member of a type can have attributes as described in §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 8.4.1). Static fields specify fields associated with the object type itself (see §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 §8.5.2). Fields are declared using a field definition (see §8.11.2).

 ©ISO/IEC 2006 – All rights reserved

- Methods of an object type specify operations on values of the type (see §8.4.2). Static methods specify operations on the type itself (see §8.4.3). Methods are named and they have a method signature. The names of methods are scoped to the type (see §8.5.2). Methods are declared using a method definition (see §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 §8.5.2). Properties are declared using a property definition (see §8.11.3).
- Events of an object type specify named state transitions in which subscribers can 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 §8.5.2). Events are declared using an event definition (see §8.11.4).

8.9.6.4 Supporting interface contracts

Object type definitions can 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 `Foo` 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 can 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 members.

[*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. If a CLS-compliant framework exposes a class implementing a non-CLS-compliant interface, the framework shall provide concrete implementations of all non-CLS-compliant members. This ensures that CLS extenders do not need syntax for implementing non-CLS-compliant members. *end note*]

8.9.6.5 Supporting class contracts

Object type definitions can declare support for one other class contract. Declaring support for another class contract is synonymous with object type inheritance (see §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 can 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. It can issue a compiler error if the constructor does not obey these rules.

CLS (framework): Can assume that object creation includes a call to one of the constructors, and that no object is initialized twice. `System.Object.MemberwiseClone` (see Partition IV) and deserialization (including object remoting) shall not run constructors. *end note*]

8.9.6.7 Finalizers

A class definition that creates an object type can supply an instance method (called a *finalizer*) to be called when an instance of the class is no longer reachable. 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 can 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. *end rationale*]

It is valid to define a finalizer for a value type. However, that finalizer will only be run for *boxed* instances of that value type.

[*Note:* Since programmers might depend on finalizers to be called, the CLI should make every effort, before it shuts down, to ensure that finalizers are called 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. *end note*]

[*Note:* Since resources might 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. *end note*]

8.9.7 Value type definition

Not all types defined by a class definition are object types (see §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 §8.2.4). The members of the class definition define the representation of both:

 ©ISO/IEC 2006 – All rights reserved

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.

2. Value types do not support interface contracts, but their associated boxed types do.

3. A value type does not inherit; rather the base type specified in the class definition defines the base type of the boxed type.

4. The base type of a boxed type shall not have any fields.

5. 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 implements 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 can 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 (i.e., 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`. The introduction of generic types makes it more difficult to give a precise definition; see §Partition II.

An object type declares that 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 array object types are manufactured by the VES, the inheritance of arrays is fixed. See §8.9.1.

8.9.10 Value type inheritance

In their unboxed form value types 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 derivation

Interface types can require the implementation of one or more other interfaces. Any type that implements support for an interface type shall also implement support for any required interfaces specified by that interface. 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; required interfaces do not, since interfaces do not define implementation. Required interfaces specify 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 derives 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 can inherit implementations, hence only object types can inherit members (see §8.9.8). While interface types can be derived from other interface types, they 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 can be read or written. The type visibility, field accessibility, and security attributes of the field definition (see §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 can 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 §8.5.3) determine if a method is accessible to the derived object type.

A derived object type can 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 can still be invoked, subject to method accessibility rules, since the type that contains the method always qualifies a method reference.

Virtual methods can 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 can 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 §8.5.3.

8.10.3 Property and event inheritance

Fundamentally, properties and events are constructs of the metadata intended for use by tools that target the CLI and are not directly supported by the VES itself. Therefore, it is 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.

 ©ISO/IEC 2006 – All rights reserved

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 can 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 from 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

©ISO/IEC 2006 – All rights reserved

- 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 can 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 shall only be defined on object types. Properties 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 §8.6.1). The implementation is the code to execute when the method is invoked. A value type or object type shall define only one method of a given name and signature. However, a derived object type can have methods that are of the same name and signature as its base object type. See §8.10.2 and §8.10.4.

The name of the method is scoped to the type (see §8.5.2). Methods can be given accessibility attributes (see §8.5.3). Methods shall only be invoked with arguments that are assignment compatible with the parameter 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 can 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 can be marked as **virtual**, indicating that an alternate implementation can be provided in derived types. All non-static method definitions in interface definitions shall be virtual methods. Virtual method can be marked as **final**, indicating that derived object types are not allowed to override the method implementation.

Method definitions can be parameterized, a feature known as *generic method definitions*. When used, a specific instantiation of the generic method is made, at which point the generic parameters are bound to specific generic arguments. Generic methods can be defined as members of a non-generic type; or can be defined as members of a generic type, but parameterized by different generic parameter (or parameters) than its owner type. For example, the `Stack<T>` class might include a generic method `S ConvertTo<S> ()`, where the `S` generic parameter is distinct from the `T` generic parameter in `Stack<T>`.

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 §8.6.1. A value type or object type shall define only one field of a given name and type. However, a derived object type can have fields that are of the same name and type as its base object type. See §8.10.1 and §8.10.4.

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

 ©ISO/IEC 2006 – All rights reserved

Fields can 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 §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 require that a conforming implementation provide support for serialization (or its counterpart, deserialization), not does it specify the mechanism by which these operations might be 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 type that includes the property definition. 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: No longer used. [*Note*: In an earlier version of this standard, this rule stated "The accessibility of a property and of its accessors shall be identical." The removal of this rule allows, for example, public access to a getter while restricting access to the setter. *end note*]

CLS Rule 26: A property's 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 §10.4. The `SpecialName` attribute referred to in CLS rule 24 shall be ignored in appropriate name comparisons and shall adhere to identifier rules. A property shall have a getter method, a setter method, or both.

[*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. *end note*]

8.11.4 Event definitions

The CTS supports events in precisely the same way that it supports properties (see §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 §10.4. The `SpecialName` attribute referred to in CLS rule 29 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. *end note*]

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 §8.5.3.

 ©ISO/IEC 2006 – All rights reserved

9 Metadata

This clause and its subclauses contain only informative text, with the exception of the CLS rules introduced here and repeated in §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 can help a developer to understand the structure of metadata, the rules outlined in this clause 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 subclause. 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 II and Partition III.

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 (such as type declaration and member declaration). 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 can 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:
    - A method implementation to be located by implementation-specific means described outside this Standard.
    - 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:

 ©ISO/IEC 2006 – All rights reserved

- **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 might have been specified is ignored. This is the default.
- **sequentiallayout**: A class marked `sequentiallayout` 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 can 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 valid 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 can 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. Generic types shall not be marked `explicitlayout`.

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 exported 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 can also supply runtime configuration information. A CLI module can be thought of as a packaging of type declarations and implementations, where the packaging decisions can 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 can 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 can be deployed by itself, with the assumption that any other referenced assemblies will be available in the deployed environment. Or, it can be deployed with its dependent assemblies.

Manifests: Every assembly has a manifest that declares which 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 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 headers 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 can 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 can 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 can 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).

 ©ISO/IEC 2006 – All rights reserved

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

- 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 can 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) it does not understand.

[*Note*:

CLS (consumer): Shall be able to read metadata containing optional modifiers and correctly copy signatures that include them. Can ignore these modifiers in type matching and overload resolution. Can 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. *end note*]

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. *end note*]

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 or 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 exported to other languages, **private** members or types that aren't exported from an assembly can 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 might 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 they are persisted in Canonical form C, which preserves case but forces combining characters into a standard representation. See CLS Rule 4, in §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 §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:* Although the CTS describes inheritance, object layout, name hiding, and overriding of virtual methods, it 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. *end note*]

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 can refer to any number of methods provided they differ in any of the following:

- Number of parameters
- Type of any parameter

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

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

 ©ISO/IEC 2006 – All rights reserved

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

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

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

[*Note*:

CLS (consumer): Can 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` can be ignored since the functionality shall be provided in some other way by a CLS-compliant framework. Consumers must first apply the hide-by-name and hide-by-signature-and-name rules (§8.10.4) to avoid any ambiguity.

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. *end note*]

[*Note*: The names visible on any class `C`, are the names visible in that class and its base classes. As a consequence, the names of methods on interfaces implemented by `C` that are only implemented via MethodImpls (see Partition II) are not visible on class `C`. The names visible on an interface `I`, consist only of the names directly defined on this interface. As a consequence, the names of methods from other interfaces (which `I` requires be implemented) are not visible on `I` itself. *end note*]

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.

end note]

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 operand, 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. Table 4: Unary Operator Names shows the names that are defined.

Table 4: Unary Operator Names

Name	ISO/IEC 14882:2003 C++ Operator Symbol (This column is informative.)
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 operands, perform some operation on them, and return a value. They are represented as static methods on the class that defines the type of one of their two operands. Table 5: Binary Operator Names shows the names that are defined.

Table 5: Binary Operator Names

Name	ISO/IEC 14882:2003 C++ Operator Symbol (This column is informative.)
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	Not defined (= is not the same)
op_LeftShift	<<
op_RightShift	>>
op_SignedRightShift	Not defined
op_UnsignedRightShift	Not defined
op_Equality	==

 ©ISO/IEC 2006 – All rights reserved

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 can 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 do 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. It is recommended that the more common ToXxx (where Xxx is the target type) and FromYyy (where Yyy is the name of the source type) naming pattern be used. *end note*]

Because these operations can 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. *end note*]

10.4 Naming patterns

See also Partition VI.

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 indicate 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 indicate 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 can 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 §12.4.2. CLS-compliant frameworks can 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 a type inheriting 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 the throwing of objects having other types.

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

 ©ISO/IEC 2006 – All rights reserved

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 rule defined by the CLS:

CLS Rule 41: Attributes shall be of type `System.Attribute`, or a type inheriting 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. *end note*]

The use of a particular attribute class can 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.) can 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 can be combined.
- Multiple instances of the attribute class can 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 base class and those explicitly applied to the derived class type. If multiple instances are not allowed, then an attribute of that type applied directly to the derived class overrides the attribute supplied by the base class. 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. *end note*]

10.7 Generic types and methods

The following subclauses describe the CLS rules for generic types and methods.

10.7.1 Nested type parameter re-declaration

Any type exported by a CLS-compliant framework, that is nested in a generic type, itself declares, by position, all the generic parameters of that enclosing type. (The nested type can also introduce new generic parameters.) As such, any CLS-compliant type nested inside a generic type is itself generic. Such redeclared generic parameters shall precede any newly introduced generic parameters. [*Example:* Consider the following C# source code:

```
public class A<T> {
  public class B {}
  public class C<U,V> {
    public class D<W> {}
  }
}
public class X {
  public class Y<T> {}
}
```

The relevant corresponding ILAsm code is:

```
.class … A`1<T> … {                          // T is introduced
  .class … nested … B<T> … { }               // T is redeclared
  .class … nested … C`2<T,U,V> … {           // T is redeclared; U and V are introduced
    .class … nested … D`1<T,U,V,W> … { }     // T, U, and V are redeclared; W is introduced
  }
}

.class … X … {
  .class … nested Y`1<T> … { }               // Nothing is redeclared; T is introduced
}
```

As generic parameter re-declaration is based on parameter position matching, not on parameter name matching, the name of a redeclared generic parameter need not be the same as the one it re-declares. For example:

```
.class … A`1<T> … {                          // T is introduced
  .class … nested … B<Q> … { }               // T is redeclared (as Q)
  .class … nested … C`2<T1,U,V> … {          // T is redeclared (as T1); U and V
                                             // are introduced
    .class … nested … D`1<R1,R2,R3,W> … { }  // T1, U, and V are redeclared (as R1, R2,
                                             // and R3); W is introduced
  }
}
```

A CLS-compliant Framework should therefore expose the following types:

Lexical Name	Total Generic Parameters	Redeclared Generic Parameters	Introduced Generic Parameters
A<T>	1 (T)	0	1 T
A<T>.B	1 (T)	1 T	0
A<T>.C<U,V>	3 (T,U,V)	1 T	2 U,V
A<T>.C<U,V>.D<W>	4 (T,U,V,W)	3 T,U,V	1 W
X	0	0	0
A.Y<T>	1 (T)	0	1 T

end example]

CLS Rule 42: Nested types shall have at least as many generic parameters as the enclosing type. Generic parameters in a nested type correspond by position to the generic parameters in its enclosing type.

[*Note*:

CLS (consumer): Need not consume types that violate this rule.

CLS (extender): Same as consumers. Extenders choosing to support definition of types nested in generic types shall follow this rule for externally visible types.

CLS (framework): Shall not expose types that violate this rule. *end note*]

10.7.2 Type names and arity encoding

CLS-compliant generic type names are encoded using the format "*name***[**`*arity***]**" , where **[**…**]** indicates that the grave accent character "`" and *arity* together are optional. The encoded name shall follow these rules:

1. *name* shall be an *ID* (see Partition II) that does not contain the "`" character.

2. *arity* is specified as an unsigned decimal number without leading zeros or spaces.

3. For a normal generic type, *arity* is the number of type parameters declared on the type.

4. For a nested generic type, *arity* is the number of newly introduced type parameters.

[*Example:* Consider the following C# source code:

 ©ISO/IEC 2006 – All rights reserved

```
public class A<T> {
  public class B {}
  public class C<U,V> {
    public class D<W> {}
  }
}

public class X {
  public class Y<T> {}
}
```

The relevant corresponding ILAsm code is:

```
.class … A`1<T> … {                          // T is introduced
  .class … nested … B<T> … { }               // T is redeclared
  .class … nested … C`2<T,U,V> … {           // T is redeclared; U and V are introduced
    .class … nested … D`1<T,U,V,W> … { }     // T, U, and V are redeclared; W is introduced
  }
}

.class … X … {
  .class … nested Y`1<T> … { }               // Nothing is redeclared; T is introduced
}
```

A CLS-compliant Framework should expose the following types:

Lexical Name	Total Generic Parameters	Redeclared Generic Parameters	Introduced Generic Parameters	Metadata Encoding
A<T>	1 (T)	0	1 T	A`1
A<T>.B	1 (T)	1 T	0	B
A<T>.C<U,V>	3 (T,U,V)	1 T	2 U,V	C`2
A<T>.C<U,V>.D<W>	4 (T,U,V,W)	3 T,U,V	1 W	D`1
X	0	0	0	X
A.Y<T>	1 (T)	0	1 T	Y`1

While a type name encoded in metadata does not explicitly mention its enclosing type, the CIL and Reflection type name grammars do include this detail:

Lexical Name	Metadata Encoding	CIL	Reflection
A<T>	A`1	A`1	A`1[T]
A<T>.B	B	A`1/B	A`1+B[T]
A<T>.C<U,V>	C`2	A`1/C`2	A`1+C`2[T,U,V]
A<T>.C<U,V>.D<W>	D`1	A`1/C`2/D`1	A`1+C`2+D`1[T,U,V,W]
X	X	X	X
A.Y<T>	Y`1	X/Y`1	X+Y`1[T]

end example]

CLS Rule 43: The name of a generic type shall encode the number of type parameters declared on the non-nested type, or newly introduced to the type if nested, according to the rules defined above.

[*Note*:

CLS (consumer): Need not consume types that violate this rule.

CLS (extender): Same as consumers. Extenders choosing to support definition of generic types shall follow this rule for externally visible types.

CLS (framework): Shall not expose types that violate this rule. *end note*]

10.7.3 Type constraint re-declaration

CLS Frameworks shall ensure that a generic type explicitly re-declares any constraints present on generic parameters in its base class and all implemented interfaces. Put another way, CLS Extenders and Consumers should be able to examine just the specific type in question, to determine the set of constraints that need to be satisfied.

CLS Rule 44: A generic type shall redeclare sufficient constraints to guarantee that any constraints on the base type, or interfaces would be satisfied by the generic type constraints.

[*Note*:

CLS (consumer): Need not consume types that violate this rule. Consumers who check constraints need only look at the type being instantiated to determine the applicable constraints.

CLS (extender): Same as consumers. Extenders choosing to support definition of generic types shall follow this rule.

CLS (framework): Shall not expose types that violate this rule. *end note*]

10.7.4 Constraint type restrictions

CLS Rule 45: Types used as constraints on generic parameters shall themselves be CLS-compliant.

[*Note*:

CLS (consumer): Need not consume types that violate this rule.

CLS (extender): Same as consumers. Extenders choosing to support definition of generic types shall follow this rule when checking for CLS compliance, and need not provide syntax to violate this rule.

CLS (framework): Shall not expose types that violate this rule. *end note*]

10.7.5 Frameworks and accessibility of nested types

CLI generics treat the generic type declaration and all instantiations of that generic type as having the same accessibility scope. However, language accessibility rules may differ in this regard, with some choosing to follow the CLI accessibility model, while others use a more restrictive, per-instantiation model. To enable consumption by all CLS languages, CLS frameworks shall be designed with a conservative per-instantiation model of accessibility in mind, and not expose nested types or require access to protected members based on specific, alternate instantiations of a generic type.

This has implications for signatures containing nested types with **family** accessibility. Open generic types shall not expose fields or members with signatures containing a specific instantiation of a nested generic type with family accessibility. Non-generic types extending a specific instantiation of a generic base class or interface, shall not expose fields or members with signatures containing a different instantiation of a nested generic type with family accessibility. [*Example:* Consider the following C# source code:

```
public class C<T> {
  protected class N {…}
  protected void M1(C<int>.N n) {…}   // Not CLS-compliant - C<int>.N not
                                      // accessible from within C<T> in all languages
  protected void M2(C<T>.N n) {…}     // CLS-compliant – C<T>.N accessible inside C<T>
}

public class D : C<long> {
  protected void M3(C<int>.N n) {…}   // Not CLS-compliant – C<int>.N is not
                                      // accessible in D (extends C<long>)
  protected void M4(C<long>.N n) {…}  // CLS-compliant, C<long>.N is
                                      // accessible in D (extends C<long>)
}
```

The relevant corresponding ILASM code is:

 ©ISO/IEC 2006 – All rights reserved

```
.class public … C`1<T> … {
  .class … nested … N<T> … {}
  .method family hidebysig instance void  M1(class C`1/N<int32> n) … {}
  // Not CLS-compliant - C<int>.N is not accessible from within C<T> in all languages

  .method family hidebysig instance void  M2(class C`1/N<!0> n) … {}
  // CLS-compliant – C<T>.N is accessible inside C<T>
}

.class public … D extends class C`1<int64> {
  .method family hidebysig instance void  M3(class C`1/N<int32> n) … {}
  // Not CLS-compliant – C<int>.N is not accessible in D (extends C<long>)

  .method family hidebysig instance void  M4(class C`1/N<int64> n) … {}
  // CLS-compliant, C<long>.N is accessible in D (extends C<long>)
}
```

end example]

CLS Rule 46: The visibility and accessibility of members (including nested types) in an instantiated generic type shall be considered to be scoped to the specific instantiation rather than the generic type declaration as a whole. Assuming this, the visibility and accessibility rules of CLS rule 12 still apply.

[*Note*:

CLS (consumer): Need not consume types that violate this rule.

CLS (extender): Shall use this more restrictive notion of accessibility when determining CLS compliance.

CLS (framework): Shall not expose members that violate this rule. *end note*]

10.7.6 Frameworks and abstract or virtual methods

CLS Frameworks shall not expose libraries that require CLS Extenders to override or implement generic methods to use the framework. This does not imply that virtual or abstract generic methods are non-compliant; rather, the framework shall also provide concrete implementations with appropriate default behavior.

CLS Rule 47: For each abstract or virtual generic method, there shall be a default concrete (non-abstract) implementation.

[*Note*:

CLS (consumer): No impact.

CLS (extender): Need not provide syntax for overriding generic methods.

CLS (framework): Shall not expose generic methods that violate this rule without also providing appropriate concrete implementations. *end note*]

11 Collected Common Language Specification 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 can 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.

CLS Rule 1: CLS rules apply only to those parts of a type that are accessible or visible outside of the defining assembly. (§0)

CLS Rule 2: Members of non-CLS compliant types shall not be marked CLS-compliant. (§7.3.1)

CLS Rule 3: Boxed value types are not CLS-compliant. (§8.2.4.)

CLS Rule 4: Assemblies shall follow Annex 7 of Technical Report 15 of the Unicode Standard 3.0 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, one-to-one 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. (§8.5.1)

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. (§8.5.2)

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. (§8.5.2)

CLS Rule 7: The underlying type of an enum shall be a built-in CLS integer type, the name of the field shall be "value__", and that field shall be marked `RTSpecialName`. (§8.5.2)

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 represents named bit flags that can be combined to generate an unnamed value. The value of an enum is not limited to the specified values. (§8.5.2)

CLS Rule 9: Literal static fields (see §8.6.1) of an enum shall have the type of the enum itself. (§8.5.2)

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**. (§8.5.3.2)

CLS Rule 11: All types appearing in a signature shall be CLS-compliant. All types composing an instantiated generic type shall be CLS-compliant. (§8.6.1)

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. The visibility and accessibility of types composing an instantiated generic type used in the signature of any member shall be visible and accessible whenever the member itself is visible and accessible. For example, an instantiated generic type present in the signature of a member that is visible outside its assembly shall not have a generic argument whose type is visible only within the assembly. (§8.6.1)

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

 ©ISO/IEC 2006 – All rights reserved

that is of exactly the same type as the literal (or of the underlying type, if that literal is an **enum**). (§8.6.1.2)

CLS Rule 14: Typed references are not CLS-compliant. (§8.6.1.3)

CLS Rule 15: The vararg constraint is not part of the CLS, and the only calling convention supported by the CLS is the standard managed calling convention. (§8.6.1.5)

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. (§8.9.1)

CLS Rule 17: Unmanaged pointer types are not CLS-compliant. (§8.9.2)

CLS Rule 18: CLS-compliant interfaces shall not require the definition of non-CLS compliant methods in order to implement them. (§8.9.4)

CLS Rule 19: CLS-compliant interfaces shall not define static methods, nor shall they define fields. (§8.9.4)

CLS Rule 20: CLS-compliant classes, value types, and interfaces shall not require the implementation of non-CLS-compliant members. (§8.9.6.4)

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.) (§8.9.6.6)

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. (§8.9.6.6)

CLS Rule 23: `System.Object` is CLS-compliant. Any other CLS-compliant class shall inherit from a CLS-compliant class. (§8.9.9)

CLS Rule 24: The methods that implement the `getter` and `setter` methods of a property shall be marked `SpecialName` in the metadata. (§8.11.3)

CLS Rule 25: No longer used. [*Note*: In an earlier version of this standard, this rule stated "The accessibility of a property and of its accessors shall be identical." The removal of this rule allows, for example, public access to a getter while restricting access to the setter. *end note*] (§8.11.3)

CLS Rule 26: A property's accessors shall all be static, all be virtual, or all be instance. (§8.11.3)

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). (§8.11.3)

CLS Rule 28: Properties shall adhere to a specific naming pattern. See §10.4. The `SpecialName` attribute referred to in CLS rule 24 shall be ignored in appropriate name comparisons and shall adhere to identifier rules. A property shall have a getter method, a setter method, or both. (§8.11.3)

CLS Rule 29: The methods that implement an event shall be marked `SpecialName` in the metadata. (§8.11.4)

CLS Rule 30: The accessibility of an event and of its accessors shall be identical. (§8.11.4)

CLS Rule 31: The `add` and `remove` methods for an event shall both either be present or absent. (§8.11.4)

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`. (§8.11.4)

CLS Rule 33: Events shall adhere to a specific naming pattern. See §10.4. The `SpecialName` attribute referred to in CLS rule 29 shall be ignored in appropriate name comparisons and shall adhere to identifier rules. (§8.11.4)

©ISO/IEC 2006 – All rights reserved

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. (§9.7)

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) it does not understand. (§9.7)

CLS Rule 36: Global static fields and methods are not CLS-compliant. (§9.8)

CLS Rule 37: Only properties and methods can be overloaded. (§10.2)

CLS Rule 38: Properties and methods can be overloaded based only on the number and types of their parameters, except the conversion operators named `op_Implicit` and `op_Explicit`, which can also be overloaded based on their return type. (§10.2)

CLS Rule 39: If either `op_Implicit` or `op_Explicit` is provided, an alternate means of providing the coercion shall be provided. (§10.3.3)

CLS Rule 40: Objects that are thrown shall be of type `System.Exception` or a type inheriting from it. Nonetheless, CLS-compliant methods are not required to block the propagation of other types of exceptions. (§10.5)

CLS Rule 41: Attributes shall be of type `System.Attribute`, or a type inheriting from it. (§10.6)

CLS Rule 42: Nested types shall have at least as many generic parameters as the enclosing type. Generic parameters in a nested type correspond by position to the generic parameters in its enclosing type. (§10.7.1)

CLS Rule 43: The name of a generic type shall encode the number of type parameters declared on the non-nested type, or newly introduced to the type if nested, according to the rules defined above. (§10.7.2)

CLS Rule 44: A generic type shall redeclare sufficient constraints to guarantee that any constraints on the base type, or interfaces would be satisfied by the generic type constraints. (§10.7.3)

CLS Rule 45: Types used as constraints on generic parameters shall themselves be CLS-compliant. (§10.7.4)

CLS Rule 46: The visibility and accessibility of members (including nested types) in an instantiated generic type shall be considered to be scoped to the specific instantiation rather than the generic type declaration as a whole. Assuming this, the visibility and accessibility rules of CLS rule 12 still apply. (§10.7.5)

CLS Rule 47: For each abstract or virtual generic method, there shall be a default concrete (non-abstract) implementation. (§10.7.6)

Error! Reference source not found. (§7.2.1)

 ©ISO/IEC 2006 – All rights reserved

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 CIL 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 two's-complement signed value
`unsigned int8`	8-bit unsigned binary value
`int16`	16-bit two's-complement signed value
`unsigned int16`	16-bit unsigned binary value
`int32`	32-bit two's-complement signed value
`unsigned int32`	32-bit unsigned binary value
`int64`	64-bit two'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 two'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 (can 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`, and `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 §12.1.3 and Partition III. Short numeric values (`int8`, `int16`, `unsigned int8`, and `unsigned int16`) are widened when loaded and narrowed when stored. 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 can 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, stelem, stind
- Data conversion: conv, conv.ovf

- Array creation: newarr

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

This instruction set design simplifies CIL-to-native code (e.g., JIT) compilers and interpreters of CIL by allowing them to internally track a smaller number of data types. See §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 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; however, 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`) can 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. *end rationale*]

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` data type 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 `&` data type (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 (`&`) can be changed during garbage collection, since the data to which they refer might be moved.

 ©ISO/IEC 2006 – All rights reserved

[*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 "byref 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. *end note*]

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.un instruction be used to convert the pointer from its memory format before its use as a pointer. This might 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 an `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; however, it also has a memory model that encompasses 1- and 2-byte integers. To be more precise, the following rules are part of the CLI model:

- Loading from 1- 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- 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 argument is larger than the parameter.
- 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 data types

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** (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
```

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. *end note*]

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. For comparison purposes, **NaN** is 'unordered' (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, can 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". Class libraries are 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 `float32`or `float64`, but its value can be represented internally with additional range and/or precision. The size of the internal floating-point representation is implementation-dependent, can 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. *end note*]

[*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 might be able to leave floating-point variables in hardware registers that provide more precision than a user has requested. At the

 ©ISO/IEC 2006 – All rights reserved

same time, CIL generators can force operations to respect language-specific rules for representations through the use of conversion instructions. *end rationale*]

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 can involve a loss of precision or the creation of an out-of-range value (NaN, +infinity, or -infinity). However, the value might 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 (§12.6)). 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) can be used, followed by a check for a non-finite value using ckfinite. Underflow can be detected by converting to a particular storage type, comparing to zero before and after the conversion. *end note*]

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

12.1.4 CIL instructions and numeric types

This subclause contains only informative text

Most CIL instructions that deal with numbers take their operands from the evaluation stack (see §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 §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). The notation "[.s]" means both the long and short forms of these instructions.

©ISO/IEC 2006 – All rights reserved

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

End informative text

12.1.5 CIL instructions and pointer types

This subclause 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 can only be used in certain ways. The o (object reference) and & (managed pointer) data types are the formalization of these restrictions. *end rationale*]

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)

 ©ISO/IEC 2006 – All rights reserved

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. If other uses of arithmetic on managed pointers are made, the behavior 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 rationale*]

End informative text

12.1.6 Aggregate data

This subclause 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 §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	JIT compile time Type Determination
Argument	ldarga for by-value arguments or ldarg for byref arguments	Method signature
Local Variable	ldloca for by-value locals or ldloc for byref 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 can 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 `System.Object`; in this respect, they act like the built-in types int32, int64, 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 **localsinit** 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 §12.1.6.2.2.

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.

 ©ISO/IEC 2006 – All rights reserved

- **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 reference type (rather than a value type) expects a **this** pointer that is an instance of that class. This makes sense for reference types, 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 byref parameter of the value type rather than an ordinary by-value parameter.

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

- For unboxed instances of a value type, the exact type is known 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 base 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

Boxing and **unboxing** are conceptually equivalent to (and can 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 type-safe) 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 can be generated) operation that converts a `System.Object` (whose exact 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 §12.4.2.
- The size of the evaluation stack that the method will require.
- The size of the locals array that the method will require.
- A "localsinit flag" that indicates whether the local variables and memory pool (§12.3.2.4) should be initialized by the CLI (see also localloc §III.3.47).
- 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 can be described:

- Restriction to a specific 32-bit 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. *end note*]

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 can be visible to multiple threads, and modification of such values is considered a side-effect.

 ©ISO/IEC 2006 – All rights reserved

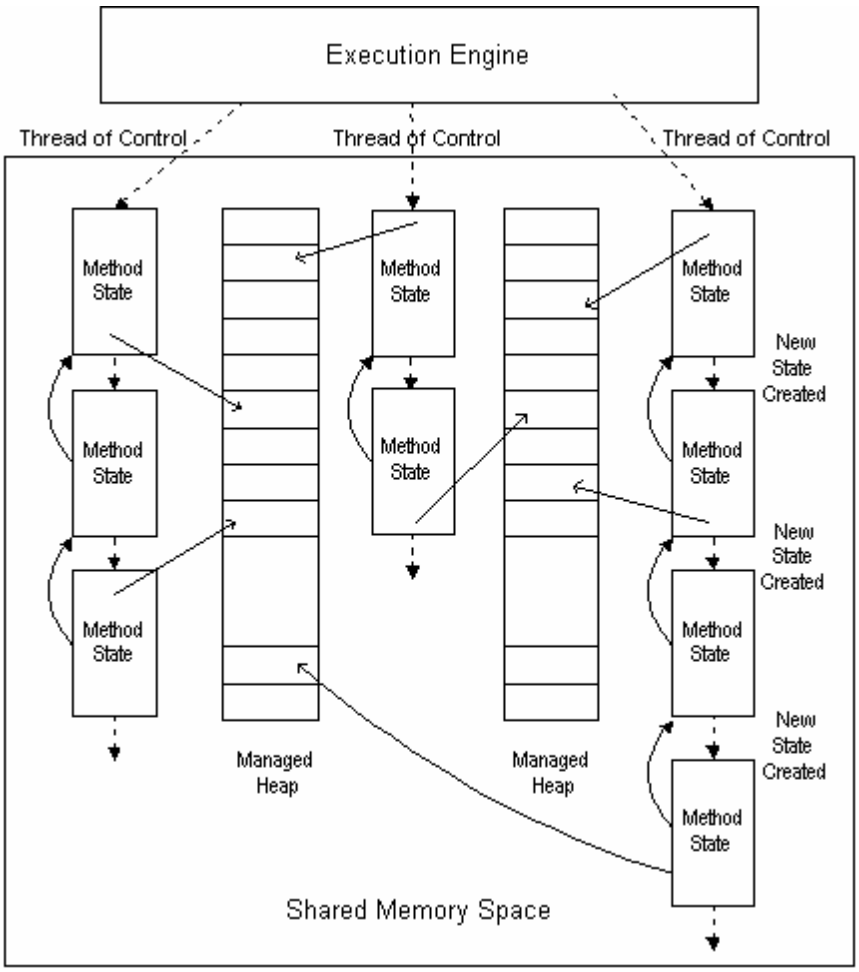


Figure 2: Machine State Model

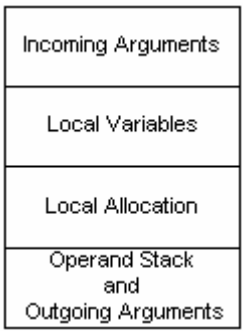


Figure 3: Method State

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 §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 can 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 §12.3.2.1). Local variables are initialized to 0 before entry if the localsinit flag for the method is set (see §12.2). The address of an individual local variable can 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 type-safe 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 can 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 §12.3.2.2) to the method. This can 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 §12.1, the CLI treats the evaluation stack in a special way. While some JIT compilers might 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

 ©ISO/IEC 2006 – All rights reserved

- `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 can 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 can 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 `&`) can be explicitly converted to an unmanaged pointer (type `native unsigned int`), although this is not verifiable and might 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 can be 64-bit aligned, while on others they can 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 implementations might 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 ("vararg methods"). Access to these arguments is through a type-safe iterator in the library, called `System.ArgIterator` (see Partition IV).

The CIL includes one instruction provided specifically to support the argument iterator, arglist. This instruction shall only be used 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, vararg 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 can 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 §12.4.2).
- **Method call** instructions to compute new arguments, transfer them and control to a known or computed destination method (see §12.4.1).
- **Tail call** prefix to indicate that a method should relinquish its stack frame before executing a method call (see §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 §12.4.1).
- **Exception-related** instructions (see §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 §12.4.2) except through the exception handling mechanism.
2. Control transfer out of a protected region is covered in §12.4.2.
3. The evaluation stack shall be empty after the return value is popped by a **ret** instruction.
4. Regardless of the control flow that allows execution to arrive there, each slot on the stack shall have the same data type at any given point within the method body.
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, endfilter, endfault, or endfinally). The stack shall 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.

 ©ISO/IEC 2006 – All rights reserved

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 conventions. All method calls initialize the method state areas (see §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 localsinit flag is set in the method header, then the local variables array is initialized to 0 for all integer types and to 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 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 can be used to call static or instance methods or the (statically known) base class 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 derived class to be supplied and the method appropriate for that derived class to be invoked. The callvirt instruction is used both for instance methods and methods on interfaces. For further details, see the CTS specification and the CIL instruction set specification in Partition III.

In addition, each of these instructions can 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 can 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. *end note*]

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 an 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 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 compiler into a native calling convention. The JIT compiler 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 compiler 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 implementation 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 can 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 byref 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). §12.4.1.5 describes how each of the three parameter passing conventions (by-value, byref, 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 can be passed by-value while all others are passed byref). Parameters shall be passed in one of the following ways (see detailed descriptions below):

 ©ISO/IEC 2006 – All rights reserved

- **By-value** – where the **value** of an object is passed from the caller to the callee.
- **By-ref** – 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** – 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 §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 §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 byref parameters because they have no home.

The CLI provides instructions to support byref 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 §8.8) byref parameters shall **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 byref 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 byref parameter but the static data type is passed as well as the address of the data. Like byref 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 byref parameter could be implemented as a standard value type with two fields: the address of the data and the type of the data. *end note*]

Like a regular byref 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 byref parameters; and verification imposes the same restrictions on the use of typed reference parameters as it does on byref parameters (see §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 can be passed using any one of the parameter passing conventions: by-value, byref, or typed reference. No combination of these is allowed for a single parameter, although a method can have different parameters with different calling mechanisms.

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

A byref parameter can 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 exception objects are instances of some class (i.e., they can be boxed value types, but not pointers, unboxed value types, etc.). Users can create their own exception classes, typically by deriving from `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 **catch handler** that handles any exception of a specified class or any of its sub-classes.

 ©ISO/IEC 2006 – All rights reserved

- A **filter handler** that runs a user-specified set of CIL instructions to determine if the exception should be 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 `System.ExecutionEngineException` described below that can be generated by all instructions).

Base Instructions (see Partition III)

- `System.ArithmeticException`
- `System.DivideByZeroException`
- `System.ExecutionEngineException`
- `System.InvalidAddressException`
- `System.OverflowException`
- `System.SecurityException`
- `System.StackOverflowException`

Object Model Instructions (see Partition III)

- `System.TypeLoadException`
- `System.IndexOutOfRangeException`
- `System.InvalidAddressException`
- `System.InvalidCastException`
- `System.MissingFieldException`
- `System.MissingMethodException`
- `System.NullReferenceException`
- `System.OutOfMemoryException`
- `System.SecurityException`
- `System.StackOverflowException`

The `System.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 might corrupt memory. Any attempt to execute non-conforming CIL or non-conforming file formats can result in 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 III) 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 `System.ExecutionEngineException`.

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

12.4.2.2 Deriving from exception classes

Certain types of exceptions thrown by the CLI can be derived from 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 derived class of that type.

[*Note:* For instance, the specification of the ckfinite instruction requires that an exception of type `System.ArithmeticException` or a derived class of `ArithmeticException` be thrown by the CLI. A conforming implementation might simply throw an exception of type `ArithmeticException`, but it might also choose to provide more information to the programmer by throwing an exception of type `NotFiniteNumberException` with the offending number. *end note*]

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 might 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 might 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.

 ©ISO/IEC 2006 – All rights reserved

12.4.2.4 Timing and choice of exceptions

Certain types of exceptions thrown by CIL instructions might 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. Relaxation of the timing of exceptions is provided so that an implementation can 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 can be detected early (e.g., at JIT time), but the condition can be signaled later (e.g., at the execution time of the offending instruction) by throwing an exception.

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

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

In addition, as to when class initialization (see Partition II) occurs is not fully specified. In particular, there is no guarantee when `System.TypeInitializationException` might be thrown.

If more than one exception's conditions are met by a method invocation, as to which exception is thrown is unspecified.

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 shall 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. *end note*]

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 can 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 **filter 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 an address and a length: the **trystart** is the address of the first instruction to be protected, and the **trylength** is the length of the protected region. (The **tryend**, the address immediately following the last instruction to be protected, can be trivially computed from these two). A *handler region* is described by an address and a length: the **handlerstart** is the address of the first instruction of the handler and the **handlerlength** is the length of the handler region. (The **handlerend**, the address immediately following the last instruction of the handler, can be trivially computed from these two.)

Every method can have associated with it a set of **exception entries**, called the **exception set**. Each **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 block**
- Required: **handler region**. There are four kinds of handler regions: catch handlers, filtered handlers, finally handlers, and fault handlers. (A filtered handler is the code that runs if the filter evaluates to true.)

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

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

Each region must begin and end on an instruction boundary.

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 shall be within a single region of the other entry, with the further restriction that the enclosing region shall not be a filter. [*Note*: Functions called from within a filter can contain exception handling. *end note*]
- They are **disjoint**: all six regions of the two entries are pairwise-disjoint (no addresses overlap).
- They **mutually protect**: the protected blocks are the same and the other regions are pairwise-disjoint. In this case, all handlers shall be either catch handlers or filtered handlers. The precedence of the handler regions is determined by their ordering in the Exception Handler Table (Partition II).

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

An *exception-handling block* is either a protected region, a filter, a catch handler, a filter handler, a fault handler, or a finally handler.

 ©ISO/IEC 2006 – All rights reserved

12.4.2.8 Control flow restrictions on protected blocks

12.4.2.8.1 Fall Through

An instruction I1 is capable of *fall through* if one of the following is true:

- I1 is not a control-flow instruction (i.e., the only way control flow could be altered by I1 would be if it threw an exception).
- I1 is a switch or conditional branch. [*Note*: Fall through would be the not-taken case. *end note*]
- I1 is a method call instruction.

[*Note*: For the purposes of this section, the ability of an instruction to fall through can be determined purely by the type of the instruction. *end note*]

[*Note*: Most instructions can allow control to fall through after their execution—only unconditional branches, ret, jmp, leave(.s), endfinally, endfault, endfilter, throw, and rethrow do not. Call instructions do allow control to fall through, since the next instruction to be executed in the current method is the one lexically following the call instruction, which executes after the call returns. *end note*]

[*Note*: The determination of validity with respect to fall through can be done lexically; no control-flow or data-flow analysis is required. *end note*]

Entry to filters or handlers can only be accomplished through the CLI exception system; that is, it is not valid for control to fall through into such blocks. This means filters and handlers cannot appear at the beginning of a method, or immediately following any instruction that can cause control flow to fall through.

[*Note*: Conditional branches can have multiple effects on control flow. Since one of the possible effects is to allow control flow to fall through, a filter or handler cannot appear immediately following a conditional branch. *end note*]

Entry to protected blocks can be accomplished by fall-through, at which time the evaluation stack shall be empty.

Exit from protected blocks, filters, or handlers cannot be accomplished via fall through.

12.4.2.8.2 Control-flow Instructions

Instructions that affect control flow have restrictions on how they are used in protected blocks, filters, and handlers. The particular rules depend on the type of instruction. This subclause describes restrictions based on the following:

- The **source** of the instruction; i.e., the address of the start of the instruction.
- The **target(s)** of the instruction; i.e., the address(es) of all instructions within the same method that might be executed following it, excluding fall through (which has been addressed above). If an instruction has a target rule, the exact definition of the target precedes that rule.

For the source and each target of an instruction, consider each protected block, filter, or handler that encloses that address. If all rules are satisfied for all enclosing protected blocks, filters, or handlers, for the source of an instruction and all targets, then the instruction is valid with respect to exception-handling. (Obviously, the instruction shall still follow all other validity rules.) An instruction is considered to be within a block even if the source of the instruction is at the very start of that block.

12.4.2.8.2.1 throw (and all CIL instructions not listed below)

source

1. There are no source restrictions.

target

1. There are no target restrictions.

12.4.2.8.2.2 rethrow:

source

1. Shall be enclosed in a catch handler

[*Note*: The catch handler need not be the innermost enclosing exception-handling block. For example, the rethrow may be within a finally that is within a catch. In such a case, the exception to be rethrown is the one caught by the innermost enclosing catch handler. *end note*]

target

1. There are no target restrictions.

12.4.2.8.2.3 ret:

source

1. Shall not be enclosed in any protected block, filter, or handler.

[*Note*: To return from a protected block, filtered handler, or catch handler, a leave(.s) instruction is needed to transfer control to an address outside all exception-handling blocks, then a ret instruction is needed at that address. *end note*]

[*Note*: Since the tail. prefix on an instruction requires that that instruction be followed by ret, tail calls are not allowed from within protected blocks, filters, or handlers. *end note*]

target

1. There are no target restrictions.

12.4.2.8.2.4 jmp:

source

1. Shall not be enclosed in any protected block, filter, or handler

target

1. There are no target restrictions.

12.4.2.8.2.5 endfilter:

source

1. Shall appear as the lexically last instruction in the filter.

[*Note*: The endfilter is required even if no control-flow path reaches it. This can happen if, for example, the filter does a throw. *end note*]

[*Note*: The lexical nesting rules prohibit nesting other exception-handling entries inside a filter. Thus the innermost exception-handling block enclosing an endfilter instruction shall be a filter. *end note*]

target

1. There are no target restrictions.

12.4.2.8.2.6 endfinally/endfault:

source

1. The innermost enclosing protected block, filter, or handler shall be a finally or fault handler

[*Note*: endfinally and endfault are aliases for the same CIL opcode. Conventionally, CIL assemblers require that endfinally be used within a finally handler, and endfault be used within a fault handler, but the instruction emitted is exactly the same by either name. *end note*]

[*Note*: A finally or fault handler can contain more than one endfinally/endfault. The lexically last instruction in the finally or fault handler need not be endfinally/endfault. In fact, a finally or fault handler might not

 ©ISO/IEC 2006 – All rights reserved

require an endfinally/endfault at all if all control-flow paths terminate through other means. This can happen if, for example, the finally or fault handler throws. *end note*]

target

1. There are no target restrictions.

12.4.2.8.2.7 Branches (br, br.s, conditional branches, switch):

source

1. If the source of the branch is within a protected block, filter, or handler, the target(s) shall be within the same protected block, filter, or handler

target

The target of br, br.s, and the conditional branches, is the address specified. The targets of switch are all of the addresses specified in the jump table.

1. If any target of the branch is within a protected block, except the first instruction of that protected block, the source shall be within the same protected block.
2. If any target of the branch is within a filter or handler, the source shall be within the same filter or handler.

[*Note*: Code can branch to the first instruction of a protected block, but not into the middle of one. *end note*]

[*Note*: Since the conditional branches and switch have a fall-through case, they shall also obey the rules for fall through. *end note*]

12.4.2.8.2.8 leave and leave.s:

source

1. If the source is within a filter, fault handler, or finally handler, the target shall be within the same filter, fault handler, or finally handler.

   [*Note*: This means control cannot be transferred out of a filter, fault handler, or finally handler via the leave(.s) instruction. *end note*]
2. If the source is within a protected block, the target shall be within the same protected block, within an enclosing protected block, the first instruction of a disjoint protected block, or not within any protected block.
3. If the source is within a catch handler or filtered handler, the target shall be within the same catch handler or filtered handler, within the associated protected block, within a protected block that encloses the catch handler or filtered handler, the first instruction of a disjoint protected block, or not within any protected block.

[*Note*: If the source is outside any exception-handling block, that fact implies no additional restrictions on the target. In effect, a leave from outside of exception handling acts like a branch, with the side-effect of emptying the evaluation stack. *end note*]

target

The target of leave(.s) is the address specified by leave(.s).

1. If the target is within a filter or handler, the source shall be within the same filter or handler.
2. If the target is within a protected block, except the first instruction of that protected block, the source shall be within the same protected block, or within the associated catch handler or filtered handler.

[*Note*: To be clear, if the target is the first instruction of a protected block, the source can be outside of the protected block. *end note*]

[*Note*: This means that it is possible to transfer control from a catch handler or a filtered handler to the associated protected block. *end note*]

12.4.2.8.2.9 Examples

[*Example*: Example 1

```
{
EX1:
        br TryStart2
        .try
        {
TryStart1:
                .try
                {
TryStart2:
                        leave End
                }
                finally
                {
                        endfinally
                }
        }
        finally
        {
                endfinally
        }
End:
        ret
}
```

Consider the `br TryStart2` instruction at `EX1`. It is not contained within any exception-handling block, so the source rules do not apply and are thus satisfied. The target is contained within two protected regions, so the target rules are applied once for each region.

Considering the outermost protected region, branch target rule 1 is satisfied since the target is the first instruction of the outermost protected region. Branch target rule 2 does not apply to protected regions and is thus satisfied.

Considering the innermost protected region, branch target rule 1 is satisfied since the target is the first instruction of the innermost protected region. Branch target rule 2 does not apply to protected regions and is thus satisfied.

Thus, the branch instruction at `EX1` is valid from the exception-handling perspective. *end example*]

[*Example*: Example 2

```
{
        ldc.i4.0
EX2:
        brtrue TryStart2
        .try
        {
TryStart1:
EX3:
                br TryStart2
                .try
                {
TryStart2:
                        leave End
                }
                finally
                {
                        endfinally
                }
        }
```

 ©ISO/IEC 2006 – All rights reserved

```
        finally
        {
                endfinally
        }
End:
        ret
}
```

Consider the `brtrue TryStart2` instruction at `EX2`. It is not contained within any exception-handling block, so the source rules do not apply and are thus satisfied. The target is contained within two protected regions, so the target rules are applied once for each region.

Branch target rule 1 is satisfied for the inner protected block since the target is the first instruction of the block. However, branch target rule 1 is *not* satisfied for the outer protected block since the source is not within the outer protected block and the target is not the first instruction of that block.

Thus the conditional branch instruction at `EX2` is invalid from an exception-handling perspective.

Now consider the `br TryStart2` instruction at `EX3`. It is within one protected block, so the source rules are applied considering that protected block. Branch source rule 1 is satisfied since the target is within that protected block. The target is contained within two protected regions, so the target rules are applied once for each region.

Considering the outer protected block, branch target rule 1 is satisfied since the source is also within the outer protected block. Branch target rule 2 does not apply to protected blocks, and is thus satisfied.

Considering the inner protected block, branch target rule 1 is satisfied since the target is the first instruction of the inner protected block. Branch target rule 2 does not apply to protected blocks, and is thus satisfied.

Thus, the branch instruction at `EX3` is valid from an exception-handling perspective. *end example*]

[*Example*: Example 3

```
{
        .try
        {
                newobj instance void [mscorlib]System.Exception::.ctor()
                throw
AfterThrow:
                leave End
        }

        catch [mscorlib]System.Exception
        {
                .try
                {
                        newobj instance void [mscorlib]System.Exception::.ctor()
                        throw
                }

                catch [mscorlib]System.Exception
                {
EX4:
                        leave AfterThrow
                }
                leave End
        }
End:
        ret
}
```

Consider the `leave` instruction at `EX4`. It is contained within two catch handlers, so the source rules are applied once for each region.

Considering the outer catch handler, leave source rules 1 and 2 do not apply to catch handlers and are thus satisfied. Leave source rule 3 is satisfied since the target is within the associated protected region.

Considering the inner catch handler, leave source rules 1 and 2 do not apply to catch handlers and are thus satisfied. Leave source rule 3 is not satisfied since the target is in the middle of a disjoint protected region.

©ISO/IEC 2006 – All rights reserved

Thus, the `leave` instruction at `EX4` is invalid from an exception-handling perspective. However, for illustration purposes, consider the target rules as well.

The target is within one protected region, so the target rules are applied considering that protected region. Leave target rule 1 does not apply to protected regions, and is thus satisfied. Leave target rule 2 is satisfied because the source is within a catch block associated with the protected region. *end example*]

[*Example*: Example 4

```
  {
          .try
          {
                  .try
                  {
                          newobj instance void [mscorlib]System.Exception::.ctor()
                          throw
                  }

                  catch [mscorlib] System.Exception
                  {
EX5:
                          leave EndOfOuterTry
                  }
EndOfOuterTry:
                  // …
                  leave End
          }

          catch [mscorlib]System.Exception
          {
                  leave End
          }
End:
          ret
}
```

Consider the `leave` instruction at `EX5`. It is contained within a protected region and within a catch handler, so the source rules are applied once for each.

Considering the protected region, leave source rules 1 and 3 do not apply to protected regions and are thus satisfied. Leave source rule 2 is satisfied because the target is within the same protected region.

Considering the catch handler, leave source rules 1 and 2 do not apply to catch handlers and are thus satisfied. Leave source rule 3 is satisfied because the target is within a protected block that encloses the catch handler.

The target is within one protected region, so the target rules are applied considering that protected region. Target rule 1 does not apply to protected regions and is thus satisfied. Target rule 2 is satisfied because the source is within the same protected block.

Thus the `leave` instruction at `EX5` is valid from an exception-handling perspective. *end example*]

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.

 ©ISO/IEC 2006 – All rights reserved

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 data types 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 the architecture). When generated externally, these should also be aligned to their natural size, although portable code can 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 can immediately precede an ldind, stind, initblk, or cpblk instruction. This prefix indicates that the data can have arbitrary alignment; the JIT compiler 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 unaligned. prefix has been specified, executing the instruction can generate unaligned memory faults or incorrect data.

12.6.3 Byte ordering

For data types larger than 1 byte, the byte ordering is dependent on the target CPU. Code that depends on byte ordering might not run on all platforms. The PE file format (see §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 only volatile operations (including volatile reads) constitute visible side-effects. (Note that while only volatile operations constitute visible side-effects, volatile operations also affect the visibility of non-volatile references.) Volatile operations are specified in §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. *end rationale*]

[*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). *end note*]

Optimizers are granted additional latitude for relaxed exceptions in methods. A method is *E-relaxed* for a kind of exception if the innermost custom attribute `System.Runtime.CompilerServices.CompilationRelaxationsAttribute` pertaining to exceptions of kind *E* is present and specifies to relax exceptions of kind *E*. (Here, "innermost" means inspecting the method, its class, and its assembly, in that order.)

A *E-relaxed sequence* is a sequence of instructions executed by a thread, where

- Each instruction causing visible side effects or exceptions is in an *E*-relaxed method.
- The sequence does not cross the boundary of a non-trivial protected or handler region. A region is trivial if it can be optimized away under the rules for non-relaxed methods.

Below, an *E-check* is defined as a test performed by a CIL instruction that upon failure causes an exception of kind *E* to be thrown. Furthermore, the type and range tests performed by the methods that set or get an array element's value, or that get an array element's address are considered checks here.

A conforming implementation of the CLI is free to change the timing of relaxed *E*-checks in an *E*-relaxed sequence, with respect to other checks and instructions as long as the observable behavior of the program is changed only in the case that a relaxed *E*-check fails. If an *E*-check fails in an *E*-relaxed sequence:

- The rest of the associated instruction must be suppressed, in order to preserve verifiability. If the instruction was expected to push a value on the VES stack, no subsequent instruction that uses that value should visibly execute.
- It is unspecified whether or not any or all of the side effects in the *E*-relaxed sequence are made visible by the VES.
- The check's exception is thrown some time in the sequence, unless the sequence throws another exception. When multiple relaxed checks fail, it is unspecified as to which exception is thrown by the VES.

[*Note*: Relaxed checks preserve verifiability, but not necessarily security. Because a relaxed check's exception might be deferred and subsequent code allowed to execute, programmers should never rely on implicit checks to preserve security, but instead use explicit checks and throws when security is an issue. *end note*]

[*Rationale*: Different programmers have different goals. For some, trading away precise exception behavior is unacceptable. For others, optimization is more important. The programmer must specify their preference. Different kinds of exceptions may be relaxed or not relaxed separately because different programmers have different notions of which kinds of exceptions must be timed precisely. *end rationale*]

[*Note*: For background and implementation information for relaxed exception handling , plus examples, see Annex F of Partition VI. *end note*]

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 prefix "`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 can 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

 ©ISO/IEC 2006 – All rights reserved

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 §12.6.7. In addition, the class library provides methods to perform explicit volatile reads (`System.Thread.VolatileRead`) and writes (`System.Thread.VolatileWrite`), as well as barrier synchronization (`System.Thread.MemoryBarrier`).
4. **Built-in atomic reads and writes.** All reads and writes of certain properly aligned data types are guaranteed to occur atomically. See §12.6.6.
5. **Explicit atomic operations.** The class library provides a variety of atomic operations in the `System.Threading.Interlocked` class. These operations (e.g., Increment, Decrement, Exchange, and CompareExchange) perform implicit acquire/release operations.

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 §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 §12.6.2) when all the write accesses to a location are the same size. 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/modify/write on hardware that does not support direct writes to small data items. *end note*]

[*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 might perform atomic operations when the data is aligned on an 8-byte boundary. *end note*]

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 §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 shall 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 might change the behavior of the program. *end rationale*]

[*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 §12.6 and the cross-thread semantics of volatile operations. *end note*]

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 invalid 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 Execution Engine 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 can ensure this requirement themselves by inserting appropriate calls to the memory barrier or volatile write instructions.

 ©ISO/IEC 2006 – All rights reserved

13 Index

& **65**
accessibility 26
  assembly 27
  compiler-controlled 27
  family....27
  family-and-assembly 27
  family-or-assembly....27
  private....27
  public 27
alignment **93**
application **50**
application domain 50, 92
ArgIterator....77
argument array....**76**
ArithmeticException....83
array....23, 34
  jagged 34
  storage layout....**34**
  zero-dimensional 35
Array....34
array element 23
assembly **6**, 25, 47, **49**
assembly dependency **49**
assignment compatibility....22, 25, 29, **32**
atomicity....**95**
attribute....**6**, 57
Attribute....57
AttributeUsageAttribute 50, 57
BadImageFormatException 84
behavior 18
  implementation-specific **6**
  undefined **6**
  unspecified....**6**
bool....**19**
Boolean....19, 50
boxing....**6**, 20, 73
byref....67
Byte 19, 50
byte ordering....**93**
cast....22
  explicit 55
  implicit....55
  narrowing....55
  widening 55
char **19**
Char 19, 50
character
  ANSI....**6**
  combining 52
CIL....**6**, 10, 65
class contract **28**
class definition....**28**, 37
class layout **48**
  autolayout 49
  explicitlayout 49
  layoutsequential 49
class, abstract....18
CLI....**6**, 9
CLS....**6**, 9, 12
CLS compliance **14**
  identifying....**15**
  rules for....**15**
CLS consumer **13**
CLS extender **13**
CLS framework **12**
CLSCompliantAttribute....15
code
  managed....**7**, 10
  unmanaged....**8**
coercion **22**
COFF module ....*See* PE module
Common Intermediate Language....*See* CIL

Common Language Infrastructure ..... *See* CLI
Common Language Specification ..... *See* CLS
Common Type System ..... *See* CTS
CompilationRelaxationsAttribute 94
component 47
  self-describing 47
component metadata **47**
conformance **2**
constraint **29**
  byref 31
  location ..... *See* location constraint
    init-only 30
    literal 30
  vararg ..... *See* vararg
  volatile 30
constructor **39**
consumer **12**
contract 16, **28**, 48
  class ..... *See* class contract
  event ..... *See* event contract
  interface ..... *See* interface contract
  method ..... *See* method contract
  property ..... *See* property contract
contravariance 16
conversion
  explicit 22
  implicit 22
covariance 16
CTS **6**, 9
data
  aggregate **71**
  managed 7, 10, 11
  unmanaged **8**
delegate **6**, **35**
Delegate 35, 46
DivideByZeroException 83
Double 19, 50
  Equals 22
EntryPointNotFoundException 84
enum **25**
Enum 20, 25
equality 21
evaluation stack **76**
event **6**, 39
  naming pattern for **56**
event contract **29**
event definition **29**, **46**
examples 5
exception 56
  relaxed **94**
  timing of **85**
Exception 56, 82
exception handling 82
execution engine **6**
execution model 10
ExecutionEngineException 83
export
  member **27**
  nested type **27**
  type **26**
extender **12**
extensions 2
F **65**
field **6**, **23**, 38
  instance 23
  serializable 45
  static 23, 45
field definition 44
finalizer 40
FlagsAttribute 25
float32 ..... **19**, **65**
float64 ..... **19**, **65**
framework 9, **12**
garbage collection **6**, 11
GC 40
  ReRegisterForFinalize 40

 ©ISO/IEC 2006 – All rights reserved

SuppressFinalize....40
generic argument....**6**
generic parameter....**6**
generics....**7**
global static....51
handle
methodInfo....**76**
return state....**76**
handler
catch....**82**
fault....**82**
filter....**83**
finally....**82**
handler region....**86**
hiding....42, 43, 44
by name....43
by name and signature....43
home....71
identical....21
identifier....52
case folding of....52
case-insensitive....52
case-sensitive....52
identity....21
casting and....22
coercion and....22
indexed property....*See* property, indexed
IndexOutOfRangeException....83
infinity....68
inheritance....28, **41**
instance....**18**
instruction pointer....**75**
int
native....**19**, **65**
native unsigned....**19**, **65**
int16....**19**, **65**
unsigned....**19**, **65**
Int16....19, 50
int32....**19**, **65**
unsigned....**19**, **65**
Int32....19, 50
int64....**19**, **65**
unsigned....**19**, **65**
Int64....19, 50
int8....**19**, **65**
unsigned....**19**, **65**
interface contract....**28**
interface definition....**28**, **36**
Intermediate Language....*See* CIL
IntPtr....19
InvalidAddressException....83
InvalidCastException....83
JIT....66
kind....**24**
layout....43
library....**7**
literal....25
local signature....31
local variable array....**76**
localsinit flag....72, 74, 76, 79
location....22
location signature....**30**
lock....**94**
manifest....**7**, **49**
MarshalByRefObject....93
marshalling....48
member....**7**, **22**, **38**
member signatures....48
message....18
metadata....**7**, 9, 47
component....*See* component metadata
metadata extensibility....**50**
metadata token....**47**
method....**7**, **19**, **23**, 39
abstract....44
add....46

final ... 24, 42, 44
generic ... **7**
getter ... 45
instance ... **23**
non-generic ... **7**
remove ... 46
setter ... 45
static ... **23**, 44
virtual ... **23**, 44
method contract ... **29**
method definition ... **29**, 31, **44**
method signature ... **31**
method state ... **75**
MissingFieldException ... 83, 84, 85
MissingMemberException ... 84
MissingMethodException ... 83, 84, 85
modifier
optional ... 51
required ... 51
module ... **7**, **9**
Monitor ... 94
name ... 24
overloading of ... **52**
qualified ... 25
scope of ... *See* scope
special ... 53
uniqueness of ... 24
NaN ... 68
narrowing ... **22**, 55
notes ... 5
NotFiniteNumberException ... 84
NotSupportedException ... 84
null ... **23**
NullReferenceException ... 83
O **65**
object ... **7**, **16**, **19**, 20
Object ... 20, 41
Equals ... 21
MemberwiseClone ... 40
ReferenceEquals ... 21
OOP ... *See* Programming, Object-Oriented
op_Addition ... 54
op_AdditionAssignment ... 55
op_AddressOf ... 54
op_Assign ... 54
op_BitwiseAnd ... 54
op_BitwiseAndAssignment ... 55
op_BitwiseOr ... 54
op_BitwiseOrAssignment ... 55
op_Comma ... 55
op_Decrement ... 54, 58, 59
op_Division ... 54
op_DivisionAssignment ... 55
op_Equality ... 54
op_ExclusiveOr ... 54
op_ExclusiveOrAssignment ... 55
op_Explicit ... 52, 55
op_False ... 54
op_GreaterThan ... 55
op_GreaterThanOrEqual ... 55
op_Implicit ... 52, 55
op_Increment ... 54, 58, 59
op_Inequality ... 55
op_LeftShift ... 54
op_LeftShiftAssignment ... 55
op_LessThan ... 55
op_LessThanOrEqual ... 55
op_LogicalAnd ... 54
op_LogicalNot ... 54, 58, 59
op_LogicalOr ... 54
op_MemberSelection ... 55
op_Modulus ... 54
op_ModulusAssignment ... 55
op_MultiplicationAssignment ... 55
op_Multiply ... 54
op_OnesComplement ... 54

 ©ISO/IEC 2006 – All rights reserved

op_PointerDereference 54
op_PointerToMemberSelection 55
op_RightShift 54
op_RightShiftAssignment 55
op_SignedRightShift 54
op_Subtraction 54
op_SubtractionAssignment 55
op_True 54
op_UnaryNegation 54, 58, 59
op_UnaryPlus 54, 58, 59
op_UnsignedRightShift 54
op_UnsignedRightShiftAssignment 55
operator
addition .....*See* op_Addition
address-of .....*See* op_AddressOf
assignment, compound, addition .....*See* op_AdditionAssignment
assignment, compound, bitwise AND .....*See* op_BitwiseAndAssignment
assignment, compound, bitwise OR .....*See* op_BitwiseOrAssignment
assignment, compound, division .....*See* op_DivisionAssignment
assignment, compound, exclusive OR .....*See* op_ExclusiveOrAssignment
assignment, compound, left-shift .....*See* op_LeftShiftAssignment
assignment, compound, multiplication .....*See* op_MultiplicationAssignment
assignment, compound, remainder .....*See* op_ModulusAssignment
assignment, compound, right-shift .....*See* op_RightShiftAssignment
assignment, compound, right-shift, unsigned .....*See* op_UnsignedRightShiftAssignment
assignment, compound, subtraction .....*See* op_SubtractionAssignment
assignment, simple .....*See* op_Assign
bitwise AND .....*See* op_BitwiseAnd
bitwise OR .....*See* op_BitwiseOr
comma .....*See* op_Comma
conversion, explicit .....*See* op_Explicit
conversion, implicit .....*See* op_Implicit
decrement .....*See* op_Decrement
division .....*See* op_Division
equality .....*See* op_Equality
exclusive OR .....*See* op_ExclusiveOr
false .....*See* op_False
greater-than .....*See* op_GreaterThan
greater-than-or-equal ..*See* op_GreaterThanOrEqual
increment .....*See* op_Increment
inequality .....*See* op_Inequality
left-shift .....*See* op_LeftShift
less-than .....*See* op_LessThan
less-than-or-equal .....*See* op_LessThanOrEqual
logical AND .....*See* op_LogicalAnd
logical NOT .....*See* op_LogicalNot
logical OR .....*See* op_LogicalOr
member selection .....*See* op_MemberSelection
multiplication .....*See* op_Multiply
negation .....*See* op_UnaryNegation
ones-complement .....*See* op_OnesComplement
pointer dereference .....*See* op_PointerDereference
pointer-to-member, compound, ...*See* op_PointerToMemberSelection
remainder .....*See* op_Modulus
right-shift .....*See* op_RightShift
right-shift, signed .....*See* op_SignedRightShift
right-shift, unsigned .....*See* op_UnsignedRightShift
subtraction .....*See* op_Subtraction
true .....*See* op_True
unary plus .....*See* op_UnaryPlus
operator overloading 53
optimization 10, 93
OutOfMemoryException 83
OverflowException 83
overriding **23**, 43
parameter passing 80
parameter signature **31**

PE module 47
pointer
  function 35
  managed 31
pool
  local memory **76**
prefix
  constrained. 20
  tail. 78, 79
  unaligned. 93
  volatile. 30, 95
profile **7**
programming
  functional 16
  object-oriented 16
  procedural 16
  typeless 18
property **7**, 39
  indexed 56
  naming pattern for **56**
property contract **29**
property definition **29**, 45
protected region **86**
proxy 92
publicly accessible parts 14
rank **34**
reference 50
referenced entity 26
referent 26
Relative Virtual Address ..... *See* RVA
remoting boundary 92
representation 18
rounding mode 68
Runtime.CompilerServices 38
RVA 48
SByte 19
scope 24, 38
  assembly 25
  member 25
sealed 42
security 28
Security 28
security demand
  inheritance 28
  reference 28
security descriptor **76**
SecurityException 83, 85
serialization 45
signature **7**, 28, **29**
  local ..... *See* local signature
  location ..... *See* location signature
  method ..... *See* method signature
  parameter ..... *See* parameter signature
  type ..... *See* type signature
Single 19, 50
  Equals 22
slot
  expect existing 43
  new 43
SpecialName 45, 46
StackOverflowException 83
string **19**
  ANSI **6**
String 19, 50
System.ArgIterator ..... *See* ArgIterator
System.Array ..... *See* Array
System.Attribute ..... *See* Attribute
System.AttributeUsageAttribute ..... *See* AttributeUsageAttribute
System.Boolean ..... *See* Boolean
System.Byte ..... *See* Byte
System.Char ..... *See* Boolean
System.CLSCompliantAttribute ..... *See* CLSCompliantAttribute
System.Delegate ..... *See* Delegate
System.Double ..... *See* Double

 ©ISO/IEC 2006 – All rights reserved

System.Enum....41, *See* Enum
System.Exception....*See* Exception
System.FlagsAttribute....*See* FlagsAttribute
System.GC....*See* GC
System.Int16....*See* Int16
System.Int32....*See* Int32
System.Int64....*See* Int64
System.IntPtr....*See* IntPtr
System.MarshalByRefObject....*See* MarshalByRefObject
System.Object....*See* Object
System.SByte....*See* SByte
System.Security....*See* Security
System.Single....*See* Single
System.String....*See* String
System.Threading....*See* Threading
System.Type....*See* Type
System.TypedReference....*See* TypedReference
System.TypeInitializationException....*See* TypeInitializationException
System.UInt16....*See* UInt16
System.UInt32....*See* UInt32
System.UInt64....*See* UInt64
System.UIntPtr....*See* UIntPtr
System.ValueType....*See* ValueType
this....23, 44
thread....**94**
Thread....94
ThreadAbortException....93
try block....*See* protected region
type....18
& *See* &
abstract....37, 38
array....*See* array
boxed....20
built-in....**19**
class....28, **37**
compound....23
concrete....38
enclosing....26, 28
enumeration....*See* enum
exact....**18**, 23
exact array....**34**
explicit....**33**
F *See* F
float32....*See* float32
float64....*See* float64
generic....**7**
implicit....33
int, native....*See* int, native
int, native unsigned....*See* int, native unsigned
int16....*See* int16
int16, unsigned....*See* int16, unsigned
int32....*See* int32
int32, unsigned....*See* int32, unsigned
int64....*See* int64
int64, unsigned....*See* int64, unsigned
int8....*See* int8
int8, unsigned....*See* int8, unsigned
interface....**18**, 19, 28
nested....**26**, 28
O *See* O
object....**18**, **20**, 38
pointer....**18**, 35
reference....**7**, 18
sealed....41
value....**7**, 18
Type....50
type definer....33
type definition....25
type member....*See* member
type name....25
exportable....**26**
nested....**26**
non-exportable....**26**
type safety....**9**, 33

type signature **30**
typed reference 31
typedref ..... **19**, See typed reference
TypedReference 19, 31
TypeInitializationException 85
TypeLoadException 83, 84, 85
TypeUnloadedException 84
UInt16 19
UInt32 19
UInt64 19
UIntPtr 19
unboxing **7**, 20, 73
Unicode standard 4
validation
  metadata 47
value **8**, **16**, **18**
  boxed 20
  coersion of a ..... *See* coersion
  null ..... *See* null
  partial description 18
ValueType 20, 41
vararg 31
vector 35
verification **8**, 10, 33
versioning 10
VES **8**, 9, **65**
virtual call 23
Virtual Execution System ..... *See* VES
visibility **26**, 38
  assembly 38
  public 38
widening **22**, 55

 ©ISO/IEC 2006 – All rights reserved

Common Language Infrastructure (CLI)

Partition II:
Metadata Definition and Semantics

Table of contents

1	**Introduction**	**1**
2	**Overview**	**2**
3	**Validation and verification**	**3**
4	**Introductory examples**	**4**
4.1	"Hello world!"	4
4.2	Other examples	4
5	**General syntax**	**5**
5.1	General syntax notation	5
5.2	Basic syntax categories	5
5.3	Identifiers	6
5.4	Labels and lists of labels	7
5.5	Lists of hex bytes	7
5.6	Floating-point numbers	7
5.7	Source line information	8
5.8	File names	8
5.9	Attributes and metadata	8
5.10	*ilasm* source files	9
6	**Assemblies, manifests and modules**	**10**
6.1	Overview of modules, assemblies, and files	10
6.2	Defining an assembly	11
6.2.1	Information about the assembly (*AsmDecl*)	12
6.2.2	Manifest resources	14
6.2.3	Associating files with an assembly	14
6.3	Referencing assemblies	14
6.4	Declaring modules	15
6.5	Referencing modules	16
6.6	Declarations inside a module or assembly	16
6.7	Exported type definitions	16
7	**Types and signatures**	**18**
7.1	Types	18

©ISO/IEC 2006 – All rights reserved

7.1.1	modreq and modopt	19
7.1.2	pinned	20
7.2	Built-in types	20
7.3	References to user-defined types (*TypeReference*)	20
7.4	Native data types	21
8	**Visibility, accessibility and hiding**	**23**
8.1	Visibility of top-level types and accessibility of nested types	23
8.2	Accessibility	23
8.3	Hiding	23
9	**Generics**	**24**
9.1	Generic type definitions	25
9.2	Generics and recursive inheritance graphs	25
9.3	Generic method definitions	26
9.4	Instantiating generic types	27
9.5	Generics variance	28
9.6	Assignment compatibility of instantiated types	28
9.7	Validity of member signatures	29
9.8	Signatures and binding	30
9.9	Inheritance and overriding	31
9.10	Explicit method overrides	32
9.11	Constraints on generic parameters	33
9.12	References to members of generic types	34
10	**Defining types**	**35**
10.1	Type header (*ClassHeader*)	35
10.1.1	Visibility and accessibility attributes	36
10.1.2	Type layout attributes	37
10.1.3	Type semantics attributes	37
10.1.4	Inheritance attributes	38
10.1.5	Interoperation attributes	38
10.1.6	Special handling attributes	38
10.1.7	Generic parameters (*GenPars*)	39
10.2	Body of a type definition	43
10.3	Introducing and overriding virtual methods	44
10.3.1	Introducing a virtual method	44
10.3.2	The .override directive	44
10.3.3	Accessibility and overriding	45

 ©ISO/IEC 2006 – All rights reserved

10.4	Method implementation requirements	46
10.5	Special members	46
10.5.1	Instance constructor	46
10.5.2	Instance finalizer	47
10.5.3	Type initializer	47
10.6	Nested types	49
10.7	Controlling instance layout	49
10.8	Global fields and methods	50
11	**Semantics of classes**	**52**
12	**Semantics of interfaces**	**53**
12.1	Implementing interfaces	53
12.2	Implementing virtual methods on interfaces	53
13	**Semantics of value types**	**55**
13.1	Referencing value types	56
13.2	Initializing value types	56
13.3	Methods of value types	57
14	**Semantics of special types**	**59**
14.1	Vectors	59
14.2	Arrays	59
14.3	Enums	61
14.4	Pointer types	62
14.4.1	Unmanaged pointers	63
14.4.2	Managed pointers	64
14.5	Method pointers	64
14.6	Delegates	65
14.6.1	Delegate signature compatibility	66
14.6.2	Synchronous calls to delegates	67
14.6.3	Asynchronous calls to delegates	68
15	**Defining, referencing, and calling methods**	**70**
15.1	Method descriptors	70
15.1.1	Method declarations	70
15.1.2	Method definitions	70
15.1.3	Method references	70
15.1.4	Method implementations	70
15.2	Static, instance, and virtual methods	70

15.3	Calling convention	71
15.4	Defining methods	72
15.4.1	Method body	73
15.4.2	Predefined attributes on methods	76
15.4.3	Implementation attributes of methods	78
15.4.4	Scope blocks	79
15.4.5	vararg methods	79
15.5	Unmanaged methods	80
15.5.1	Method transition thunks	80
15.5.2	Platform invoke	81
15.5.3	Method calls via function pointers	82
15.5.4	Data type marshaling	82
16	**Defining and referencing fields**	**84**
16.1	Attributes of fields	84
16.1.1	Accessibility information	85
16.1.2	Field contract attributes	85
16.1.3	Interoperation attributes	85
16.1.4	Other attributes	86
16.2	Field init metadata	86
16.3	Embedding data in a PE file	87
16.3.1	Data declaration	87
16.3.2	Accessing data from the PE file	88
16.4	Initialization of non-literal static data	88
16.4.1	Data known at link time	88
16.5	Data known at load time	89
16.5.1	Data known at run time	89
17	**Defining properties**	**90**
18	**Defining events**	**92**
19	**Exception handling**	**95**
19.1	Protected blocks	95
19.2	Handler blocks	96
19.3	Catch blocks	96
19.4	Filter blocks	96
19.5	Finally blocks	97
19.6	Fault handlers	97

 ©ISO/IEC 2006 – All rights reserved

20	**Declarative security**	**98**
21	**Custom attributes**	**99**
21.1	CLS conventions: custom attribute usage	99
21.2	Attributes used by the CLI	99
21.2.1	Pseudo custom attributes	100
21.2.2	Custom attributes defined by the CLS	101
21.2.3	Custom attributes for security	101
21.2.4	Custom attributes for TLS	102
21.2.5	Custom attributes, various	102
22	**Metadata logical format: tables**	**103**
22.1	Metadata validation rules	104
22.2	Assembly : 0x20	105
22.3	AssemblyOS : 0x22	106
22.4	AssemblyProcessor : 0x21	106
22.5	AssemblyRef : 0x23	106
22.6	AssemblyRefOS : 0x25	107
22.7	AssemblyRefProcessor : 0x24	107
22.8	ClassLayout : 0x0F	108
22.9	Constant : 0x0B	110
22.10	CustomAttribute : 0x0C	110
22.11	DeclSecurity : 0x0E	112
22.12	EventMap : 0x12	114
22.13	Event : 0x14	114
22.14	ExportedType : 0x27	116
22.15	Field : 0x04	117
22.16	FieldLayout : 0x10	119
22.17	FieldMarshal : 0x0D	120
22.18	FieldRVA : 0x1D	121
22.19	File : 0x26	121
22.20	GenericParam : 0x2A	122
22.21	GenericParamConstraint : 0x2C	123
22.22	ImplMap : 0x1C	124
22.23	InterfaceImpl : 0x09	125
22.24	ManifestResource : 0x28	125
22.25	MemberRef : 0x0A	126
22.26	MethodDef : 0x06	127
22.27	MethodImpl : 0x19	130

©ISO/IEC 2006 – All rights reserved

22.28	MethodSemantics : 0x18	131
22.29	MethodSpec : 0x2B	132
22.30	Module : 0x00	133
22.31	ModuleRef : 0x1A	133
22.32	NestedClass : 0x29	134
22.33	Param : 0x08	134
22.34	Property : 0x17	135
22.35	PropertyMap : 0x15	136
22.36	StandAloneSig : 0x11	137
22.37	TypeDef : 0x02	138
22.38	TypeRef : 0x01	141
22.39	TypeSpec : 0x1B	142
23	**Metadata logical format: other structures**	**143**
23.1	Bitmasks and flags	143
23.1.1	Values for AssemblyHashAlgorithm	143
23.1.2	Values for AssemblyFlags	143
23.1.3	Values for Culture	143
23.1.4	Flags for events [EventAttributes]	144
23.1.5	Flags for fields [FieldAttributes]	144
23.1.6	Flags for files [FileAttributes]	145
23.1.7	Flags for Generic Parameters [GenericParamAttributes]	145
23.1.8	Flags for ImplMap [PInvokeAttributes]	146
23.1.9	Flags for ManifestResource [ManifestResourceAttributes]	146
23.1.10	Flags for methods [MethodAttributes]	146
23.1.11	Flags for methods [MethodImplAttributes]	147
23.1.12	Flags for MethodSemantics [MethodSemanticsAttributes]	148
23.1.13	Flags for params [ParamAttributes]	148
23.1.14	Flags for properties [PropertyAttributes]	148
23.1.15	Flags for types [TypeAttributes]	148
23.1.16	Element types used in signatures	150
23.2	Blobs and signatures	152
23.2.1	MethodDefSig	153
23.2.2	MethodRefSig	154
23.2.3	StandAloneMethodSig	155
23.2.4	FieldSig	157
23.2.5	PropertySig	157
23.2.6	LocalVarSig	157

 ©ISO/IEC 2006 – All rights reserved

23.2.7	CustomMod	158
23.2.8	TypeDefOrRefEncoded	158
23.2.9	Constraint	159
23.2.10	Param	159
23.2.11	RetType	159
23.2.12	Type	160
23.2.13	ArrayShape	160
23.2.14	TypeSpec	161
23.2.15	MethodSpec	161
23.2.16	Short form signatures	162
23.3	Custom attributes	162
23.4	Marshalling descriptors	164
24	**Metadata physical layout**	**167**
24.1	Fixed fields	167
24.2	File headers	167
24.2.1	Metadata root	167
24.2.2	Stream header	168
24.2.3	#Strings heap	168
24.2.4	#US and #Blob heaps	168
24.2.5	#GUID heap	169
24.2.6	#~ stream	169
25	**File format extensions to PE**	**173**
25.1	Structure of the runtime file format	173
25.2	PE headers	173
25.2.1	MS-DOS header	174
25.2.2	PE file header	174
25.2.3	PE optional header	175
25.3	Section headers	177
25.3.1	Import Table and Import Address Table (IAT)	178
25.3.2	Relocations	178
25.3.3	CLI header	179
25.4	Common Intermediate Language physical layout	180
25.4.1	Method header type values	181
25.4.2	Tiny format	181
25.4.3	Fat format	181
25.4.4	Flags for method headers	182
25.4.5	Method data section	182

25.4.6 Exception handling clauses 183

26 Index 184

 ©ISO/IEC 2006 – All rights reserved

1 Introduction

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

©ISO/IEC 2006 – All rights reserved

2 Overview

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

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

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

 ©ISO/IEC 2006 – All rights reserved

3 Validation and verification

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

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

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

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

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

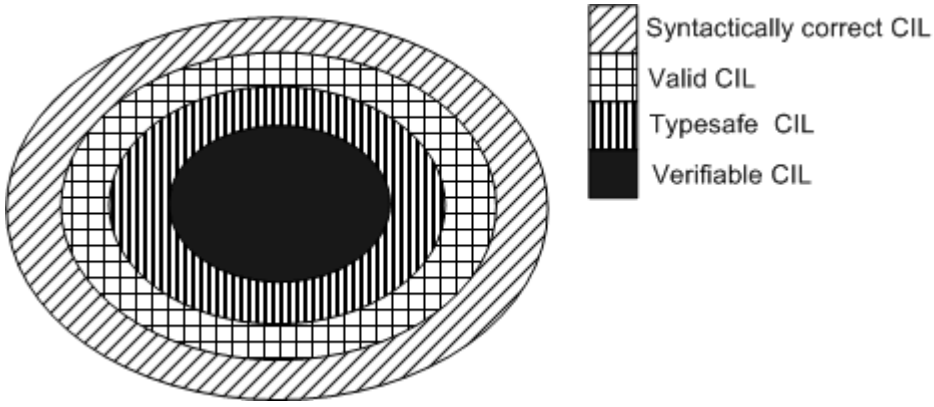


Figure 1: Relationship between correct and verifiable CIL

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

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

©ISO/IEC 2006 – All rights reserved

4 Introductory examples

This clause and its subclauses contain only informative text.

4.1 "Hello world!"

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

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

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

end example]

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

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

4.2 Other examples

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

End informative text

 ©ISO/IEC 2006 – All rights reserved

5 General syntax

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

5.1 General syntax notation

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

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

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

[*Example:* A grammar such as

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

would consider all of the following to be valid:

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

but all of the following to be invalid:

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

end example]

5.2 Basic syntax categories

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

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

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

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

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

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

©ISO/IEC 2006 – All rights reserved

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

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

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

and

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

end example]

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

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

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

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

5.3 Identifiers

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

Id ::=
ID
| *SQSTRING*

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

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

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

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

[*Example:* The following are simple identifiers:

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

The following are identifiers in single quotes:

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

 ©ISO/IEC 2006 – All rights reserved

The following are dotted names:

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

end example]

5.4 Labels and lists of labels

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

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

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

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

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

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

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

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

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

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

end example]

5.5 Lists of hex bytes

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

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

5.6 Floating-point numbers

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

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

Float32 ::=

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

[*Example:*

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

end example]

5.7 Source line information

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

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

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

5.8 File names

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

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

5.9 Attributes and metadata

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

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

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

 ©ISO/IEC 2006 – All rights reserved

5.10 *ilasm* source files

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

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

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

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

©ISO/IEC 2006 – All rights reserved

6 Assemblies, manifests and modules

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

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

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

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

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

6.1 Overview of modules, assemblies, and files

This subclause contains informative text only.

Consider the following figure:

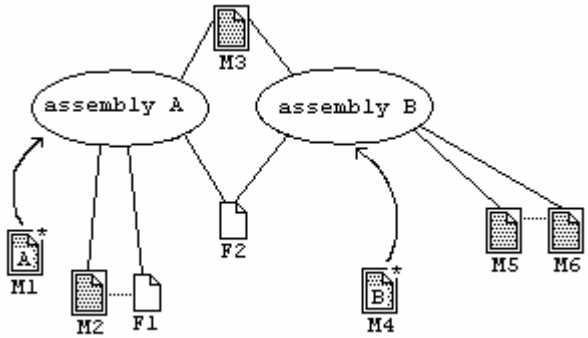


Figure 2: References to Modules and Files

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

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

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

 ©ISO/IEC 2006 – All rights reserved

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

End informative text

6.2 Defining an assembly

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

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

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

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

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

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

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

[*Example:*

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

end example]

6.2.1 Information about the assembly (*AsmDecl*)

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

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

6.2.1.1 Hash algorithm

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

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

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

6.2.1.2 Culture

AsmDecl ::= .culture *QSTRING* | …

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

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

6.2.1.3 Originator's public key

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

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

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

 ©ISO/IEC 2006 – All rights reserved

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

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

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

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

6.2.1.4 Version numbers

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

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

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

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

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

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

end note]

6.2.2 Manifest resources

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

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

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

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

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

6.2.3 Associating files with an assembly

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

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

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

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

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

6.3 Referencing assemblies

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

An assembly mediates all accesses to other assemblies from the files that it contains. This is done through the metadata by requiring that the manifest for the executing assembly contain a declaration for any assembly

 ©ISO/IEC 2006 – All rights reserved

referenced by the executing code. A top-level `.assembly extern` declaration is used for this purpose. The optional **as** clause provides an alias, which allows ILAsm to address external assemblies that have the same name, but differing in version, culture, etc.

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

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

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

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

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

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

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

[*Example:*

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

end example]

6.4 Declaring modules

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

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

[*Example:*

```
.module CountDown.exe
```

end example]

6.5 Referencing modules

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

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

[*Example:*

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

end example]

6.6 Declarations inside a module or assembly

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

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

6.7 Exported type definitions

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

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

 ©ISO/IEC 2006 – All rights reserved

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

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

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

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

7 Types and signatures

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

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

7.1 Types

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

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

 ©ISO/IEC 2006 – All rights reserved

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

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

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

7.1.1 modreq and modopt

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

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

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

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

©ISO/IEC 2006 – All rights reserved

7.1.2 pinned

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

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

7.2 Built-in types

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

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

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

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

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

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

AssemblyRefName ::=	**Clause**
DottedName	5.1

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

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

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

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

 ©ISO/IEC 2006 – All rights reserved

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

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

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

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

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

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

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

end example]

7.4 Native data types

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

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

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

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

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

[*Example:*

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

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

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

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

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

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

 ©ISO/IEC 2006 – All rights reserved

8 Visibility, accessibility and hiding

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

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

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

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

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

8.2 Accessibility

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

8.3 Hiding

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

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

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

9 Generics

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

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

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

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

[*Note*: Conside the following definition:

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

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

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

 ©ISO/IEC 2006 – All rights reserved

9.1 Generic type definitions

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

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

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

Generic type definitions are subject to the following restrictions:

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

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

- Varargs methods cannot be members of generic types

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

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

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

9.2 Generics and recursive inheritance graphs

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

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

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

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

[*Example*:

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

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

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

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

[*Example*:

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

generates the edges

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

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

[*Example*:

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

generates the edges

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

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

9.3 Generic method definitions

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

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

 ©ISO/IEC 2006 – All rights reserved

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

```
.method … !!0 M`1<T>() { … }
```

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

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

9.4 Instantiating generic types

GenArgs is used to represent a generic argument list:

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

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

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

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

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

end example]

[*Example*:

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

end example]

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

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

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

- Byref types (e.g., `System.Generic.Collection.List`1<string&>` is invalid)
- Value types that contain fields that can point into the CIL evaluation stack (e.g., `List<System.RuntimeArgumentHandle>`)
- void (e.g., `List<System.Void>` is invalid)

Unmanaged pointer types (e.g., `int32*`) can be used as generic arguments to generic types and methods.

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

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

©ISO/IEC 2006 – All rights reserved

9.5 Generics variance

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

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

This block contains only informative text

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

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

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

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

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

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

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

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

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

end example]

End informative text

9.6 Assignment compatibility of instantiated types

- Assignment compatibility is defined in Partition I.8.7.

[*Example*:

Assuming `Employee` := `Manager`,

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

 ©ISO/IEC 2006 – All rights reserved

end example]

[*Example*: Given the following:

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

`IConverter<string, object>` ≔ `IConverter<object, string>`

Given the following:

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

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

[*Example*:

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

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

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

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

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

end example]

9.7 Validity of member signatures

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

This block contains only informative text

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

End informative text

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

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

- Every instance method and virtual method declaration is valid with respect to S

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

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

Think of

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

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

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

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

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

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

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

9.8 Signatures and binding

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

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

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

[*Example*:

 ©ISO/IEC 2006 – All rights reserved

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

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

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

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

end example]

9.9 Inheritance and overriding

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

end example]

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

[*Example:*

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

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

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

```
void V(string)
```

However, the following version of D is valid:

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

end example]

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

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

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

9.10 Explicit method overrides

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

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

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

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

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

  end example]

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

 ©ISO/IEC 2006 – All rights reserved

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

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

neither of the following .override statements is allowed

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

end example]

9.11 Constraints on generic parameters

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

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

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

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

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

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

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

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

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

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

end note]

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

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

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

end example]

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

This block contains only informative text

9.12 References to members of generic types

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

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

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

End informative text

 ©ISO/IEC 2006 – All rights reserved

10 Defining types

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

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

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

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

10.1 Type header (*ClassHeader*)

A type header consists of

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

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

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

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

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

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

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

end example]

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

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

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

10.1.1 Visibility and accessibility attributes

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

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

 ©ISO/IEC 2006 – All rights reserved

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

10.1.2 Type layout attributes

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

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

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

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

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

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

10.1.3 Type semantics attributes

ClassAttr `::= …`
`| interface`

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

[*Example:*

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

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

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

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

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

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

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

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

end example]

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

10.1.4 Inheritance attributes

ClassAttr ::= …
| `abstract`
| `sealed`

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

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

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

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

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

10.1.5 Interoperation attributes

ClassAttr ::= …
| `ansi`
| `autochar`
| `unicode`

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

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

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

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

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

10.1.6 Special handling attributes

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

These attributes can be combined in any way.

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

 ©ISO/IEC 2006 – All rights reserved

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

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

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

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

10.1.7 Generic parameters (*GenPars*)

Generic parameters are included when defining a generic type.

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

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

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

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

+ denotes a covariant generic parameter (§9.5).

- denotes a contravariant generic parameter (§9.5).

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

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

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

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

[*Example*:

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

This declares a generic class `C<T>`, which has a covariant generic parameter named `T`. `T` is constrained such that

it must implement `System.IComparable<T>`, and must be a concrete class with a public default constructor. *end example*]

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

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

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

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

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

end example]

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

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

 ©ISO/IEC 2006 – All rights reserved

	×	✓	*	
	×	×	×	Any enum type **and** `System.Enum`
	✓	×	×	System.Enum **only**
	✓	×	✓	**Not meaningful**: Cannot be instantiated (no instantiable reference type can be derived from `System.Enum`)
	✓	✓	*	**Invalid**
System.INullableValue	×	×	×	Any `System.Nullable<T>` **or** other type implementing interface
	×	×	✓	Any `System.Nullable<T>` **or** other type implementing interface with default constructor
	✓	×	×	Any reference type implementing `System.INullableValue` (note: this **excludes** `System.Nullable<T>`)
	✓	×	✓	Any reference type implementing `System.INullableValue` with a default constructor (note: this **excludes** `System.Nullable<T>`)
	×	✓	*	Any valuetype implementing `System.INullableValue` (note: this **includes** `System.Nullable<T>`)
	✓	✓	*	**Invalid**
System.Exception (an example of any non-special reference Type)	×	×	×	`System.Exception`, or any class derived from `System.Exception`
	×	×	✓	Any `System.Exception` with a public default constructor
	✓	×	×	`System.Exception`, or any class derived from `System.Exception`. This is exactly the same result as if the class constraint was not specified
	✓	×	✓	Any Exception with a public default constructor.
	×	✓	*	**Not meaningful**: Cannot be instantiated (a value type cannot be derived from a reference type)
	✓	✓	*	**Invalid**
System.Delegate	×	×	×	`System.Delegate`, or any class derived from `System.Delegate`
	×	×	✓	**Not meaningful**: Cannot be instantiated (there is no default constructor)

	✓	×	×	`System.Delegate`, or any class derived from `System.Delegate`
	✓	×	✓	Any Delegate with a public .ctor. Invalid for known delegates (`System.Delegate`)
	×	✓	*	**Not meaningful**: Cannot be instantiated (a value type cannot be derived from a reference type)
	✓	✓	*	**Invalid**
System.Array	×	×	×	Any array
	*	×	✓	**Not meaningful**: Cannot be instantiated (no default constructor)
	✓	×	×	Any array
	×	✓	*	**Not meaningful**: Cannot be instantiated (a value type cannot be derived from a reference type)
	✓	✓	*	**Invalid**

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

```
.class public auto ansi beforefieldinit Bar`1<valuetype T>
       Valid   ldtoken       class Bar`1<int32>
       Invalid ldtoken       class Bar`1<class [mscorlib]System.Exception>
       Invalid ldtoken       class Bar`1<Nullable`1<int32>>
       Invalid ldtoken       class Bar`1<class [mscorlib]System.ValueType>
.class public auto ansi beforefieldinit 'Bar`1'<class T>
       Invalid ldtoken       class Bar`1<int32>
       Valid   ldtoken       class Bar`1<class [mscorlib]System.Exception>
       Invalid ldtoken       class Bar`1<valuetype [mscorlib]System.Nullable`1<int32>>
       Valid   ldtoken       class Bar`1<class [mscorlib]System.ValueType>
.class public auto ansi beforefieldinit Bar`1<(class
       [mscorlib]System.ValueType) T>
       Valid   ldtoken       class Bar`1<int32>
       Invalid ldtoken       class Bar`1<class [mscorlib]System.Exception>
       Valid   ldtoken       class Bar`1<valuetype [mscorlib]System.Nullable`1<int32>>
       Valid   ldtoken       class Bar`1<class [mscorlib]System.ValueType>
.class public auto ansi beforefieldinit Bar`1<class (int32)> T>
       Invalid ldtoken       class Bar`1<int32>
       Invalid ldtoken       class Bar`1<class [mscorlib]System.Exception>
       Invalid ldtoken       class Bar`1<valuetype [mscorlib]System.Nullable`1<int32>>
       Invalid ldtoken       class Bar`1<class [mscorlib]System.ValueType>
```

 ©ISO/IEC 2006 – All rights reserved

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

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

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

end example]

10.2 Body of a type definition

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

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

©ISO/IEC 2006 – All rights reserved

ClassMember ::=	**Description**	**Clause**
| *SecurityDecl*	Declarative security permissions.	20

10.3 Introducing and overriding virtual methods

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

10.3.1 Introducing a virtual method

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

- If the definition is marked `newslot`, the definition always creates a new virtual method, even if a base class provides a matching virtual method. A reference to the virtual method via the class containing the method definition, or via a class derived from that class, refers to the new definition (unless hidden by a `newslot` definition in a derived class). Any reference to the virtual method not via the class containing the method definition, nor via its derived classes, refers to the original definition.
- If the definition is not marked `newslot`, the definition creates a new virtual method only if there is not virtual method of the same name and signature inherited from a base class.

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

10.3.2 The .override directive

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

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

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

Int32 is the number of generic parameters.

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

 ©ISO/IEC 2006 – All rights reserved

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

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

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

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

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

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

end example]

10.3.3 Accessibility and overriding

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

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

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

Table 7.1: Valid Widening of Access to a Virtual Method

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

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

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

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

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

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

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

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

10.4 Method implementation requirements

A type (concrete or abstract) can provide

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

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

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

10.5 Special members

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

10.5.1 Instance constructor

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

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

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

 ©ISO/IEC 2006 – All rights reserved

end example]

10.5.2 Instance finalizer

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

10.5.3 Type initializer

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

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

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

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

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

end example]

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

10.5.3.1 Type initialization guarantees

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

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

10.5.3.2 Relaxed guarantees

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

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

this guarantee is required therefore provides efficiency when it is desired at the cost of consistency guarantees. *end rationale*]

10.5.3.3 Races and deadlocks

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

 ©ISO/IEC 2006 – All rights reserved

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

2.4 Execute the type initialization code for this type.

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

end rationale]

10.6 Nested types

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

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

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

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

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

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

end example]

10.7 Controlling instance layout

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

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

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

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

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

The `.pack` directive specifies that fields should be placed within the runtime object at byte addresses which are a multiple of the specified number, or at natural alignment for that field type, whichever is *smaller*. For example, `.pack 2` would allow 32-bit-wide fields to be started on even addresses, whereas without any

`.pack` directive, they would be naturally aligned; that is, placed on addresses that are a multiple of 4. The integer following `.pack` shall be one of the following: 0, 1, 2, 4, 8, 16, 32, 64, or 128. (A value of zero indicates that the pack size used should match the default for the current platform.) The `.pack` directive shall not be supplied for any type with explicit layout control.

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

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

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

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

The following class uses explicit layout of its fields:

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

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

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

The following class specifies a contiguous block of 16 bytes:

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

end example]

10.8 Global fields and methods

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

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

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

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

 ©ISO/IEC 2006 – All rights reserved

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

11 Semantics of classes

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

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

 ©ISO/IEC 2006 – All rights reserved

12 Semantics of interfaces

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

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

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

12.1 Implementing interfaces

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

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

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

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

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

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

end example]

12.2 Implementing virtual methods on interfaces

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

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

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

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

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

 ©ISO/IEC 2006 – All rights reserved

13 Semantics of value types

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

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

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

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

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

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

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

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

[*Example*:

```
class C<U> { }

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

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

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

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

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

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

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

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

13.1 Referencing value types

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

ValueTypeReference `::=`

`boxed` *TypeReference*

| `valuetype` *TypeReference*

13.2 Initializing value types

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

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

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

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

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

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

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

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

 ©ISO/IEC 2006 – All rights reserved

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

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

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

end example]

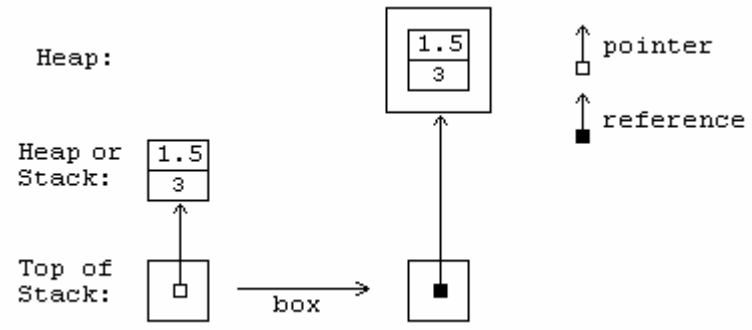
13.3 Methods of value types

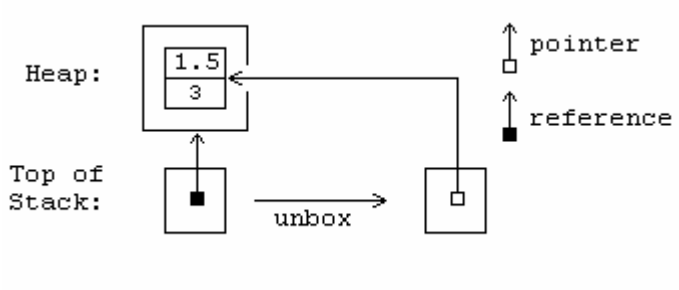
Value types can have static, instance and virtual methods. Static methods of value types are defined and called the same way as static methods of class types. As with classes, both instance and virtual methods of a boxed or unboxed value type can be called using the call instruction. The callvirt instruction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

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

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

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





end note]

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

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

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

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

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

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

Then the call is made as shown below:

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

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

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

end example]

 ©ISO/IEC 2006 – All rights reserved

14 Semantics of special types

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

14.1 Vectors

Type ::= …
| *Type* '[' ']'

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

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

[*Example:* Declare a vector of Strings:

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

Declare a vector of function pointers:

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

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

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

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

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

end example]

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

14.2 Arrays

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

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

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

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

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

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

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

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

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

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

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

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

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

 ©ISO/IEC 2006 – All rights reserved

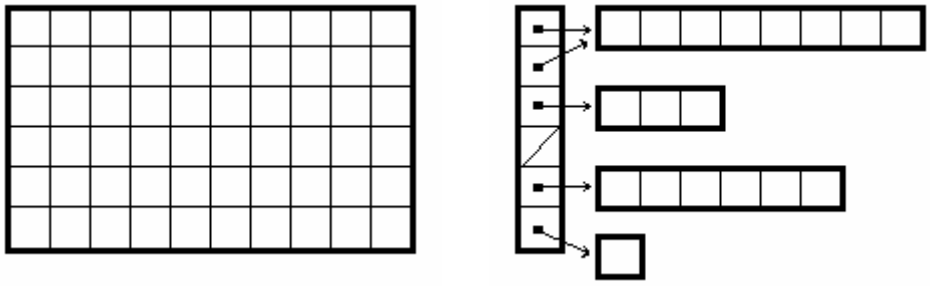
```
    ldloc myArray
    ldc.i4.5
    ldc.i4.4
    call instance string & string[,]::Address(int32, int32)
    ldstr "Test"
    stind.ref

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

end example]

The following text is informative

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



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

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

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

End of informative text

14.3 Enums

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

[*Note:* Unlike Pascal, the CLI does *not* provide a guarantee that values of the enum type are integers corresponding to one of the symbols. In fact, the CLS (see Partition I, CLS) defines a convention for using enums to represent bit flags which can be combined to form integral value that are not named by the enum type itself. *end note*]

Enums obey additional restrictions beyond those on other value types. Enums shall contain only fields as members (they shall not even define type initializers or instance constructors); they shall not implement any

interfaces; they shall have auto field layout (§10.1.2); they shall have exactly one instance field and it shall be of the underlying type of the enum; all other fields shall be static and literal (§16.1); and they shall not be initialized with the initobj instruction.

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

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

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

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

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

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

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

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

end example]

14.4 Pointer types

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

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

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

- *Loading* the value from the location referenced by the pointer.

 ©ISO/IEC 2006 – All rights reserved

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

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

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

The following is informative text

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

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

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

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

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

14.4.1 Unmanaged pointers

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

- Unmanaged pointers are not reported to the garbage collector and can be used in any way that an integer can be used.
- Verifiable code cannot dereference unmanaged pointers.
- Unverified code can pass an unmanaged pointer to a method that expects a managed pointer. This is safe only if one of the following is true:
    a. The unmanaged pointer refers to memory that is not in memory used by the CLI for storing instances of objects (“garbage-collected memory” or “managed memory”).
    b. The unmanaged pointer contains the address of a field within an object.
    c. The unmanaged pointer contains the address of an element within an array.

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

14.4.2 Managed pointers

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

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

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

End informative text

14.5 Method pointers

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

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

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

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

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

 ©ISO/IEC 2006 – All rights reserved

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

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

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

NegateIt:
    ldftn int32 Negate(int32)
    ret
  }
}

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

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

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

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

end example]

14.6 Delegates

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

Delegates shall be declared sealed, and the only members a delegate shall have are either the first two or all four methods as specified here. These methods shall be declared `runtime` and `managed` (§15.4.3). They shall not have a body, since that body shall be created automatically by the VES. Other methods available on delegates are inherited from the class `System.Delegate` in the Base Class Library (see Partition IV). The delegate methods are:

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

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

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

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

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

end example]

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

14.6.1 Delegate signature compatibility

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

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

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

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

 ©ISO/IEC 2006 – All rights reserved

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

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

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

14.6.2 Synchronous calls to delegates

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

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

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

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

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

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

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

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

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

end example]

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

14.6.3 Asynchronous calls to delegates

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

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

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

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

14.6.3.1 The BeginInvoke method

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

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

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

 ©ISO/IEC 2006 – All rights reserved

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

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

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

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

14.6.3.2 The EndInvoke method

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

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

15 Defining, referencing, and calling methods

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

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

as well as inside a type:

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

15.1 Method descriptors

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

15.1.1 Method declarations

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

15.1.2 Method definitions

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

15.1.3 Method references

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

15.1.4 Method implementations

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

15.2 Static, instance, and virtual methods

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

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

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

 ©ISO/IEC 2006 – All rights reserved

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

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

4. The *this* pointer can be null.

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

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

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

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

15.3 Calling convention

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

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

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

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

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

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

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

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

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

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

15.4 Defining methods

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

The method head (see also §10) consists of

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

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

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

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

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

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

 ©ISO/IEC 2006 – All rights reserved

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

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

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

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

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

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

15.4.1 Method body

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

MethodBodyItem ::=	**Description**	**Clause**
.custom *CustomDecl*	Definition of custom attributes.	21
| .data *DataDecl*	Emits data to the data section	16.3
| .emitbyte *Int32*	Emits an unsigned byte to the code section of the method.	15.4.1.1
| .entrypoint	Specifies that this method is the entry point to the application (only one such method is allowed).	15.4.1.2
| .locals [init] '(' *LocalsSignature* ')'	Defines a set of local variables for this method.	15.4.1.3
| .maxstack *Int32*	The int32 specifies the maximum number of elements on the evaluation stack during the execution of the method.	15.4.1
| .override *TypeSpec* '::' *MethodName*	Use current method as the implementation for the method specified.	10.3.2
| .override method *CallConv Type TypeSpec* '::' *MethodName GenArity* '(' *Parameters* ')'	Use current method as the implementation for the method specified.	10.3.2
| .param '[' *Int32* ']' ['=' *FieldInit*]	Store a constant *FieldInit* value for parameter *Int32*	15.4.1.4
| .param type '[' *Int32* ']'	Specifies a type parameter for a generic method	15.4.1.5

MethodBodyItem ::=	**Description**	**Clause**
| *ExternSourceDecl*	`.line` or `#line`	5.7
| *Instr*	An instruction	Partition VI
| *Id* `':'`	A label	5.4
| *ScopeBlock*	Lexical scope of local variables	15.4.4
| *SecurityDecl*	`.permission` or `.permissionset`	20
| *SEHBlock*	An exception block	19

15.4.1.1 The .emitbyte directive

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

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

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

15.4.1.2 The .entrypoint directive

MethodBodyItem ::= …
| `.entrypoint`

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

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

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

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

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

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

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

end example]

 ©ISO/IEC 2006 – All rights reserved

15.4.1.3 The .locals directive

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

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

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

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

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

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

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

end example]

15.4.1.4 The .param directive

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

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

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

15.4.1.5 The .param type directive

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

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

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

[*Example:*

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

end example]

15.4.2 Predefined attributes on methods

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

The following combinations of predefined attributes are invalid:

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

15.4.2.1 Accessibility information

MethAttr ::= …
| assembly
| compilercontrolled
| famandassem
| family

 ©ISO/IEC 2006 – All rights reserved

| `famorassem`
| `private`
| `public`

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

15.4.2.2 Method contract attributes

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

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

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

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

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

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

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

15.4.2.3 Overriding behavior

MethAttr `::=` …
| `newslot`

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

15.4.2.4 Method attributes

MethAttr `::=` …
| `abstract`

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

15.4.2.5 Interoperation attributes

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

See §15.5.2and §22.20.

15.4.2.6 Special handling attributes

MethAttr ::= …
| rtspecialname
| specialname

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

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

15.4.3 Implementation attributes of methods

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

15.4.3.1 Code implementation attributes

ImplAttr ::= …
| cil
| native
| runtime

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

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

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

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

15.4.3.2 Managed or unmanaged

ImplAttr ::= …

 ©ISO/IEC 2006 – All rights reserved

| `managed`
| `unmanaged`

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

15.4.3.3 Implementation information

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

These attributes can be combined.

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

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

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

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

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

15.4.4 Scope blocks

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

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

15.4.5 vararg methods

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

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

©ISO/IEC 2006 – All rights reserved

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

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

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

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

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

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

end example]

15.5 Unmanaged methods

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

This Standard specifies:

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

15.5.1 Method transition thunks

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

In order to call managed code from unmanaged code, some platforms require a specific transition sequence to be performed. In addition, some platforms require that the representation of data types be converted (data

 ©ISO/IEC 2006 – All rights reserved

marshaling). Both of these problems are solved by the `.vtfixup` directive. This directive can appear several times, but only at the top level of a CIL assembly file, as shown by the following grammar:

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

The `.vtfixup` directive declares that at a certain memory location there is a table that contains metadata tokens referring to methods that shall be converted into method pointers. The CLI will do this conversion automatically when the file containing the `.vtfixup` directive is loaded into memory for execution. The declaration specifies the number of entries in the table, the kind of method pointer that is required, the width of an entry in the table, and the location of the table:

VTFixupDecl ::=
[*Int32*] *VTFixupAttr** `at` *DataLabel*

VTFixupAttr ::=
`fromunmanaged`
| `int32`
| `int64`

The attributes `int32` and `int64` are mutually exclusive, with `int32` being the default. These attributes specify the width of each slot in the table. Each slot contains a 32-bit metadata token (zero-padded if the table has 64-bit slots), and the CLI converts it into a method pointer of the same width as the slot.

If `fromunmanaged` is specified, the CLI will generate a thunk that will convert the unmanaged method call to a managed call, call the method, and return the result to the unmanaged environment. The thunk will also perform data marshalling in the platform-specific manner described for *platform invoke*.

The ILAsm syntax does not specify a mechanism for creating the table of tokens, but a compiler can simply emit the tokens as byte literals into a block specified using the `.data` directive.

15.5.2 Platform invoke

Methods defined in native code can be invoked using the *platform invoke* (also know as PInvoke or p/invoke) functionality of the CLI. Platform invoke will switch from managed to unmanaged state and back, and also handle necessary data marshalling. Methods that need to be called using PInvoke are marked as `pinvokeimpl`. In addition, the methods shall have the implementation attributes `native` and `unmanaged` (§15.4.2.4).

MethAttr ::=	**Description**	**Clause**
`pinvokeimpl` `'('` *QSTRING* [`as` *QSTRING*] *PinvAttr** `')'`	Implemented in native code	
| …		15.4.1.5

The first quoted string is a platform-specific description indicating where the implementation of the method is located (for example, on Microsoft Windows™ this would be the name of the DLL that implements the method). The second (optional) string is the name of the method as it exists on that platform, since the platform can use name-mangling rules that force the name as it appears to a managed program to differ from the name as seen in the native implementation (this is common, for example, when the native code is generated by a C++ compiler).

Only static methods, defined at global scope (i.e., outside of any type), can be marked `pinvokeimpl`. A method declared with `pinvokeimpl` shall not have a body specified as part of the definition.

©ISO/IEC 2006 – All rights reserved

PinvAttr ::=	**Description (platform-specific, suggestion only)**
`ansi`	ANSI character set.
| `autochar`	Determine character set automatically.
| `cdecl`	Standard C style call
| `fastcall`	C style fastcall.
| `stdcall`	Standard C++ style call.
| `thiscall`	The method accepts an implicit *this* pointer.
| `unicode`	Unicode character set.
| `platformapi`	Use call convention appropriate to target platform.

The attributes `ansi`, `autochar`, and `unicode` are mutually exclusive. They govern how strings will be marshaled for calls to this method: `ansi` indicates that the native code will receive (and possibly return) a platform-specific representation that corresponds to a string encoded in the ANSI character set (typically this would match the representation of a C or C++ string constant); `autochar` indicates a platform-specific representation that is "natural" for the underlying platform; and `unicode` indicates a platform-specific representation that corresponds to a string encoded for use with Unicode methods on that platform.

The attributes `cdecl`, `fastcall`, `stdcall`, `thiscall`, and `platformapi` are mutually exclusive. They are platform-specific and specify the calling conventions for native code.

[*Example:* The following shows the declaration of the method `MessageBeep` located in the Microsoft Windows™ DLL user32.dll:

```
.method public static pinvokeimpl("user32.dll" stdcall) int8
       MessageBeep(unsigned int32) native unmanaged {}
```

end example]

15.5.3 Method calls via function pointers

Unmanaged methods can also be called via function pointers. There is no difference between calling managed or unmanaged methods with pointers. However, the unmanaged method needs to be declared with `pinvokeimpl` as described in §15.5.2. Calling managed methods with function pointers is described in §14.5.

15.5.4 Data type marshaling

While data type marshaling is necessarily platform-specific, this Standard specifies a minimum set of data types that shall be supported by all conforming implementations of the CLI. Additional data types can be supported in a platform-specific manner, using custom attributes and/or custom modifiers to specify any special handling required on the particular implementation.

The following data types shall be marshaled by all conforming implementations of the CLI; the native data type to which they conform is implementation-specific:

- All integer data types (`int8`, `int16`, `unsigned int8`, `bool`, `char`, etc.) including the `native` integer types.
- Enumerations, as their underlying data type.
- All floating-point data types (`float32` and `float64`), if they are supported by the CLI implementation for managed code.
- The type `string`.
- Unmanaged pointers to any of the above types.

 ©ISO/IEC 2006 – All rights reserved

In addition, the following types shall be supported for marshaling from managed code to unmanaged code, but need not be supported in the reverse direction (i.e., as return types when calling unmanaged methods or as parameters when calling from unmanaged methods into managed methods):

- One-dimensional zero-based arrays of any of the above
- Delegates (the mechanism for calling from unmanaged code into a delegate is platform-specific; it should not be assumed that marshaling a delegate will produce a function pointer that can be used directly from unmanaged code).

Finally, the type `System.Runtime.InteropServices.GCHandle` can be used to marshal an object to unmanaged code. The unmanaged code receives a platform-specific data type that can be used as an "opaque handle" to a specific object. See Partition IV.

16 Defining and referencing fields

Fields are typed memory locations that store the data of a program. The CLI allows the declaration of both instance and static fields. While static fields are associated with a type, and are shared across all instances of that type, instance fields are associated with a particular instance of that type. Once instantiated, an instance has its own copy of each instance field.

The CLI also supports global fields, which are fields declared outside of any type definition. Global fields shall be static.

A field is defined by the `.field` directive: (§22.15)

Field ::= `.field` *FieldDecl*

FieldDecl ::=
[`'[' Int32 ']'`] *FieldAttr** *Type* *Id* [`'='` *FieldInit* | `at` *DataLabel*]

The *FieldDecl* has the following parts:

- An optional integer specifying the byte offset of the field within an instance (§10.7). If present, the type containing this field shall have the `explicit` layout attribute. An offset shall not be supplied for global or static fields.
- Any number of field attributes (§16.2).
- Type.
- Name.
- Optionally, either a *FieldInit* clause (§16.2) or a *DataLabel* (§5.4) clause.

Global fields shall have a data label associated with them. This specifies where, in the PE file, the data for that field is located. Static fields of a type can, but need not, be assigned a data label.

[*Example:*

```
.field private class [.module Counter.dll]Counter counter
.field public static initonly int32 pointCount
.field private int32 xOrigin
.field public static int32 count at D_0001B040
```

end example]

16.1 Attributes of fields

Attributes of a field specify information about accessibility, contract information, interoperation attributes, as well as information on special handling.

The following subclauses contain additional information on each group of predefined attributes of a field.

FieldAttr ::=	**Description**	**Clause**
`assembly`	Assembly accessibility.	16.1.1
| `famandassem`	Family and Assembly accessibility.	16.1.1
| `family`	Family accessibility.	16.1.1
| `famorassem`	Family or Assembly accessibility.	16.1.1
| `initonly`	Marks a constant field.	16.1.2
| `literal`	Specifies metadata field. No memory is allocated at runtime for this field.	16.1.2

 ©ISO/IEC 2006 – All rights reserved

FieldAttr ::=	**Description**	**Clause**
| `marshal` '(' *NativeType* ')'	Marshaling information.	16.1.3
| `notserialized`	Reserved (indicates this field is not to be serialized).	16.1.2
| `private`	Private accessibility.	16.1.1
| `compilercontrolled`	Compiler controlled accessibility.	16.1.1
| `public`	Public accessibility.	16.1.1
| `rtspecialname`	Special treatment by runtime.	16.1.4
| `specialname`	Special name for other tools.	16.1.4
| `static`	Static field.	16.1.2

16.1.1 Accessibility information

The accessibility attributes are `assembly`, `famandassem`, `family`, `famorassem`, `private`, `compilercontrolled`, and `public`. These attributes are mutually exclusive.

Accessibility attributes are described in §8.2.

16.1.2 Field contract attributes

Field contract attributes are `initonly`, `literal`, `static` and `notserialized`. These attributes can be combined; however, only `static` fields shall be `literal`. The default is an instance field that can be serialized.

`static` specifies that the field is associated with the type itself rather than with an instance of the type. Static fields can be accessed without having an instance of a type, e.g., by static methods. As a consequence, within an application domain, a static field is shared between all instances of a type, and any modification of this field will affect all instances. If `static` is not specified, an instance field is created.

`initonly` marks fields which are constant after they are initialized. These fields shall only be mutated inside a constructor. If the field is a static field, then it shall be mutated only inside the type initializer of the type in which it was declared. If it is an instance field, then it shall be mutated only in one of the instance constructors of the type in which it was defined. It shall not be mutated in any other method or in any other constructor, including constructors of derived classes.

[*Note:* The use of ldflda or ldsflda on an `initonly` field makes code unverifiable. In unverifiable code, the VES need not check whether `initonly` fields are mutated outside the constructors. The VES need not report any errors if a method changes the value of a constant. However, such code is not valid. *end note*]

`literal` specifies that this field represents a constant value; such fields shall be assigned a value. In contrast to `initonly` fields, `literal` fields do not exist at runtime. There is no memory allocated for them. `literal` fields become part of the metadata, but cannot be accessed by the code. `literal` fields are assigned a value by using the *FieldInit* syntax (§16.2).

[*Note:* It is the responsibility of tools generating CIL to replace source code references to the literal with its actual value. Hence changing the value of a literal requires recompilation of any code that references the literal. Literal values are, thus, not version-resilient. *end note*]

16.1.3 Interoperation attributes

There is one attribute for interoperation with pre-existing native applications; it is platform-specific and shall not be used in code intended to run on multiple implementations of the CLI. The attribute is `marshal` and specifies that the field's contents should be converted to and from a specified native data type when passed to unmanaged code. Every conforming implementation of the CLI will have default marshaling rules as well as restrictions on what automatic conversions can be specified using the `marshal` attribute. See also §15.5.4.

©ISO/IEC 2006 – All rights reserved

[*Note:* Marshaling of user-defined types is not required of all implementations of the CLI. It is specified in this standard so that implementations which choose to provide it will allow control over its behavior in a consistent manner. While this is not sufficient to guarantee portability of code that uses this feature, it does increase the likelihood that such code will be portable. *end note*]

16.1.4 Other attributes

The attribute `rtspecialname` indicates that the field name shall be treated in a special way by the runtime.

[*Rationale:* There are currently no field names that are required to be marked with `rtspecialname`. It is provided for extensions, future standardization, and to increase consistency between the declaration of fields and methods (instance and type initializer methods shall be marked with this attribute). *end rationale*]

The attribute `specialname` indicates that the field name has special meaning to tools other than the runtime, typically because it marks a name that has meaning for the CLS (see Partition I).

16.2 Field init metadata

The *FieldInit* metadata can optionally be added to a field declaration. The use of this feature shall not be combined with a data label.

The *FieldInit* information is stored in metadata and this information can be queried from metadata. But the CLI does not use this information to automatically initialize the corresponding fields. The field initializer is typically used with `literal` fields (§16.1.2) or parameters with default values. See §22.9.

The following table lists the options for a field initializer. Note that while both the type and the field initializer are stored in metadata there is no requirement that they match. (Any importing compiler is responsible for coercing the stored value to the target field type). The description column in the table below provides additional information.

FieldInit ::=	**Description**
`bool '(' true | false ')'`	Boolean value, encoded as **true** or **false**
`| bytearray '(' Bytes ')'`	String of bytes, stored without conversion. Can be padded with one zero byte to make the total byte-count an even number
`| char '(' Int32 ')'`	16-bit unsigned integer (Unicode character)
`| float32 '(' Float64 ')'`	32-bit floating-point number, with the floating-point number specified in parentheses.
`| float32 '(' Int32 ')'`	*Int32* is binary representation of float
`| float64 '(' Float64 ')'`	64-bit floating-point number, with the floating-point number specified in parentheses.
`| float64 '(' Int64 ')'`	*Int64* is binary representation of double
`| [unsigned] int8 '(' Int32 ')'`	8-bit integer with the value specified in parentheses.
`| [unsigned] int16 '(' Int32 ')'`	16-bit integer with the value specified in parentheses.
`| [unsigned] int32 '(' Int32 ')'`	32-bit integer with the value specified in parentheses.
`| [unsigned] int64 '(' Int64 ')'`	64-bit integer with the value specified in parentheses.
`| QSTRING`	String. *QSTRING* is stored as Unicode
`| nullref`	Null object reference

[*Example:* The following shows a typical use of this:

```
.field public static literal valuetype ErrorCodes no_error = int8(0)
```

 ©ISO/IEC 2006 – All rights reserved

The field named `no_error` is a literal of type `ErrorCodes` (a value type) for which no memory is allocated. Tools and compilers can look up the value and detect that it is intended to be an 8-bit signed integer whose value is 0. *end example*]

16.3 Embedding data in a PE file

There are several ways to declare a data field that is stored in a PE file. In all cases, the `.data` directive is used.

Data can be embedded in a PE file by using the `.data` directive at the top-level.

Decl ::=	**Clause**
`.data` *DataDecl*	
| …	6.6

Data can also be declared as part of a type:

ClassMember ::=	**Clause**
`.data` *DataDecl*	
| …	10.2

Yet another alternative is to declare data inside a method:

MethodBodyItem ::=	**Clause**
`.data` *DataDecl*	
| …	15.4.1

16.3.1 Data declaration

A `.data` directive contains an optional data label and the body which defines the actual data. A data label shall be used if the data is to be accessed by the code.

DataDecl ::= [*DataLabel* `'='`] *DdBody*

The body consists either of one data item or a list of data items in braces. A list of data items is similar to an array.

DdBody ::=
DdItem
| `'{'` *DdItemList* `'}'`

A list of items consists of any number of items:

DdItemList ::= *DdItem* [`','` *DdItemList*]

The list can be used to declare multiple data items associated with one label. The items will be laid out in the order declared. The first data item is accessible directly through the label. To access the other items, pointer arithmetic is used, adding the size of each data item to get to the next one in the list. The use of pointer arithmetic will make the application non-verifiable. (Each data item shall have a *DataLabel* if it is to be referenced afterwards; missing a *DataLabel* is useful in order to insert alignment padding between data items)

A data item declares the type of the data and provides the data in parentheses. If a list of data items contains items of the same type and initial value, the grammar below can be used as a short cut for some of the types: the number of times the item shall be replicated is put in brackets after the declaration.

DdItem ::=	**Description**

`'&' '(' Id ')'`	Address of label
| `bytearray '(' Bytes ')'`	Array of bytes
| `char '*' '(' QSTRING ')'`	Array of (Unicode) characters
| `float32 [ '(' Float64 ')' ][ '[' Int32 ']' ]`	32-bit floating-point number, can be replicated
| `float64 [ '(' Float64 ')' ][ '[' Int32 ']' ]`	64-bit floating-point number, can be replicated
| `int8 [ '(' Int32 ')' ][ '[' Int32 ']' ]`	8-bit integer, can be replicated
| `int16 [ '(' Int32 ')' ][ '[' Int32 ']' ]`	16-bit integer, can be replicated
| `int32 [ '(' Int32 ')' ][ '[' Int32 ']' ]`	32-bit integer, can be replicated
| `int64 [ '(' Int64 ')' ][ '[' Int32 ']' ]`	64-bit integer, can be replicated

[*Example:*

The following declares a 32-bit signed integer with value 123:

```
.data theInt = int32(123)
```

The following declares 10 replications of an 8-bit unsigned integer with value 3:

```
.data theBytes = int8 (3) [10]
```

end example]

16.3.2 Accessing data from the PE file

The data stored in a PE File using the `.data` directive can be accessed through a `static` variable, either global or a member of a type, declared at a particular position of the data:

FieldDecl `::=` *FieldAttr** *Type Id* `at` *DataLabel*

The data is then accessed by a program as it would access any other static variable, using instructions such as `ldsfld`, `ldsflda`, and so on (see Partition III).

The ability to access data from within the PE File can be subject to platform-specific rules, typically related to section access permissions within the PE File format itself.

[*Example:* The following accesses the data declared in the example of §16.3.1. First a static variable needs to be declared for the data, e.g., a global static variable:

```
.field public static int32 myInt at theInt
```

Then the static variable can be used to load the data:

```
ldsfld int32 myInt
// data on stack
```

end example]

16.4 Initialization of non-literal static data

This subclause and its subclauses contain only informative text.

Many languages that support static data provide for a means to initialize that data before the program begins execution. There are three common mechanisms for doing this, and each is supported in the CLI.

16.4.1 Data known at link time

When the correct value to be stored into the static data is known at the time the program is linked (or compiled for those languages with no linker step), the actual value can be stored directly into the PE file, typically into

 ©ISO/IEC 2006 – All rights reserved

the data area (§16.3). References to the variable are made directly to the location where this data has been placed in memory, using the OS-supplied fix-up mechanism to adjust any references to this area if the file loads at an address other than the one assumed by the linker.

In the CLI, this technique can be used directly if the static variable has one of the primitive numeric types or is a value type with explicit type layout and no embedded references to managed objects. In this case the data is laid out in the data area as usual and the static variable is assigned a particular RVA (i.e., offset from the start of the PE file) by using a data label with the field declaration (using the `at` syntax).

This mechanism, however, does not interact well with the CLI notion of an application domain (see Partition I). An application domain is intended to isolate two applications running in the same OS process from one another by guaranteeing that they have no shared data. Since the PE file is shared across the entire process, any data accessed via this mechanism is visible to all application domains in the process, thus violating the application domain isolation boundary.

16.5 Data known at load time

When the correct value is not known until the PE file is loaded (for example, if it contains values computed based on the load addresses of several PE files) it can be possible to supply arbitrary code to run as the PE file is loaded, but this mechanism is platform-specific and might not be available in all conforming implementations of the CLI.

16.5.1 Data known at run time

When the correct value cannot be determined until type layout is computed, the user shall supply code as part of a type initializer to initialize the static data. The guarantees about type initialization are covered in §10.5.3.1. As will be explained below, global statics are modeled in the CLI as though they belonged to a type, so the same guarantees apply to both global and type statics.

Because the layout of managed types need not occur until a type is first referenced, it is not possible to statically initialize managed types by simply laying out the data in the PE file. Instead, there is a type initialization process that proceeds in the following steps:

1. All static variables are zeroed.
2. The user-supplied type initialization procedure, if any, is invoked as described in §10.5.3.

Within a type initialization procedure there are several techniques:

- *Generate explicit code* that stores constants into the appropriate fields of the static variables. For small data structures this can be efficient, but it requires that the initializer be converted to native code, which can prove to be both a code space and an execution time problem.
- *Box value types.* When the static variable is simply a boxed version of a primitive numeric type or a value type with explicit layout, introduce an additional static variable with known RVA that holds the unboxed instance and then simply use the `box` instruction to create the boxed copy.
- *Create a managed array from a static native array of data.* This can be done by marshaling the native array to a managed array. The specific marshaler to be used depends on the native array. e.g., it can be a safearray.
- *Default initialize a managed array of a value type.* The Base Class Library provides a method that zeroes the storage for every element of an array of unboxed value types (`System.Runtime.CompilerServices.InitializeArray`)

End informative text

17 Defining properties

A Property is declared by the using the `.property` directive. Properties shall only be declared inside of types (i.e., global properties are not supported).

ClassMember `::=`
`.property` *PropHeader* `'{'` *PropMember*`*` `'}'`

See §22.34 and §22.35 for how property information is stored in metadata.

PropHeader `::=`
[`specialname`][`rtspecialname`] *CallConv Type Id* `'('` *Parameters* `')'`

The `.property` directive specifies a calling convention (§15.3), type, name, and parameters in parentheses. `specialname` marks the property as *special* to other tools, while `rtspecialname` marks the property as *special* to the CLI. The signature for the property (i.e., the *PropHeader* production) shall match the signature of the property's `.get` method (see below)

[*Rationale:* There are currently no property names that are required to be marked with `rtspecialname`. It is provided for extensions, future standardization, and to increase consistency between the declaration of properties and methods (instance and type initializer methods shall be marked with this attribute). *end rationale*]

While the CLI places no constraints on the methods that make up a property, the CLS (see Partition I) specifies a set of consistency constraints.

A property can contain any number of methods in its body. The following table shows how these methods are identified, and provides short descriptions of each kind of item:

PropMember `::=`	**`Description`**	**`Clause`**
`|` `.custom` *CustomDecl*	Custom attribute.	21
`|` `.get` *CallConv Type* [*TypeSpec* `'::'`] *MethodName* `'('` *Parameters* `')'`	Specifies the getter for the property.	
`|` `.other` *CallConv Type* [*TypeSpec* `'::'`] *MethodName* `'('` *Parameters* `')'`	Specifies a method for the property other than the getter or setter.	
`|` `.set` *CallConv Type* [*TypeSpec* `'::'`] *MethodName* `'('` *Parameters* `')'`	Specifies the setter for the property.	
`|` *ExternSourceDecl*	`.line` or #line	5.7

`.get` specifies the *getter* for this property. The *TypeSpec* defaults to the current type. Only one *getter* can be specified for a property. To be CLS-compliant, the definition of *getter* shall be marked `specialname`.

`.set` specifies the *setter* for this property. The *TypeSpec* defaults to the current type. Only one *setter* can be specified for a property. To be CLS-compliant, the definition of *setter* shall be marked `specialname`.

`.other` is used to specify any other methods that this property comprises.

In addition, custom attributes (§21) or source line declarations can be specified.

[*Example:* This shows the declaration of the property called `count`.

```
.class public auto autochar MyCount extends [mscorlib]System.Object {
  .method virtual hidebysig public specialname instance int32 get_Count() {
  // body of getter
  }
```

 ©ISO/IEC 2006 – All rights reserved

```
    .method virtual hidebysig public specialname instance void set_Count(
        int32 newCount) {
    // body of setter
    }

    .method virtual hidebysig public instance void reset_Count() {
    // body of refresh method
    }

    // the declaration of the property
    .property int32 Count() {
      .get instance int32 MyCount::get_Count()
      .set instance void MyCount::set_Count(int32)
      .other instance void MyCount::reset_Count()
    }
  }
```

end example]

18 Defining events

Events are declared inside types, using the `.event` directive; there are no global events.

ClassMember ::=	**Clause**
`.event` *EventHeader* `'{'` *EventMember** `'}'`	
| …	9

See §22.13 and §22.11

EventHeader ::=
[`specialname`] [`rtspecialname`] [*TypeSpec*] *Id*

In typical usage, the *TypeSpec* (if present) identifies a delegate whose signature matches the arguments passed to the event's fire method.

The event head can contain the keywords `specialname` or `rtspecialname`. `specialname` marks the name of the property for other tools, while `rtspecialname` marks the name of the event as special for the runtime.

[*Rationale:* There are currently no event names that are required to be marked with `rtspecialname`. It is provided for extensions, future standardization, and to increase consistency between the declaration of events and methods (instance and type initializer methods shall be marked with this attribute). *end rationale*]

EventMember ::=	**Description**	**Clause**
`.addon` *CallConv Type* [*TypeSpec* `'::'`] *MethodName* `'('` *Parameters* `')'`	`Add` method for event.	
| `.custom` *CustomDecl*	Custom attribute.	21
| `.fire` *CallConv Type* [*TypeSpec* `'::'`] *MethodName* `'('` *Parameters* `')'`	Fire method for event.	
| `.other` *CallConv Type* [*TypeSpec* `'::'`] *MethodName* `'('` *Parameters* `')'`	Other method.	
| `.removeon` *CallConv Type* [*TypeSpec* `'::'`] *MethodName* `'('` *Parameters* `')'`	Remove method for event.	
| *ExternSourceDecl*	`.line` or `#line`	5.7

The `.addon` directive specifies the *add* method, and the *TypeSpec* defaults to the same type as the event. The CLS specifies naming conventions and consistency constraints for events, and requires that the definition of the *add* method be marked with `specialname`.

The `.removeon` directive specifies the *remove* method, and the *TypeSpec* defaults to the same type as the event. The CLS specifies naming conventions and consistency constraints for events, and requires that the definition of the *remove* method be marked with `specialname`.

The `.fire` directive specifies the *fire* method, and the *TypeSpec* defaults to the same type as the event. The CLS specifies naming conventions and consistency constraints for events, and requires that the definition of the *fire* method be marked with `specialname`.

An event can contain any number of other methods specified with the `.other` directive. From the point of view of the CLI, these methods are only associated with each other through the event. If they have special semantics, this needs to be documented by the implementer.

Events can also have custom attributes (§21) associated with them and they can declare source line information.

 ©ISO/IEC 2006 – All rights reserved

[*Example:* This shows the declaration of an event, its corresponding delegate, and typical implementations of the add, remove, and fire method of the event. The event and the methods are declared in a class called `Counter`.

```
// the delegate
.class private sealed auto autochar TimeUpEventHandler extends
     [mscorlib]System.Delegate {
  .method public hidebysig specialname rtspecialname instance void .ctor(object
      'object', native int 'method') runtime managed {}

  .method public hidebysig virtual instance void Invoke() runtime managed {}

  .method public hidebysig newslot virtual instance class
    [mscorlib]System.IAsyncResult BeginInvoke(class
    mscorlib]System.AsyncCallback callback, object 'object') runtime managed {}

  .method public hidebysig newslot virtual instance void EndInvoke(class
     [mscorlib]System.IAsyncResult result) runtime managed {}
}


// the class that declares the event
.class public auto autochar Counter extends [mscorlib]System.Object {
  // field to store the handlers, initialized to null
  .field private class TimeUpEventHandler timeUpEventHandler
  // the event declaration
  .event TimeUpEventHandler startStopEvent {
    .addon instance void Counter::add_TimeUp(class TimeUpEventHandler 'handler')
    .removeon instance void Counter::remove_TimeUp(class TimeUpEventHandler
'handler')
    .fire instance void Counter::fire_TimeUpEvent()
  }
  // the add method, combines the handler with existing delegates
  .method public hidebysig virtual specialname instance void add_TimeUp(class
      TimeUpEventHandler 'handler') {
    .maxstack 4
    ldarg.0
    dup

    ldfld    class TimeUpEventHandler Counter::TimeUpEventHandler
    ldarg    'handler'
    call     class[mscorlib]System.Delegate
      [mscorlib]System.Delegate::Combine(class [mscorlib]System.Delegate, class
      [mscorlib]System.Delegate)
    castclass TimeUpEventHandler
    stfld    class TimeUpEventHandler Counter::timeUpEventHandler
    ret
  }

  // the remove method, removes the handler from the delegate
  .method virtual public specialname void remove_TimeUp(class TimeUpEventHandler
       'handler') {
    .maxstack 4
    ldarg.0
    dup
    ldfld    class TimeUpEventHandler Counter::timeUpEventHandler

    ldarg    'handler'
    call     class[mscorlib]System.Delegate
       [mscorlib]System.Delegate::Remove(class
       [mscorlib]System.Delegate, class [mscorlib]System.Delegate)
    castclass TimeUpEventHandler
    stfld    class TimeUpEventHandler Counter::timeUpEventHandler
    ret
  }
```

```
    // the fire method
    .method virtual family specialname void fire_TimeUpEvent() {
      .maxstack 3
      ldarg.0
      ldfld    class TimeUpEventHandler Counter::timeUpEventHandler
      callvirt instance void TimeUpEventHandler::Invoke()
      ret
    }
  }      // end of class Counter
```

end example]

 ©ISO/IEC 2006 – All rights reserved

19 Exception handling

In the CLI, a method can define a range of CIL instructions that are said to be *protected*. This is called a *try block*. It can then associate one or more *handlers* with that try block. If an exception occurs during execution anywhere within the try block, an exception object is created that describes the problem. The CLI then takes over, transferring control from the point at which the exception was thrown, to the block of code that is willing to handle that exception. See Partition I.

No two handlers (fault, filter, catch, or finally) can have the same starting address. When an exception occurs it is necessary to convert the execution address to the correct most lexically nested try block in which the exception occurred.

SEHBlock ::=
TryBlock SEHClause [*SEHClause**]

The next few subclauses expand upon this simple description, by describing the five kinds of code block that take part in exception processing: `try`, `catch`, `filter`, `finally`, and `fault`. (Note that there are restrictions upon how many, and what kinds of *SEHClause* a given *TryBlock* can have; see Partition I for details.)

The remaining syntax items are described in detail below; they are collected here for reference.

TryBlock ::=	**Description**
`.try` *Label* `to` *Label*	Protect region from first label to prior to second
| `.try` *ScopeBlock*	*ScopeBlock* is protected

SEHClause ::=	**Description**
`catch` *TypeReference HandlerBlock*	Catch all objects of the specified type
| `fault` *HandlerBlock*	Handle all exceptions but not normal exit
| `filter` *Label HandlerBlock*	Enter handler only if filter succeeds
| `finally` *HandlerBlock*	Handle all exceptions and normal exit

HandlerBlock::=	**Description**
`handler` *Label* `to` *Label*	Handler range is from first label to prior to second
| *ScopeBlock*	*ScopeBlock* is the handler block

19.1 Protected blocks

A *try*, or *protected*, or *guarded*, block is declared with the `.try` directive.

TryBlock ::=	**Descriptions**
`.try` *Label* `to` *Label*	Protect region from first label to prior to second.
| `.try` *ScopeBlock*	*ScopeBlock* is protected

In the first case, the protected block is delimited by two labels. The first label is the first instruction to be protected, while the second label is the instruction just beyond the last one to be protected. Both labels shall be defined prior to this point.

The second case uses a scope block (§15.4.4) after the `.try` directive—the instructions within that scope are the ones to be protected.

©ISO/IEC 2006 – All rights reserved

19.2 Handler blocks

HandlerBlock `::=`	**Description**
| `handler` *Label* `to` *Label*	Handler range is from first label to prior to second
| *ScopeBlock*	*ScopeBlock* is the handler block

In the first case, the labels enclose the instructions of the handler block, the first label being the first instruction of the handler while the second is the instruction immediately after the handler. In the second case, the handler block is just a scope block.

19.3 Catch blocks

A catch block is declared using the `catch` keyword. This specifies the type of exception object the clause is designed to handle, and the handler code itself.

SEHClause `::=`
`catch` *TypeReference HandlerBlock*

[*Example:*

```
.try {
       …                            // protected instructions
       leave  exitSEH               // normal exit
} catch [mscorlib]System.FormatException {
       …                            // handle the exception
       pop                          // pop the exception object
       leave  exitSEH               // leave catch handler
}
exitSEH:                            // continue here
```

end example]

19.4 Filter blocks

A filter block is declared using the filter keyword.

SEHClause `::= …`
| `filter` *Label HandlerBlock*
| `filter` *Scope HandlerBlock*

The filter code begins at the specified label and ends at the first instruction of the handler block. (Note that the CLI demands that the filter block shall immediately precede, within the CIL stream, its corresponding handler block.)

[*Example:*

```
.method public static void m () {
    .try {
      …                     // protected instructions
      leave   exitSEH       // normal exit
    }

    filter {
      …                     // decide whether to handle
      pop                   // pop exception object
      ldc.i4.1              // EXCEPTION_EXECUTE_HANDLER
      endfilter             // return answer to CLI
    }
```

 ©ISO/IEC 2006 – All rights reserved

```
    {
      …                    // handle the exception
      pop                  // pop the exception object
      leave   exitSEH      // leave filter handler
    }
  exitSEH:
    …
  }
```

end example]

19.5 Finally blocks

A finally block is declared using the finally keyword. This specifies the handler code, with this grammar:

SEHClause `::= …`
`| finally` *HandlerBlock*

The last possible CIL instruction that can be executed in a finally handler shall be `endfinally`.

[*Example:*

```
.try {
       …                   // protected instructions
       leave exitTry       // shall use leave
} finally {
       …                   // finally handler
       endfinally
}
exitTry:                   // back to normal
```

19.6 Fault handlers

end example]

A fault block is declared using the fault keyword. This specifies the handler code, with this grammar:

SEHClause `::= …`
`| fault` *HandlerBlock*

The last possible CIL instruction that can be executed in a fault handler shall be `endfault`.

[*Example:*

```
.method public static void m() {
  startTry:
       …                   // protected instructions
       leave  exitSEH      // shall use leave
  endTry:

startFault:
       …                   // fault handler instructions
       endfault

endFault:
       .try startTry to endTry fault handler startFault to endFault

exitSEH:                   // back to normal
}
```

end example]

20 Declarative security

Many languages that target the CLI use attribute syntax to attach declarative security attributes to items in the metadata. This information is actually converted by the compiler into an XML-based representation that is stored in the metadata, see §22.11. By contrast, *ilasm* requires the conversion information to be represented in its input.

SecurityDecl ::=
.permissionset *SecAction* = '(' *Bytes* ')'
| .permission *SecAction TypeReference* '(' *NameValPairs* ')'

NameValPairs ::= *NameValPair* [',' *NameValPair*]*

NameValPair ::= *SQSTRING* '=' *SQSTRING*

In .permission, *TypeReference* specifies the permission class and *NameValPairs* specifies the settings. See §22.11

In .permissionset the bytes specify the encoded version of the security settings:

SecAction ::=	**Description**
assert	Assert permission so that callers do not need it.
| demand	Demand permission of all callers.
| deny	Deny permission so checks will fail.
| inheritcheck	Demand permission of a derived class.
| linkcheck	Demand permission of caller.
| permitonly	Reduce permissions so check will fail.
| reqopt	Request optional additional permissions.
| reqrefuse	Refuse to be granted these permissions.
| request	Hint that permission might be required.

 ©ISO/IEC 2006 – All rights reserved

21 Custom attributes

Custom attributes add user-defined annotations to the metadata. Custom attributes allow an instance of a type to be stored with any element of the metadata. This mechanism can be used to store application-specific information at compile time, and to access it either at runtime or when another tool reads the metadata. While any user-defined type can be used as an attribute, CLS compliance requires that attributes will be instances of types whose base class is `System.Attribute`. The CLI predefines some attribute types and uses them to control runtime behavior. Some languages predefine attribute types to represent language features not directly represented in the CTS. Users or other tools are welcome to define and use additional attribute types.

Custom attributes are declared using the directive `.custom`, followed by the method declaration for a type constructor, optionally followed by a *Bytes* in parentheses:

CustomDecl ::=
Ctor ['=' '(' *Bytes* ')']

The *Ctor* item represents a method declaration (§15.4), specific for the case where the method's name is `.ctor`. [*Example:*

```
.custom instance void myAttribute::.ctor(bool, bool) = ( 01 00 00 01 00
00 )
```

end example]

Custom attributes can be attached to *any* item in metadata, except a custom attribute itself. Commonly, custom attributes are attached to assemblies, modules, classes, interfaces, value types, methods, fields, properties, generic parameters, and events (the custom attribute is attached to the immediately preceding declaration)

The *Bytes* item is not required if the constructor takes no arguments. In such cases, all that matters is the presence of the custom attribute.

If the constructor takes parameters, their values shall be specified in the *Bytes* item. The format for this 'blob' is defined in §23.3.

[*Example:* The following shows a class that is marked with the attribute called `System.CLSCompliantAttribute` and a method that is marked with the attribute called `System.ObsoleteAttribute`.

```
.class public MyClass extends [mscorlib]System.Object
{ .custom instance void [mscorlib]System.CLSCompliantAttribute::.ctor(bool) =
    ( 01 00 01 00 00 )
  .method public static void CalculateTotals() cil managed
{ .custom instance void [mscorlib]System.ObsoleteAttribute::.ctor() =
    ( 01 00 00 00 )
  ret
}
```

end example]

21.1 CLS conventions: custom attribute usage

CLS imposes certain conventions upon the use of custom attributes in order to improve cross-language operation. See Partition I for details.

21.2 Attributes used by the CLI

There are two kinds of custom attributes, called *genuine custom attributes*, and *pseudo custom attributes*. Custom attributes and pseudo custom attributes are treated differently, at the time they are defined, as follows:

- A custom attribute is stored directly into the metadata; the'blob' which holds its defining data is stored as-is. That 'blob' can be retrieved later.

- A pseudo custom attribute is recognized because its name is one of a short list. Rather than store its 'blob' directly in metadata, that 'blob' is parsed, and the information it contains is used to set bits and/or fields within metadata tables. The 'blob' is then discarded; it cannot be retrieved later.

Pseudo custom attributes therefore serve to capture user directives, using the same familiar syntax the compiler provides for genuine custom attributes, but these user directives are then stored into the more space-efficient form of metadata tables. Tables are also faster to check at runtime than are genuine custom attributes.

Many custom attributes are invented by higher layers of software. They are stored and returned by the CLI, without its knowing or caring what they 'mean'. But all pseudo custom attributes, plus a collection of genuine custom attributes, are of special interest to compilers and to the CLI. An example of such custom attributes is `System.Reflection.DefaultMemberAttribute`. This is stored in metadata as a genuine custom attribute 'blob', but reflection uses this custom attribute when called to invoke the default member (property) for a type.

The following subclauses list all of the pseudo custom attributes and *distinguished* custom attributes, where *distinguished* means that the CLI and/or compilers pay direct attention to them, and their behavior is affected in some way.

In order to prevent name collisions into the future, all custom attributes in the `System` namespace are reserved for standardization.

21.2.1 Pseudo custom attributes

The following table lists the CLI pseudo custom attributes. (Not all of these attributes are specified in this Standard, but all of their names are reserved and shall not be used for other purposes. For details on these attributes, see the documentation for the corresponding class in Partition IV.) They are defined in the namespaces `System.Reflection`, `System.Runtime.CompilerServices`, and `System.Runtime.InteropServices` namespaces.

Attribute	Description
`AssemblyAlgorithmIDAttribute`	Records the ID of the hash algorithm used (reserved only)
`AssemblyFlagsAttribute`	Records the flags for this assembly (reserved only)
`DllImportAttribute`	Provides information about code implemented within an unmanaged library
`FieldOffsetAttribute`	Specifies the byte offset of fields within their enclosing class or value type
`InAttribute`	Indicates that a method parameter is an [in] argument
`MarshalAsAttribute`	Specifies how a data item should be marshalled between managed and unmanaged code (see §23.4).
`MethodImplAttribute`	Specifies details of how a method is implemented
`OutAttribute`	Indicates that a method parameter is an [out] argument
`StructLayoutAttribute`	Allows the caller to control how the fields of a class or value type are laid out in managed memory

These attributes affect bits and fields in metadata, as follows:

`AssemblyAlgorithmIDAttribute`: sets the *Assembly.HashAlgId* field.

`AssemblyFlagsAttribute`: sets the *Assembly.Flags* field.

`DllImportAttribute`: sets the *Method.Flags.PinvokeImpl* bit for the attributed method; also, adds a new row into the *ImplMap* table (setting *MappingFlags*, *MemberForwarded*, *ImportName* and *ImportScope* columns).

`FieldOffsetAttribute`: sets the *FieldLayout.OffSet* value for the attributed field.

`InAttribute`: sets the *Param.Flags.In* bit for the attributed parameter.

 ©ISO/IEC 2006 – All rights reserved

`MarshalAsAttribute`: sets the *Field.Flags.HasFieldMarshal* bit for the attributed field (or the *Param.Flags.HasFieldMarshal* bit for the attributed parameter); also enters a new row into the FieldMarshal table for both *Parent* and *NativeType* columns.

`MethodImplAttribute`: sets the *Method.ImplFlags* field of the attributed method.

`OutAttribute`: sets the *Param.Flags.Out* bit for the attributed parameter.

`StructLayoutAttribute`: sets the *TypeDef.Flags.LayoutMask* sub-field for the attributed type, and, optionally, the *TypeDef.Flags.StringFormatMask* sub-field, the *ClassLayout.PackingSize,*and *ClassLayout.ClassSize* fields for that type.

21.2.2 Custom attributes defined by the CLS

The CLS specifies certain Custom Attributes and requires that conformant languages support them. These attributes are located under `System`.

Attribute	Description
`AttributeUsageAttribute`	Used to specify how an attribute is intended to be used.
`ObsoleteAttribute`	Indicates that an element is not to be used.
`CLSCompliantAttribute`	Indicates whether or not an element is declared to be CLS compliant through an instance field on the attribute object.

21.2.3 Custom attributes for security

The following custom attributes are defined in the `System.Net` and `System.Security.Permissions` namespaces. Note that these are all base classes; the actual instances of security attributes found in assemblies will be sub-classes of these.

Attribute	Description
`CodeAccessSecurityAttribute`	This is the base attribute class for declarative security using custom attributes.
`DnsPermissionAttribute`	Custom attribute class for declarative security with DnsPermission
`EnvironmentPermissionAttribute`	Custom attribute class for declarative security with EnvironmentPermission.
`FileIOPermissionAttribute`	Custom attribute class for declarative security with FileIOPermission.
`ReflectionPermissionAttribute`	Custom attribute class for declarative security with ReflectionPermission.
`SecurityAttribute`	This is the base attribute class for declarative security from which CodeAccessSecurityAttribute is derived.
`SecurityPermissionAttribute`	Indicates whether the attributed method can affect security settings
`SocketPermissionAttribute`	Custom attribute class for declarative security with SocketPermission.
`WebPermissionAttribute`	Custom attribute class for declarative security with WebPermission.

Note that any other security-related custom attributes (i.e., any custom attributes that derive from `System.Security.Permissions.SecurityAttribute`) included into an assembly, can cause a conforming

implementaion of the CLI to reject such an assembly when it is loaded, or throw an exception at runtime if any attempt is made to access those security-related custom attributes. (This statement holds true for any custom attributes that cannot be resolved; security-related custom attributes are just one particular case)

21.2.4 Custom attributes for TLS

A custom attribute that denotes a TLS (thread-local storage) field is defined in the `System` namespace.

Attribute	Description
`ThreadStaticAttribute`	Provides for type member fields that are relative for the thread.

21.2.5 Custom attributes, various

The following custom attributes control various aspects of the CLI:

Attribute	Namespace	Description
`ConditionalAttribute`	`System.Diagnostics`	Used to mark methods as callable, based on some compile-time condition. If the condition is false, the method will not be called
`DecimalConstantAttribute`	`System.Runtime.CompilerServices`	Stores the value of a decimal constant in metadata
`DefaultMemberAttribute`	`System.Reflection`	Defines the member of a type that is the default member used by reflection's InvokeMember.
`FaultModeAttribute`	`System.Runtime.CompilerServices`	Indicates whether exceptions from instruction checks are precise or imprecise.
`FlagsAttribute`	`System`	Custom attribute indicating an enumeration should be treated as a bitfield; that is, a set of flags
`IndexerNameAttribute`	`System.Runtime.CompilerServices`	Indicates the name by which a property having one or more parameters will be known in programming languages that do not support such a facility directly
`ParamArrayAttribute`	`System`	Indicates that the method will allow a variable number of arguments in its invocation

 ©ISO/IEC 2006 – All rights reserved

22 Metadata logical format: tables

This clause defines the structures that describe metadata, and how they are cross-indexed. This corresponds to how metadata is laid out, after being read into memory from a PE file. (For a description of metadata layout inside the PE file itself, see §24)

Metadata is stored in two kinds of structure: tables (arrays of records) and heaps. There are four heaps in any module: String, Blob, Userstring, and Guid. The first three are byte arrays (so valid indexes into these heaps might be 0, 23, 25, 39, etc). The Guid heap is an array of GUIDs, each 16 bytes wide. Its first element is numbered 1, its second 2, and so on.

Each entry in each column of each table is either a constant or an index.

Constants are either literal values (e.g., ALG_SID_SHA1 = 4, stored in the *HashAlgId* column of the *Assembly* table), or, more commonly, bitmasks. Most bitmasks (they are almost all called *Flags*) are 2 bytes wide (e.g., the *Flags* column in the *Field* table), but there are a few that are 4 bytes (e.g., the *Flags* column in the *TypeDef* table).

Each index is either 2 or 4 bytes wide. The index points into the same or another table, or into one of the four heaps. The size of each index column in a table is only made 4 bytes if it needs to be for that particular module. So, if a particular column indexes a table, or tables, whose highest row number fits in a 2-byte value, the indexer column need only be 2 bytes wide. Conversely, for tables containing 64K or more rows, an indexer of that table will be 4 bytes wide.

Indexes to tables begin at 1, so index 1 means the first row in any given metadata table. (An index value of zero denotes that it does not index a row at all; that is, it behaves like a null reference.)

There are two kinds of columns that index a metadata table. (For details of the physical representation of these tables, see §24.2.6):

- Simple – such a column indexes one, and only one, table. For example, the *FieldList* column in the *TypeDef* table always indexes the *Field* table. So all values in that column are simple integers, giving the row number in the target table
- Coded – such a column indexes any of several tables. For example, the *Extends* column in the *TypeDef* table can index into the *TypeDef* or *TypeRef* table. A few bits of that index value are reserved to define which table it targets. For the most part, this specification talks of index values after being decoded into row numbers within the target table. However, the specification includes a description of these coded indexes in the section that describes the physical layout of Metadata (§24).

Metadata preserves name strings, as created by a compiler or code generator, unchanged. Essentially, it treats each string as an opaque *blob*. In particular, it preserves case. The CLI imposes no limit on the length of names stored in metadata and subsequently processed by the CLI

Matching AssemblyRefs and ModuleRefs to their corresponding Assembly and Module shall be performed case-blind (see Partition I). However, all other name matches (type, field, method, property, event) shall be exact – so that this level of resolution is the same across all platforms, whether their OS is case-sensitive or not.

Tables are given both a name (e.g., "Assembly") and a number (e.g., 0x20). The number for each table is listed immediately with its title in the following subclauses. The table numbers indicate the order in which their corresponding table shall appear in the PE file, and there is a set of bits (§24.2.6) saying whether a given table exists or not. The number of a table is the position within that set of bits.

A few of the tables represent extensions to regular CLI files. Specifically, ENCLog and ENCMap, which occur in temporary images, generated during "Edit and Continue" or "incremental compilation" scenarios, whilst debugging. Both table types are reserved for future use.

References to the methods or fields of a type are stored together in a metadata table called the *MemberRef* table. However, sometimes, for clearer explanation, this standard distinguishes between these two kinds of reference, calling them “MethodRef” and “FieldRef”.

Certain tables are required to be sorted by a primary key, as follows:

Table	Primary Key Column
ClassLayout	Parent
Constant	Parent
CustomAttribute	Parent
DeclSecurity	Parent
FieldLayout	Field
FieldMarshal	Parent
FieldRVA	Field
GenericParam	Owner
GenericParamConstraint	Owner
ImplMap	MemberForwarded
InterfaceImpl	Class
MethodImpl	Class
MethodSemantics	Association
NestedClass	NestedClass

Furthermore, the InterfaceImpl table is sorted using the Interface column as a secondary key, and the GenericParam table is sorted using the Number column as a secondary key.

Finally, the TypeDef table has a special ordering constraint: the definition of an enclosing class shall precede the definition of all classes it encloses.

Metadata items (records in the metadata tables) are addressed by metadata tokens. Uncoded metadata tokens are 4-byte unsigned integers, which contain the metadata table index in the most significant byte and a 1-based record index in the three least-significant bytes. Metadata tables and their respective indexes are described in §22.2 and later subclauses.

Coded metadata tokens also contain table and record indexes, but in a different format. For details on the encoding, see §24.2.6.

22.1 Metadata validation rules

This contains informative text only

The subclauses that follow describe the schema for each kind of metadata table, and explain the detailed rules that guarantee metadata emitted into any PE file is valid. Checking that metadata is valid ensures that later processing (such as checking the CIL instruction stream for type safety, building method tables, CIL-to-native-code compilation, and data marshalling) will not cause the CLI to crash or behave in an insecure fashion.

In addition, some of the rules are used to check compliance with the CLS requirements (see Partition I) even though these are not related to valid Metadata. These are marked with a trailing `[CLS]` tag.

The rules for valid metadata refer to an individual module. A module is any collection of metadata that *could* typically be saved to a disk file. This includes the output of compilers and linkers, or the output of script compilers (where the metadata is often held only in memory, but never actually saved to a file on disk).

The rules address intra-module validation only. As such, software that checks conformance with this standard need not resolve references or walk type hierarchies defined in other modules. However, even if two modules, A and B, analyzed separately, contain only valid metadata, they can still be in error when viewed together (e.g.,

 ©ISO/IEC 2006 – All rights reserved

a call from Module A, to a method defined in module B, might specify a call site signature that does not match the signatures defined for that method in B).

All checks are categorized as ERROR, WARNING, or CLS.

- An ERROR check reports something that might cause a CLI to crash or hang, it might run but produce wrong answers; or it might be entirely benign. Conforming implementations of the CLI can exist that will not accept metadata that violates an ERROR rule, and therefore such metadata is invalid and is not portable.
- A WARNING check reports something, not actually wrong, but possibly a slip on the part of the compiler. Normally, it indicates a case where a compiler could have encoded the same information in a more compact fashion or where the metadata represents a construct that can have no actual use at runtime. All conforming implementations shall support metadata that violate only WARNING rules; hence such metadata is both valid and portable.
- A CLS check reports lack of compliance with the Common Language Specification (see Partition I). Such metadata is both valid and portable, but programming languages might exist that cannot process it, even though all conforming implementations of the CLI support the constructs.

Validation rules fall into the following broad categories:

- **Number of Rows:** A few tables are allowed only one row (e.g., Module table). Most have no such restriction.
- **Unique Rows:** No table shall contain duplicate rows, where "duplicate" is defined in terms of its *key* column, or combination of columns.
- **Valid Indexes:** Columns which are indexes shall point somewhere sensible, as follows:
  - Every index into the String, Blob, or Userstring heaps shall point *into* that heap, neither before its start (offset 0), nor after its end.
  - Every index into the Guid heap shall lie between 1 and the maximum element number in this module, inclusive.
  - Every index (row number) into another metadata table shall lie between 0 and that table's row count + 1 (for some tables, the index can point just past the end of any target table, meaning it indexes nothing).
- **Valid Bitmasks:** Columns which are bitmasks shall have only valid permutations of bits set.
- **Valid RVAs:** There are restrictions upon fields and methods that are assigned RVAs (Relative Virtual Addresses, which are byte offsets, expressed from the address at which the corresponding PE file is loaded into memory).

Note that some of the rules listed below really don't say anything—for example, some rules state that a particular table is allowed zero or more rows—in which case, there is no way that the check can fail. This is done simply for completeness, to record that such details have indeed been addressed, rather than overlooked.

End informative text

The CLI imposes no limit on the length of names stored in metadata, and subsequently processed by a CLI implementation.

22.2 Assembly : 0x20

The *Assembly* table has the following columns:

- *HashAlgId* (a 4-byte constant of type AssemblyHashAlgorithm, §23.1.1)
- *MajorVersion*, *MinorVersion*, *BuildNumber*, *RevisionNumber* (each being 2-byte constants)
- *Flags* (a 4-byte bitmask of type AssemblyFlags, §23.1.2)

- *PublicKey* (an index into the Blob heap)
- *Name* (an index into the String heap)
- *Culture* (an index into the String heap)

The *Assembly* table is defined using the `.assembly` directive (§6.2); its columns are obtained from the respective `.hash algorithm`, `.ver`, `.publickey`, and `.culture` (§6.2.1). (For an example, see §6.2.)

This contains informative text only

1. The *Assembly* table shall contain zero or one row [ERROR]
2. *HashAlgId* shall be one of the specified values [ERROR]
3. *MajorVersion*, *MinorVersion*, *BuildNumber*, and *RevisionNumber* can each have any value
4. *Flags* shall have only those values set that are specified [ERROR]
5. *PublicKey* can be null or non-null
6. *Name* shall index a non-empty string in the String heap [ERROR]
7. The string indexed by *Name* can be of unlimited length
8. *Culture* can be null or non-null
9. If *Culture* is non-null, it shall index a single string from the list specified (§23.1.3) [ERROR]

[*Note: Name* is a simple name (e.g., "Foo", with no drive letter, no path, and no file extension); on POSIX-compliant systems, *Name* contains no colon, no forward-slash, no backslash, and no period. *end note*]

End informative text

22.3 AssemblyOS : 0x22

The *AssemblyOS* table has the following columns:

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

This record should not be emitted into any PE file. However, if present in a PE file, it shall be treated as if all its fields were zero. It shall be ignored by the CLI.

22.4 AssemblyProcessor : 0x21

The *AssemblyProcessor* table has the following column:

- *Processor* (a 4-byte constant)

This record should not be emitted into any PE file. However, if present in a PE file, it should be treated as if its field were zero. It should be ignored by the CLI.

22.5 AssemblyRef : 0x23

The *AssemblyRef* table has the following columns:

- *MajorVersion*, *MinorVersion*, *BuildNumber*, *RevisionNumber* (each being 2-byte constants)
- *Flags* (a 4-byte bitmask of type AssemblyFlags, §23.1.2)
- *PublicKeyOrToken* (an index into the Blob heap, indicating the public key or token that identifies the author of this Assembly)
- *Name* (an index into the String heap)

 ©ISO/IEC 2006 – All rights reserved

- *Culture* (an index into the String heap)
- *HashValue* (an index into the Blob heap)

The table is defined by the `.assembly extern` directive (§6.3). Its columns are filled using directives similar to those of the *Assembly* table except for the *PublicKeyOrToken* column, which is defined using the `.publickeytoken` directive. (For an example, see §6.3.)

This contains informative text only

1. *MajorVersion*, *MinorVersion*, *BuildNumber*, and *RevisionNumber* can each have any value
2. *Flags* shall have only one bit set, the `PublicKey` bit (§23.1.2). All other bits shall be zero. [ERROR]
3. *PublicKeyOrToken* can be null, or non-null (note that the *Flags*`.PublicKey` bit specifies whether the 'blob' is a full public key, or the short hashed token)
4. If non-null, then *PublicKeyOrToken* shall index a valid offset in the Blob heap [ERROR]
5. *Name* shall index a non-empty string, in the String heap (there is no limit to its length) [ERROR]
6. *Culture* can be null or non-null.
7. If non-null, it shall index a single string from the list specified (§23.1.3) [ERROR]
8. *HashValue* can be null or non-null
9. If non-null, then *HashValue* shall index a non-empty 'blob' in the Blob heap [ERROR]
10. The *AssemblyRef* table shall contain no duplicates (where duplicate rows are deemd to be those having the same *MajorVersion*, *MinorVersion*, *BuildNumber*, *RevisionNumber*, *PublicKeyOrToken*, *Name,* and *Culture*) [WARNING]

[*Note: Name* is a simple name (e.g., "Foo", with no drive letter, no path, and no file extension); on POSIX-compliant systems *Name* contains no colon, no forward-slash, no backslash, and no period. *end note*]

End informative text

22.6 AssemblyRefOS : 0x25

The *AssemblyRefOS* table has the following columns:

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

These records should not be emitted into any PE file. However, if present in a PE file, they should be treated as-if their fields were zero. They should be ignored by the CLI.

22.7 AssemblyRefProcessor : 0x24

The *AssemblyRefProcessor* table has the following columns:

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

These records should not be emitted into any PE file. However, if present in a PE file, they should be treated as-if their fields were zero. They should be ignored by the CLI.

©ISO/IEC 2006 – All rights reserved

22.8 ClassLayout : 0x0F

The *ClassLayout* table is used to define how the fields of a class or value type shall be laid out by the CLI. (Normally, the CLI is free to reorder and/or insert gaps between the fields defined for a class or value type.)

[*Rationale:* This feature is used to lay out a managed value type in exactly the same way as an unmanaged C struct, allowing a managed value type to be handed to unmanaged code, which then accesses the fields exactly as if that block of memory had been laid out by unmanaged code. *end rationale*]

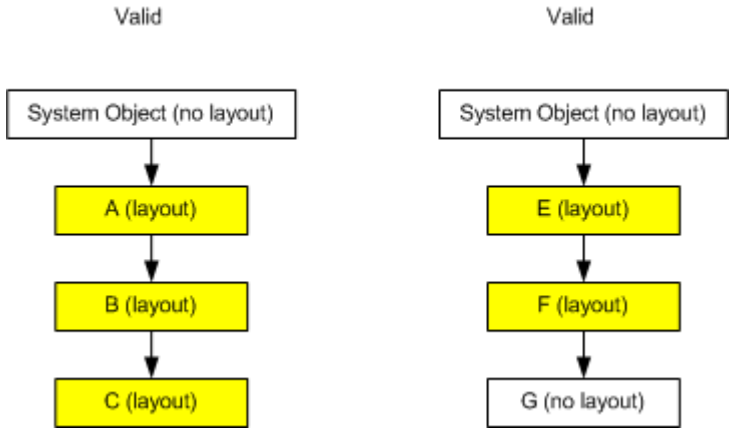
The information held in the *ClassLayout* table depends upon the *Flags* value for {*AutoLayout, SequentialLayout, ExplicitLayout*} in the owner class or value type.

A type *has layout* if it is marked *SequentialLayout* or *ExplicitLayout*. If any type within an inheritance chain has layout, then so shall all its base classes, up to the one that descends immediately from `System.ValueType` (if it exists in the type's hierarchy); otherwise, from `System.Object`.

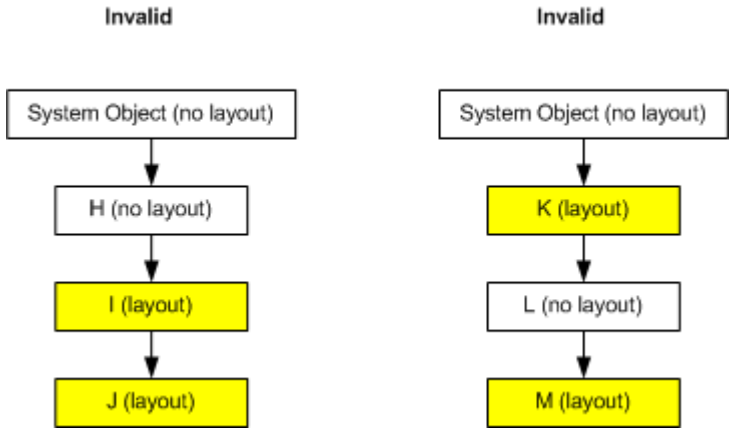
This contains informative text only

Layout cannot begin part way down the chain. But it *is* valid to *stop* "having layout" at any point down the chain.

For example, in the diagrams below, Class A derives from `System.Object`; class B derives from A; class C derives from B. `System.Object` has no layout. But A, B and C are all defined with layout, and that is valid.



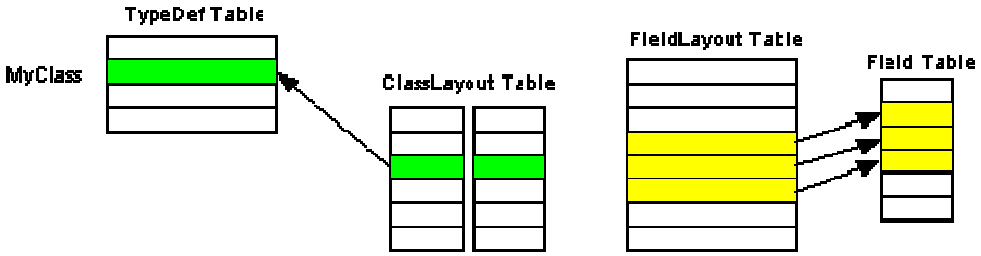
The situation with classes E, F, and G is similar. G has no layout, and this too is valid. The following picture shows two *invalid* setups:



On the left, the "chain with layout" does not start at the 'highest' class. And on the right, there is a 'hole' in the "chain with layout"

 ©ISO/IEC 2006 – All rights reserved

Layout information for a class or value type is held in two tables (*ClassLayout* and *FieldLayout*), as shown in the following diagram:



In this example, row 3 of the *ClassLayout* table points to row 2 in the *TypeDef* table (the definition for a Class, called "MyClass"). Rows 4–6 of the *FieldLayout* table point to corresponding rows in the *Field* table. This illustrates how the CLI stores the explicit offsets for the three fields that are defined in "MyClass" (there is always one row in the *FieldLayout* table for each field in the owning class or value type) So, the *ClassLayout* table acts as an extension to those rows of the *TypeDef* table that have layout info; since many classes do not have layout info, overall, this design saves space.

End informative text

The *ClassLayout* table has the following columns:

- *PackingSize* (a 2-byte constant)
- *ClassSize* (a 4-byte constant)
- *Parent* (an index into the *TypeDef* table)

The rows of the *ClassLayout* table are defined by placing `.pack` and `.size` directives on the body of the type declaration in which this type is declared (§10.2). When either of these directives is omitted, its corresponding value is zero. (See §10.7.)

ClassSize of zero does not mean the class has zero size. It means that no `.size` directive was specified at definition time, in which case, the actual size is calculated from the field types, taking account of packing size (default or specified) and natural alignment on the target, runtime platform.

This contains informative text only

1. A *ClassLayout* table can contain zero or more rows
2. *Parent* shall index a valid row in the *TypeDef* table, corresponding to a Class or ValueType (but not to an Interface) [ERROR]
3. The Class or ValueType indexed by *Parent* shall be *SequentialLayout* or *ExplicitLayout* (§23.1.15). (That is, *AutoLayout* types shall not own any rows in the *ClassLayout* table.) [ERROR]
4. If *Parent* indexes a *SequentialLayout* type, then:
   - *PackingSize* shall be one of {0, 1, 2, 4, 8, 16, 32, 64, 128}. (0 means use the default pack size for the platform on which the application is running.) [ERROR]
   - If *Parent* indexes a ValueType, then *ClassSize* shall be less than 1 MByte (0x100000 bytes). [ERROR]
5. If *Parent* indexes an *ExplicitLayout* type, then
   - if *Parent* indexes a ValueType, then *ClassSize* shall be less than 1 MByte (0x100000 bytes) [ERROR]
   - *PackingSize* shall be 0. (It makes no sense to provide explicit offsets for each field, as well as a packing size.) [ERROR]

©ISO/IEC 2006 – All rights reserved

6. Note that an *ExplicitLayout* type *might* result in a verifiable type, provided the layout does not create a type whose fields overlap.

7. Layout along the length of an inheritance chain shall follow the rules specified above (starting at 'highest' Type, with no 'holes', etc.) [ERROR]

End informative text

22.9 Constant : 0x0B

The *Constant* table is used to store compile-time, constant values for fields, parameters, and properties.

The *Constant* table has the following columns:

- *Type* (a 1-byte constant, followed by a 1-byte padding zero); see §23.1.16 . The encoding of *Type* for the **nullref** value for *FieldInit* in *ilasm* (§16.2) is `ELEMENT_TYPE_CLASS` with a *Value* of a 4-byte zero. Unlike uses of `ELEMENT_TYPE_CLASS` in signatures, this one is *not* followed by a type token.
- *Parent* (an index into the *Param*, *Field*, or *Property* table; more precisely, a *HasConstant* (§24.2.6) coded index)
- *Value* (an index into the Blob heap)

Note that *Constant* information does not directly influence runtime behavior, although it is visible via Reflection (and hence can be used to implement functionality such as that provided by `System.Enum.ToString`). Compilers inspect this information, at compile time, when importing metadata, but the value of the constant itself, if used, becomes embedded into the CIL stream the compiler emits. There are no CIL instructions to access the *Constant* table at runtime.

A row in the *Constant* table for a parent is created whenever a compile-time value is specified for that parent. (For an example, see §16.2.)

This contains informative text only

1. *Type* shall be exactly one of: `ELEMENT_TYPE_BOOLEAN`, `ELEMENT_TYPE_CHAR`, `ELEMENT_TYPE_I1`, `ELEMENT_TYPE_U1`, `ELEMENT_TYPE_I2`, `ELEMENT_TYPE_U2`, `ELEMENT_TYPE_I4`, `ELEMENT_TYPE_U4`, `ELEMENT_TYPE_I8`, `ELEMENT_TYPE_U8`, `ELEMENT_TYPE_R4`, `ELEMENT_TYPE_R8`, or `ELEMENT_TYPE_STRING`; or `ELEMENT_TYPE_CLASS` with a *Value* of zero (§23.1.16) [ERROR]
2. *Type* shall not be any of: `ELEMENT_TYPE_I1`, `ELEMENT_TYPE_U2`, `ELEMENT_TYPE_U4`, or `ELEMENT_TYPE_U8` (§23.1.16) [CLS]
3. *Parent* shall index a valid row in the *Field*, *Property*, or *Param* table. [ERROR]
4. There shall be no duplicate rows, based upon *Parent* [ERROR]
5. *Type* shall match exactly the declared type of the *Param*, *Field*, or *Property* identified by *Parent* (in the case where the parent is an enum, it shall match exactly the underlying type of that enum). [CLS]

End informative text

22.10 CustomAttribute : 0x0C

The *CustomAttribute* table has the following columns:

- *Parent* (an index into *any* metadata table, except the *CustomAttribute* table itself; more precisely, a *HasCustomAttribute* (§24.2.6) coded index)
- *Type* (an index into the *MethodDef* or *MemberRef* table; more precisely, a *CustomAttributeType* (§24.2.6) coded index)
- *Value* (an index into the Blob heap)

 ©ISO/IEC 2006 – All rights reserved

The *CustomAttribute* table stores data that can be used to instantiate a Custom Attribute (more precisely, an object of the specified Custom Attribute class) at runtime. The column called *Type* is slightly misleading—it actually indexes a constructor method—the owner of that constructor method is the Type of the Custom Attribute.

A row in the *CustomAttribute* table for a parent is created by the `.custom` attribute, which gives the value of the *Type* column and optionally that of the *Value* column (§21).

This contains informative text only

All binary values are stored in little-endian format (except for *PackedLen* items, which are used only as a count for the number of bytes to follow in a UTF8 string).

1. No *CustomAttribute* is required; that is, *Value* is permitted to be null.
2. *Parent* can be an index into *any* metadata table, *except* the *CustomAttribute* table itself [ERROR]
3. *Type* shall index a valid row in the *Method* or *MemberRef* table. That row shall be a constructor method (for the class of which this information forms an instance) [ERROR]
4. *Value* can be null or non-null.
5. If *Value* is non-null, it shall index a 'blob' in the Blob heap [ERROR]
6. The following rules apply to the overall structure of the *Value* 'blob' (§23.3):
   - *Prolog* shall be 0x0001 [ERROR]
   - There shall be as many occurrences of *FixedArg* as are declared in the Constructor method [ERROR]
   - *NumNamed* can be zero or more
   - There shall be exactly *NumNamed* occurrences of *NamedArg* [ERROR]
   - Each *NamedArg* shall be accessible by the caller [ERROR]
   - If *NumNamed* = 0 then there shall be no further items in the *CustomAttrib* [ERROR]
7. The following rules apply to the structure of *FixedArg* (§23.3):
   - If this item is not for a vector (a single-dimension array with lower bound of 0), then there shall be exactly one *Elem* [ERROR]
   - If this item is for a vector, then:
   - *NumElem* shall be 1 or more [ERROR]
   - This shall be followed by *NumElem* occurrences of *Elem* [ERROR]
8. The following rules apply to the structure of *Elem* (§23.3):
   - If this is a simple type or an enum (see §23.3 for how this is defined), then *Elem* consists simply of its value [ERROR]
   - If this is a string or a Type, then *Elem* consists of a *SerString* – *PackedLen* count of bytes, followed by the UTF8 characters [ERROR]
   - If this is a boxed simple value type (`bool`, `char`, `float32`, `float64`, `int8`, `int16`, `int32`, `int64`, `unsigned int8`, `unsigned int16`, `unsigned int32`, or `unsigned int64`), then *Elem* consists of the corresponding type denoter (`ELEMENT_TYPE_BOOLEAN`, `ELEMENT_TYPE_CHAR`, `ELEMENT_TYPE_I1`, `ELEMENT_TYPE_U1`, `ELEMENT_TYPE_I2`, `ELEMENT_TYPE_U2`, `ELEMENT_TYPE_I4`, `ELEMENT_TYPE_U4`, `ELEMENT_TYPE_I8`, `ELEMENT_TYPE_U8`, `ELEMENT_TYPE_R4`, or `ELEMENT_TYPE_R8`), followed by its value. [ERROR]
9. The following rules apply to the structure of *NamedArg* (§23.3):
   - The single byte `FIELD` (0x53) or `PROPERTY` (0x54) [ERROR]

- The type of the Field or Property is one of `ELEMENT_TYPE_BOOLEAN`, `ELEMENT_TYPE_CHAR`, `ELEMENT_TYPE_I1`, `ELEMENT_TYPE_U1`, `ELEMENT_TYPE_I2`, `ELEMENT_TYPE_U2`, `ELEMENT_TYPE_I4`, `ELEMENT_TYPE_U4`, `ELEMENT_TYPE_I8`, `ELEMENT_TYPE_U8`, `ELEMENT_TYPE_R4`, `ELEMENT_TYPE_R8`, `ELEMENT_TYPE_STRING`, or the constant 0x50 (for an argument of type `System.Type`) [ERROR]
- The name of the Field or Property, respectively with the previous item, as a *SerString* – *PackedLen* count of bytes, followed by the UTF8 characters of the name [ERROR]
- A *FixedArg* (see above) [ERROR]

End informative text

22.11 DeclSecurity : 0x0E

Security attributes, which derive from `System.Security.Permissions.SecurityAttribute` (see Partition IV), can be attached to a *TypeDef*, a *Method*, or an *Assembly*. All constructors of this class shall take a `System.Security.Permissions.SecurityAction` value as their first parameter, describing what should be done with the permission on the type, method or assembly to which it is attached. Code access security attributes, which derive from `System.Security.Permissions. CodeAccessSecurityAttribute`, can have any of the security actions.

These different security actions are encoded in the *DeclSecurity* table as a 2-byte enum (see below). All security custom attributes for a given security action on a method, type, or assembly shall be gathered together, and one `System.Security.PermissionSet` instance shall be created, stored in the Blob heap, and referenced from the *DeclSecurity* table.

[*Note:* The general flow from a compiler's point of view is as follows. The user specifies a custom attribute through some language-specific syntax that encodes a call to the attribute's constructor. If the attribute's type is derived (directly or indirectly) from `System.Security.Permissions.SecurityAttribute` then it is a security custom attribute and requires special treatment, as follows (other custom attributes are handled by simply recording the constructor in the metadata as described in §22.10). The attribute object is constructed, and provides a method (`CreatePermission`) to convert it into a security permission object (an object derived from `System.Security.Permission`). All the permission objects attached to a given metadata item with the same security action are combined together into a `System.Security.PermissionSet`. This permission set is converted into a form that is ready to be stored in XML using its `ToXML` method to create a `System.Security.SecurityElement`. Finally, the XML that is required for the metadata is created using the `ToString` method on the security element. *end note*]

The *DeclSecurity* table has the following columns:

- *Action* (a 2-byte value)
- *Parent* (an index into the *TypeDef*, *MethodDef*, or *Assembly* table; more precisely, a *HasDeclSecurity* (§24.2.6) coded index)
- *PermissionSet* (an index into the Blob heap)

Action is a 2-byte representation of Security Actions (see `System.Security.SecurityAction` in Partition IV). The values 0–0xFF are reserved for future standards use. Values 0x20–0x7F and 0x100–0x07FF are for uses where the action can be ignored if it is not understood or supported. Values 0x80–0xFF and 0x0800–0xFFFF are for uses where the action shall be implemented for secure operation; in implementations where the action is not available, no access to the assembly, type, or method shall be permitted.

Security Action	Note	Explanation of behavior	Valid Scope
Assert	1	Without further checks, satisfy Demand for the specified permission.	Method, Type
Demand	1	Check that all callers in the call chain have been granted specified permission, throw `SecurityException` (see Partition IV) on failure.	Method, Type

 ©ISO/IEC 2006 – All rights reserved

Deny	1	Without further checks refuse Demand for the specified permission.	Method, Type
InheritanceDemand	1	The specified permission shall be granted in order to inherit from class or override virtual method.	Method, Type
LinkDemand	1	Check that the immediate caller has been granted the specified permission; throw `SecurityException` (see Partition IV) on failure.	Method, Type
NonCasDemand	2	Check that the current assembly has been granted the specified permission; throw `SecurityException` (see Partition IV) otherwise.	Method, Type
NonCasLinkDemand	2	Check that the immediate caller has been granted the specified permission; throw `SecurityException` (see Partition IV) otherwise.	Method, Type
PrejitGrant		Reserved for implementation-specific use.	Assembly
PermitOnly	1	Without further checks, refuse Demand for all permissions other than those specified.	Method, Type
RequestMinimum		Specify the minimum permissions required to run.	Assembly
RequestOptional		Specify the optional permissions to grant.	Assembly
RequestRefuse		Specify the permissions not to be granted.	Assembly

Note 1: The specified attribute shall derive from `System.Security.Permissions.CodeAccess-SecurityAttribute`

Note 2: The attribute shall derive from `System.Security.Permissions.SecurityAttribute`, but shall not derive from `System.Security.Permissions.CodeAccessSecurityAttribute`

Parent is a metadata token that identifies the *Method*, *Type*, or *Assembly* on which security custom attributes encoded in *PermissionSet* was defined.

PermissionSet is a 'blob' having the following format:

- A byte containing a period (.).
- A compressed int32 containing the number of attributes encoded in the blob.
- An array of attributes each containing the following:
    - A String, which is the fully-qualified type name of the attribute. (Strings are encoded as a compressed int to indicate the size followed by an array of UTF8 characters.)
    - A set of properties, encoded as the named arguments to a custom attribute would be (as in §23.3, beginning with NumNamed).

The permission set contains the permissions that were requested with an *Action* on a specific *Method*, *Type*, or *Assembly* (see *Parent*). In other words, the blob will contain an encoding of all the attributes on the *Parent* with that particular *Action*.

[*Note*: The first edition of this standard specified an XML encoding of a permission set. Implementations should continue supporting this encoding for backward compatibility. *end note*]

The rows of the *DeclSecurity* table are filled by attaching a `.permission` or `.permissionset` directive that specifies the *Action* and *PermissionSet* on a parent assembly (§6.6) or parent type or method (§10.2).

This contains informative text only

1. *Action* shall have only those values set that are specified [ERROR]

2. *Parent* shall be one of *TypeDef*, *MethodDef*, or *Assembly*. That is, it shall index a valid row in the *TypeDef* table, the *MethodDef* table, or the *Assembly* table. [ERROR]
3. If *Parent* indexes a row in the *TypeDef* table, that row should not define an Interface. The security system ignores any such parent; compilers should not emit such permissions sets. [WARNING]
4. If *Parent* indexes a TypeDef, then its *TypeDef.Flags.HasSecurity* bit shall be set [ERROR]
5. If *Parent* indexes a *MethodDef*, then its *MethodDef.Flags.HasSecurity* bit shall be set [ERROR]
6. *PermissionSet* shall index a 'blob' in the Blob heap [ERROR]
7. The format of the 'blob' indexed by *PermissionSet* shall represent a valid, encoded CLI object graph. (The encoded form of all standardized permissions is specified in Partition IV.) [ERROR]

End informative text

22.12 EventMap : 0x12

The *EventMap* table has the following columns:

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

Note that *EventMap* info does not directly influence runtime behavior; what counts is the information stored for each method that the event comprises.

The *EventMap* and *Event* tables result from putting the `.event` directive on a class (§18).

This contains informative text only

1. *EventMap* table can contain zero or more rows
2. There shall be no duplicate rows, based upon *Parent* (a given class has only one ‘pointer’ to the start of its event list) [ERROR]
3. There shall be no duplicate rows, based upon *EventList* (different classes cannot share rows in the *Event* table) [ERROR]

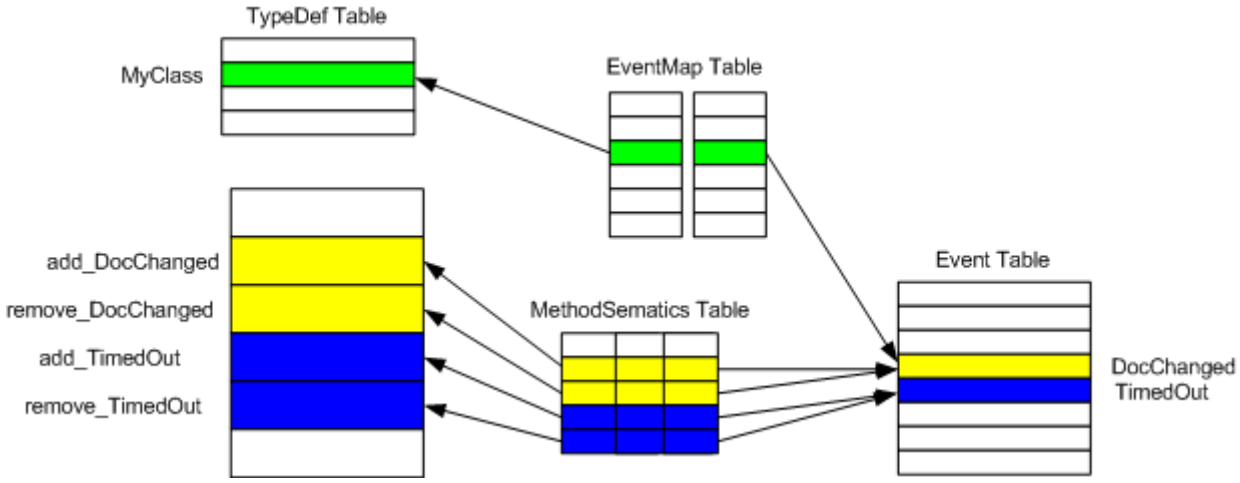
End informative text

22.13 Event : 0x14

Events are treated within metadata much like Properties; that is, as a way to associate a collection of methods defined on a given class. There are two required methods (*add_* and *remove_*) plus an optional one (*raise_*); others are permitted. All of the methods gathered together as an Event shall be defined on the class.

The association between a row in the *TypeDef* table and the collection of methods that make up a given Event is held in three separate tables (exactly analogous to the approach used for Properties), as follows:

 ©ISO/IEC 2006 – All rights reserved



Row 3 of the *EventMap* table indexes row 2 of the *TypeDef* table on the left (*MyClass*), whilst indexing row 4 of the *Event* table on the right (the row for an Event called *DocChanged*). This setup establishes that *MyClass* has an Event called *DocChanged*. But what methods in the *MethodDef* table are gathered together as 'belonging' to event *DocChanged*? That association is contained in the *MethodSemantics* table – its row 2 indexes event *DocChanged* to the right, and row 2 in the *MethodDef* table to the left (a method called add_DocChanged). Also, row 3 of the *MethodSemantics* table indexes *DocChanged* to the right, and row 3 in the *MethodDef* table to the left (a method called *remove_DocChanged*). As the shading suggests, *MyClass* has another event, called *TimedOut*, with two methods, *add_TimedOut* and *remove_TimedOut*.

Event tables do a little more than group together existing rows from other tables. The *Event* table has columns for *EventFlags*, *Name* (e.g., *DocChanged* and *TimedOut* in the example here), and *EventType*. In addition, the *MethodSemantics* table has a column to record whether the method it indexes is an *add_*, a *remove_*, a *raise_*, or *other* function.

The *Event* table has the following columns:

- *EventFlags* (a 2-byte bitmask of type *EventAttributes*, §23.1.4)
- *Name* (an index into the String heap)
- *EventType* (an index into a *TypeDef*, a *TypeRef*, or *TypeSpec* table; more precisely, a *TypeDefOrRef* (§24.2.6) coded index) (This corresponds to the Type of the Event; it is *not* the Type that owns this event.)

Note that *Event* information does not directly influence runtime behavior; what counts is the information stored for each method that the event comprises.

The *EventMap* and *Event* tables result from putting the `.event` directive on a class (§18).

This contains informative text only

1. The *Event* table can contain zero or more rows
2. Each row shall have one, and only one, owner row in the *EventMap* table [ERROR]
3. *EventFlags* shall have only those values set that are specified (all combinations valid) [ERROR]
4. *Name* shall index a non-empty string in the String heap [ERROR]
5. The *Name* string shall be a valid CLS identifier [CLS]
6. *EventType* can be null or non-null
7. If *EventType* is non-null, then it shall index a valid row in the *TypeDef* or *TypeRef* table [ERROR]

©ISO/IEC 2006 – All rights reserved

8. If *EventType* is non-null, then the row in the *TypeDef*, *TypeRef*, or *TypeSpec* table that it indexes shall be a Class (not an Interface or a ValueType) [ERROR]
9. For each row, there shall be one *add_* and one *remove_* row in the *MethodSemantics* table [ERROR]
10. For each row, there can be zero or one *raise_* row, as well as zero or more *other* rows in the *MethodSemantics* table [ERROR]
11. Within the rows owned by a given row in the *TypeDef* table, there shall be no duplicates based upon *Name* [ERROR]
12. There shall be no duplicate rows based upon *Name*, where *Name* fields are compared using CLS conflicting-identifier-rules [CLS]

End informative text

22.14 ExportedType : 0x27

The *ExportedType* table holds a row for each type, defined within *other* modules of this Assembly; that is exported out of this Assembly. In essence, it stores *TypeDef* row numbers of all types that are marked public in *other* modules that this Assembly comprises.

The actual target row in a *TypeDef* table is given by the combination of *TypeDefId* (in effect, row number) and *Implementation* (in effect, the module that holds the target *TypeDef* table). Note that this is the only occurrence in metadata of *foreign* tokens; that is, token values that have a meaning in *another* module. (A regular token value is an index into a table in the *current* module)

The full name of the type need not be stored directly. Instead, it can be split into two parts at any included "." (although typically this is done at the last "." in the full name). The part preceding the "." is stored as the *TypeNamespace* and that following the "." is stored as the *TypeName*. If there is no "." in the full name, then the *TypeNamespace* shall be the index of the empty string.

The *ExportedType* table has the following columns:

- *Flags* (a 4-byte bitmask of type *TypeAttributes*, §23.1.15)
- *TypeDefId* (a 4-byte index into a *TypeDef* table of another module in this Assembly). This column is used as a hint only. If the entry in the target *TypeDef* table matches the *TypeName* and *TypeNamespace* entries in this table, resolution has succeeded. But if there is a mismatch, the CLI shall fall back to a search of the target *TypeDef* table
- *TypeName* (an index into the String heap)
- *TypeNamespace* (an index into the String heap)
- *Implementation*. This is an index (more precisely, an *Implementation* (§24.2.6) coded index) into either of the following tables:
    - *File* table, where that entry says which module in the current assembly holds the *TypeDef*
    - *ExportedType* table, where that entry is the enclosing Type of the current nested Type

The rows in the *ExportedType* table are the result of the `.class extern` directive (§6.7).

This contains informative text only

The term "*FullName*" refers to the string created as follows: if the *TypeNamespace* is null, then use the *TypeName*, otherwise use the concatenation of *Typenamespace*, ".", and *TypeName*.

1. The *ExportedType* table can contain zero or more rows
2. There shall be no entries in the *ExportedType* table for Types that are defined in the current module—just for Types defined in other modules within the Assembly [ERROR]

 ©ISO/IEC 2006 – All rights reserved

3. *Flags* shall have only those values set that are specified [ERROR]
4. If *Implementation* indexes the *File* table, then *Flags.VisibilityMask* shall be `public` (§23.1.15) [ERROR]
5. If *Implementation* indexes the *ExportedType* table, then *Flags.VisibilityMask* shall be `NestedPublic` (§23.1.15) [ERROR]
6. If non-null, *TypeDefId* should index a valid row in a *TypeDef* table in a module somewhere within this Assembly (but not this module), and the row so indexed should have its *Flags.Public* = 1 (§23.1.15) [WARNING]
7. *TypeName* shall index a non-empty string in the String heap [ERROR]
8. *TypeNamespace* can be null, or non-null
9. If *TypeNamespace* is non-null, then it shall index a non-empty string in the String heap [ERROR]
10. *FullName* shall be a valid CLS identifier [CLS]
11. If this is a nested Type, then *TypeNamespace* should be null, and *TypeName* should represent the unmangled, simple name of the nested Type [ERROR]
12. *Implementation* shall be a valid index into either of the following: [ERROR]
    - the *File* table; that file shall hold a definition of the target Type in its *TypeDef* table
    - a different row in the current *ExportedType* table—this identifies the enclosing Type of the current, nested Type
13. *FullName* shall match exactly the corresponding *FullName* for the row in the *TypeDef* table indexed by *TypeDefId* [ERROR]
14. Ignoring nested Types, there shall be no duplicate rows, based upon *FullName* [ERROR]
15. For nested Types, there shall be no duplicate rows, based upon *TypeName* and enclosing Type [ERROR]
16. The complete list of Types exported from the current Assembly is given as the catenation of the *ExportedType* table with all public Types in the current *TypeDef* table, where "public" means a *Flags.VisibilityMask* of either *Public* or *NestedPublic*. There shall be no duplicate rows, in this concatenated table, based upon *FullName* (add Enclosing Type into the duplicates check if this is a nested Type) [ERROR]

End informative text

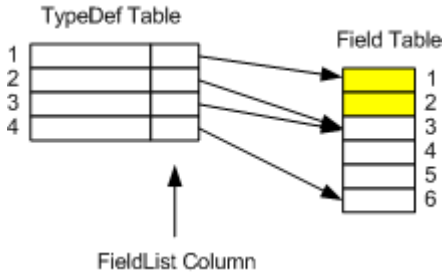
22.15 Field : 0x04

The *Field* table has the following columns:

- *Flags* (a 2-byte bitmask of type *FieldAttributes*, §23.1.5)
- *Name* (an index into the String heap)
- *Signature* (an index into the Blob heap)

Conceptually, each row in the *Field* table is owned by one, and only one, row in the *TypeDef* table. However, the owner of any row in the *Field* table is not stored anywhere in the *Field* table itself. There is merely a 'forward-pointer' from each row in the *TypeDef* table (the *FieldList* column), as shown in the following illustration.

©ISO/IEC 2006 – All rights reserved



The *TypeDef* table has rows 1–4. The first row in the *TypeDef* table corresponds to a pseudo type, inserted automatically by the CLI. It is used to denote those rows in the *Field* table corresponding to global variables. The *Field* table has rows 1–6. Type 1 (pseudo type for 'module') owns rows 1 and 2 in the *Field* table. Type 2 owns no rows in the *Field* table, even though its *FieldList* indexes row 3 in the *Field* table. Type 3 owns rows 3–5 in the *Field* table. Type 4 owns row 6 in the *Field* table. So, in the *Field* table, rows 1 and 2 belong to Type 1 (global variables); rows 3–5 belong to Type 3; row 6 belongs to Type 4.

Each row in the *Field* table results from a top-level `.field` directive (§5.10), or a `.field` directive inside a Type (§10.2). (For an example, see §14.5.)

This contains informative text only

1. The *Field* table can contain zero or more rows
2. Each row shall have one, and only one, owner row in the *TypeDef* table [ERROR]
3. The owner row in the *TypeDef* table shall not be an Interface [CLS]
4. *Flags* shall have only those values set that are specified [ERROR]
5. The *FieldAccessMask* subfield of Flags shall contain precisely one of `CompilerControlled`, `Private`, `FamANDAssem`, `Assembly`, `Family`, `FamORAssem`, or `Public` (§23.1.5) [ERROR]
6. *Flags* can set either or neither of `Literal` or `InitOnly`, but not both (§23.1.5) [ERROR]
7. If *Flags.Literal* = 1 then *Flags.Static* shall also be 1 (§23.1.5) [ERROR]
8. If *Flags.RTSpecialName* = 1, then *Flags.SpecialName* shall also be 1 (§23.1.5) [ERROR]
9. If *Flags.HasFieldMarshal* = 1, then this row shall 'own' exactly one row in the *FieldMarshal* table (§23.1.5) [ERROR]
10. If *Flags.HasDefault* = 1, then this row shall 'own' exactly one row in the *Constant* table (§23.1.5) [ERROR]
11. If *Flags.HasFieldRVA* = 1, then this row shall 'own' exactly one row in the *Field's RVA* table (§23.1.5) [ERROR]
12. *Name* shall index a non-empty string in the String heap [ERROR]
13. The *Name* string shall be a valid CLS identifier [CLS]
14. *Signature* shall index a valid field signature in the Blob heap [ERROR]
15. If *Flags.CompilerControlled* = 1 (§23.1.5), then this row is ignored completely in duplicate checking.
16. If the owner of this field is the internally-generated type called `<Module>`, it denotes that this field is defined at module scope (commonly called a global variable). In this case:
    - *Flags.Static* shall be 1 [ERROR]
    - *Flags.MemberAccessMask* subfield shall be one of `Public`, `CompilerControlled`, or `Private` (§23.1.5) [ERROR]
    - module-scope fields are not allowed [CLS]

 ©ISO/IEC 2006 – All rights reserved

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 `ELEMENT_TYPE_U1`, `ELEMENT_TYPE_I2`, `ELEMENT_TYPE_I4`, or `ELEMENT_TYPE_I8` (§23.1.16): [CLS]
20. its *Signature* shall be an integral type. [ERROR]

End informative text

22.16 FieldLayout : 0x10

The *FieldLayout* table has the following columns:

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

Note that each Field in any Type is defined by its Signature. When a Type instance (i.e., an object) is laid out by the CLI, each Field is one of four kinds:

- Scalar: for any member of built-in type, such as `int32`. The size of the field is given by the size of that intrinsic, which varies between 1 and 8 bytes
- ObjectRef: for `ELEMENT_TYPE_CLASS`, `ELEMENT_TYPE_STRING`, `ELEMENT_TYPE_OBJECT`, `ELEMENT_TYPE_ARRAY`, `ELEMENT_TYPE_SZARRAY`
- Pointer: for `ELEMENT_TYPE_PTR`, `ELEMENT_TYPE_FNPTR`
- ValueType: for `ELEMENT_TYPE_VALUETYPE`. The instance of that ValueType is actually laid out in this object, so the size of the field is the size of that ValueType

Note that metadata specifying explicit structure layout can be valid for use on one platform but not on another, since some of the rules specified here are dependent on platform-specific alignment rules.

A row in the *FieldLayout* table is created if the `.field` directive for the parent field has specified a field offset (§16).

This contains informative text only

1. A *FieldLayout* table can contain zero or more or rows
2. The Type whose Fields are described by each row of the *FieldLayout* table shall have *Flags.ExplicitLayout* (§23.1.15) set [ERROR]
3. *Offset* shall be zero or more [ERROR]
4. *Field* shall index a valid row in the *Field* table [ERROR]
5. *Flags.Static* for the row in the *Field* table indexed by *Field* shall be non-static (i.e., zero 0) [ERROR]

©ISO/IEC 2006 – All rights reserved

6. Among the rows owned by a given Type there shall be no duplicates, based upon *Field*. That is, a given Field of a Type cannot be given two offsets. [ERROR]
7. Each Field of kind *ObjectRef* shall be naturally aligned within the Type [ERROR]
8. Among the rows owned by a given Type it is perfectly valid for several rows to have the same value of *Offset*. ObjectRef and a valuetype cannot have the same offset [ERROR]
9. Every Field of an ExplicitLayout Type shall be given an offset; that is, it shall have a row in the *FieldLayout* table [ERROR]

End informative text

22.17 FieldMarshal : 0x0D

The *FieldMarshal* table has two columns. It 'links' an existing row in the *Field* or *Param* table, to information in the Blob heap that defines how that field or parameter (which, as usual, covers the method return, as parameter number 0) shall be marshalled when calling to or from unmanaged code via PInvoke dispatch.

Note that *FieldMarshal* information is used only by code paths that arbitrate operation with unmanaged code. In order to execute such paths, the caller, on most platforms, would be installed with elevated security permission. Once it invokes unmanaged code, it lies outside the regime that the CLI can check—it is simply trusted not to violate the type system.

The *FieldMarshal* table has the following columns:

- *Parent* (an index into *Field* or *Param* table; more precisely, a *HasFieldMarshal* (§24.2.6) coded index)
- *NativeType* (an index into the Blob heap)

For the detailed format of the 'blob', see §23.4

A row in the *FieldMarshal* table is created if the `.field` directive for the parent field has specified a `marshal` attribute (§16.1).

This contains informative text only

1. A *FieldMarshal* table can contain zero or more rows
2. *Parent* shall index a valid row in the Field or Param table (*Parent* values are encoded to say which of these two tables each refers to) [ERROR]
3. *NativeType* shall index a non-null 'blob' in the Blob heap [ERROR]
4. No two rows shall point to the same parent. In other words, after the *Parent* values have been decoded to determine whether they refer to the *Field* or the *Param* table, no two rows can point to the same row in the Field table or in the Param table [ERROR]
5. The following checks apply to the *MarshalSpec* 'blob' (§23.4):
   a. *NativeIntrinsic* shall be exactly one of the constant values in its production (§23.4) [ERROR]
   b. If `ARRAY`, then *ArrayElemType* shall be exactly one of the constant values in its production [ERROR]
   c. If `ARRAY`, then *ParamNum* can be zero
   d. If `ARRAY`, then *ParamNum* cannot be < 0 [ERROR]
   e. If `ARRAY`, and *ParamNum* > 0, then *Parent* shall point to a row in the Param table, not in the Field table [ERROR]

 ©ISO/IEC 2006 – All rights reserved

f. If ARRAY, and *ParamNum* > 0, then *ParamNum* cannot exceed the number of parameters supplied to the *MethodDef* (or *MethodRef* if a VARARG call) of which the parent Param is a member [ERROR]

g. If ARRAY, then *ElemMult* shall be >= 1 [ERROR]

h. If ARRAY and *ElemMult* != 1 issue a warning, because it is probably a mistake [WARNING]

i. If ARRAY and *ParamNum* = 0, then *NumElem* shall be >= 1 [ERROR]

j. If ARRAY and *ParamNum* != 0 and NumElem != 0 then issue a warning, because it is probably a mistake [WARNING]

End informative text

22.18 FieldRVA : 0x1D

The *FieldRVA* table has the following columns:

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

Conceptually, each row in the *FieldRVA* table is an extension to exactly one row in the *Field table*, and records the RVA (Relative Virtual Address) within the image file at which this field's initial value is stored.

A row in the *FieldRVA* table is created for each static parent field that has specified the optional `data` label §16). The RVA column is the relative virtual address of the data in the PE file (§16.3).

This contains informative text only

1. *RVA* shall be non-zero [ERROR]
2. *RVA* shall point into the current module's data area (not its metadata area) [ERROR]
3. *Field* shall index a valid row in the *Field* table [ERROR]
4. Any field with an RVA shall be a ValueType (not a Class or an Interface). Moreover, it shall not have any private fields (and likewise for any of its fields that are themselves ValueTypes). (If any of these conditions were breached, code could overlay that global static and access its private fields.) Moreover, no fields of that ValueType can be Object References (into the GC heap) [ERROR]
5. So long as two RVA-based fields comply with the previous conditions, the ranges of memory spanned by the two ValueTypes can overlap, with no further constraints. This is not actually an additional rule; it simply clarifies the position with regard to overlapped RVA-based fields

End informative text

22.19 File : 0x26

The *File* table has the following columns:

- *Flags* (a 4-byte bitmask of type FileAttributes, §23.1.6)
- *Name* (an index into the String heap)
- *HashValue* (an index into the Blob heap)

The rows of the *File* table result from `.file` directives in an Assembly (§6.2.3)

This contains informative text only

1. *Flags* shall have only those values set that are specified (all combinations valid) [ERROR]

2. *Name* shall index a non-empty string in the String heap. It shall be in the format <filename>.<extension> (e.g., "foo.dll", but *not* "c:\utils\foo.dll") [ERROR]
3. *HashValue* shall index a non-empty 'blob' in the Blob heap [ERROR]
4. There shall be no duplicate rows; that is, rows with the same *Name* value [ERROR]
5. If this module contains a row in the *Assembly* table (that is, if this module "holds the manifest") then there shall not be any row in the *File* table for this module; i.e., no self-reference [ERROR]
6. If the *File* table is empty, then this, by definition, is a single-file assembly. In this case, the *ExportedType* table should be empty [WARNING]

End informative text

22.20 GenericParam : 0x2A

The *GenericParam* table has the following columns:

- *Number* (the 2-byte index of the generic parameter, numbered left-to-right, from zero)
- *Flags* (a 2-byte bitmask of type *GenericParamAttributes*, §23.1.7)
- *Owner* (an index into the *TypeDef* or *MethodDef* table, specifying the Type or Method to which this generic parameter applies; more precisely, a *TypeOrMethodDef* (§24.2.6) coded index)
- *Name* (a non-null index into the String heap, giving the name for the generic parameter. This is purely descriptive and is used only by source language compilers and by Reflection)

The *GenericParam* table stores the generic parameters used in generic type definitions and generic method definitions. These generic parameters can be constrained (i.e., generic arguments shall extend some class and/or implement certain interfaces) or unconstrained. (Such constraints are stored in the *GenericParamConstraint* table.)

Conceptually, each row in the *GenericParam* table is owned by one, and only one, row in either the *TypeDef* or *MethodDef* tables.

[*Example*:

```
.class Dict`2<([mscorlib]System.IComparable) K, V>
```

The generic parameter *K* of class *Dict* is constrained to implement `System.IComparable`.

```
.method static void ReverseArray<T>(!!0[] 'array')
```

There is no constraint on the generic parameter *T* of the generic method `ReverseArray`.

end example]

This contains informative text only

1. *GenericParam* table can contain zero or more rows
2. Each row shall have one, and only one, owner row in the *TypeDef* or *MethodDef* table (i.e., no row sharing) [ERROR]
3. Every generic type shall own one row in the *GenericParam* table for each of its generic parameters [ERROR]
4. Every generic method shall own one row in the *GenericParam* table for each of its generic parameters [ERROR]

Flags:

- Can hold the value *Covariant* or *Contravariant*, but only if the owner row corresponds to a generic interface, or a generic delegate class. [ERROR]

 ©ISO/IEC 2006 – All rights reserved

- Otherwise, shall hold the value *NonVariant* (i.e., where the owner is a non delegate class, a value-type, or a generic method) [ERROR]

If *Flags* == Covariant then the corresponding generic parameter can appear in a type definition only as [ERROR]:

- The result type of a method
- A generic parameter to an inherited interface

If *Flags* == Contravariant then the corresponding generic parameter can appear in a type definition only as the argument to a method [ERROR]

Number shall have a value >= 0 and < the number of generic parameters in owner type or method. [ERROR]

Successive rows of the *GenericParam* table that are owned by the same method shall be ordered by increasing *Number* value; there shall be no gaps in the *Number* sequence [ERROR]

Name shall be non-null and index a string in the String heap [ERROR]

[*Rationale*: Otherwise, Reflection output is not fully usable. *end rationale*]

There shall be no duplicate rows based upon *Owner+Name* [ERROR] [*Rationale*: Otherwise, code using Reflection cannot disambiguate the different generic parameters. *end rationale*]

There shall be no duplicate rows based upon *Owner+Number* [ERROR]

End informative text

22.21 GenericParamConstraint : 0x2C

The *GenericParamConstraint* table has the following columns:

- *Owner* (an index into the *GenericParam* table, specifying to which generic parameter this row refers)
- *Constraint* (an index into the *TypeDef*, *TypeRef*, or *TypeSpec* tables, specifying from which class this generic parameter is constrained to derive; or which interface this generic parameter is constrained to implement; more precisely, a *TypeDefOrRef* (§24.2.6) coded index)

The *GenericParamConstraint* table records the constraints for each generic parameter. Each generic parameter can be constrained to derive from zero or one class. Each generic parameter can be constrained to implement zero or more interfaces.

Conceptually, each row in the *GenericParamConstraint* table is 'owned' by a row in the *GenericParam* table.

All rows in the *GenericParamConstraint* table for a given *Owner* shall refer to distinct constraints.

Note that if *Constraint* is a *TypeRef* to `System.ValueType`, then it means the constraint type shall be `System.ValueType`, or one of its sub types. However, since `System.ValueType` itself is a reference type, this particular mechanism does not guarantee that the type is a non-reference type.

This contains informative text only

1. The *GenericParamConstraint* table can contain zero or more rows
2. Each row shall have one, and only one, owner row in the *GenericParam* table (i.e., no row sharing) [ERROR]
3. Each row in the *GenericParam* table shall 'own' a separate row in the *GenericParamConstraint* table for each constraint that generic parameter has [ERROR]
4. All of the rows in the *GenericParamConstraint* table that are owned by a given row in the *GenericParam* table shall form a contiguous range (of rows) [ERROR]

©ISO/IEC 2006 – All rights reserved

5. Any generic parameter (corresponding to a row in the *GenericParam table*) shall own zero or one row in the *GenericParamConstraint table* corresponding to a class constraint. [ERROR]
6. Any generic parameter (corresponding to a row in the *GenericParam table*) shall own zero or more rows in the *GenericParamConstraint* table corresponding to an interface constraint. [ERROR]
7. There shall be no duplicate rows based upon *Owner+Constraint* [ERROR]
8. *Constraint* shall not reference `System.Void`. [ERROR]

End informative text

22.22 ImplMap : 0x1C

The *ImplMap* table holds information about unmanaged methods that can be reached from managed code, using PInvoke dispatch.

Each row of the *ImplMap* table associates a row in the *MethodDef* table (*MemberForwarded*) with the name of a routine (*ImportName*) in some unmanaged DLL (*ImportScope*).

[*Note:* A typical example would be: associate the managed Method stored in row N of the *Method* table (so *MemberForwarded* would have the value N) with the routine called "GetEnvironmentVariable" (the string indexed by *ImportName)* in the DLL called "kernel32" (the string in the *ModuleRef* table indexed by *ImportScope*). The CLI intercepts calls to managed Method number N, and instead forwards them as calls to the unmanaged routine called "GetEnvironmentVariable" in "kernel32.dll" (including marshalling any arguments, as required)

The CLI does not support this mechanism to access *fields* that are exported from a DLL, only methods. *end note*]

The *ImplMap* table has the following columns:

- *MappingFlags* (a 2-byte bitmask of type *PInvokeAttributes*, §23.1.7)
- *MemberForwarded* (an index into the *Field* or *MethodDef* table; more precisely, a *MemberForwarded* (§24.2.6) coded index). However, it only ever indexes the *MethodDef* table, since *Field* export is not supported.
- *ImportName* (an index into the String heap)
- *ImportScope* (an index into the *ModuleRef* table)

A row is entered in the *ImplMap* table for each parent Method (§15.5) that is defined with a `.pinvokeimpl` interoperation attribute specifying the *MappingFlags*, *ImportName*, and *ImportScope*.

This contains informative text only

1. *ImplMap* can contain zero or more rows
2. *MappingFlags* shall have only those values set that are specified [ERROR]
3. *MemberForwarded* shall index a valid row in the *MethodDef* table [ERROR]
4. The *MappingFlags.CharSetMask* (§23.1.7) in the row of the *MethodDef* table indexed by *MemberForwarded* shall have at most one of the following bits set: `CharSetAnsi`, `CharSetUnicode`, or `CharSetAuto` (if none is set, the default is `CharSetNotSpec`) [ERROR]
5. *ImportName* shall index a non-empty string in the String heap [ERROR]
6. *ImportScope* shall index a valid row in the *ModuleRef* table [ERROR]
7. The row indexed in the *MethodDef* table by *MemberForwarded* shall have its *Flags.PinvokeImpl* = 1, and *Flags.Static* = 1 [ERROR]

 ©ISO/IEC 2006 – All rights reserved

End informative text

22.23 InterfaceImpl : 0x09

The *InterfaceImpl* table has the following columns:

- *Class* (an index into the *TypeDef* table)
- *Interface* (an index into the *TypeDef*, *TypeRef*, or *TypeSpec* table; more precisely, a *TypeDefOrRef* (§24.2.6) coded index)

The *InterfaceImpl* table records the interfaces a type implements explicitly. Conceptually, each row in the *InterfaceImpl* table indicates that *Class* implements *Interface.*

This contains informative text only

1. The *InterfaceImpl* table can contain zero or more rows
2. *Class* shall be non-null [ERROR]
3. If *Class* is non-null, then:
   a. *Class* shall index a valid row in the *TypeDef* table [ERROR]
   b. *Interface* shall index a valid row in the *TypeDef* or *TypeRef* table [ERROR]
   c. The row in the *TypeDef, TypeRef*, or *TypeSpec* table indexed by *Interface* shall be an interface (*Flags.Interface* = 1), not a Class or ValueType [ERROR]
4. There should be no duplicates in the *InterfaceImpl* table, based upon non-null *Class* and *Interface* values [WARNING]
5. There can be many rows with the same value for *Class* (since a class can implement many interfaces)
6. There can be many rows with the same value for *Interface* (since many classes can implement the same interface)

End informative text

22.24 ManifestResource : 0x28

The *ManifestResource* table has the following columns:

- *Offset* (a 4-byte constant)
- *Flags* (a 4-byte bitmask of type *ManifestResourceAttributes*, §23.1.9)
- *Name* (an index into the String heap)
- *Implementation* (an index into a *File* table, a *AssemblyRef* table, or null; more precisely, an *Implementation* (§24.2.6) coded index)

The *Offset* specifies the byte offset within the referenced file at which this resource record begins. The *Implementation* specifies which file holds this resource. The rows in the table result from `.mresource` directives on the Assembly (§6.2.2).

This contains informative text only

1. The *ManifestResource* table can contain zero or more rows
2. *Offset* shall be a valid offset into the target file, starting from the Resource entry in the CLI header [ERROR]
3. *Flags* shall have only those values set that are specified [ERROR]

4. The *VisibilityMask* (§23.1.9) subfield of *Flags* shall be one of `Public` or `Private` [ERROR]
5. *Name* shall index a non-empty string in the String heap [ERROR]
6. *Implementation* can be null or non-null (if null, it means the resource is stored in the current file)
7. If *Implementation* is null, then *Offset* shall be a valid offset in the current file, starting from the Resource entry in the CLI header [ERROR]
8. If *Implementation* is non-null, then it shall index a valid row in the *File* or *AssemblyRef* table [ERROR]
9. There shall be no duplicate rows, based upon *Name* [ERROR]
10. If the resource is an index into the *File* table, *Offset* shall be zero [ERROR]

End informative text

22.25 MemberRef : 0x0A

The *MemberRef* table combines two sorts of references, to Methods and to Fields of a class, known as 'MethodRef' and 'FieldRef', respectively. The *MemberRef* table has the following columns:

- *Class* (an index into the *MethodDef*, *ModuleRef*,*TypeDef*, *TypeRef*, or *TypeSpec* tables; more precisely, a MemberRefParent (§24.2.6) coded index)
- *Name* (an index into the String heap)
- *Signature* (an index into the Blob heap)

An entry is made into the *MemberRef* table whenever a reference is made in the CIL code to a method or field which is defined in another module or assembly. (Also, an entry is made for a call to a method with a `VARARG` signature, even when it is defined in the same module as the call site.)

This contains informative text only

1. *Class* shall be one of the following: [ERROR]
   a. a *TypeRef* token, if the class that defines the member is defined in another module. (Note that it is unusual, but valid, to use a *TypeRef* token when the member is defined in this same module, in which case, its *TypeDef* token can be used instead.)
   b. a *ModuleRef* token, if the member is defined, in another module of the same assembly, as a global function or variable.
   c. a *MethodDef* token, when used to supply a call-site signature for a vararg method that is defined in this module. The *Name* shall match the *Name* in the corresponding *MethodDef* row. The *Signature* shall match the *Signature* in the target method definition [ERROR]
   d. a *TypeSpec* token, if the member is a member of a generic type
2. *Class* shall not be null (as this would indicate an unresolved reference to a global function or variable) [ERROR]
3. *Name* shall index a non-empty string in the String heap [ERROR]
4. The *Name* string shall be a valid CLS identifier [CLS]
5. *Signature* shall index a valid field or method signature in the Blob heap. In particular, it shall embed exactly one of the following 'calling conventions': [ERROR]
   a. `DEFAULT` (0x0)
   b. `VARARG` (0x5)
   c. `FIELD` (0x6)
   d. `GENERIC (0x10)`

 ©ISO/IEC 2006 – All rights reserved

6. The *MemberRef* table shall contain no duplicates, where duplicate rows have the same *Class, Name*, and *Signature* [WARNING]
7. *Signature* shall not have the `VARARG` (0x5) calling convention [CLS]
8. There shall be no duplicate rows, where *Name* fields are compared using CLS conflicting-identifier-rules. (In particular, note that the return type and whether parameters are marked `ELEMENT_TYPE_BYREF` (§23.1.16) are ignored in the CLS. For example, `.method int32 M()` and `.method float64 M()` result in duplicate rows by CLS rules. Similarly, `.method void N(int32 i)` and `.method void N(int32& i)` also result in duplicate rows by CLS rules.) [CLS]
9. If *Class* and *Name* resolve to a field, then that field shall not have a value of `CompilerControlled` (§23.1.5) in its *Flags.FieldAccessMask* subfield [ERROR]
10. If *Class* and *Name* resolve to a method, then that method shall not have a value of `CompilerControlled` in its *Flags.MemberAccessMask* (§23.1.10) subfield [ERROR]
11. The type containing the definition of a *MemberRef* shall be a *TypeSpec* representing an instantiated type.

End informative text

22.26 MethodDef : 0x06

The *MethodDef* table has the following columns:

- *RVA* (a 4-byte constant)
- *ImplFlags* (a 2-byte bitmask of type *MethodImplAttributes*, §23.1.10)
- *Flags* (a 2-byte bitmask of type *MethodAttributes*, §23.1.10)
- *Name* (an index into the String heap)
- *Signature* (an index into the Blob heap)
- *ParamList* (an index into the *Param* table). It marks the first of a contiguous run of Parameters owned by this method. The run continues to the smaller of:
  - the last row of the *Param* table
  - the next run of Parameters, found by inspecting the *ParamList* of the next row in the *MethodDef* table

Conceptually, every row in the *MethodDef* table is owned by one, and only one, row in the *TypeDef* table.

The rows in the *MethodDef* table result from `.method` directives (§15). The RVA column is computed when the image for the PE file is emitted and points to the `COR_ILMETHOD` structure for the body of the method (§25.4)

[*Note*: If *Signature* is `GENERIC` (0x10), the generic arguments are described in the *GenericParam* table (§22.20). *end note*]

This contains informative text only

1. The *MethodDef* table can contain zero or more rows
2. Each row shall have one, and only one, owner row in the *TypeDef* table [ERROR]
3. *ImplFlags* shall have only those values set that are specified [ERROR]
4. *Flags* shall have only those values set that are specified [ERROR]
5. If Name is .ctor and the method is marked `SpecialName`, there shall not be a row in the *GenericParam* table which has this *MethodDef* as its owner. [ERROR]

©ISO/IEC 2006 – All rights reserved

6. The *MemberAccessMask* (§23.1.10) subfield of *Flags* shall contain precisely one of `CompilerControlled`, `Private`, `FamANDAssem`, `Assem`, `Family`, `FamORAssem`, or `Public` [ERROR]
7. The following combined bit settings in *Flags* are invalid [ERROR]
   a. `Static | Final`
   b. `Static | Virtual`
   c. `Static | NewSlot`
   d. `Final | Abstract`
   e. `Abstract | PinvokeImpl`
   f. `CompilerControlled | SpecialName`
   g. `CompilerControlled | RTSpecialName`
8. An abstract method shall be virtual. So, if *Flags.Abstract* = 1 then *Flags.Virtual* shall also be 1 [ERROR]
9. If *Flags.RTSpecialName* = 1 then *Flags.SpecialName* shall also be 1 [ERROR]
10. If *Flags.HasSecurity* = 1, then at least one of the following conditions shall be true: [ERROR]
    - this Method owns at least row in the *DeclSecurity* table
    - this Method has a custom attribute called *SuppressUnmanagedCodeSecurityAttribute*
11. If this Method owns one (or more) rows in the *DeclSecurity* table then *Flags.HasSecurity* shall be 1 [ERROR]
12. If this Method has a custom attribute called *SuppressUnmanagedCodeSecurityAttribute* then *Flags.HasSecurity* shall be 1 [ERROR]
13. A Method can have a custom attribute called *DynamicSecurityMethodAttribute*, but this has no effect whatsoever upon the value of its *Flags.HasSecurity*
14. *Name* shall index a non-empty string in the String heap [ERROR]
15. Interfaces cannot have instance constructors. So, if this Method is owned by an Interface, then its *Name* cannot be `.ctor` [ERROR]
16. 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]
17. The *Name* string shall be a valid CLS identifier (unless *Flags.RTSpecialName* is set - for example, `.cctor` is valid) [CLS]
18. *Signature* shall index a valid method signature in the Blob heap [ERROR]
19. If *Flags.CompilerControlled* = 1, then this row is ignored completely in duplicate checking
20. If the owner of this method is the internally-generated type called `<Module>`, it denotes that this method is defined at module scope. [*Note*: In C++, the method is called *global* and can be referenced only within its compiland, from its point of declaration forwards. *end note*] In this case:
    a. *Flags.Static* shall be 1 [ERROR]
    b. *Flags.Abstract* shall be 0 [ERROR]
    c. *Flags.Virtual* shall be 0 [ERROR]
    d. *Flags.MemberAccessMask* subfield shall be one of `CompilerControlled`, `Public`, or `Private` [ERROR]
    e. module-scope methods are not allowed [CLS]

 ©ISO/IEC 2006 – All rights reserved

21. It makes no sense for ValueTypes, which have no *identity*, to have synchronized methods (unless they are boxed). So, if the owner of this method is a ValueType then the method cannot be synchronized. That is, *ImplFlags.Synchronized* shall be 0 [ERROR]
22. There shall be no duplicate rows in the *MethodDef* table, based upon owner + *Name* + *Signature* (where owner is the owning row in the *TypeDef* table). (Note that the *Signature* encodes whether or not the method is generic, and for generic methods, it encodes the number of generic parameters.) (Note, however, that if *Flags.CompilerControlled* = 1, then this row is excluded from duplicate checking) [ERROR]
23. There shall be no duplicate rows in the *MethodDef* table, based upon owner + *Name* + *Signature*, where *Name* fields are compared using CLS conflicting-identifier-rules; also, the Type defined in the signatures shall be different. So, for example, `"int i"` and `"float i"` would be considered CLS duplicates; also, the return type of the method is ignored (Note, however, that if *Flags.CompilerControlled* = 1, this row is excluded from duplicate checking as explained above.) [CLS]
24. If `Final`, `NewSlot`, or `Strict` are set in *Flags*, then *Flags.Virtual* shall also be set [ERROR]
25. If *Flags.PInvokeImpl* is set, then *Flags.Virtual* shall be 0 [ERROR]
26. If *Flags.Abstract != 1* then exactly one of the following shall also be true: [ERROR]
    - RVA != 0
    - Flags.PInvokeImpl = 1
    - ImplFlags.Runtime = 1
27. If the method is `CompilerControlled`, then the RVA shall be non-zero or marked with `PinvokeImpl` = 1 [ERROR]
28. *Signature* shall have exactly one of the following managed calling conventions [ERROR]
    a. `DEFAULT` (0x0)
    b. `VARARG` (0x5)
    c. `GENERIC` (0x10)
29. *Signature* shall have the calling convention `DEFAULT` (0x0) or `GENERIC` (0x10). [CLS]
30. *Signature:* If and only if the method is not `Static` then the calling convention byte in *Signature* has its `HASTHIS` (0x20) bit set [ERROR]
31. *Signature:* If the method is `static`, then the `HASTHIS` (0x20) bit in the calling convention shall be 0 [ERROR]
32. If `EXPLICITTHIS` (0x40) in the signature is set, then `HASTHIS` (0x20) shall also be set (note that if `EXPLICITTHIS` is set, then the code is not verifiable) [ERROR]
33. The `EXPLICITTHIS` (0x40) bit can be set only in signatures for function pointers: signatures whose MethodDefSig is preceded by `FNPTR` (0x1B) [ERROR]
34. If *RVA* = 0, then either: [ERROR]
    - Flags.Abstract = 1, or
    - ImplFlags.Runtime = 1, or
    - Flags.PinvokeImpl = 1, or
35. If *RVA* != 0, then: [ERROR]
    a. *Flags.Abstract* shall be 0, and
    b. *ImplFlags.CodeTypeMask* shall have exactly one of the following values: `Native`, `CIL`, or `Runtime`, and

c. *RVA* shall point into the CIL code stream in this file

36. If *Flags.PinvokeImpl* = 1 then [ERROR]
    - *RVA* = 0 *and* the method owns a row in the *ImplMap* table
37. If *Flags.RTSpecialName* = 1 then *Name* shall be one of: [ERROR]
    a. `.ctor` (an object constructor method)
    b. `.cctor` (a class constructor method)
38. Conversely, if *Name* is any of the above special names then *Flags.RTSpecialName* shall be set [ERROR]
39. If *Name* = `.ctor` (an object constructor method) then:
    a. return type in *Signature* shall be `ELEMENT_TYPE_VOID` (§23.1.16) [ERROR]
    b. *Flags.Static* shall be 0 [ERROR]
    c. *Flags.Abstract* shall be 0 [ERROR]
    d. *Flags.Virtual* shall be 0 [ERROR]
    e. 'Owner' type shall be a valid Class or ValueType (not `<Module>` and not an Interface) in the TypeDef table [ERROR]
    f. there can be zero or more `.ctors` for any given 'owner'
40. If *Name* = `.cctor` (a class constructor method) then:
    a. the return type in *Signature* shall be `ELEMENT_TYPE_VOID` (§23.1.16) [ERROR]
    b. *Signature* shall have `DEFAULT` (0x0) for its calling convention [ERROR]
    c. there shall be no parameters supplied in *Signature* [ERROR]
    d. *Flags.Static* shall be set [ERROR]
    e. *Flags.Virtual* shall be clear [ERROR]
    f. *Flags.Abstract* shall be clear [ERROR]
41. Among the set of methods owned by any given row in the *TypeDef* table there can only be 0 or 1 methods named `.cctor` [ERROR]

End informative text

22.27 MethodImpl : 0x19

MethodImpl tables let a compiler override the default inheritance rules provided by the CLI. Their original use was to allow a class `C`, that inherited method `M` from both interfaces `I` and `J`, to provide implementations for both methods (rather than have only one slot for `M` in its vtable). However, *MethodImpl*s can be used for other reasons too, limited only by the compiler writer's ingenuity within the constraints defined in the Validation rules below.

In the example above, *Class* specifies `C`, *MethodDeclaration* specifies `I::M`, *MethodBody* specifies the method which provides the implementation for `I::M` (either a method body within `C`, or a method body implemented by a base class of `C`).

The *MethodImpl* table has the following columns:

- *Class* (an index into the *TypeDef* table)
- *MethodBody* (an index into the *MethodDef* or *MemberRef* table; more precisely, a *MethodDefOrRef* (§24.2.6) coded index)

 ©ISO/IEC 2006 – All rights reserved

- *MethodDeclaration* (an index into the *MethodDef* or *MemberRef* table; more precisely, a *MethodDefOrRef* (§24.2.6) coded index)

ILAsm uses the `.override` directive to specify the rows of the *MethodImpl* table (§10.3.2 and §15.4.1).

This contains informative text only

1. The *MethodImpl* table can contain zero or more rows
2. *Class* shall index a valid row in the *TypeDef* table [ERROR]
3. *MethodBody* shall index a valid row in the *MethodDef* or *MemberRef* table [ERROR]
4. The method indexed by *MethodDeclaration* shall have *Flags.Virtual* set [ERROR]
5. The owner Type of the method indexed by *MethodDeclaration* shall not have *Flags.Sealed* = 0 [ERROR]
6. The method indexed by *MethodBody* shall be a member of *Class* or some base class of *Class* (*MethodImpl*s do not allow compilers to 'hook' arbitrary method bodies) [ERROR]
7. The method indexed by *MethodBody* shall be virtual [ERROR]
8. The method indexed by *MethodBody* shall have its *Method.RVA* != 0 (cannot be an unmanaged method reached via PInvoke, for example) [ERROR]
9. *MethodDeclaration* shall index a method in the ancestor chain of *Class* (reached via its *Extends* chain) or in the interface tree of *Class* (reached via its *InterfaceImpl* entries) [ERROR]
10. The method indexed by *MethodDeclaration* shall not be final (its *Flags.Final* shall be 0) [ERROR]
11. If MethodDeclaration has the Strict flag set, the method indexed by *MethodDeclaration* shall be accessible to *Class*. [ERROR]
12. The method signature defined by *MethodBody* shall match those defined by *MethodDeclaration* [ERROR]
13. There shall be no duplicate rows, based upon *Class+MethodDeclaration* [ERROR]

End informative text

22.28 MethodSemantics : 0x18

The *MethodSemantics* table has the following columns:

- *Semantics* (a 2-byte bitmask of type *MethodSemanticsAttributes*, §23.1.12)
- *Method* (an index into the *MethodDef* table)
- *Association* (an index into the *Event* or *Property* table; more precisely, a *HasSemantics* (§24.2.6) coded index)

The rows of the *MethodSemantics* table are filled by `.property` (§17) and `.event` directives (§18). (See §22.13 for more information.)

This contains informative text only

1. *MethodSemantics* table can contain zero or more rows
2. *Semantics* shall have only those values set that are specified [ERROR]
3. *Method* shall index a valid row in the *MethodDef* table, and that row shall be for a method defined on the same class as the Property or Event this row describes [ERROR]
4. All methods for a given Property or Event shall have the same accessibility (ie the *MemberAccessMask* subfield of their *Flags* row) and cannot be `CompilerControlled` [CLS]

5. *Semantics:* constrained as follows:
    - o If this row is for a Property, then exactly one of `Setter`, `Getter`, or `Other` shall be set [ERROR]
    - o If this row is for an Event, then exactly one of `AddOn`, `RemoveOn`, `Fire`, or `Other` shall be set [ERROR]
6. If this row is for an Event, and its *Semantics* is `Addon` or `RemoveOn`, then the row in the *MethodDef* table indexed by *Method* shall take a Delegate as a parameter, and return void [ERROR]
7. If this row is for an Event, and its *Semantics* is `Fire`, then the row indexed in the *MethodDef* table by *Method* can return any type
8. For each property, there shall be a setter, or a getter, or both [CLS]
9. Any getter method for a property whose *Name* is **xxx** shall be called **get_xxx** [CLS]
10. Any setter method for a property whose *Name* is **xxx** shall be called **set_xxx** [CLS]
11. If a property provides both getter and setter methods, then these methods shall have the same value in the *Flags.MemberAccessMask* subfield [CLS]
12. If a property provides both getter and setter methods, then these methods shall have the same value for their *Method.Flags.Virtual* [CLS]
13. Any getter and setter methods shall have *Method.Flags.SpecialName* = 1 [CLS]
14. Any getter method shall have a return type which matches the signature indexed by the *Property.Type* field [CLS]
15. The last parameter for any setter method shall have a type which matches the signature indexed by the *Property.Type* field [CLS]
16. Any setter method shall have return type `ELEMENT_TYPE_VOID` (§23.1.16) in *Method.Signature* [CLS]
17. If the property is indexed, the indexes for getter and setter shall agree in number and type [CLS]
18. Any *AddOn* method for an event whose *Name* is **xxx** shall have the signature: `void add_xxx (<DelegateType> handler)` [CLS]
19. Any *RemoveOn* method for an event whose *Name* is **xxx** shall have the signature: `void remove_xxx(<DelegateType> handler)` [CLS]
20. Any *Fire* method for an event whose *Name* is **xxx** shall have the signature: `void raise_xxx(Event e)` [CLS]

End informative text

22.29 MethodSpec : 0x2B

The *MethodSpec* table has the following columns:

- *Method* (an index into the *MethodDef* or *MemberRef* table, specifying to which generic method this row refers; that is, which generic method this row is an instantiation of; more precisely, a *MethodDefOrRef* (§24.2.6) coded index)
- *Instantiation* (an index into the *Blob* heap (§23.2.15), holding the signature of this instantiation)

The *MethodSpec* table records the signature of an instantiated generic method.

Each unique instantiation of a generic method (i.e., a combination of Method and Instantiation) shall be represented by a single row in the table.

This contains informative text only

 ©ISO/IEC 2006 – All rights reserved

1. The *MethodSpec* table can contain zero or more rows
2. One or more rows can refer to the same row in the *MethodDef* or *MemberRef* table. (There can be multiple instantiations of the same generic method.)
3. The signature stored at *Instantiation* shall be a valid instantiation of the signature of the generic method stored at *Method* [ERROR]
4. There shall be no duplicate rows based upon *Method+Instantiation* [ERROR]

End informative text

22.30 Module : 0x00

The *Module* table has the following columns:

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

The *Mvid* column shall index a unique GUID in the GUID heap (§24.2.5) that identifies this instance of the module. The *Mvid* can be ignored on read by conforming implementations of the CLI. The *Mvid* should be newly generated for every module, using the algorithm specified in ISO/IEC 11578:1996 (Annex A) or another compatible algorithm.

[*Note:* The term GUID stands for Globally Unique IDentifier, a 16-byte long number typically displayed using its hexadecimal encoding. A GUID can be generated by several well-known algorithms including those used for UUIDs (Universally Unique IDentifiers) in RPC and CORBA, as well as CLSIDs, GUIDs, and IIDs in COM. *end note*]

[*Rationale:* While the VES itself makes no use of the Mvid, other tools (such as debuggers, which are outside the scope of this standard) rely on the fact that the Mvid almost always differs from one module to another. *end rationale*]

The *Generation*, *EncId*, and *EncBaseId* columns can be written as zero, and can be ignored by conforming implementations of the CLI. The rows in the *Module* table result from `.module` directives in the Assembly (§6.4).

This contains informative text only

1. The *Module* table shall contain one and only one row [ERROR]
2. *Name* shall index a non-empty string. This string should match exactly any corresponding *ModuleRef.Name* string that resolves to this module. [ERROR]
3. *Mvid* shall index a non-null GUID in the Guid heap [ERROR]

End informative text

22.31 ModuleRef : 0x1A

The *ModuleRef* table has the following column:

- *Name* (an index into the String heap)

The rows in the *ModuleRef* table result from `.module extern` directives in the Assembly (§6.5).

This contains informative text only

1. *Name* shall index a non-empty string in the String heap. This string shall enable the CLI to locate the target module (typically, it might name the file used to hold the module) [ERROR]
2. There should be no duplicate rows [WARNING]
3. *Name* should match an entry in the *Name* column of the *File* table. Moreover, that entry shall enable the CLI to locate the target module (typically it might name the file used to hold the module) [ERROR]

End informative text

22.32 NestedClass : 0x29

The *NestedClass* table has the following columns:

- *NestedClass* (an index into the *TypeDef* table)
- *EnclosingClass* (an index into the *TypeDef* table)

NestedClass is defined as lexically 'inside' the text of its enclosing Type.

This contains informative text only

The *NestedClass* table records which Type definitions are nested within which other Type definition. In a typical high-level language, the nested class is defined as lexically 'inside' the text of its enclosing Type

1. The *NestedClass* table can contain zero or more rows
2. *NestedClass* shall index a valid row in the *TypeDef* table [ERROR]
3. *EnclosingClass* shall index a valid row in the *TypeDef* table (note particularly, it is not allowed to index the *TypeRef* table) [ERROR]
4. There should be no duplicate rows (ie same values for *NestedClass* and *EnclosingClass*) [WARNING]
5. A given Type can only be nested by one encloser. So, there cannot be two rows with the same value for *NestedClass*, but different value for *EnclosingClass* [ERROR]
6. A given Type can 'own' several different nested Types, so it is perfectly valid to have two or more rows with the same value for *EnclosingClass* but different values for *NestedClass*

End informative text

22.33 Param : 0x08

The *Param* table has the following columns:

- *Flags* (a 2-byte bitmask of type ParamAttributes, §23.1.13)
- *Sequence* (a 2-byte constant)
- *Name* (an index into the String heap)

Conceptually, every row in the *Param* table is owned by one, and only one, row in the *MethodDef* table

The rows in the *Param* table result from the parameters in a method declaration (§15.4), or from a `.param` attribute attached to a method (§15.4.1).

This contains informative text only

1. *Param* table can contain zero or more rows

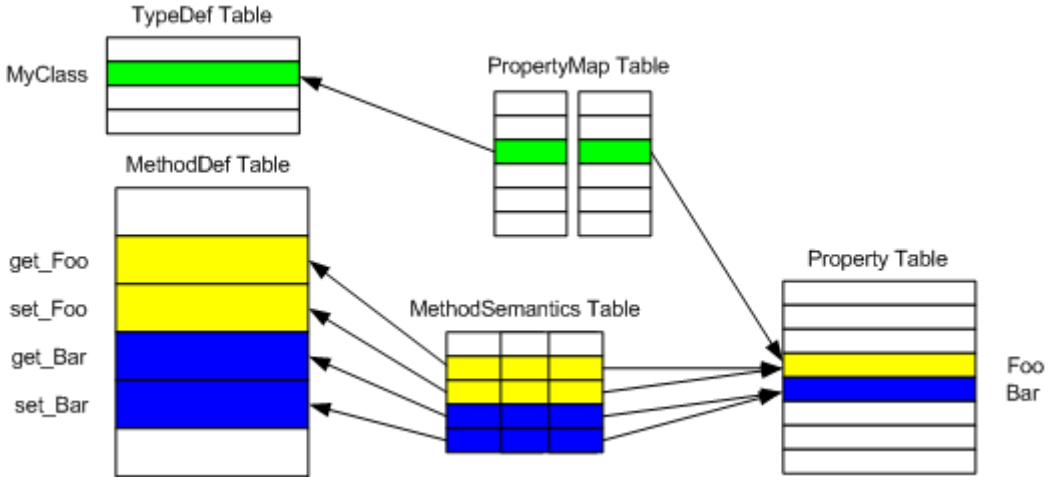
 ©ISO/IEC 2006 – All rights reserved

2. Each row shall have one, and only one, owner row in the *MethodDef* table [ERROR]
3. *Flags* shall have only those values set that are specified (all combinations valid) [ERROR]
4. *Sequence* shall have a value >= 0 and <= number of parameters in owner method. A *Sequence* value of 0 refers to the owner method's return type; its parameters are then numbered from 1 onwards [ERROR]
5. Successive rows of the *Param* table that are owned by the same method shall be ordered by increasing *Sequence* value - although gaps in the sequence are allowed [WARNING]
6. If *Flags.HasDefault* = 1 then this row shall own exactly one row in the *Constant* table [ERROR]
7. If *Flags.HasDefault* = 0, then there shall be no rows in the *Constant* table owned by this row [ERROR]
8. parameters cannot be given default values, so *Flags.HasDefault* shall be 0 [CLS]
9. if *Flags.FieldMarshal* = 1 then this row shall own exactly one row in the *FieldMarshal* table [ERROR]
10. *Name* can be null or non-null
11. If *Name* is non-null, then it shall index a non-empty string in the String heap [WARNING]

End informative text

22.34 Property : 0x17

Properties within metadata are best viewed as a means to gather together collections of methods defined on a class, give them a name, and not much else. The methods are typically *get_* and *set_* methods, already defined on the class, and inserted like any other methods into the *MethodDef* table. The association is held together by three separate tables, as shown below:



Row 3 of the *PropertyMap* table indexes row 2 of the *TypeDef* table on the left (*MyClass*), whilst indexing row 4 of the *Property* table on the right – the row for a property called *Foo*. This setup establishes that *MyClass* has a property called *Foo*. But what methods in the *MethodDef* table are gathered together as 'belonging' to property *Foo*? That association is contained in the *MethodSemantics* table – its row 2 indexes property *Foo* to the right, and row 2 in the *MethodDef* table to the left (a method called *get_Foo*). Also, row 3 of the *MethodSemantics* table indexes *Foo* to the right, and row 3 in the *MethodDef* table to the left (a method called *set_Foo*). As the shading suggests, *MyClass* has another property, called *Bar*, with two methods, *get_Bar* and *set_Bar*.

Property tables do a little more than group together existing rows from other tables. The *Property* table has columns for *Flags*, *Name* (eg *Foo* and *Bar* in the example here) and *Type*. In addition, the *MethodSemantics* table has a column to record whether the method it points at is a *set_*, a *get_* or *other*.

©ISO/IEC 2006 – All rights reserved

[*Note:* The CLS (see Partition I) refers to instance, virtual, and static properties. The signature of a property (from the *Type* column) can be used to distinguish a static property, since instance and virtual properties will have the "HASTHIS" bit set in the signature (§23.2.1) while a static property will not. The distinction between an instance and a virtual property depends on the signature of the getter and setter methods, which the CLS requires to be either both virtual or both instance. *end note*]

The *Property* (0x17) table has the following columns:

- *Flags* (a 2-byte bitmask of type PropertyAttributes, §23.1.14)
- *Name* (an index into the String heap)
- *Type* (an index into the Blob heap) (The name of this column is misleading. It does not index a *TypeDef* or *TypeRef* table—instead it indexes the signature in the Blob heap of the Property)

This contains informative text only

1. *Property* table can contain zero or more rows
2. Each row shall have one, and only one, owner row in the *PropertyMap* table (as described above) [ERROR]
3. *PropFlags* shall have only those values set that are specified (all combinations valid) [ERROR]
4. *Name* shall index a non-empty string in the String heap [ERROR]
5. The *Name* string shall be a valid CLS identifier [CLS]
6. *Type* shall index a non-null signature in the Blob heap [ERROR]
7. The signature indexed by *Type* shall be a valid signature for a property (ie, low nibble of leading byte is 0x8). Apart from this leading byte, the signature is the same as the property's *get_* method [ERROR]
8. Within the rows owned by a given row in the *TypeDef* table, there shall be no duplicates based upon *Name+Type* [ERROR]
9. There shall be no duplicate rows based upon *Name*, where *Name* fields are compared using CLS conflicting-identifier-rules (in particular, properties cannot be overloaded by their *Type* – a class cannot have two properties, `"int Foo"` and `"String Foo"`, for example) [CLS]

End informative text

22.35 PropertyMap : 0x15

The *PropertyMap* table has the following columns:

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

The *PropertyMap* and *Property* tables result from putting the `.property` directive on a class (§17).

This contains informative text only

1. *PropertyMap* table can contain zero or more rows
2. There shall be no duplicate rows, based upon *Parent* (a given class has only one 'pointer' to the start of its property list) [ERROR]

 ©ISO/IEC 2006 – All rights reserved

3. There shall be no duplicate rows, based upon *PropertyList* (different classes cannot share rows in the *Property* table) [ERROR]

End informative text

22.36 StandAloneSig : 0x11

Signatures are stored in the metadata Blob heap. In most cases, they are indexed by a column in some table—*Field.Signature*, *Method.Signature*, *MemberRef.Signature*, etc. However, there are two cases that require a metadata token for a signature that is not indexed by any metadata table. The *StandAloneSig* table fulfils this need. It has just one column, which points to a Signature in the Blob heap.

The signature shall describe either:

- a method – code generators create a row in the *StandAloneSig* table for each occurrence of a calli CIL instruction. That row indexes the call-site signature for the function pointer operand of the calli instruction
- local variables – code generators create one row in the *StandAloneSig* table for each method, to describe all of its local variables. The `.locals` directive (§15.4.1) in ILAsm generates a row in the *StandAloneSig* table.

The*StandAloneSig* table has the following column:

- *Signature* (an index into the Blob heap)

[*Example:*

```
// On encountering the calli instruction, ilasm generates a signature
// in the blob heap (DEFAULT, ParamCount = 1, RetType = int32, Param1 = int32),
// indexed by the StandAloneSig table:
.assembly Test {}
.method static int32 AddTen(int32)
{ ldarg.0
  ldc.i4  10
  add
  ret
}

.class Test
{ .method static void main()
  { .entrypoint
    ldc.i4.1
    ldftn int32 AddTen(int32)
    calli int32(int32)
    pop
    ret
  }
}
```

end example]

This contains informative text only

1. The *StandAloneSig* table can contain zero or more rows
2. *Signature* shall index a valid signature in the Blob heap [ERROR]
3. The signature 'blob' indexed by *Signature* shall be a valid METHOD or LOCALS signature [ERROR]
4. Duplicate rows are allowed

End informative text

22.37 TypeDef : 0x02

The *TypeDef* table has the following columns:

- *Flags* (a 4-byte bitmask of type *TypeAttributes*, §23.1.15)
- *TypeName* (an index into the String heap)
- *TypeNamespace* (an index into the String heap)
- *Extends* (an index into the *TypeDef*, *TypeRef*, or *TypeSpec* table; more precisely, a *TypeDefOrRef* (§24.2.6) coded index)
- *FieldList* (an index into the *Field* table; it marks the first of a contiguous run of Fields owned by this Type). The run continues to the smaller of:
    - the last row of the *Field* table
    - the next run of Fields, found by inspecting the *FieldList* of the next row in this *TypeDef* table
- *MethodList* (an index into the *MethodDef* table; it marks the first of a continguous run of Methods owned by this Type). The run continues to the smaller of:
    - the last row of the *MethodDef* table
    - the next run of Methods, found by inspecting the *MethodList* of the next row in this *TypeDef* table

The first row of the *TypeDef* table represents the pseudo class that acts as *parent* for functions and variables defined at module scope.

Note that any *type* shall be one, and only one, of

- Class (*Flags.Interface* = 0, and derives ultimately from `System.Object`)
- Interface (*Flags.Interface* = 1)
- Value type, derived ultimately from `System.ValueType`

For any given type, there are two separate and distinct chains of pointers to other types (the pointers are actually implemented as indexes into metadata tables). The two chains are:

- Extension chain – defined via the *Extends* column of the *TypeDef* table. Typically, a *derived Class extends* a *base Class* (always one, and only one, base *Class*)
- Interface chains – defined via the *InterfaceImpl* table. Typically, a *Class* implements zero, one or more *Interface*s

These two chains (extension and interface) are always kept separate in metadata. The *Extends* chain represents one-to-one relations—that is, one *Class extends* (or 'derives from') exactly one other *Class* (called its immediate base class). The *Interface* chains can represent one-to-many relations—that is, one *Class* might well implement two or more *Interface*s.

An interface can also implement one or more other interfaces—metadata stores those links via the *InterfaceImpl* table (the nomenclature is a little inappropriate here—there is no "implementation" involved; perhaps a clearer name might have been *Interface* table, or *InterfaceInherit* table)

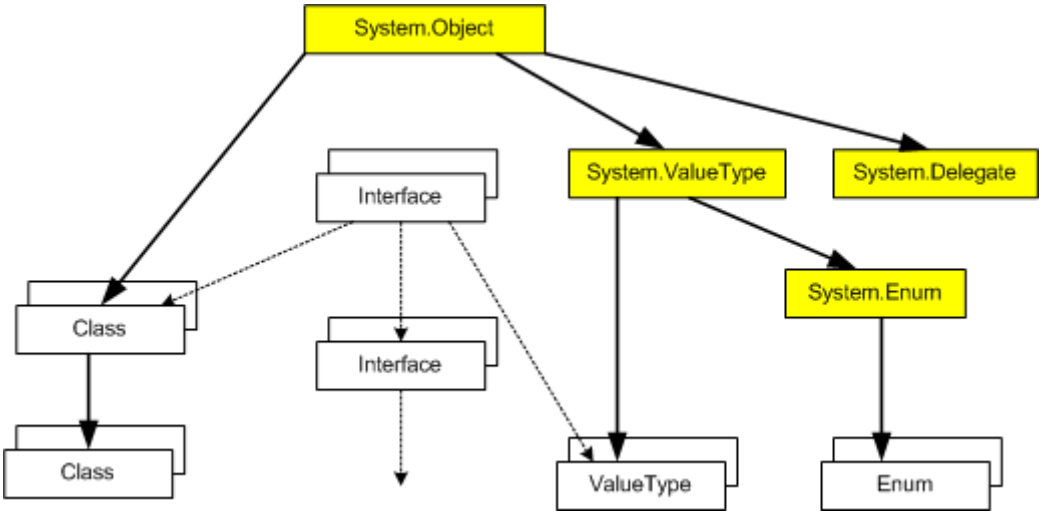
Another slightly specialized type is a *nested* type which is declared in ILAsm as lexically nested within an enclosing type declaration. Whether a type is nested can be determined by the value of its *Flags.Visibility* sub-field – it shall be one of the set {*NestedPublic*, *NestedPrivate*, *NestedFamily*, *NestedAssembly*, *NestedFamANDAssem*, *NestedFamORAssem*}.

If a type is generic, its parameters are defined in the *GenericParam* table (§22.20). Entries in the *GenericParam* table reference entries in the *TypeDef* table; there is no reference from the *TypeDef* table to the *GenericParam* table.

 ©ISO/IEC 2006 – All rights reserved

This contains informative text only

The roots of the inheritance hierarchies look like this:



There is one system-defined root, `System.Object`. All Classes and ValueTypes shall derive, ultimately, from `System.Object`; Classes can derive from other Classes (through a single, non-looping chain) to any depth required. This *Extends* inheritance chain is shown with heavy arrows.

(See below for details of the `System.Delegate` Class)

Interfaces do not inherit from one another; however, they can have zero or more required interfaces, which shall be implemented. The *Interface* requirement chain is shown as light, dashed arrows. This includes links between Interfaces and Classes/ValueTypes – where the latter are said to *implement* that interface or interfaces.

Regular ValueTypes (i.e., excluding Enums – see later) are defined as deriving directly from System.ValueType. Regular ValueTypes cannot be derived to a depth of more than one. (Another way to state this is that user-defined ValueTypes shall be *sealed*.) User-defined Enums shall derive directly from System.Enum. Enums cannot be derived to a depth of more than one below System.Enum. (Another way to state this is that user-defined Enums shall be *sealed*.) System.Enum derives directly from System.ValueType.

User-defined delegates derive from `System.Delegate`. Delegates cannot be derived to a depth of more than one.

For the directives to declare types see §9.

1. A *TypeDef* table can contain one or more rows.
2. Flags:
   a. *Flags* shall have only those values set that are specified [ERROR]
   b. can set 0 or 1 of `SequentialLayout` and `ExplicitLayout` (if none set, then defaults to `AutoLayout`) [ERROR]
   c. can set 0 or 1 of `UnicodeClass` and `AutoClass` (if none set, then defaults to `AnsiClass`) [ERROR]
   d. If *Flags.HasSecurity* = 1, then at least one of the following conditions shall be true: [ERROR]
      - this Type owns at least one row in the *DeclSecurity* table
      - this Type has a custom attribute called `SuppressUnmanagedCodeSecurityAttribute`
   e. If this Type owns one (or more) rows in the *DeclSecurity* table then *Flags.HasSecurity* shall be 1 [ERROR]

©ISO/IEC 2006 – All rights reserved

f. If this Type has a custom attribute called `SuppressUnmanagedCodeSecurityAttribute` then *Flags.HasSecurity* shall be 1 [ERROR]

g. Note that it is valid for an Interface to have `HasSecurity` set. However, the security system ignores any permission requests attached to that Interface

3. *Name* shall index a non-empty string in the String heap [ERROR]
4. The *TypeName* string shall be a valid CLS identifier [CLS]
5. *TypeNamespace* can be null or non-null
6. If non-null, then *TypeNamespace* shall index a non-empty string in the String heap [ERROR]
7. If non-null, *TypeNamespace*'s string shall be a valid CLS Identifier [CLS]
8. Every Class (with the exception of `System.Object` and the special class `<Module>`) shall extend one, and only one, other Class - so *Extends* for a Class shall be non-null [ERROR]
9. `System.Object` shall have an *Extends* value of null [ERROR]
10. `System.ValueType` shall have an *Extends* value of `System.Object` [ERROR]
11. With the exception of `System.Object` and the special class `<Module>`, for any Class, *Extends* shall index a valid row in the *TypeDef*, *TypeRef*, or *TypeSpec* table, where valid means 1 <= row <= rowcount. In addition, that row itself shall be a Class (not an Interface or ValueType) In addition, that base Class shall not be sealed (its *Flags.Sealed* shall be 0) [ERROR]
12. A Class cannot extend itself, or any of its children (i.e., its derived Classes), since this would introduce loops in the hierarchy tree [ERROR] (For generic types, see §9.1 and §9.2.)
13. An Interface never *extends* another Type - so *Extends* shall be null (Interfaces *do* implement other Interfaces, but recall that this relationship is captured via the *InterfaceImpl* table, rather than the *Extends* column) [ERROR]
14. *FieldList* can be null or non-null
15. A Class or Interface can 'own' zero or more fields
16. A ValueType shall have a non-zero size - either by defining at least one field, or by providing a non-zero *ClassSize* [ERROR]
17. If *FieldList* is non-null, it shall index a valid row in the *Field* table, where valid means 1 <= row <= rowcount+1 [ERROR]
18. *MethodList* can be null or non-null
19. A Type can 'own' zero or more methods
20. The runtime size of a ValueType shall not exceed 1 MByte (0x100000 bytes) [ERROR]
21. If *MethodList* is non-null, it shall index a valid row in the *MethodDef* table, where valid means 1 <= row <= rowcount+1 [ERROR]
22. A Class which `has` one or more abstract methods cannot be instantiated, and shall have *Flags.Abstract* = 1. Note that the methods *owned* by the class include all of those inherited from its base class and interfaces it implements, plus those defined via its *MethodList*. (The CLI shall analyze class definitions at runtime; if it finds a class to have one or more abstract methods, but has *Flags.Abstract* = 0, it will throw an exception) [ERROR]
23. An Interface shall have *Flags.Abstract* = 1 [ERROR]
24. It is valid for an abstract Type to have a constructor method (ie, a method named `.ctor`)
25. Any non-abstract Type (ie *Flags.Abstract* = 0) shall provide an implementation (body) for every method its contract requires. Its methods can be inherited from its base class, from the interfaces it implements, or defined by itself. The implementations can be inherited from its base class, or defined by itself [ERROR]

 ©ISO/IEC 2006 – All rights reserved

26. An Interface (*Flags.Interface* = 1) can own static fields (*Field.Static* = 1) but cannot own instance fields (*Field.Static* = 0) [ERROR]
27. An Interface cannot be sealed (if *Flags.Interface = 1,* then *Flags.Sealed* shall be 0) [ERROR]
28. All of the methods owned by an Interface (*Flags.Interface* = 1) shall be abstract (*Flags.Abstract* = 1) [ERROR]
29. There shall be no duplicate rows in the *TypeDef* table, based on *TypeNamespace+TypeName* (unless this is a nested type - see below) [ERROR]
30. If this is a nested type, there shall be no duplicate row in the *TypeDef* table, based upon *TypeNamespace+TypeName+OwnerRowInNestedClassTable* [ERROR]
31. There shall be no duplicate rows, where *TypeNamespace+TypeName* fields are compared using CLS conflicting-identifier-rules (unless this is a nested type - see below) [CLS]
32. If this is a nested type, there shall be no duplicate rows, based upon *TypeNamespace+TypeName+OwnerRowInNestedClassTable* and where *TypeNamespace+TypeName* fields are compared using CLS conflicting-identifier-rules [CLS]
33. If *Extends* = `System.Enum` (i.e., type is a user-defined Enum) then:
    a. shall be sealed (`Sealed` = 1) [ERROR]
    b. shall not have any methods of its own (*MethodList* chain shall be zero length) [ERROR]
    c. shall not implement any interfaces (no entries in *InterfaceImpl* table for this type) [ERROR]
    d. shall not have any properties [ERROR]
    e. shall not have any events [ERROR]
    f. any static fields shall be literal (have *Flags.Literal* = 1) [ERROR]
    g. shall have one or more static, literal fields, each of which has the type of the Enum [CLS]
    h. shall be exactly one instance field, of built-in integer type [ERROR]
    i. the *Name* string of the instance field shall be "value__", the field shall be marked `RTSpecialName`, and that field shall have one of the CLS integer types [CLS]
    j. shall not have any static fields unless they are literal [ERROR]
34. A Nested type (defined above) shall own exactly one row in the *NestedClass* table, where 'owns' means a row in that *NestedClass* table whose *NestedClass* column holds the TypeDef token for this type definition [ERROR]
35. A ValueType shall be sealed [ERROR]

End informative text

22.38 TypeRef : 0x01

The *TypeRef* table has the following columns:

- *ResolutionScope* (an index into a *Module*, *ModuleRef*, *AssemblyRef* or *TypeRef* table, or null; more precisely, a *ResolutionScope* (§24.2.6) coded index)
- *TypeName* (an index into the String heap)
- *TypeNamespace* (an index into the String heap)

This contains informative text only

1. *ResolutionScope* shall be exactly one of:

a. null - in this case, there shall be a row in the *ExportedType* table for this Type - its *Implementation* field shall contain a *File* token or an *AssemblyRef* token that says where the type is defined [ERROR]

b. a *TypeRef* token, if this is a nested type (which can be determined by, for example, inspecting the *Flags* column in its *TypeDef* table - the accessibility subfield is one of the `tdNestedXXX` set) [ERROR]

c. a *ModuleRef* token, if the target type is defined in another module within the same Assembly as this one [ERROR]

d. a *Module* token, if the target type is defined in the current module - this should not occur in a CLI ("compressed metadata") module [WARNING]

e. an *AssemblyRef* token, if the target type is defined in a different Assembly from the current module [ERROR]

2. *TypeName* shall index a non-empty string in the String heap [ERROR]

3. *TypeNamespace* can be null, or non-null

4. If non-null, *TypeNamespace* shall index a non-empty string in the String heap [ERROR]

5. The *TypeName* string shall be a valid CLS identifier [CLS]

6. There shall be no duplicate rows, where a duplicate has the same *ResolutionScope, TypeName* and *TypeNamespace* [ERROR]

7. There shall be no duplicate rows, where *TypeName* and *TypeNamespace* fields are compared using CLS conflicting-identifier-rules [CLS]

End informative text

22.39 TypeSpec : 0x1B

The *TypeSpec* table has just one column, which indexes the specification of a Type, stored in the Blob heap. This provides a metadata token for that Type (rather than simply an index into the Blob heap). This is required, typically, for array operations, such as creating, or calling methods on the array class.

The *TypeSpec* table has the following column:

- *Signature* (index into the Blob heap, where the blob is formatted as specified in §23.2.14)

Note that TypeSpec tokens can be used with any of the CIL instructions that take a *TypeDef* or *TypeRef* token; specifically, castclass, cpobj, initobj, isinst, ldelema, ldobj, mkrefany, newarr, refanyval, sizeof, stobj, box, and unbox.

This contains informative text only

1. The *TypeSpec* table can contain zero or more rows

2. *Signature* shall index a valid Type specification in the Blob heap [ERROR]

3. There shall be no duplicate rows, based upon *Signature* [ERROR]

End informative text

 ©ISO/IEC 2006 – All rights reserved

23 Metadata logical format: other structures

23.1 Bitmasks and flags

This subclause explains the flags and bitmasks used in the metadata tables. When a conforming implementation encounters a metadata structure (such as a flag) that is not specified in this standard, the behavior of the implementation is unspecified.

23.1.1 Values for AssemblyHashAlgorithm

Algorithm	Value
None	0x0000
Reserved (MD5)	0x8003
SHA1	0x8004

23.1.2 Values for AssemblyFlags

Flag	Value	Description
PublicKey	0x0001	The assembly reference holds the full (unhashed) public key.
SideBySideCompatible	0x0000	The assembly is side-by-side compatible
<reserved>	0x0030	Reserved: both bits shall be zero
Retargetable	0x0100	The implementation of this assembly used at runtime is not expected to match the version seen at compile time. (See the text following this table.)
EnableJITcompileTracking	0x8000	Reserved (a conforming implementation of the CLI can ignore this setting on read; some implementations might use this bit to indicate that a CIL-to-native-code compiler should generate CIL-to-native code map)
DisableJITcompileOptimizer	0x4000	Reserved (a conforming implementation of the CLI can ignore this setting on read; some implementations might use this bit to indicate that a CIL-to-native-code compiler should not generate optimized code)

In portable programs, the Retargetable (0x100) bit shall be set on all references to assemblies specified in this Standard.

23.1.3 Values for Culture

ar-SA	ar-IQ	ar-EG	ar-LY
ar-DZ	ar-MA	ar-TN	ar-OM
ar-YE	ar-SY	ar-JO	ar-LB
ar-KW	ar-AE	ar-BH	ar-QA
bg-BG	ca-ES	zh-TW	zh-CN
zh-HK	zh-SG	zh-MO	cs-CZ
da-DK	de-DE	de-CH	de-AT
de-LU	de-LI	el-GR	en-US
en-GB	en-AU	en-CA	en-NZ

en-IE	en-ZA	en-JM	en-CB
en-BZ	en-TT	en-ZW	en-PH
es-ES-Ts	es-MX	es-ES-Is	es-GT
es-CR	es-PA	es-DO	es-VE
es-CO	es-PE	es-AR	es-EC
es-CL	es-UY	es-PY	es-BO
es-SV	es-HN	es-NI	es-PR
fi-FI	fr-FR	fr-BE	fr-CA
fr-CH	fr-LU	fr-MC	he-IL
hu-HU	is-IS	it-IT	it-CH
ja-JP	ko-KR	nl-NL	nl-BE
nb-NO	nn-NO	pl-PL	pt-BR
pt-PT	ro-RO	ru-RU	hr-HR
lt-sr-SP	cy-sr-SP	sk-SK	sq-AL
sv-SE	sv-FI	th-TH	tr-TR
ur-PK	id-ID	uk-UA	be-BY
sl-SI	et-EE	lv-LV	lt-LT
fa-IR	vi-VN	hy-AM	lt-az-AZ
cy-az-AZ	eu-ES	mk-MK	af-ZA
ka-GE	fo-FO	hi-IN	ms-MY
ms-BN	kk-KZ	ky-KZ	sw-KE
lt-uz-UZ	cy-uz-UZ	tt-TA	pa-IN
gu-IN	ta-IN	te-IN	kn-IN
mr-IN	sa-IN	mn-MN	gl-ES
kok-IN	syr-SY	div-MV	

Note on RFC 1766, Locale names: a typical string would be “en-US”. The first part (“en” in the example) uses ISO 639 characters (“Latin-alphabet characters in lowercase. No diacritical marks of modified characters are used”). The second part (“US” in the example) uses ISO 3166 characters (similar to ISO 639, but uppercase); that is, the familiar ASCII characters a–z and A–Z, respectively. However, whilst RFC 1766 recommends the first part be lowercase and the second part be uppercase, it allows mixed case. Therefore, the validation rule checks only that *Culture* is one of the strings in the list above—but the check is totally case-blind—where case-blind is the familiar fold on values less than U+0080

23.1.4 Flags for events [EventAttributes]

Flag	Value	Description
SpecialName	0x0200	Event is special.
RTSpecialName	0x0400	CLI provides 'special' behavior, depending upon the name of the event

23.1.5 Flags for fields [FieldAttributes]

Flag	Value	Description
FieldAccessMask	0x0007	These 3 bits contain one of the following values:
CompilerControlled	0x0000	Member not referenceable

 ©ISO/IEC 2006 – All rights reserved

Private	0x0001	Accessible only by the parent type
FamANDAssem	0x0002	Accessible by sub-types only in this Assembly
Assembly	0x0003	Accessibly by anyone in the Assembly
Family	0x0004	Accessible only by type and sub-types
FamORAssem	0x0005	Accessibly by sub-types anywhere, plus anyone in assembly
Public	0x0006	Accessibly by anyone who has visibility to this scope field contract attributes
Static	0x0010	Defined on type, else per instance
InitOnly	0x0020	Field can only be initialized, not written to after init
Literal	0x0040	Value is compile time constant
NotSerialized	0x0080	Reserved (to indicate this field should not be serialized when type is remoted)
SpecialName	0x0200	Field is special
Interop Attributes		
PInvokeImpl	0x2000	Implementation is forwarded through PInvoke.
Additional flags		
RTSpecialName	0x0400	CLI provides 'special' behavior, depending upon the name of the field
HasFieldMarshal	0x1000	Field has marshalling information
HasDefault	0x8000	Field has default
HasFieldRVA	0x0100	Field has RVA

23.1.6 Flags for files [FileAttributes]

Flag	Value	Description
ContainsMetaData	0x0000	This is not a resource file
ContainsNoMetaData	0x0001	This is a resource file or other non-metadata-containing file

23.1.7 Flags for Generic Parameters [GenericParamAttributes]

Flag	Value	Description
VarianceMask	0x0003	These 2 bits contain one of the following values:
None	0x0000	The generic parameter is non-variant and has no special constraints
Covariant	0x0001	The generic parameter is covariant
Contravariant	0x0002	The generic parameter is contravariant
SpecialConstraintMask	0x001C	These 3 bits contain one of the following values:
ReferenceTypeConstraint	0x0004	The generic parameter has the `class` special constraint
NotNullableValueTypeConstraint	0x0008	The generic parameter has the `valuetype` special constraint
DefaultConstructorConstraint	0x0010	The generic parameter has the `.ctor` special constraint

©ISO/IEC 2006 – All rights reserved

23.1.8 Flags for ImplMap [PInvokeAttributes]

Flag	Value	Description
NoMangle	0x0001	PInvoke is to use the member name as specified
Character set		
CharSetMask	0x0006	This is a resource file or other non-metadata-containing file. These 2 bits contain one of the following values:
CharSetNotSpec	0x0000	
CharSetAnsi	0x0002	
CharSetUnicode	0x0004	
CharSetAuto	0x0006	
SupportsLastError	0x0040	Information about target function. Not relevant for fields
Calling convention		
CallConvMask	0x0700	These 3 bits contain one of the following values:
CallConvWinapi	0x0100	
CallConvCdecl	0x0200	
CallConvStdcall	0x0300	
CallConvThiscall	0x0400	
CallConvFastcall	0x0500	

23.1.9 Flags for ManifestResource [ManifestResourceAttributes]

Flag	Value	Description
VisibilityMask	0x0007	These 3 bits contain one of the following values:
Public	0x0001	The Resource is exported from the Assembly
Private	0x0002	The Resource is private to the Assembly

23.1.10 Flags for methods [MethodAttributes]

Flag	Value	Description
MemberAccessMask	0x0007	These 3 bits contain one of the following values:
CompilerControlled	0x0000	Member not referenceable
Private	0x0001	Accessible only by the parent type
FamANDAssem	0x0002	Accessible by sub-types only in this Assembly
Assem	0x0003	Accessibly by anyone in the Assembly
Family	0x0004	Accessible only by type and sub-types
FamORAssem	0x0005	Accessibly by sub-types anywhere, plus anyone in assembly
Public	0x0006	Accessibly by anyone who has visibility to this scope
Static	0x0010	Defined on type, else per instance
Final	0x0020	Method cannot be overridden

 ©ISO/IEC 2006 – All rights reserved

Virtual	0x0040	Method is virtual
HideBySig	0x0080	Method hides by name+sig, else just by name
VtableLayoutMask	0x0100	Use this mask to retrieve vtable attributes. This bit contains one of the following values:
ReuseSlot	0x0000	Method reuses existing slot in vtable
NewSlot	0x0100	Method always gets a new slot in the vtable
Strict	0x0200	Method can only be overriden if also accessible
Abstract	0x0400	Method does not provide an implementation
SpecialName	0x0800	Method is special
Interop attributes		
PInvokeImpl	0x2000	Implementation is forwarded through PInvoke
UnmanagedExport	0x0008	Reserved: shall be zero for conforming implementations
Additional flags		
RTSpecialName	0x1000	CLI provides 'special' behavior, depending upon the name of the method
HasSecurity	0x4000	Method has security associate with it
RequireSecObject	0x8000	Method calls another method containing security code.

23.1.11 Flags for methods [MethodImplAttributes]

Flag	Value	Description
CodeTypeMask	0x0003	These 2 bits contain one of the following values:
IL	0x0000	Method impl is CIL
Native	0x0001	Method impl is native
OPTIL	0x0002	Reserved: shall be zero in conforming implementations
Runtime	0x0003	Method impl is provided by the runtime
ManagedMask	0x0004	Flags specifying whether the code is managed or unmanaged. This bit contains one of the following values:
Unmanaged	0x0004	Method impl is unmanaged, otherwise managed
Managed	0x0000	Method impl is managed
Implementation info and interop		
ForwardRef	0x0010	Indicates method is defined; used primarily in merge scenarios
PreserveSig	0x0080	Reserved: conforming implementations can ignore
InternalCall	0x1000	Reserved: shall be zero in conforming implementations
Synchronized	0x0020	Method is single threaded through the body
NoInlining	0x0008	Method cannot be inlined
MaxMethodImplVal	0xffff	Range check value

23.1.12 Flags for MethodSemantics [MethodSemanticsAttributes]

Flag	Value	Description
Setter	0x0001	Setter for property
Getter	0x0002	Getter for property
Other	0x0004	Other method for property or event
AddOn	0x0008	AddOn method for event
RemoveOn	0x0010	RemoveOn method for event
Fire	0x0020	Fire method for event

23.1.13 Flags for params [ParamAttributes]

Flag	Value	Description
In	0x0001	Param is [In]
Out	0x0002	Param is [out]
Optional	0x0010	Param is optional
HasDefault	0x1000	Param has default value
HasFieldMarshal	0x2000	Param has FieldMarshal
Unused	0xcfe0	Reserved: shall be zero in a conforming implementation

23.1.14 Flags for properties [PropertyAttributes]

Flag	Value	Description
SpecialName	0x0200	Property is special
RTSpecialName	0x0400	Runtime(metadata internal APIs) should check name encoding
HasDefault	0x1000	Property has default
Unused	0xe9ff	Reserved: shall be zero in a conforming implementation

23.1.15 Flags for types [TypeAttributes]

Flag	Value	Description
Visibility attributes		
VisibilityMask	0x00000007	Use this mask to retrieve visibility information. These 3 bits contain one of the following values:
NotPublic	0x00000000	Class has no public scope
Public	0x00000001	Class has public scope
NestedPublic	0x00000002	Class is nested with public visibility
NestedPrivate	0x00000003	Class is nested with private visibility
NestedFamily	0x00000004	Class is nested with family visibility
NestedAssembly	0x00000005	Class is nested with assembly visibility

 ©ISO/IEC 2006 – All rights reserved

NestedFamANDAssem	0x00000006	Class is nested with family and assembly visibility
NestedFamORAssem	0x00000007	Class is nested with family or assembly visibility
Class layout attributes		
LayoutMask	0x00000018	Use this mask to retrieve class layout information. These 2 bits contain one of the following values:
AutoLayout	0x00000000	Class fields are auto-laid out
SequentialLayout	0x00000008	Class fields are laid out sequentially
ExplicitLayout	0x00000010	Layout is supplied explicitly
Class semantics attributes		
ClassSemanticsMask	0x00000020	Use this mask to retrive class semantics information. This bit contains one of the following values:
Class	0x00000000	Type is a class
Interface	0x00000020	Type is an interface
Special semantics in addition to class semantics		
Abstract	0x00000080	Class is abstract
Sealed	0x00000100	Class cannot be extended
SpecialName	0x00000400	Class name is special
Implementation Attributes		
Import	0x00001000	Class/Interface is imported
Serializable	0x00002000	Reserved (Class is serializable)
String formatting Attributes		
StringFormatMask	0x00030000	Use this mask to retrieve string information for native interop. These 2 bits contain one of the following values:
AnsiClass	0x00000000	LPSTR is interpreted as ANSI
UnicodeClass	0x00010000	LPSTR is interpreted as Unicode
AutoClass	0x00020000	LPSTR is interpreted automatically
CustomFormatClass	0x00030000	A non-standard encoding specified by CustomStringFormatMask
CustomStringFormatMask	0x00C00000	Use this mask to retrieve non-standard encoding information for native interop. The meaning of the values of these 2 bits is unspecified.
Class Initialization Attributes		
BeforeFieldInit	0x00100000	Initialize the class before first static field access
Additional Flags		

RTSpecialName	0x00000800	CLI provides 'special' behavior, depending upon the name of the Type
HasSecurity	0x00040000	Type has security associate with it

23.1.16 Element types used in signatures

The following table lists the values for ELEMENT_TYPE constants. These are used extensively in metadata signature *blobs* – see §23.2

Name	Value	Remarks
ELEMENT_TYPE_END	0x00	Marks end of a list
ELEMENT_TYPE_VOID	0x01	
ELEMENT_TYPE_BOOLEAN	0x02	
ELEMENT_TYPE_CHAR	0x03	
ELEMENT_TYPE_I1	0x04	
ELEMENT_TYPE_U1	0x05	
ELEMENT_TYPE_I2	0x06	
ELEMENT_TYPE_U2	0x07	
ELEMENT_TYPE_I4	0x08	
ELEMENT_TYPE_U4	0x09	
ELEMENT_TYPE_I8	0x0a	
ELEMENT_TYPE_U8	0x0b	
ELEMENT_TYPE_R4	0x0c	
ELEMENT_TYPE_R8	0x0d	
ELEMENT_TYPE_STRING	0x0e	
ELEMENT_TYPE_PTR	0x0f	Followed by *type*
ELEMENT_TYPE_BYREF	0x10	Followed by *type*
ELEMENT_TYPE_VALUETYPE	0x11	Followed by TypeDef or TypeRef token
ELEMENT_TYPE_CLASS	0x12	Followed by TypeDef or TypeRef token
ELEMENT_TYPE_VAR	0x13	Generic parameter in a generic type definition, represented as *number*
ELEMENT_TYPE_ARRAY	0x14	*type rank boundsCount bound1 … loCount lo1 …*
ELEMENT_TYPE_GENERICINST	0x15	Generic type instantiation. Followed by *type type-arg-count type-1 ... type-n*
ELEMENT_TYPE_TYPEDBYREF	0x16	
ELEMENT_TYPE_I	0x18	System.IntPtr
ELEMENT_TYPE_U	0x19	System.UIntPtr
ELEMENT_TYPE_FNPTR	0x1b	Followed by full method signature
ELEMENT_TYPE_OBJECT	0x1c	System.Object
ELEMENT_TYPE_SZARRAY	0x1d	Single-dim array with 0 lower bound

 ©ISO/IEC 2006 – All rights reserved

Name	Value	Remarks
ELEMENT_TYPE_END	0x00	Marks end of a list
ELEMENT_TYPE_VOID	0x01	
ELEMENT_TYPE_BOOLEAN	0x02	
ELEMENT_TYPE_CHAR	0x03	
ELEMENT_TYPE_I1	0x04	
ELEMENT_TYPE_U1	0x05	
ELEMENT_TYPE_I2	0x06	
ELEMENT_TYPE_U2	0x07	
ELEMENT_TYPE_I4	0x08	
ELEMENT_TYPE_U4	0x09	
ELEMENT_TYPE_I8	0x0a	
ELEMENT_TYPE_U8	0x0b	
ELEMENT_TYPE_R4	0x0c	
ELEMENT_TYPE_R8	0x0d	
ELEMENT_TYPE_STRING	0x0e	
ELEMENT_TYPE_PTR	0x0f	Followed by *type*
ELEMENT_TYPE_BYREF	0x10	Followed by *type*
ELEMENT_TYPE_VALUETYPE	0x11	Followed by TypeDef or TypeRef token
ELEMENT_TYPE_CLASS	0x12	Followed by TypeDef or TypeRef token
ELEMENT_TYPE_MVAR	0x1e	Generic parameter in a generic method definition, represented as *number*
ELEMENT_TYPE_CMOD_REQD	0x1f	Required modifier : followed by a TypeDef or TypeRef token
ELEMENT_TYPE_CMOD_OPT	0x20	Optional modifier : followed by a TypeDef or TypeRef token
ELEMENT_TYPE_INTERNAL	0x21	Implemented within the CLI
ELEMENT_TYPE_MODIFIER	0x40	Or'd with following element types
ELEMENT_TYPE_SENTINEL	0x41	Sentinel for vararg method signature
ELEMENT_TYPE_PINNED	0x45	Denotes a local variable that points at a pinned object
	0x50	Indicates an argument of type System.Type.
	0x51	Used in custom attributes to specify a boxed object (§23.3).
	0x52	Reserved
	0x53	Used in custom attributes to indicate a FIELD (§22.10, 23.3).
	0x54	Used in custom attributes to indicate a PROPERTY (§22.10, 23.3).

Name	Value	Remarks
ELEMENT_TYPE_END	0x00	Marks end of a list
ELEMENT_TYPE_VOID	0x01	
ELEMENT_TYPE_BOOLEAN	0x02	
ELEMENT_TYPE_CHAR	0x03	
ELEMENT_TYPE_I1	0x04	
ELEMENT_TYPE_U1	0x05	
ELEMENT_TYPE_I2	0x06	
ELEMENT_TYPE_U2	0x07	
ELEMENT_TYPE_I4	0x08	
ELEMENT_TYPE_U4	0x09	
ELEMENT_TYPE_I8	0x0a	
ELEMENT_TYPE_U8	0x0b	
ELEMENT_TYPE_R4	0x0c	
ELEMENT_TYPE_R8	0x0d	
ELEMENT_TYPE_STRING	0x0e	
ELEMENT_TYPE_PTR	0x0f	Followed by *type*
ELEMENT_TYPE_BYREF	0x10	Followed by *type*
ELEMENT_TYPE_VALUETYPE	0x11	Followed by TypeDef or TypeRef token
ELEMENT_TYPE_CLASS	0x12	Followed by TypeDef or TypeRef token
	0x55	Used in custom attributes to specify an enum (§23.3).

23.2 Blobs and signatures

The word *signature* is conventionally used to describe the type info for a function or method; that is, the type of each of its parameters, and the type of its return value. Within metadata, the word *signature* is also used to describe the type info for fields, properties, and local variables. Each Signature is stored as a (counted) byte array in the Blob heap. There are several kinds of Signature, as follows:

- MethodRefSig (differs from a MethodDefSig only for VARARG calls)
- MethodDefSig
- FieldSig
- PropertySig
- LocalVarSig
- TypeSpec
- MethodSpec

The value of the first byte of a Signature 'blob' indicates what kind of Signature it is. Its lowest 4 bits hold one of the following: C, DEFAULT, FASTCALL, STDCALL, THISCALL, or VARARG (whose values are defined in §23.2.3), which qualify method signatures; FIELD, which denotes a field signature (whose value is defined in §23.2.4); or PROPERTY, which denotes a property signature (whose value is defined in §23.2.5). This subclause defines the binary 'blob' format for each kind of Signature. In the syntax diagrams that accompany many of the definitions,

 ©ISO/IEC 2006 – All rights reserved

shading is used to combine into a single diagram what would otherwise be multiple diagrams; the accompanying text describes the use of shading.

Signatures are compressed before being stored into the Blob heap (described below) by compressing the integers embedded in the signature. The maximum encodable 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 2-byte integer with bit 15 set, bit 14 clear (value held in bits 13 through 0)
- Otherwise, encode as a 4-byte integer, with bit 31 set, bit 30 set, bit 29 clear (value held in bits 28 through 0)
- 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.) *end note*]

Original Value	**Compressed Representation**
0x03	03
0x7F	7F (7 bits set)
0x80	8080
0x2E57	AE57
0x3FFF	BFFF
0x4000	C000 4000
0x1FFF FFFF	DFFF FFFF

The most significant bits (the first ones encountered in a PE file) of a "compressed" field, can reveal whether it occupies 1, 2, or 4 bytes, as well as its value. For this to work, the "compressed" value, as explained above, is stored in big-endian order; i.e., with the most significant byte at the smallest offset within the file.

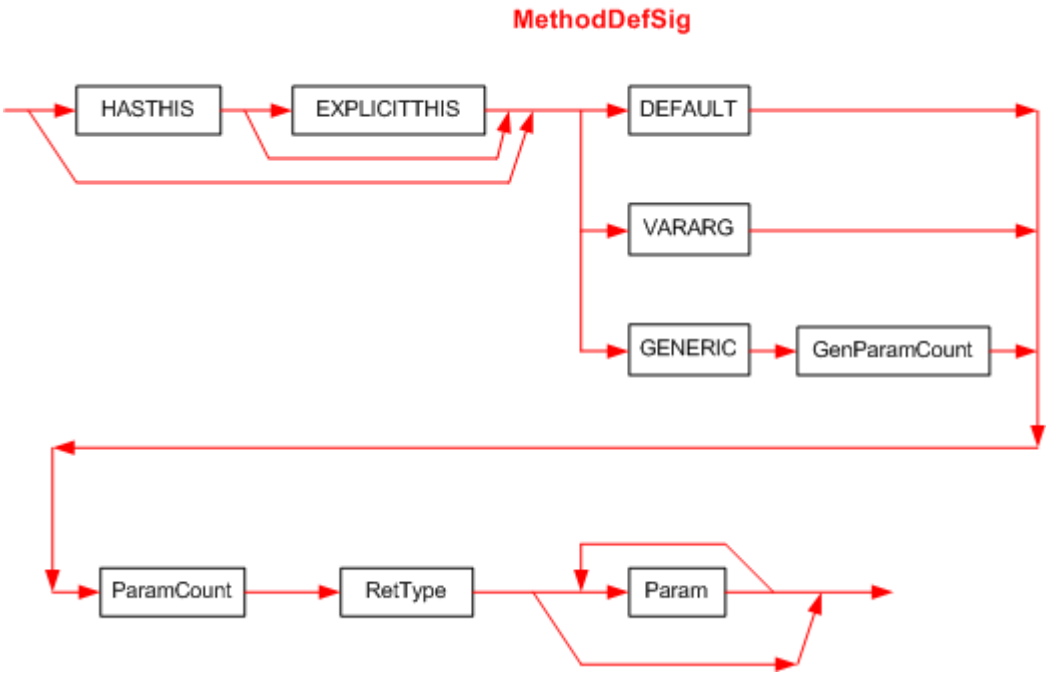
Signatures make extensive use of constant values called `ELEMENT_TYPE_xxx` – see §23.1.16. In particular, signatures include two modifiers called:

`ELEMENT_TYPE_BYREF` – this element is a managed pointer (see Partition I). This modifier can only occur in the definition of *LocalVarSig* (§23.2.6), *Param* (§23.2.10) or *RetType* (§23.2.11). It shall not occur within the definition of a Field (§23.2.4)

`ELEMENT_TYPE_PTR` – this element is an unmanaged pointer (see Partition I). This modifier can occur in the definition of *LocalVarSig* (§23.2.6), *Param* (§23.2.10), *RetType* (§23.2.11) or *Field* (§23.2.4)

23.2.1 MethodDefSig

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



This diagram uses the following abbreviations:

`HASTHIS` = 0x20, used to encode the keyword `instance` in the calling convention, see §15.3

`EXPLICITTHIS` = 0x40, used to encode the keyword `explicit` in the calling convention, see §15.3

`DEFAULT` = 0x0, used to encode the keyword `default` in the calling convention, see §15.3

`VARARG` = 0x5, used to encode the keyword `vararg` in the calling convention, see §15.3

`GENERIC` = 0x10, used to indicate that the method has one or more generic parameters.

The first byte of the Signature holds bits for `HASTHIS`, `EXPLICITTHIS` and calling convention (`DEFAULT`, `VARARG`, or `GENERIC`). These are ORed together.

GenParamCount is the number of generic parameters for the method. This is a compressed int32. [*Note*: For generic methods, both *MethodDef* and *MemberRef* shall include the `GENERIC` calling convention, together with *GenParamCount*; these are significant for binding—they enable the CLI to overload on generic methods by the number of generic parameters they include. *end note*]

ParamCount is an integer that holds the number of parameters (0 or more). It can be any number between 0 and 0x1FFFFFFF. The compiler compresses it too (see Partition II Metadata Validation) – before storing into the 'blob' (*ParamCount* counts just the method parameters – it does not include the method's return type)

The *RetType* item describes the type of the method's return value (§23.2.11)

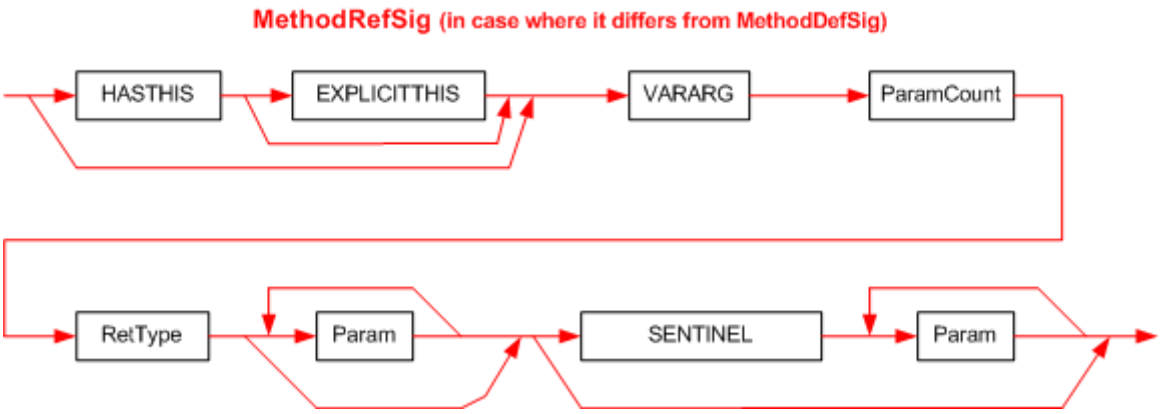
The *Param* item describes the type of each of the method's parameters. There shall be *ParamCount* instances of the *Param* item (§23.2.10).

23.2.2 MethodRefSig

A MethodRefSig is indexed by the `MemberRef.Signature` column. This provides the *call site* Signature for a method. Normally, this call site Signature shall match exactly the Signature specified in the definition of the target method. For example, if a method Foo is defined that takes two `unsigned int32`s and returns void; then any call site shall index a signature that takes exactly two `unsigned int32`s and returns void. In this case, the syntax diagram for a MethodRefSig is identical with that for a MethodDefSig – see §23.2.1

 ©ISO/IEC 2006 – All rights reserved

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



This diagram uses the following abbreviations:

`HASTHIS` = `0x20`, used to encode the keyword instance in the calling convention, see §15.3

`EXPLICITTHIS` = `0x40`, used to encode the keyword explicit in the calling convention, see §15.3

`VARARG` = `0x5`, used to encode the keyword vararg in the calling convention, see 15.3

`SENTINEL` = `0x41` (§23.1.16), used to encode "..." in the parameter list, see §15.3

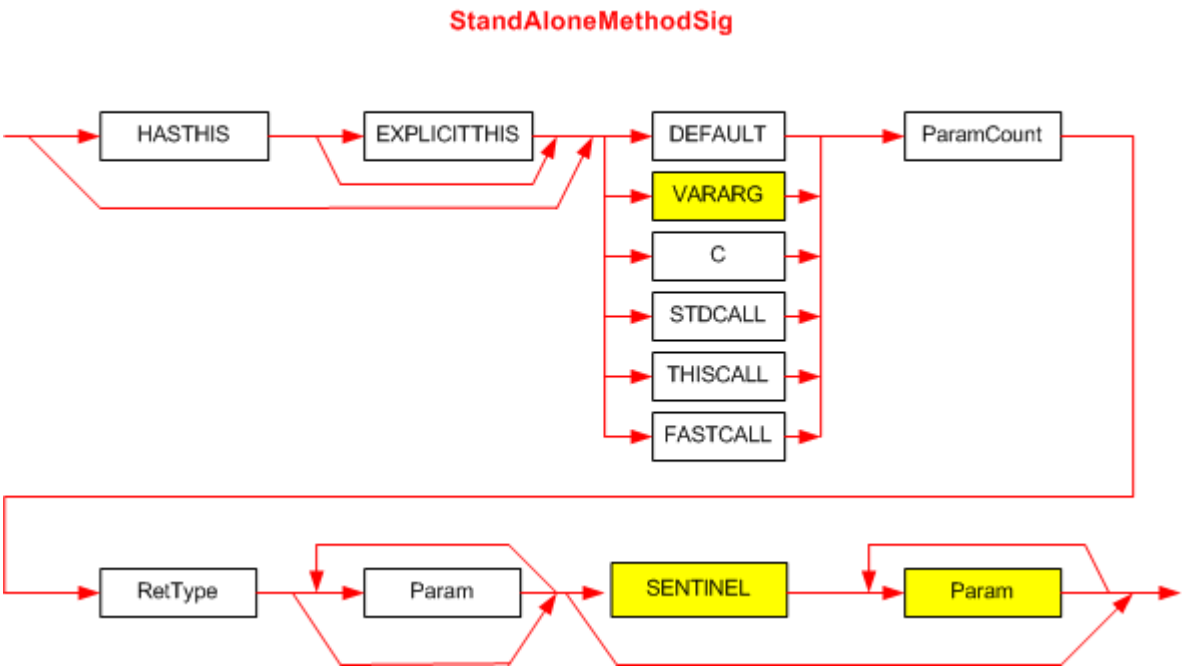
- The first byte of the Signature holds bits for `HASTHIS`, `EXPLICITTHIS`, and the calling convention `VARARG`. These are ORed together.
- *ParamCount* is an integer that holds the number of parameters (0 or more). It can be any number between 0 and 0x1FFFFFFF The compiler compresses it too (see Partition II Metadata Validation) – before storing into the 'blob' (*ParamCount* counts just the method parameters – it does not include the method's return type)
- The *RetType* item describes the type of the method's return value (§23.2.11)
- The *Param* item describes the type of each of the method's parameters. There shall be *ParamCount* instances of the *Param* item (§23.2.10).

The *Param* item describes the type of each of the method's parameters. There shall be *ParamCount* instances of the *Param* item.This starts just like the MethodDefSig for a `VARARG` method (§23.2.1). But then a `SENTINEL` token is appended, followed by extra *Param* items to describe the extra `VARARG` arguments. Note that the *ParamCount* item shall indicate the total number of *Param* items in the Signature – before and after the `SENTINEL` byte (0x41).

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

23.2.3 StandAloneMethodSig

A StandAloneMethodSig is indexed by the `StandAloneSig.Signature` column. It is typically created as preparation for executing a calli instruction. It is similar to a MethodRefSig, in that it represents a call site signature, but its calling convention can specify an unmanaged target (the calli instruction invokes either managed, or unmanaged code). Its syntax diagram is:



This diagram uses the following abbreviations (§15.3):

`HASTHIS` for 0x20

`EXPLICITTHIS` for 0x40

`DEFAULT` for 0x0

`VARARG` for 0x5

`C` for 0x1

`STDCALL` for 0x2

`THISCALL` for 0x3

`FASTCALL` for 0x4

`SENTINEL` for 0x41 (§23.1.16 and §15.3)

- The first byte of the Signature holds bits for `HASTHIS`, `EXPLICITTHIS` and calling convention – `DEFAULT`, `VARARG`, `C`, `STDCALL`, `THISCALL`, or `FASTCALL`. These are OR'd together.
- *ParamCount* is an integer that holds the number of non-vararg and vararg parameters, combined. It can be any number between 0 and 0x1FFFFFFF The compiler compresses it too (see Partition II Metadata Validation) – before storing into the `blob` (*ParamCount* counts just the method parameters – it does not include the method's return type)
- The *RetType* item describes the type of the method's return value (§23.2.11)
- The first *Param* item describes the type of each of the method's non-vararg parameters. The (optional) second *Param* item describes the type of each of the method's vararg parameters. There shall be *ParamCount* instances of *Param* (§23.2.10).

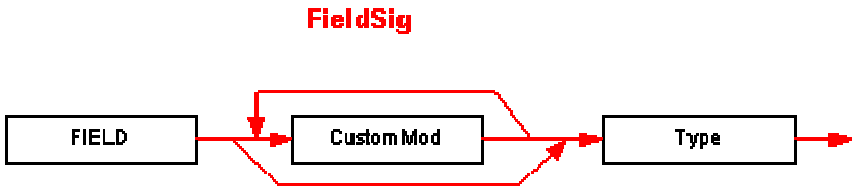
This is the most complex of the various method signatures. Two separate diagrams have been combined into one in this diagram, using shading to distinguish between them. Thus, for the following calling conventions: `DEFAULT` (managed), `STDCALL`, `THISCALL` and `FASTCALL` (unmanaged), the signature ends just before the `SENTINEL` item (these are all non vararg signatures). However, for the managed and unmanaged vararg calling conventions:

`VARARG` (managed) and `C` (unmanaged), the signature can include the `SENTINEL` and final Param items (they are not required, however). These options are indicated by the shading of boxes in the syntax diagram.

 ©ISO/IEC 2006 – All rights reserved

23.2.4 FieldSig

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



This diagram uses the following abbreviations:

`FIELD` for 0x6

CustomMod is defined in §23.2.7. *Type* is defined in §23.2.12

23.2.5 PropertySig

A PropertySig is indexed by the Property.Type column. It captures the type information for a Property – essentially, the signature of its *getter* method:

the number of parameters supplied to its *getter* method

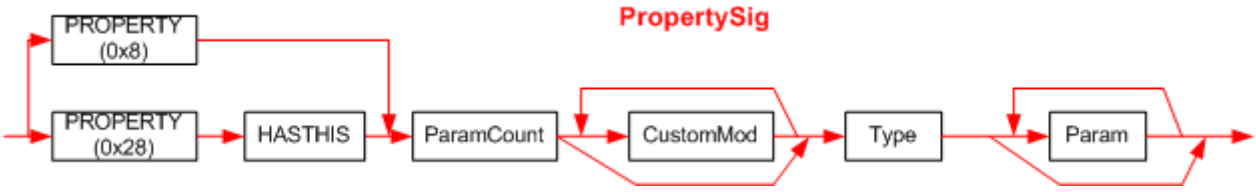
the base type of the Property (the type returned by its *getter* method)

type information for each parameter in the *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 diagram for a PropertySig looks like this:



The first byte of the Signature holds bits for `HASTHIS` and `PROPERTY`. These are OR'd together.

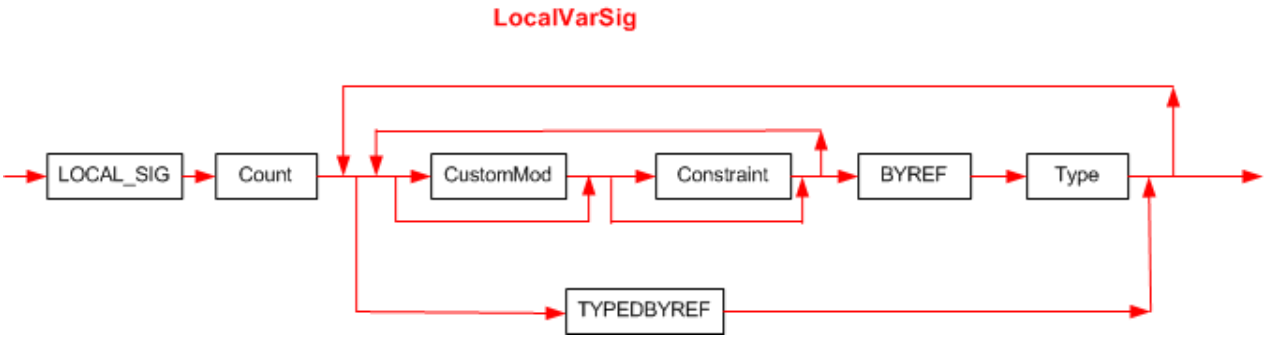
Type specifies the type returned by the *Getter* method for this property. *Type* is defined in §23.2.12. *Param* is defined in §23.2.10.

ParamCount is an integer that holds the number of index parameters in the *getter* methods (0 or more). (§23.2.1) (*ParamCount* counts just the method parameters – it does not include the method's base type of the Property)

23.2.6 LocalVarSig

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

©ISO/IEC 2006 – All rights reserved



This diagram uses the following abbreviations:

`LOCAL_SIG` for 0x7, used for the `.locals` directive, see§15.4.1.3

`BYREF` for `ELEMENT_TYPE_BYREF` (§23.1.16)

Constraint is defined in §23.2.9.

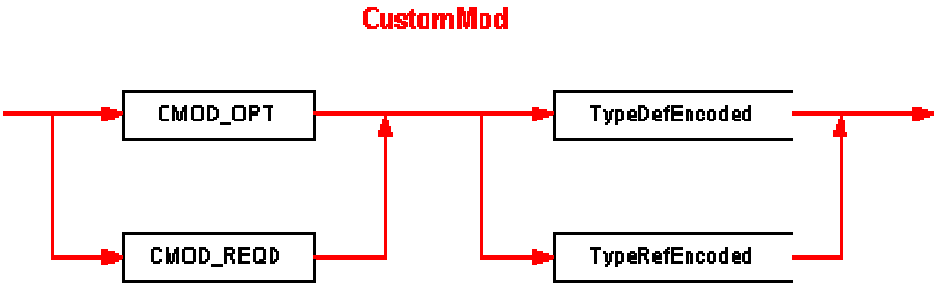
Type is defined in §23.2.12

Count is 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

23.2.7 CustomMod

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



This diagram uses the following abbreviations:

`CMOD_OPT` for `ELEMENT_TYPE_CMOD_OPT` (§23.1.16)

`CMOD_REQD` for `ELEMENT_TYPE_CMOD_REQD` (§23.1.16)

The `CMOD_OPT` or `CMOD_REQD` value is compressed, see §23.2.

The `CMOD_OPT` or `CMOD_REQD` is followed by a metadata token that indexes a row in the *TypeDef* table or the *TypeRef* table. However, these tokens are encoded and compressed – see §23.2.8 for details

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

23.2.8 TypeDefOrRefEncoded

These items are compact ways to store a *TypeDef*, *TypeRef,* or *TypeSpec* token in a Signature (§23.2.12).

Consider a regular *TypeRef* token, such as 0x01000012. The top byte of 0x01 indicates that this is a *TypeRef* token (see Partition VI for a list of the supported metadata token types). The lower 3 bytes (0x000012) index row number 0x12 in the *TypeRef* table.

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

 ©ISO/IEC 2006 – All rights reserved

1. encode the table that this token indexes as the least significant 2 bits. The bit values to use are 0, 1 and 2, specifying the target table is the *TypeDef*, *TypeRef* or *TypeSpec* table, respectively
2. shift the 3-byte row index (0x000012 in this example) left by 2 bits and OR into the 2-bit encoding from step 1
3. compress the resulting value (§23.2). This example yields the following encoded value:

```
a)  encoded = value for TypeRef table = 0x01 (from 1. above)
b)  encoded = ( 0x000012 << 2 ) |  0x01
            = 0x48 | 0x01
            = 0x49
c)  encoded = Compress (0x49)
            = 0x49
```

So, instead of the original, regular *TypeRef* token value of 0x01000012, requiring 4 bytes of space in the Signature 'blob', this TypeRef token is encoded as a single byte.

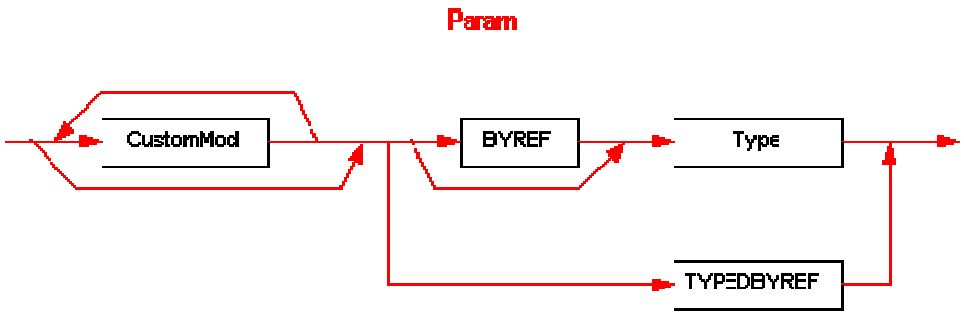
23.2.9 Constraint

The *Constraint* item in Signatures currently has only one possible value, `ELEMENT_TYPE_PINNED` (§23.1.16), which specifies that the target type is pinned in the runtime heap, and will not be moved by the actions of garbage collection.

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

23.2.10 Param

The *Param* (parameter) item in *Signatures* has this syntax diagram:



This diagram uses the following abbreviations:

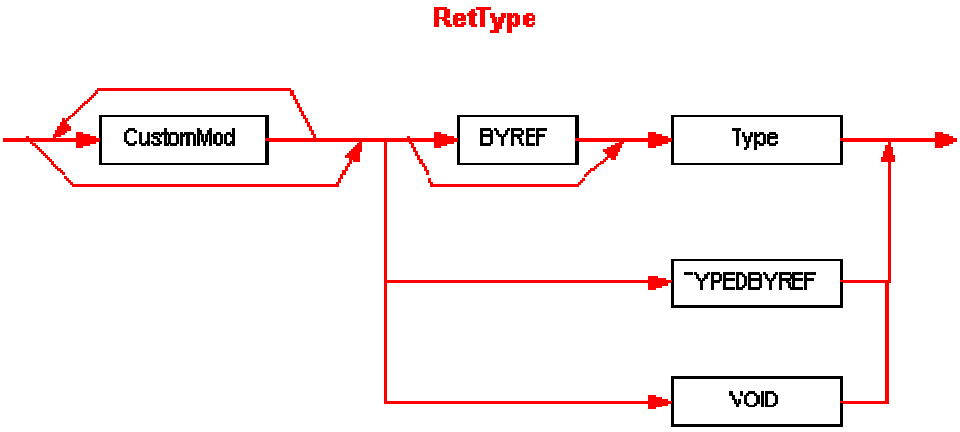
`BYREF` for 0x10 (§23.1.16)

`TYPEDBYREF` for 0x16 (§23.1.16)

CustomMod is defined in §23.2.7. *Type* is defined in §23.2.12

23.2.11 RetType

The *RetType* (return type) item in Signatures has this syntax diagram:



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

`BYREF` for `ELEMENT_TYPE_BYREF` (§23.1.16)

`TYPEDBYREF` for `ELEMENT_TYPE_TYPEDBYREF` (§23.1.16)

`VOID` for `ELEMENT_TYPE_VOID` (§23.1.16)

23.2.12 Type

Type is encoded in signatures as follows (`I1` is an abbreviation for `ELEMENT_TYPE_I1`, `U1` is an abbreviation for `ELEMENT_TYPE_U1`, and so on; see 23.1.16):

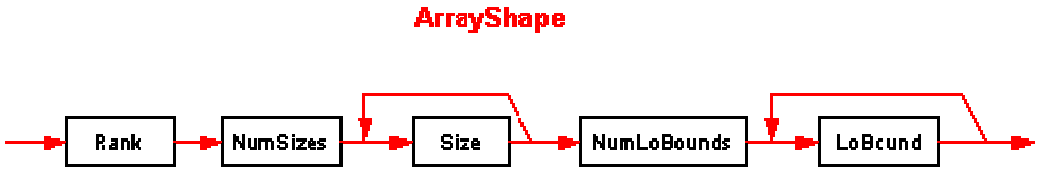
```
Type ::=
BOOLEAN | CHAR | I1 | U1 | I2 | U2 | I4 | U4 | I8 | U8 | R4 | R8 | I | U
| ARRAY Type ArrayShape (general array, see §23.2.13)
| CLASS TypeDefOrRefEncoded
| FNPTR MethodDefSig
| FNPTR MethodRefSig
| GENERICINST (CLASS | VALUETYPE) TypeDefOrRefEncoded GenArgCount Type *
| MVAR number
| OBJECT
| PTR CustomMod* Type
| PTR CustomMod* VOID
| STRING
| SZARRAY CustomMod* Type (single dimensional, zero-based array i.e., vector)
| VALUETYPE TypeDefOrRefEncoded
| VAR number
```

The *GenArgCount* non-terminal is an int32 value (compressed) specifying the number of generic arguments in this signature.

23.2.13 ArrayShape

An ArrayShape has the following syntax diagram:

 ©ISO/IEC 2006 – All rights reserved



Rank is an integer (stored in compressed form, see §23.2) that specifies the number of dimensions in the array (shall be 1 or more). *NumSizes* is a compressed 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 can itself be an array. *end note*]

23.2.14 TypeSpec

The signature in the Blob heap indexed by a *TypeSpec* token has the following format –

```
TypeSpecBlob ::=
  PTR       CustomMod*  VOID
| PTR       CustomMod*  Type
| FNPTR     MethodDefSig
| FNPTR     MethodRefSig
| ARRAY     Type  ArrayShape
| SZARRAY   CustomMod*  Type
| GENERICINST (CLASS | VALUETYPE) TypeDefOrRefEncoded GenArgCount Type Type*
```

For compactness, the `ELEMENT_TYPE_` prefixes have been omitted from this list. So, for example, "PTR" is shorthand for `ELEMENT_TYPE_PTR`. (§23.1.16) Note that a TypeSpecBlob does *not* begin with a calling-convention byte, so it differs from the various other signatures that are stored into Metadata.

23.2.15 MethodSpec

The signature in the Blob heap indexed by a *MethodSpec* token has the following format –

```
MethodSpecBlob ::=
  GENRICINST GenArgCount Type Type*
```

GENRICINST has the value 0x0A. [*Note*: This value is known as `IMAGE_CEE_CS_CALLCONV_GENERICINST` in the Microsoft CLR implementation. *end note*] The *GenArgCount* is a compressed int32 indicating the number of generic arguments in the method. The blob then specifies the instantiated type, repeating a total of *GenArgCount* times.

©ISO/IEC 2006 – All rights reserved

23.2.16 Short form signatures

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

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

short-form: ELEMENT_TYPE_STRING

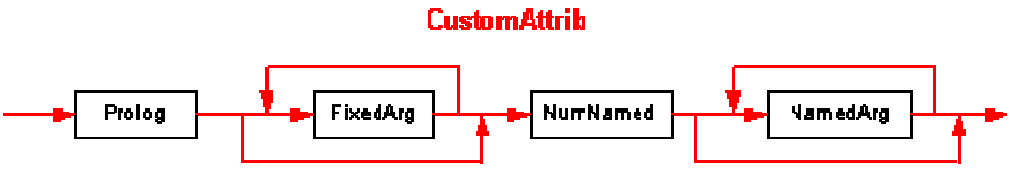
Only the short form is valid. The following table shows which short-forms should be used in place of each long-form item. (As usual, for compactness, the ELEMENT_TYPE_ prefix have been omitted here – so VALUETYPE is short for ELEMENT_TYPE_VALUETYPE)

Long Form		**Short Form**
Prefix	**TypeRef to:**	
CLASS	System.String	STRING
CLASS	System.Object	OBJECT
VALUETYPE	System.Void	VOID
VALUETYPE	System.Boolean	BOOLEAN
VALUETYPE	System.Char	CHAR
VALUETYPE	System.Byte	U1
VALUETYPE	System.Sbyte	I1
VALUETYPE	System.Int16	I2
VALUETYPE	System.UInt16	U2
VALUETYPE	System.Int32	I4
VALUETYPE	System.UInt32	U4
VALUETYPE	System.Int64	I8
VALUETYPE	System.UInt64	U8
VALUETYPE	System.IntPtr	I
VALUETYPE	System.UIntPtr	U
VALUETYPE	System.TypedReference	TYPEDBYREF

[*Note:* arrays shall be encoded in signatures using one of ELEMENT_TYPE_ARRAY or ELEMENT_TYPE_SZARRAY. There is no long form involving a TypeRef to System.Array. *end note*]

23.3 Custom attributes

A Custom Attribute has the following syntax diagram:



All binary values are stored in little-endian format (except *PackedLen* items, which are used only as counts for the number of bytes to follow in a UTF8 string). If there are no fields, parameters, or properties specified the entire attribute is represented as an empty blob.

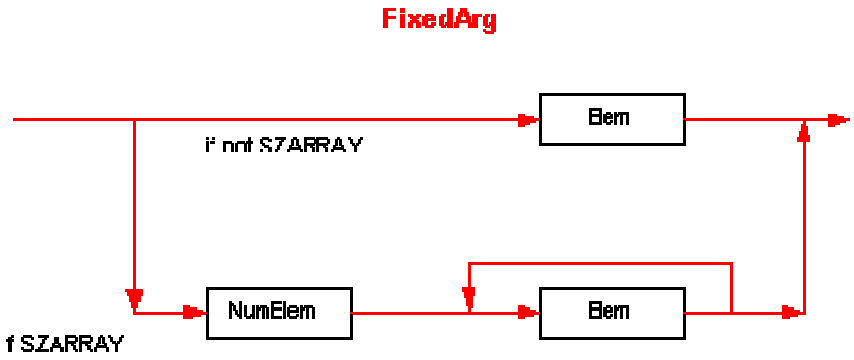
CustomAttrib starts with a *Prolog* – an unsigned *int16*, with value 0x0001.

Next comes a description of the fixed arguments for the constructor method. Their number and type is found by examining that constructor's row in the *MethodDef* table; this information is *not* repeated in the

 ©ISO/IEC 2006 – All rights reserved

CustomAttrib itself. As the syntax diagram shows, there can be zero or more *FixedArg*s. (Note that `VARARG` constructor methods are not allowed in the definition of Custom Attributes.)

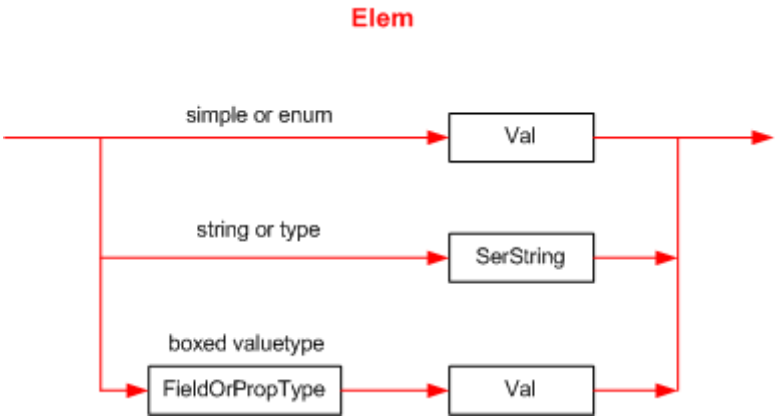
Next is a description of the optional "named" fields and properties. This starts with *NumNamed* – an unsigned *int16* giving the number of "named" properties or fields that follow. Note that *NumNamed* shall always be present. A value of zero indicates that there are no "named" properties or fields to follow (and of course, in this case, the *CustomAttrib* shall end immediately after *NumNamed*). In the case where *NumNamed* is non-zero, it is followed by *NumNamed* repeats of *NamedArg*s.



The format for each *FixedArg* depends upon whether that argument is an `SZARRAY` or not – this is shown in the lower and upper paths, respectively, of the syntax diagram. So each *FixedArg* is either a single *Elem*, or *NumElem* repeats of *Elem*.

(`SZARRAY` is the single byte 0x1D, and denotes a vector – a single-dimension array with a lower bound of zero.)

NumElem is an unsigned *int32* specifying the number of elements in the `SZARRAY`, or 0xFFFFFFFF to indicate that the value is null.



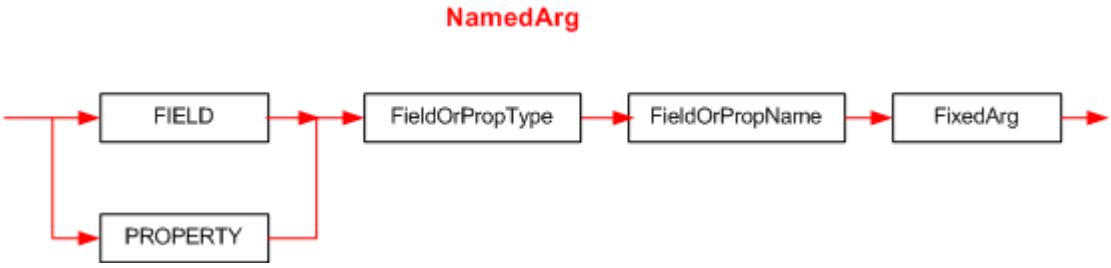
An *Elem* takes one of the forms in this diagram, as follows:

- If the parameter kind is simple (first line in the above diagram) (`bool`, `char`, `float32`, `float64`, `int8`, `int16`, `int32`, `int64`, `unsigned int8`, `unsigned int16`, `unsigned int32` or `unsigned int64`) then the 'blob' contains its binary value (*Val*). (A *bool* is a single byte with value 0 (false) or 1 (true); *char* is a two-byte Unicode character; and the others have their obvious meaning.) This pattern is also used if the parameter kind is an *enum* -- simply store the value of the enum's underlying integer type.
- If the parameter kind is *string*, (middle line in above diagram) then the blob contains a *SerString* – a *PackedLen* count of bytes, followed by the UTF8 characters. If the string is null, its *PackedLen* has the value 0xFF (with no following characters). If the string is empty (""), then *PackedLen* has the value 0x00 (with no following characters).

©ISO/IEC 2006 – All rights reserved

- If the parameter kind is *System.Type*, (also, the middle line in above diagram) its value is stored as a *SerString* (as defined in the previous paragraph), representing its canonical name. The canonical name is its full type name, followed optionally by the assembly where it is defined, its version, culture and public-key-token. If the assembly name is omitted, the CLI looks first in the current assembly, and then in the system library (mscorlib); in these two special cases, it is permitted to omit the assembly-name, version, culture and public-key-token.
- If the parameter kind is *System.Object*, (third line in the above diagram) the value stored represents the "boxed" instance of that value-type. In this case, the blob contains the actual type's *FieldOrPropType* (see below), followed by the argument's unboxed value. [*Note*: it is not possible to pass a value of **null** in this case. *end note*]
- If the type is a boxed simple value type (`bool`, `char`, `float32`, `float64`, `int8`, `int16`, `int32`, `int64`, `unsigned int8`, `unsigned int16`, `unsigned int32` or `unsigned int64`) then *FieldOrPropType* is immediately preceded by a byte containing the value 0x51 .

  The *FieldOrPropType* shall be exactly one of: `ELEMENT_TYPE_BOOLEAN`, `ELEMENT_TYPE_CHAR`, `ELEMENT_TYPE_I1`, `ELEMENT_TYPE_U1`, `ELEMENT_TYPE_I2`, `ELEMENT_TYPE_U2`, `ELEMENT_TYPE_I4`, `ELEMENT_TYPE_U4`, `ELEMENT_TYPE_I8`, `ELEMENT_TYPE_U8`, `ELEMENT_TYPE_R4`, `ELEMENT_TYPE_R8`, `ELEMENT_TYPE_STRING`. A single-dimensional, zero-based array is specified as a single byte 0x1D followed by the *FieldOrPropType* of the element type. (See §23.1.16) An enum is specified as a single byte 0x55 followed by a *SerString*.



A *NamedArg* is simply a *FixedArg* (discussed above) preceded by information to identify which field or property it represents. [*Note*: Recall that the CLI allows fields and properties to have the same name; so we require a means to disambiguate such situations. *end note*]

`FIELD` is the single byte 0x53.

`PROPERTY` is the single byte 0x54.

The *FieldOrPropName* is the name of the field or property, stored as a *SerString* (defined above).

A number of examples involving custom attributes are contained in Annex B of Partition VI.

23.4 Marshalling descriptors

A Marshalling Descriptor is like a signature – it's a 'blob' of binary data. It describes how a field or parameter (which, as usual, covers the method return, as parameter number 0) should be marshalled when calling to or from unmanaged code via PInvoke dispatch. The ILAsm syntax `marshal` can be used to create a marshalling descriptor, as can the pseudo custom attribute *MarshalAsAttribute* – see §21.2.1)

Note that a conforming implementation of the CLI need only support marshalling of the types specified earlier – see §15.5.4.

Marshalling descriptors make use of constants named NATIVE_TYPE_xxx. Their names and values are listed in the following table:

Name	Value
`NATIVE_TYPE_BOOLEAN`	`0x02`
`NATIVE_TYPE_I1`	`0x03`

 ©ISO/IEC 2006 – All rights reserved

NATIVE_TYPE_U1	0x04
NATIVE_TYPE_I2	0x05
NATIVE_TYPE_U2	0x06
NATIVE_TYPE_I4	0x07
NATIVE_TYPE_U4	0x08
NATIVE_TYPE_I8	0x09
NATIVE_TYPE_U8	0x0a
NATIVE_TYPE_R4	0x0b
NATIVE_TYPE_R8	0x0c
NATIVE_TYPE_LPSTR	0x14
NATIVE_TYPE_LPWSTR	0x15
NATIVE_TYPE_INT	0x1f
NATIVE_TYPE_UINT	0x20
NATIVE_TYPE_FUNC	0x26
NATIVE_TYPE_ARRAY	0x2a

The 'blob' has the following format –

```
MarshalSpec ::=
  NativeIntrinsic
| ARRAY ArrayElemType
| ARRAY ArrayElemType ParamNum
| ARRAY ArrayElemType ParamNum NumElem

NativeIntrinsic ::=
  BOOLEAN | I1 | U1 | I2 | U2 | I4 | U4 | I8 | U8 | R4 | R8
| LPSTR | LPSTR | INT | UINT | FUNC
```

For compactness, the NATIVE_TYPE_ prefixes have been omitted in the above lists; for example, "ARRAY" is shorthand for NATIVE_TYPE_ARRAY.

```
ArrayElemType ::=
   NativeIntrinsic
```

ParamNum is an integer (compressed as described in §23.2) specifying the parameter in the method call that provides the number of elements in the array – see below.

NumElem is an integer compressed as described in §23.2 (specifying the number of elements or additional elements – see below).

[*Note:* For example, in the method declaration:

```
.method void M(int32[] ar1, int32 size1, unsigned int8[] ar2, int32 size2) { … }
```

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

ARRAY MAX 2 1

This says the parameter is marshalled to a NATIVE_TYPE_ARRAY. There is no additional info about the type of each element (signified by that NATIVE_TYPE_MAX). The value of *ParamNum* is 2, which indicates that parameter number 2 in the method (the one called size1) will specify the number of elements in the actual array – let's suppose its value on a particular call is 42. The value of *NumElem* is 0. The calculated total size, in bytes, of the array is given by the formula:

```
if ParamNum = 0
   SizeInBytes = NumElem * sizeof (elem)
else
   SizeInBytes = ( @ParamNum +  NumElem ) * sizeof (elem)
endif
```

The syntax "*@ParamNum*" is used here to denote the value passed in for parameter number *ParamNum* – it would be 42 in this example. The size of each element is calculated from the metadata for the `ar1` parameter in *Foo*'s signature – an `ELEMENT_TYPE_I4` (§23.1.16) of size 4 bytes. *end note*]

 ©ISO/IEC 2006 – All rights reserved

24 Metadata physical layout

The physical on-disk representation of metadata is a direct reflection of the logical representation described in §22 and §23. That is, data is stored in streams representating the metadata tables and heaps. The main complication is that, where the logical representation is abstracted from the number of bytes needed for indexing into tables and columns, the physical representation has to take care of that explicitly by defining how to map logical metadata heaps and tables into their physical representations.

Unless stated otherwise, all binary values are stored in little-endian format.

24.1 Fixed fields

Complete CLI components (metadata and CIL instructions) are stored in a subset of the current Portable Executable (PE) File Format (§25). Because of this heritage, some of the fields in the physical representation of metadata have fixed values. When writing these fields it is best that they be set to the value indicated, on reading they should be ignored.

24.2 File headers

24.2.1 Metadata root

The root of the physical metadata starts with a magic signature, several bytes of version and other miscellaneous information, followed by a count and an array of stream headers, one for each stream that is present. The actual encoded tables and heaps are stored in the streams, which immediately follow this array of headers.

Offset	Size	Field	Description
0	4	**Signature**	Magic signature for physical metadata : 0x424A5342.
4	2	**MajorVersion**	Major version, 1 (ignore on read)
6	2	**MinorVersion**	Minor version, 1 (ignore on read)
8	4	**Reserved**	Reserved, always 0 (§24.1).
12	4	**Length**	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 (§24.1).
x+2	2	**Streams**	Number of streams, say *n*.
x+4		**StreamHeaders**	Array of *n* StreamHdr structures.

The Version string shall be “Standard CLI 2005” for any file that is intended to be executed on any conforming implementation of the CLI, and all conforming implementations of the CLI shall accept files that use this version string. Other strings shall be used when the file is restricted to a vendor-specific implementation of the CLI. Future versions of this standard shall specify different strings, but they shall begin “Standard CLI”. Other standards that specify additional functionality shall specify their own specific version strings beginning with “Standard□”, where “□” represents a single space. Vendors that provide implementation-specific extensions shall provide a version string that does *not* begin with “Standard□”. (For the first version of this Standard, the Version string was “Standard CLI 2002”.)

24.2.2 Stream header

A stream header gives the names, and the position and length of a particular table or heap. Note that the length of a Stream header structure is not fixed, but depends on the length of its name field (a variable length null-terminated string).

Offset	Size	Field	Description
0	4	**Offset**	Memory offset to start of this stream from start of the metadata root (§24.2.1)
4	4	**Size**	Size of this stream in bytes, shall be a multiple of 4.
8		**Name**	Name of the stream as null-terminated variable length array of ASCII characters, padded to the next 4-byte boundary with \0 characters. The name is limited to 32 characters.

Both logical tables and heaps are stored in streams. There are five possible kinds of streams. A stream header with name "#Strings" that points to the physical representation of the string heap where identifier strings are stored; a stream header with name "#US" that points to the physical representation of the user string heap; a stream header with name "#Blob" that points to the physical representation of the blob heap, a stream header with name "#GUID" that points to the physical representation of the GUID heap; and a stream header with name "#~" that points to the physical representation of a set of tables.

Each kind of stream shall occur at most once, that is, a meta-data file shall not contain two "#US" streams, or five "#Blob" streams. Streams need not be there if they are empty.

The next subclauses describe the structure of each kind of stream in more detail.

24.2.3 #Strings heap

The stream of bytes pointed to by a "#Strings" header is the physical representation of the logical string heap. The physical heap can contain garbage, that is, it can contain parts that are unreachable from any of the tables, but parts that are reachable from a table shall contain a valid null-terminated UTF8 string. When the #String heap is present, the first entry is always the empty string (i.e., \0).

24.2.4 #US and #Blob heaps

The stream of bytes pointed to by a "#US" or "#Blob" header are the physical representation of logical Userstring and 'blob' heaps respectively. Both these heaps can contain garbage, as long as any part that is reachable from any of the tables contains a valid 'blob'. Individual blobs are stored with their length encoded in the first few bytes:

- If the first one byte of the 'blob' is $0bbbbbbb_2$, then the rest of the 'blob' contains the $bbbbbbb_2$ bytes of actual data.
- If the first two bytes of the 'blob' are $10bbbbbb_2$ and x, then the rest of the 'blob' contains the ($bbbbbb_2$ << 8 + x) bytes of actual data.
- If the first four bytes of the 'blob' are $110bbbbb_2$, x, y, and z, then the rest of the 'blob' contains the ($bbbbb_2$ << 24 + x << 16 + y << 8 + z) bytes of actual data.

The first entry in both these heaps is the empty 'blob' that consists of the single byte 0x00.

Strings in the #US (user string) heap are encoded using 16-bit Unicode encodings. The count on each string is the number of bytes (not characters) in the string. Furthermore, there is an additional terminal byte (so all byte counts are odd, not even). This final byte holds the value 1 if and only if any UTF16 character within the string has any bit set in its top byte, or its low byte is any of the following: 0x01–0x08, 0x0E–0x1F, 0x27, 0x2D, 0x7F. Otherwise, it holds 0. The 1 signifies Unicode characters that require handling beyond that normally provided for 8-bit encoding sets.

 ©ISO/IEC 2006 – All rights reserved

24.2.5 #GUID heap

The "#GUID" header points to a sequence of 128-bit GUIDs. There might be unreachable GUIDs stored in the stream.

24.2.6 #~ stream

The "#~" streams contain the actual physical representations of the logical metadata tables (§22). A "#~" stream has the following top-level structure:

Offset	Size	Field	Description
0	4	**Reserved**	Reserved, always 0 (§24.1).
4	1	**MajorVersion**	Major version of table schemata; shall be 2 (§24.1).
5	1	**MinorVersion**	Minor version of table schemata; shall be 0 (§24.1).
6	1	**HeapSizes**	Bit vector for heap sizes.
7	1	**Reserved**	Reserved, always 1 (§24.1).
8	8	**Valid**	Bit vector of present tables, let *n* be the number of bits that are 1.
16	8	**Sorted**	Bit vector of sorted tables.
24	4*n	**Rows**	Array of *n* 4-byte unsigned integers indicating the number of rows for each present table.
24+4*n		**Tables**	The sequence of physical tables.

The HeapSizes field is a bitvector that encodes the width of indexes into the various heaps. If bit 0 is set, indexes into the "#String" heap are 4 bytes wide; if bit 1 is set, indexes into the "#GUID" heap are 4 bytes wide; if bit 2 is set, indexes into the "#Blob" heap are 4 bytes wide. Conversely, if the HeapSize bit for a particular heap is not set, indexes into that heap are 2 bytes wide.

Heap size flag	Description
0x01	Size of "#String" stream >= 2^{16}.
0x02	Size of "#GUID" stream >= 2^{16}.
0x04	Size of "#Blob" stream >= 2^{16}.

The Valid field is a 64-bit bitvector that has a specific bit set for each table that is stored in the stream; the mapping of tables to indexes is given at the start of §22. For example when the DeclSecurity table is present in the logical metadata, bit 0x0e should be set in the Valid vector. It is invalid to include non-existent tables in Valid, so all bits above 0x2c shall be zero.

The Rows array contains the number of rows for each of the tables that are present. When decoding physical metadata to logical metadata, the number of 1's in Valid indicates the number of elements in the Rows array.

A crucial aspect in the encoding of a logical table is its *schema*. The schema for each table is given in §22. For example, the table with assigned index 0x02 is a *TypeDef* table, which, according to its specification in §22.37, has the following columns: a 4-byte-wide flags, an index into the String heap, another index into the String heap, an index into *TypeDef* , *TypeRef* , or *TypeSpec* table, an index into *Field* table, and an index into *MethodDef* table.

The physical representation of a table with *n* columns and *m* rows with schema $(C_0,\ldots,C_{n-1})$ consists of the concatenation of the physical representation of each of its rows. The physical representation of a row with schema $(C_0,\ldots,_{n-1})$ is the concatenation of the physical representation of each of its elements. The physical representation of a row cell *e* at a column with type *C* is defined as follows:

- If e is a constant, it is stored using the number of bytes as specified for its column type *C* (i.e., a 2-bit mask of type PropertyAttributes)
- If *e* is an index into the GUID heap, 'blob', or String heap, it is stored using the number of bytes as defined in the HeapSizes field.
- If *e* is a simple index into a table with index *i*, it is stored using 2 bytes if table *i* has less than 2^{16} rows, otherwise it is stored using 4 bytes.
- If *e* is a *coded index* that points into table t_i out of *n* possible tables $t_0, \ldots t_{n-1}$, then it is stored as e << (log n) | tag{ $t_0, \ldots t_{n-1}$}[t_i] using 2 bytes if the maximum number of rows of tables $t_0, \ldots t_{n-1}$, is less than $2^{(16 - (\log n))}$, and using 4 bytes otherwise. The family of finite maps tag{ $t_0, \ldots t_{n-1}$} is defined below. Note that decoding a physical row requires the inverse of this mapping. [For example, the *Parent* column of the *Constant* table indexes a row in the *Field, Param,* or *Property* tables. The actual table is encoded into the low 2 bits of the number, using the values: 0 => *Field,* 1 => *Param,* 2 => *Property*.The remaining bits hold the actual row number being indexed. For example, a value of 0x321, indexes row number 0xC8 in the *Param* table.]

TypeDefOrRef: 2 bits to encode tag	**Tag**
TypeDef	0
TypeRef	1
TypeSpec	2

HasConstant: 2 bits to encode tag	**Tag**
Field	0
Param	1
Property	2

HasCustomAttribute: 5 bits to encode tag	**Tag**
MethodDef	0
Field	1
TypeRef	2
TypeDef	3
Param	4
InterfaceImpl	5
MemberRef	6
Module	7
Permission	8
Property	9
Event	10
StandAloneSig	11
ModuleRef	12
TypeSpec	13
Assembly	14
AssemblyRef	15
File	16
ExportedType	17

 ©ISO/IEC 2006 – All rights reserved

ManifestResource	18

[*Note*: HasCustomAttributes only has values for tables that are "externally visible"; that is, that correspond to items in a user source program. For example, an attribute can be attached to a TypeDef table and a Field table, but not a ClassLayout table. As a result, some table types are missing from the enum above. *end note*]

HasFieldMarshall: 1 bit to encode tag	**Tag**
Field	0
Param	1

HasDeclSecurity: 2 bits to encode tag	**Tag**
TypeDef	0
MethodDef	1
Assembly	2

MemberRefParent: 3 bits to encode tag	**Tag**
TypeDef	0
TypeRef	1
ModuleRef	2
MethodDef	3
TypeSpec	4

HasSemantics: 1 bit to encode tag	**Tag**
Event	0
Property	1

MethodDefOrRef: 1 bit to encode tag	**Tag**
MethodDef	0
MemberRef	1

MemberForwarded: 1 bit to encode tag	**Tag**
Field	0
MethodDef	1

Implementation: 2 bits to encode tag	**Tag**
File	0
AssemblyRef	1
ExportedType	2

CustomAttributeType: 3 bits to encode tag	**Tag**
Not used	0
Not used	1
MethodDef	2
MemberRef	3

Not used	4

ResolutionScope: 2 bits to encode tag	**Tag**
Module	0
ModuleRef	1
AssemblyRef	2
TypeRef	3

TypeOrMethodDef: 1 bit to encode tag	**Tag**
TypeDef	0
MethodDef	1

 ©ISO/IEC 2006 – All rights reserved

25 File format extensions to PE

This contains informative text only

The file format for CLI components is a strict extension of the current Portable Executable (PE) File Format. This extended PE format enables the operating system to recognize runtime images, accommodates code emitted as CIL or native code, and accommodates runtime metadata as an integral part of the emitted code. There are also specifications for a subset of the full Windows PE/COFF file format, in sufficient detail that a tool or compiler can use the specifications to emit valid CLI images.

The PE format frequently uses the term RVA (Relative Virtual Address). An RVA is the address of an item *once loaded into memory*, with the base address of the image file subtracted from it (i.e., the offset from the base address where the file is loaded). The RVA of an item will almost always differ from its position within the file on disk. To compute the file position of an item with RVA *r*, search all the sections in the PE file to find the section with RVA *s*, length *l* and file position *p* in which the RVA lies, ie $s \leq r < s+l$. The file position of the item is then given by $p+(r-s)$.

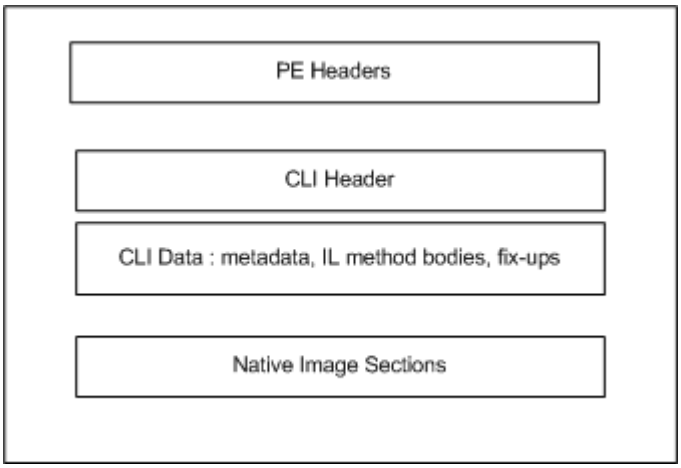
Unless stated otherwise, all binary values are stored in little-endian format.

End informative text

25.1 Structure of the runtime file format

The figure below provides a high-level view of the CLI file format. All runtime images contain the following:

- PE headers, with specific guidelines on how field values should be set in a runtime file.
- A CLI header that contains all of the runtime specific data entries. The runtime header is read-only and shall be placed in any read-only section.
- The sections that contain the actual data as described by the headers, including imports/exports, data, and code.



The CLI header (§25.3.3) is found using CLI Header directory entry in the PE header. The CLI header in turn contains the address and sizes of the runtime data (for metadata, see §24; for CIL see § 25.4) in the rest of the image. Note that the runtime data can be merged into other areas of the PE format with the other data based on the attributes of the sections (such as read only versus execute, etc.).

25.2 PE headers

A PE image starts with an MS-DOS header followed by a PE signature, followed by the PE file header, and then the PE optional header followed by PE section headers.

25.2.1 MS-DOS header

The PE format starts with an MS-DOS stub of exactly the following 128 bytes to be placed at the front of the module. At offset 0x3c in the DOS header is a 4-byte unsigned integer offset, lfanew, to the PE signature (shall be “PE\0\0”), immediately followed by the PE file header.

0x4d	0x5a	0x90	0x00	0x03	0x00	0x00	0x00
0x04	0x00	0x00	0x00	0xFF	0xFF	0x00	0x00
0xb8	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x40	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00	*lfanew*			
0x0e	0x1f	0xba	0x0e	0x00	0xb4	0x09	0xcd
0x21	0xb8	0x01	0x4c	0xcd	0x21	0x54	0x68
0x69	0x73	0x20	0x70	0x72	0x6f	0x67	0x72
0x61	0x6d	0x20	0x63	0x61	0x6e	0x6e	0x6f
0x74	0x20	0x62	0x65	0x20	0x72	0x75	0x6e
0x20	0x69	0x6e	0x20	0x44	0x4f	0x53	0x20
0x6d	0x6f	0x64	0x65	0x2e	0x0d	0x0d	0x0a
0x24	0x00	0x00	0x00	0x00	0x00	0x00	0x00

25.2.2 PE file header

Immediately after the PE signature is the PE File header consisting of the following:

Offset	Size	Field	Description
0	2	Machine	Always 0x14c (§24.1).
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 (§24.1).
12	4	Number of Symbols	Always 0 (§24.1).
16	2	Optional Header Size	Size of the optional header, the format is described below.
18	2	Characteristics	Flags indicating attributes of the file, see §25.2.2.1.

25.2.2.1 Characteristics

A CIL-only DLL sets flag 0x2000 to 1, while a CIL-only .exe has flag 0x2000 set to zero:

Flag	Value	Description
IMAGE_FILE_DLL	0x2000	The image file is a dynamic-link library (DLL).

Except for the IMAGE_FILE_DLL flag (0x2000), flags 0x0002, 0x0004, 0x008, and 0x0100 shall all be set, while all others shall always be zero (§24.1).

 ©ISO/IEC 2006 – All rights reserved

25.2.3 PE optional header

Immediately after the PE Header is the PE Optional Header. This header contains the following information:

Offset	Size	Header part	Description
0	28	Standard fields	These define general properties of the PE file, see §25.2.3.1.
28	68	NT-specific fields	These include additional fields to support specific features of Windows, see 25.2.3.2.
96	128	Data directories	These fields are address/size pairs for special tables, found in the image file (for example, Import Table and Export Table).

25.2.3.1 PE header standard fields

These fields are required for all PE files and contain the following information:

Offset	Size	Field	Description
0	2	Magic	Always 0x10B (§24.1).
2	1	LMajor	Always 6 (§24.1).
3	1	LMinor	Always 0 (§24.1).
4	4	Code Size	Size of the code (text) section, or the sum of all code sections if there are multiple sections.
8	4	Initialized Data Size	Size of the initialized data section, or the sum of all such sections if there are multiple data sections.
12	4	Uninitialized Data Size	Size of the uninitialized data section, or the sum of all such sections if there are multiple unitinitalized data sections.
16	4	Entry Point RVA	RVA of entry point , needs to point to bytes 0xFF 0x25 followed by the RVA in a section marked execute/read for EXEs or 0 for DLLs
20	4	Base Of Code	RVA of the code section. (This is a hint to the loader.)
24	4	Base Of Data	RVA of the data section. (This is a hint to the loader.)

This contains informative text only

The entry point RVA shall always be either the x86 entry point stub or be 0. On non-CLI aware platforms, this stub will call the entry point API of mscoree (_CorExeMain or _CorDllMain). The mscoree entry point will use the module handle to load the metadata from the image, and invoke the entry point specified in vthe CLI header.

End informative text

25.2.3.2 PE header Windows NT-specific fields

These fields are Windows NT specific:

Offset	Size	Field	Description
28	4	Image Base	Always 0x400000 (§24.1).
32	4	Section Alignment	Always 0x2000 (§24.1).
36	4	File Alignment	Either 0x200 or 0x1000.
40	2	OS Major	Always 4 (§24.1).

©ISO/IEC 2006 – All rights reserved

42	2	OS Minor	Always 0 (§24.1).
44	2	User Major	Always 0 (§24.1).
46	2	User Minor	Always 0 (§24.1).
48	2	SubSys Major	Always 4 (§24.1).
50	2	SubSys Minor	Always 0 (§24.1).
52	4	Reserved	Always 0 (§24.1).
56	4	Image Size	Size, in bytes, of image, including all headers and padding; shall be a multiple of Section Alignment.
60	4	Header Size	Combined size of MS-DOS Header, PE Header, PE Optional Header and padding; shall be a multiple of the file alignment.
64	4	File Checksum	Always 0 (§24.1).
68	2	SubSystem	Subsystem required to run this image. Shall be either IMAGE_SUBSYSTEM_WINDOWS_CE_GUI (0x3) or IMAGE_SUBSYSTEM_WINDOWS_GUI (0x2).
70	2	DLL Flags	Always 0 (§24.1).
72	4	Stack Reserve Size	Always 0x100000 (1Mb) (§24.1).
76	4	Stack Commit Size	Always 0x1000 (4Kb) (§24.1).
80	4	Heap Reserve Size	Always 0x100000 (1Mb) (§24.1).
84	4	Heap Commit Size	Always 0x1000 (4Kb) (§24.1).
88	4	Loader Flags	Always 0 (§24.1)
92	4	Number of Data Directories	Always 0x10 (§24.1).

25.2.3.3 PE header data directories

The optional header data directories give the address and size of several tables that appear in the sections of the PE file. Each data directory entry contains the RVA and Size of the structure it describes, in that order.

Offset	Size	Field	Description
96	8	Export Table	Always 0 (§24.1).
104	8	Import Table	RVA and Size of Import Table, (§25.3.1).
112	8	Resource Table	Always 0 (§24.1).
120	8	Exception Table	Always 0 (§24.1).
128	8	Certificate Table	Always 0 (§24.1).
136	8	Base Relocation Table	Relocation Table; set to 0 if unused (§25.3.1).
144	8	Debug	Always 0 (§24.1).
152	8	Copyright	Always 0 (§24.1).
160	8	Global Ptr	Always 0 (§24.1).
168	8	TLS Table	Always 0 (§24.1).
176	8	Load Config Table	Always 0 (§24.1).

 ©ISO/IEC 2006 – All rights reserved

184	8	Bound Import	Always 0 (§24.1).
192	8	IAT	RVA and Size of Import Address Table, (§25.3.1).
200	8	Delay Import Descriptor	Always 0 (§24.1).
208	8	CLI Header	CLI Header with directories for runtime data, (§25.3.1).
216	8	Reserved	Always 0 (§24.1).

The tables pointed to by the directory entries are stored in one of the PE file's sections; these sections themselves are described by section headers.

25.3 Section headers

Immediately following the optional header is the Section Table, which contains a number of section headers. This positioning is required because the file header does not contain a direct pointer to the section table; the location of the section table is determined by calculating the location of the first byte after the headers.

Each section header has the following format, for a total of 40 bytes per entry:

Offset	Size	Field	Description
0	8	Name	An 8-byte, null-padded ASCII string. There is no terminating null if the string is exactly eight characters long.
8	4	VirtualSize	Total size of the section in bytes. If this value is greater than SizeOfRawData, the section is zero-padded.
12	4	VirtualAddress	For executable images this is the address of the first byte of the section, when loaded into memory, relative to the image base.
16	4	SizeOfRawData	Size of the initialized data on disk in bytes, shall be a multiple of FileAlignment from the PE header. If this is less than VirtualSize the remainder of the section is zero filled. Because this field is rounded while the VirtualSize field is not it is possible for this to be greater than VirtualSize as well. When a section contains only uninitialized data, this field should be 0.
20	4	PointerToRawData	Offset of section's first page within the PE file. This shall be a multiple of FileAlignment from the optional header. When a section contains only uninitialized data, this field should be 0.
24	4	PointerToRelocations	RVA of Relocation section.
28	4	PointerToLinenumbers	Always 0 (§24.1).
32	2	NumberOfRelocations	Number of relocations, set to 0 if unused.
34	2	NumberOfLinenumbers	Always 0 (§24.1).
36	4	Characteristics	Flags describing section's characteristics, see below.

The following table defines the possible characteristics of the section.

Flag	Value	Description
IMAGE_SCN_CNT_CODE	0x00000020	Section contains executable code.
IMAGE_SCN_CNT_INITIALIZED_DATA	0x00000040	Section contains initialized data.
IMAGE_SCN_CNT_UNINITIALIZED_DATA	0x00000080	Section contains uninitialized data.
IMAGE_SCN_MEM_EXECUTE	0x20000000	Section can be executed as code.

©ISO/IEC 2006 – All rights reserved

IMAGE_SCN_MEM_READ	0x40000000	Section can be read.
IMAGE_SCN_MEM_WRITE	0x80000000	Section can be written to.

25.3.1 Import Table and Import Address Table (IAT)

The Import Table and the Import Address Table (IAT) are used to import the _CorExeMain (for a .exe) or _CorDllMain (for a .dll) entries of the runtime engine (mscoree.dll). The Import Table directory entry points to a one element zero terminated array of Import Directory entries (in a general PE file there is one entry for each imported DLL):

Offset	Size	Field	Description
0	4	ImportLookupTable	RVA of the Import Lookup Table
4	4	DateTimeStamp	Always 0 (§24.1).
8	4	ForwarderChain	Always 0 (§24.1).
12	4	Name	RVA of null-terminated ASCII string “mscoree.dll”.
16	4	ImportAddressTable	RVA of Import Address Table (this is the same as the RVA of the IAT descriptor in the optional header).
20	20		End of Import Table. Shall be filled with zeros.

The Import Lookup Table and the Import Address Table (IAT) are both one element, zero terminated arrays of RVAs into the Hint/Name table. Bit 31 of the RVA shall be set to 0. In a general PE file there is one entry in this table for every imported symbol.

Offset	Size	Field	Description
0	4	Hint/Name Table RVA	A 31-bit RVA into the Hint/Name Table. Bit 31 shall be set to 0 indicating import by name.
4	4		End of table, shall be filled with zeros.

The IAT should be in an executable and writable section as the loader will replace the pointers into the Hint/Name table by the actual entry points of the imported symbols.

The Hint/Name table contains the name of the dll-entry that is imported.

Offset	Size	Field	Description
0	2	Hint	Shall be 0.
2	variable	Name	Case sensitive, null-terminated ASCII string containing name to import. Shall be “_CorExeMain” for a .exe file and “_CorDllMain” for a .dll file.

25.3.2 Relocations

In a pure CIL image, a single fixup of type IMAGE_REL_BASED_HIGHLOW (0x3) is required for the x86 startup stub which access the IAT to load the runtime engine on down level loaders. When building a mixed CIL/native image or when the image contains embedded RVAs in user data, the relocation section contains relocations for these as well.

The relocations shall be in their own section, named “.reloc”, which shall be the final section in the PE file. The relocation section contains a Fix-Up Table. The fixup table is broken into blocks of fixups. Each block represents the fixups for a 4K page, and each block shall start on a 32-bit boundary.

Each fixup block starts with the following structure:

Offset	Size	Field	Description

 ©ISO/IEC 2006 – All rights reserved

0	4	PageRVA	The RVA of the block in which the fixup needs to be applied. The low 12 bits shall be zero.
4	4	Block Size	Total number of bytes in the fixup block, including the Page RVA and Block Size fields, as well as the Type/Offset fields that follow, rounded up to the next multiple of 4.

The Block Size field is then followed by (BlockSize –8)/2 Type/Offset. Each entry is a word (2 bytes) and has the following structure (if necessary, insert 2 bytes of 0 to pad to a multiple of 4 bytes in length):

Offset	Size	Field	Description
0	4 bits	Type	Stored in high 4 bits of word. Value indicating which type of fixup is to be applied (described above)
0	12 bits	Offset	Stored in remaining 12 bits of word. Offset from starting address specified in the Page RVA field for the block. This offset specifies where the fixup is to be applied.

25.3.3 CLI header

The CLI header contains all of the runtime-specific data entries and other information. The header should be placed in a read-only, sharable section of the image. This header is defined as follows:

Offset	Size	Field	Description
0	4	Cb	Size of the header in bytes
4	2	MajorRuntimeVersion	The minimum version of the runtime required to run this program, currently 2.
6	2	MinorRuntimeVersion	The minor portion of the version, currently 0.
8	8	MetaData	RVA and size of the physical metadata (§24).
16	4	Flags	Flags describing this runtime image. (§25.3.3.1).
20	4	EntryPointToken	Token for the *MethodDef* or File of the entry point for the image
24	8	Resources	RVA and size of implementation-specific resources.
32	8	StrongNameSignature	RVA of the hash data for this PE file used by the CLI loader for binding and versioning
40	8	CodeManagerTable	Always 0 (§24.1).
48	8	VTableFixups	RVA of an array of locations in the file that contain an array of function pointers (e.g., vtable slots), see below.
56	8	ExportAddressTableJumps	Always 0 (§24.1).
64	8	ManagedNativeHeader	Always 0 (§24.1).

25.3.3.1 Runtime flags

The following flags describe this runtime image and are used by the loader.

Flag	Value	Description
COMIMAGE_FLAGS_ILONLY	0x00000001	Always 1 (§24.1).

COMIMAGE_FLAGS_32BITREQUIRED	0x00000002	Image can only be loaded into a 32-bit process, for instance if there are 32-bit vtablefixups, or casts from native integers to int32. CLI implementations that have 64-bit native integers shall refuse loading binaries with this flag set.
COMIMAGE_FLAGS_STRONGNAMESIGNED	0x00000008	Image has a strong name signature.
COMIMAGE_FLAGS_TRACKDEBUGDATA	0x00010000	Always 0 (§24.1).

25.3.3.2 Entry point metadata token

- The entry point token (§15.4.1.2) is always a *MethodDef* token (§22.26) or File token (§22.19) when the entry point for a multi-module assembly is not in the manifest assembly. The signature and implementation flags in metadata for the method indicate how the entry is run

25.3.3.3 Vtable fixup

Certain languages, which choose not to follow the common type system runtime model, can have virtual functions which need to be represented in a v-table. These v-tables are laid out by the compiler, not by the runtime. Finding the correct v-table slot and calling indirectly through the value held in that slot is also done by the compiler. The **VtableFixups** field in the runtime header contains the location and size of an array of Vtable Fixups (§15.5.1). V-tables shall be emitted into a *read-write* section of the PE file.

Each entry in this array describes a contiguous array of v-table slots of the specified size. Each slot starts out initialized to the metadata token value for the method they need to call. At image load time, the runtime Loader will turn each entry into a pointer to machine code for the CPU and can be called directly.

Offset	Size	Field	Description
0	4	**VirtualAddress**	RVA of Vtable
4	2	**Size**	Number of entries in Vtable
6	2	**Type**	Type of the entries, as defined in table below

Constant	Value	Description
COR_VTABLE_32BIT	0x01	Vtable slots are 32 bits.
COR_VTABLE_64BIT	0x02	Vtable slots are 64 bits.
COR_VTABLE_FROM_UNMANAGED	0x04	Transition from unmanaged to managed code.
COR_VTABLE_CALL_MOST_DERIVED	0x10	Call most derived method described by the token (only valid for virtual methods).

25.3.3.4 Strong name signature

This header entry points to the strong name hash for an image that can be used to deterministically identify a module from a referencing point (§6.2.1.3).

25.4 Common Intermediate Language physical layout

This section contains the layout of the data structures used to describe a CIL method and its exceptions. Method bodies can be stored in any read-only section of a PE file. The *MethodDef* (§22.26) records in metadata carry each method's RVA.

A method consists of a method header immediately followed by the method body, possibly followed by extra method data sections (§25.4.5), typically exception handling data. If exception-handling data is present, then

 ©ISO/IEC 2006 – All rights reserved

CorILMethod_MoreSects flag (§25.4.4) shall be specified in the method header and for each chained item after that.

There are two flavors of method headers - tiny (§25.4.2) and fat (§25.4.3). The two least significant bits in a method header indicate which type is present (§25.4.1). The tiny header is 1 byte long and stores only the method's code size. A method is given a tiny header if it has no local variables, maxstack is 8 or less, the method has no exceptions, the method size is less than 64 bytes, and the method has no flags above 0x7. Fat headers carry full information - local vars signature token, maxstack, code size, flag. Tiny method headers can start on any byte boundary. Fat method headers shall start on a 4-byte boundary.

25.4.1 Method header type values

The two least significant bits of the first byte of the method header indicate what type of header is present. These 2 bits will be one and only one of the following:

Value	Value	Description
`CorILMethod_TinyFormat`	`0x2`	The method header is tiny (§25.4.2) .
`CorILMethod_FatFormat`	`0x3`	The method header is fat (§25.4.3).

25.4.2 Tiny format

Tiny headers use a 6-bit length encoding. The following is true for all tiny headers:

- No local variables are allowed
- No exceptions
- No extra data sections
- The operand stack shall be no bigger than 8 entries

A Tiny Format header is encoded as follows:

Start Bit	Count of Bits	Description
`0`	`2`	Flags (CorILMethod_TinyFormat shall be set, see §25.4.4).
`2`	`6`	Size, in bytes, of the method body immediately following this header.

25.4.3 Fat format

The fat format is used whenever the tiny format is not sufficient. This can be true for one or more of the following reasons:

- The method is too large to encode the size (i.e., at least 64 bytes)
- There are exceptions
- There are extra data sections
- There are local variables
- The operand stack needs more than 8 entries

A fat header has the following structure

Offset	Size	Field	Description
`0`	`12 (bits)`	**Flags**	Flags (CorILMethod_FatFormat shall be set in bits 0:1, see §25.4.4)
`12 (bits)`	`4 (bits)`	**Size**	Size of this header expressed as the count of 4-byte integers occupied (currently 3)

2	2	**MaxStack**	Maximum number of items on the operand stack
4	4	**CodeSize**	Size in bytes of the actual method body
8	4	**LocalVarSigTok**	Meta Data token for a signature describing the layout of the local variables for the method. 0 means there are no local variables present

25.4.4 Flags for method headers

The first byte of a method header can also contain the following flags, valid only for the Fat format, that indicate how the method is to be executed:

Flag	Value	Description
CorILMethod_FatFormat	0x3	Method header is fat.
CorILMethod_TinyFormat	0x2	Method header is tiny.
CorILMethod_MoreSects	0x8	More sections follow after this header (§25.4.5).
CorILMethod_InitLocals	0x10	Call default constructor on all local variables.

25.4.5 Method data section

At the next 4-byte boundary following the method body can be extra method data sections. These method data sections start with a two byte header (1 byte for flags, 1 byte for the length of the actual data) or a 4-byte header (1 byte for flags, and 3 bytes for length of the actual data). The first byte determines the kind of the header, and what data is in the actual section:

Flag	Value	Description
CorILMethod_Sect_EHTable	0x1	Exception handling data.
CorILMethod_Sect_OptILTable	0x2	Reserved, shall be 0.
CorILMethod_Sect_FatFormat	0x40	Data format is of the fat variety, meaning there is a 3-byte length least-significant byte first format. If not set, the header is small with a 1-byte length
CorILMethod_Sect_MoreSects	0x80	Another data section occurs after this current section

Currently, the method data sections are only used for exception tables (§19). The layout of a small exception header structure as is a follows:

Offset	Size	Field	Description
0	1	**Kind**	Flags as described above.
1	1	**DataSize**	Size of the data for the block, including the header, say *n*12+4*.
2	2	**Reserved**	Padding, always 0.
4	*n*	**Clauses**	*n* small exception clauses (§25.4.6).

The layout of a fat exception header structure is as follows:

Offset	Size	Field	Description
0	1	**Kind**	Which type of exception block is being used
1	3	**DataSize**	Size of the data for the block, including the header, say *n*24+4*.

 ©ISO/IEC 2006 – All rights reserved

4	*n*	**Clauses**	*n* fat exception clauses (§25.4.6).

25.4.6 Exception handling clauses

Exception handling clauses also come in small and fat versions.

The small form of the exception clause should be used whenever the code sizes for the try block and the handler code are both smaller than 256 bytes and both their offsets are smaller than 65536. The format for a small exception clause is as follows:

Offset	Size	Field	Description
0	2	**Flags**	Flags, see below.
2	2	**TryOffset**	Offset in bytes of try block from start of method body.
4	1	**TryLength**	Length in bytes of the try block
5	2	**HandlerOffset**	Location of the handler for this try block
7	1	**HandlerLength**	Size of the handler code in bytes
8	4	**ClassToken**	Meta data token for a type-based exception handler
8	4	**FilterOffset**	Offset in method body for filter-based exception handler

The layout of the fat form of exception handling clauses is as follows:

Offset	Size	Field	Description
0	4	**Flags**	Flags, see below.
4	4	**TryOffset**	Offset in bytes of try block from start of method body.
8	4	**TryLength**	Length in bytes of the try block
12	4	**HandlerOffset**	Location of the handler for this try block
16	4	**HandlerLength**	Size of the handler code in bytes
20	4	**ClassToken**	Meta data token for a type-based exception handler
20	4	**FilterOffset**	Offset in method body for filter-based exception handler

The following flag values are used for each exception-handling clause:

Flag	Value	Description
COR_ILEXCEPTION_CLAUSE_EXCEPTION	0x0000	A typed exception clause
COR_ILEXCEPTION_CLAUSE_FILTER	0x0001	An exception filter and handler clause
COR_ILEXCEPTION_CLAUSE_FINALLY	0x0002	A finally clause
COR_ILEXCEPTION_CLAUSE_FAULT	0x0004	Fault clause (finally that is called on exception only)

26 Index

! 18
!! 18
& 18
* 18
\n **6**
\ooo **6**
\t **6**
+ **6**
abstract 36, **38**, 76
accessibility 23
  default **37**
  overriding and **45**
.addon 93
address 63
ansi 36, **38**, 82
arglist 80
array
  jagged 61
  multi-dimensional 59
  native **21**
  rank of **59**
  single-dimensional 59
.assembly 4, 9, **11**, 14
assembly 4
assembly **10**
assembly
  defining an **11**
assembly
  version number **13**
assembly
  referencing an **14**
assembly 76
assembly 84
.assembly extern 4, 9, 14, **15**
assert 100
attribute **8**
  accessibility **36**
  custom 101
    CLS-defined 103
    thread local storage 104
  field 84
  field contract 85
  genuine custom **101**
  inheritance **38**
  interoperation **38**, 86
  pre-defined **35**
  pseudo custom **102**
  special handling **38**
  type layout **37**
  type semantics **37**
  visibility **36**
auto 36, **37**
autochar 36, **38**, 82
beforefieldinit 36, **39**, 48
BeginInvoke 66, 68, **69**
blob **105**
block
  catch **97**
  fault **98**
  filter **97**
  finally **98**
  handler **97**
  protected **96**
bool 18, **21**
boxing **55**, 89
byte list **7**
bytearray 86
call 57, 70, 71
calli 64
calling convention **71**
callvirt 57, 70, 71
.capability 44

 ©ISO/IEC 2006 – All rights reserved

catch....96, **97**
.cctor....**47**, 72, 78
cdecl....82
char....18
character
  escape....**6**
cil78, 81
.class....9, 16, **35**, 43
class....18
.class extern....9, **17**
CLS tag....**107**
code
  type-safe....**3**
  unmanaged....81
  unverifiable....**3**
  verifiable....**3**
compilercontrolled....76, 85
constraint....**33**
constructor
  class....**47**
  instance....47
conv.ovf.u....64
conv.u....64
.corflags....9, **11**
.ctor....47, 72, 78, 101
.culture....12, 15
.custom....9, 12, 14, 15, 16, 17, 43, 73, 91, 93, **101**
.data....9, 16, 43, 73, 82, **87**
data marshaling....81
deadlock....48
default....71
delegate....**65**
  creation....**66**
delegate call
  asynchronous....**68**
  synchronous....**67**
demand....100
deny....100
directive....**9**
dottedname....**6**, **45**
.emitbyte....73, **74**
endfault....98
endfinally....98
EndInvoke....**66**, 68, **69**
.entrypoint....4, **14**, 73, **74**
enum....**61**
  underlying type....**61**
enumeration....*See* enum
ERROR tag....**107**
.event....43, **93**
event....**93**
event
  declaration....**93**
event
  adder....**93**
event
  remover....**93**
event
  fire....**93**
exception handling....**96**
explicit....36, **37**, 71, 84
extends....**35**
famandassem....76, 84
family....76, 84
famorassem....76, 84
fastcall....82
fault....96, 98
.field....9, 16, 43, 84
field
  global....51
field....**84**
field
  instance....**84**
field
  static....**84**
field

global ... **84**
.file ... 9, **14**
file name ... **8**
filter ... 96, **97**
final ... 76
finalizer ... 47
finally ... 96, 98
.fire ... 93
float32 ... 7, 18, **21**, 86
float64 ... 7, 18, **22**, 86
forwardref ... 78, 81
fromunmanaged ... **81**
generic instance ... **27**
generic method definition ... **26**
generic parameter ... **39**
generic type definition ... **25**
generics ... **24**
.get ... **91**
GUID ... 105
handle
opaque ... 83
handler ... **96**, **97**
.hash ... **14**, 15
.hash algorithm ... 12, 14
heap ... **105**
Blob ... 105
Guid ... 105
String ... 105
UserString ... 105
hexbyte ... **5**, 7
hidebysig ... 76
hiding ... **23**
id **6**
ID ... **6**
identifier ... **6**
keyword as an ... **6**
ILAsm ... 2
case sensitivity of ... **5**
syntax ... **5**
implements ... **35**, 53
[in] ... **73**
inheritcheck ... 100
init ... 73, 75
initobj ... 56
initonly ... 47, 84
instance ... 47, **71**
instance explicit ... **71**
instruction
protected ... **96**
int ... **22**
native ... 18
native unsigned ... 18
int16 ... 18, **22**, 86
unsigned ... 19
int32 ... **5**, 18, **22**, **81**, 87
unsigned ... 19
int64 ... **5**, 18, **22**, **81**, 87
unsigned ... 19
int8 ... 18, **22**, 86
unsigned ... 19
interface ... 35, 36, **37**, 53
internalcall ... 78
InvalidOperationException ... 63
Invoke ... 66
isinst ... 55
label ... **7**
code ... **7**, 74
data ... **7**, 84, 87
list of ... **7**
layout ... **50**
default ... **37**
explicit ... 50
sequential ... 50
ldarga ... 63
ldelem ... 59
ldelema ... 59, 63

 ©ISO/IEC 2006 – All rights reserved

ldflda....63
ldftn....64, 71
ldind....63
ldloca....63
ldsflda....63
ldvirtftn....64, 71
#line....74
.line....**8**, 44, 74
linkcheck....100
literal....85
.locals....**75**
.locals....73
localsinit flag....56
lpstr....**22**
lpwstr....**22**
managed....65, 78
manifest....**10**
manifest resource....**14**
marshal....**21**, 72, 85
marshaling....**83**
.maxstack....4, 73
member
  special....**47**
metadata
  semantics of....2
  structure of....2
metadata merging....**51**
metadata table
  Assembly....**107**
  AssemblyOS....**108**
  AssemblyProcessor....**108**
  AssemblyRef....**108**
  AssemblyRefOS....**109**
  AssemblyRefProcessor....**109**
  ClassLayout....**110**
  Constant....**112**
  CustomAttribute....**112**
  DeclSecurity....**114**
  Event....**116**
  EventMap....**116**
  ExportedType....**118**
  Field....**119**
  FieldLayout....**121**
  FieldMarshal....**122**
  FieldRVA....**123**
  File....**123**
  GenericParam....**124**
  GenericParamConstraint....**125**
  ImplMap....**126**
  InterfaceImpl....**127**
  ManifestResource....**127**
  MemberRef....**128**
  MethodDef....**129**
  MethodImpl....**132**
  MethodSemantics....**133**
  MethodSpec....**134**
  Module....**135**
  ModuleRef....**135**
  NestedClass....**136**
  Param....**136**
  Property....**137**
  PropertyMap....**138**
  StandAloneSig....**139**
  TypeDef....**140**
  TypeRef....**143**
  TypeSpec....**144**
.method....4, 9, 16, 43, 70
method....18
method....**22**
method
  virtual....44
method
  global....51
method
  static....**70**
method

instance ... **70**
method
virtual ... **71**
method
definition ... 72
method
entry point ... **74**
method
predefined attributes for a ... 76
method
implementation attributes for a ... 78
method
vararg ... **80**
method
unmanaged ... **81**
Method ... *See* method definition
method body ... **73**
method declaration ... **70**
method definition ... **70**
method descriptor ... **70**
method implementation ... 44, **70**
method reference ... **70**
method transition thunk ... **81**
MethodDecl. *See* method implementation, *See* method declaration
MethodImpl ... 53
MethodRef ... *See* method reference
modifier
optional ... *See* modopt
required ... *See* modreq
modopt ... 19, 73
modreq ... 19, 73
.module ... 9, **16**, 19, 20
module ... **10**
module
declaring a ... **16**
module
referencing a ... **16**
module
manifest ... 16
module ... **106**
<Module> ... **51**, 120, 130, 132
.module extern ... 9, **16**
.mresource ... 9, **14**
mscorlib ... 4
namespace ... 10
native ... 78, 82
nested assembly ... 36
nested famandassem ... 36
nested family ... 36
nested famorassem ... 36
nested private ... 36
nested public ... 36
newarr ... 59
newobj ... 56, 66
newslot ... 23, 44, 71, 76
noinlining ... 78
nometadata ... **14**
notserialized ... 85
null ... **52**
object ... 18
operator
+ **6**
[opt] ... **73**
.other ... **91**, 93
[out] ... **73**
.override ... 44, 73
.override method ... 74
.pack ... 44, **50**
.param ... 73, 74, **75**
.param type ... 44, 74, **75**
.permission ... 74, 100
.permissionset ... 74, 100
permitonly ... 100
pinned ... 19, **20**
PInvoke ... *See* platform invoke

 ©ISO/IEC 2006 – All rights reserved

pinvokeimpl 76, 82
platform invoke 79, 81, **82**
platformapi 82
pointer **62**
  managed **62**, 64
  method **64**
  unmanaged **62**, 63
pointer arithmetic **63**
private 36, 76, 85
.property 44, **91**
property **91**
property
  declaration 91
property
  getter **91**
property
  setter **91**
public 36, 76, 85
.publickey 12, 15
.publickeytoken **15**
QSTRING **6**
race 48
realnumber **5**, **7**
.removeon 93
reqopt 100
reqrefuse 100
request 100
resolution scope 20
rtspecialname 36, **39**, 47, 76, 85, 91, 93
runtime 65, 78
scope block 80
sealed 36, **38**
security
  declarative **100**
sequential 36, **37**
serializable 36, **39**
serialization 39
.set **91**
signature **154**
.size 44
specialname 36, **39**, 47, 76, 85, 91, 93
SQSTRING **6**
Standard Public Key **12**
static 76, 85
static data
  initialization of **89**
stdcall 82
stelem 59
stind 63
string 18
string literal
  concatenation of **6**
.subsystem 9, **11**
synchronized 78
System.ArgIterator 80
System.Array 59
System.Array.Initialize 56
System.AsyncCallback 66, 69
System.Attribute 101
System.AttributeUsageAttribute 103
System.CLSCompliantAttribute 103
System.Console 4
System.Delegate 65
System.Diagnostics.ConditionalAttribute 104
System.Enum 61
System.Enum.ToString 112
System.FlagsAttribute 104
System.Globalization.CultureInfo 12
System.IAsyncResult 66, 69
System.IntPtr 66
System.MarshalByRefObject 63
System.MissingMethodException 70
System.Net.DnsPermissionAttribute 103
System.Net.SocketPermissionAttribute 103
System.Net.WebPermissionAttribute 103
System.Object 18, 35, 66, 69

©ISO/IEC 2006 – All rights reserved

System.Object.Finalize 47
System.ObsoleteAttribute 103
System.ParamArrayAttribute 104
System.Reflection.AssemblyAlgorithmIDAttribute 102
System.Reflection.AssemblyFlagsAttribute..... 102
System.Reflection.DefaultMemberAttribute.. 102, 104
System.Runtime.CompilerServices.DecimalConstant Attribute 104
System.Runtime.CompilerServices.FaultModeAttribute..... 104
System.Runtime.CompilerServices.IndexerNameAttribute..... 104
System.Runtime.CompilerServices.InitializeArray . 90
System.Runtime.CompilerServices.MethodImplAttribute..... 102
System.Runtime.InteropServices.DllImportAttribute 102
System.Runtime.InteropServices.FieldOffsetAttribute 102
System.Runtime.InteropServices.GCHandle 83
System.Runtime.InteropServices.InAttribute..... 102
System.Runtime.InteropServices.MarshalAsAttribute 102
System.Runtime.InteropServices.OutAttribute..... 102
System.Runtime.InteropServices.StructLayoutAttribute..... 102
System.Runtime.Interopservices.UnmanagedType.. 21
System.Security.Permissions.CodeAccessSecurityAttribute..... 103
System.Security.Permissions.ecurityAttribute 103
System.Security.Permissions.EnvironmentPermission Attribute 103
System.Security.Permissions.FileIOPermissionAttribute..... 103
System.Security.Permissions.ReflectionPermissionAttribute 103
System.Security.Permissions.SecurityAttribute 9
System.Security.Permissions.SecurityPermissionAttribute..... 103
System.String 4, 18, 38
System.ThreadStaticAttribute 104
System.ValueType 61
table..... **105**
tail. 80
terminal **5**
thiscall 82
thunk 81
token
  foreign **118**
  regular **118**
.try **96**
try 96
try block **96**
type..... **18**
  abstract **38**
  base..... **35**
  built-in..... **18**, 20
  closed **27**
  concrete **46**
  definition of a 18, **35**
  instantiated **27**
  marshalling of a..... **21**
  native data **21**
  nested 49
  open..... **27**
  pointer **62**
  reference..... **18**
  specification **19**
  user defined **18**
  value..... 55
type initializer..... 47
type layout..... **110**
typedref 19
unbox..... 57
unboxing..... **55**
unicode..... 36, **38**, 82
unmanaged 78, 82
unmanaged cdecl..... 72
unmanaged fastcall..... 72

 ©ISO/IEC 2006 – All rights reserved

unmanaged stdcall 72
unmanaged thiscall 72
unsigned int **22**
unsigned int16 ..... **22**
unsigned int32 ..... **22**
unsigned int64 ..... **22**
unsigned int8 ..... **22**
validation 3
value type 19
vararg 71, 73, 80
vector **59**
.ver **13**
.ver 12
.ver 15
verification 3
virtual 76
visibility 23
  default **37**
void 19, 72
.vtfixup 9, 81
WARNING tag **107**

Common Language Infrastructure (CLI)

Partition III:
CIL Instruction Set

Table of Contents

1	**Introduction**	**1**
1.1	Data types	1
1.1.1	Numeric data types	2
1.1.2	Boolean data type	4
1.1.3	Object references	4
1.1.4	Runtime pointer types	4
1.2	Instruction variant table	5
1.2.1	Opcode encodings	6
1.3	Stack transition diagram	11
1.4	English description	11
1.5	Operand type table	11
1.6	Implicit argument coercion	14
1.7	Restrictions on CIL code sequences	16
1.7.1	The instruction stream	16
1.7.2	Valid branch targets	16
1.7.3	Exception ranges	17
1.7.4	Must provide maxstack	17
1.7.5	Backward branch constraints	17
1.7.6	Branch verification constraints	17
1.8	Verifiability and correctness	18
1.8.1	Flow control restrictions for verifiable CIL	18
1.9	Metadata tokens	23
1.10	Exceptions thrown	23
2	**Prefixes to instructions**	**24**
2.1	constrained. – (prefix) invoke a member on a value of a variable type	25
2.2	no. – (prefix) possibly skip a fault check	27
2.3	readonly. (prefix) – following instruction returns a controlled-mutability managed pointer	28
2.4	tail. (prefix) – call terminates current method	29
2.5	unaligned. (prefix) – pointer instruction might be unaligned	30
2.6	volatile. (prefix) – pointer reference is volatile	31
3	**Base instructions**	**32**
3.1	add – add numeric values	33
3.2	add.ovf.<signed> – add integer values with overflow check	34

©ISO/IEC 2006 – All rights reserved

3.3	and – bitwise AND	35
3.4	arglist – get argument list	36
3.5	beq.<length> – branch on equal	37
3.6	bge.<length> – branch on greater than or equal to	38
3.7	bge.un.<length> – branch on greater than or equal to, unsigned or unordered	39
3.8	bgt.<length> – branch on greater than	40
3.9	bgt.un.<length> – branch on greater than, unsigned or unordered	41
3.10	ble.<length> – branch on less than or equal to	42
3.11	ble.un.<length> – branch on less than or equal to, unsigned or unordered	43
3.12	blt.<length> – branch on less than	44
3.13	blt.un.<length> – branch on less than, unsigned or unordered	45
3.14	bne.un<length> – branch on not equal or unordered	46
3.15	br.<length> – unconditional branch	47
3.16	break – breakpoint instruction	48
3.17	brfalse.<length> – branch on false, null, or zero	49
3.18	brtrue.<length> – branch on non-false or non-null	50
3.19	call – call a method	51
3.20	calli – indirect method call	53
3.21	ceq – compare equal	54
3.22	cgt – compare greater than	55
3.23	cgt.un – compare greater than, unsigned or unordered	56
3.24	ckfinite – check for a finite real number	57
3.25	clt – compare less than	58
3.26	clt.un – compare less than, unsigned or unordered	59
3.27	conv.<to type> – data conversion	60
3.28	conv.ovf.<to type> – data conversion with overflow detection	61
3.29	conv.ovf.<to type>.un – unsigned data conversion with overflow detection	62
3.30	cpblk – copy data from memory to memory	63
3.31	div – divide values	64
3.32	div.un – divide integer values, unsigned	65
3.33	dup – duplicate the top value of the stack	66
3.34	endfilter – end exception handling filter clause	67
3.35	endfinally – end the finally or fault clause of an exception block	68
3.36	initblk – initialize a block of memory to a value	69
3.37	jmp – jump to method	70
3.38	ldarg.<length> – load argument onto the stack	71
3.39	ldarga.<length> – load an argument address	72

 ©ISO/IEC 2006 – All rights reserved

3.40	ldc.<type> – load numeric constant	73
3.41	ldftn – load method pointer	74
3.42	ldind.<type> – load value indirect onto the stack	75
3.43	ldloc – load local variable onto the stack	77
3.44	ldloca.<length> – load local variable address	78
3.45	ldnull – load a null pointer	79
3.46	leave.<length> – exit a protected region of code	80
3.47	localloc – allocate space in the local dynamic memory pool	81
3.48	mul – multiply values	82
3.49	mul.ovf.<type> – multiply integer values with overflow check	83
3.50	neg – negate	84
3.51	nop – no operation	85
3.52	not – bitwise complement	86
3.53	or – bitwise OR	87
3.54	pop – remove the top element of the stack	88
3.55	rem – compute remainder	89
3.56	rem.un – compute integer remainder, unsigned	90
3.57	ret – return from method	91
3.58	shl – shift integer left	92
3.59	shr – shift integer right	93
3.60	shr.un – shift integer right, unsigned	94
3.61	starg.<length> – store a value in an argument slot	95
3.62	stind.<type> – store value indirect from stack	96
3.63	stloc – pop value from stack to local variable	97
3.64	sub – subtract numeric values	98
3.65	sub.ovf.<type> – subtract integer values, checking for overflow	99
3.66	switch – table switch based on value	100
3.67	xor – bitwise XOR	101
4	**Object model instructions**	**102**
4.1	box – convert a boxable value to its boxed form	103
4.2	callvirt – call a method associated, at runtime, with an object	104
4.3	castclass – cast an object to a class	105
4.4	cpobj – copy a value from one address to another	106
4.5	initobj – initialize the value at an address	107
4.6	isinst – test if an object is an instance of a class or interface	108
4.7	ldelem – load element from array	109
4.8	ldelem.<type> – load an element of an array	110

4.9	ldelema – load address of an element of an array	112
4.10	ldfld – load field of an object	113
4.11	ldflda – load field address	114
4.12	ldlen – load the length of an array	115
4.13	ldobj – copy a value from an address to the stack	116
4.14	ldsfld – load static field of a class	117
4.15	ldsflda – load static field address	118
4.16	ldstr – load a literal string	119
4.17	ldtoken – load the runtime representation of a metadata token	120
4.18	ldvirtftn – load a virtual method pointer	121
4.19	mkrefany – push a typed reference on the stack	122
4.20	newarr – create a zero-based, one-dimensional array	123
4.21	newobj – create a new object	124
4.22	refanytype – load the type out of a typed reference	125
4.23	refanyval – load the address out of a typed reference	126
4.24	rethrow – rethrow the current exception	127
4.25	sizeof – load the size, in bytes,of a type	128
4.26	stelem – store element to array	129
4.27	stelem.<type> – store an element of an array	130
4.28	stfld – store into a field of an object	131
4.29	stobj – store a value at an address	132
4.30	stsfld – store a static field of a class	133
4.31	throw – throw an exception	134
4.32	unbox – convert boxed value type to its raw form	135
4.33	unbox.any – convert boxed type to value	136
5	**Index**	**137**

 ©ISO/IEC 2006 – All rights reserved

1 Introduction

This partition is a detailed description of the Common Intermediate Language (CIL) instruction set, part of the specification of the CLI. 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.

In this partition, each instruction is described in its own subclause, one per page. Related CLI machine instructions are described together. Each instruction description consists of the following parts:

- A table describing the binary format, assembly language notation, and description of each variant of the instruction. See §1.2.
- A stack transition diagram, that describes the state of the evaluation stack before and after the instruction is executed. (See §1.3.)
- An English description of the instruction. See §1.4.
- A list of exceptions that might be thrown by the instruction. (See Partition I for details.) There are three exceptions which can be thrown by any instruction and are *not* listed with the instruction:

  `System.ExecutionEngineException`: indicates that the internal state of the Execution Engine is corrupted and execution cannot continue. In a system that executes only verifiable code this exception is not thrown.

  `System.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 §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.]

  `System.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 (e.g., an implementation based on JIT compilation to native code can 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 §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 §1.5.

Note that not all instructions are included in all CLI Profiles. See Partition IV for details.

1.1 Data types

While the CTS defines a rich type system and the 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, collectively called the "basic CLI types", contains the following 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 can be assigned the value `null`. This is defined throughout the CLI to be zero (a bit pattern of all-bits-zero).

[*Note*: As far as VES operations on the evaluation stack are concerned, there is only one floating-point type, and the VES does not care about its size. The VES makes the distinction about the size of numerical values only when storing these values to, or reading from, the heap, statics, local variables, or method arguments. *end note*]

©ISO/IEC 2006 – All rights reserved

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). However, the CIL instruction set 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) can 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.i1, 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 (i.e., 1- and 2-byte) integers are loaded as 4-byte numbers on all architectures and these 4-byte numbers are always 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 is executed) will have identical results on all implementations. *end note*]

  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 a short integer 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 cannot be directly combined with 8-byte integers (they shall 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 shall 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 can 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

 ©ISO/IEC 2006 – All rights reserved

input), but these types *cannot* be combined with `int64`s. Instead, a `native int` or `int32` shall 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 might be represented internally with additional range and/or precision. The size of the internal floating-point representation is implementation-dependent, might 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:

  o The internal representation shall have precision and range greater than or equal to the nominal type.

  o 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 (such as contractions), which result in the same or greater precision. Where exactly reproducible behavior precision is required by a language or application (e.g., the Kahan Summation Formula), explicit conversions can be used. Reproducible precision does not guarantee reproducible behavior, however. Implementations with extra precision might round twice: once for the floating-point operation, and once for the explicit conversion. Implementations without extra precision effectively round only once. In rare cases, rounding twice versus rounding once can yield results differing by one unit of least precision. *end note*]

  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 might involve a loss of precision or the creation of an out-of-range value (NaN, +infinity, or -infinity). However, the value might 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 shall 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. *end note*]

  [*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. *end note*]

1.1.2 Boolean data type

A CLI Boolean type occupies 1 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 one from the other, producing a signed 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 shall 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 *shall* 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 behavior 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 valid to mark locations that contain unmanaged pointers as though they were unsigned integers (and this is, in fact, how they are treated by the 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., using 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:

 ©ISO/IEC 2006 – All rights reserved

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 (`&`) can point to a local variable, a method argument, a field of an object, a field of a value type, an element of an array, a static field, 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 shall 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 byref 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 §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 operands 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 show how this instruction is encoded (its "opcode"). For example, the ldarga instruction is encoded as a byte holding FE, followed by another holding 0A. Italicized type names delimited by < and > represent numbers that should follow in the instruction stream; for example, a 2-byte quantity that is to be treated as an unsigned integer directly follows the FE 0A opcode. [*Example*: One of

the forms of the ldc.<*type*> instruction is ldc.r8 num, which has a Format "23 <*float64*>". For the instruction ldc.r8 3.1415926535897931, the resulting code is 23 182D4454FB210940, where 182D4454FB210940 is the 8-byte hex representation for 3.1415926535897931.

Similarly, another of the forms of the ldc.<*type*> instruction is ldc.i4.s *num*, which a Format of "1F <*int8*>". For the instruction ldc.i4.s -3, the resulting code is 1F FD, where FD is the 1-byte hex representation for -3. The .s suffix indicates an instruction is a short-form instruction. In this case, it requires 2 bytes rather than the long form ldc.i4, which requires 5 bytes. *end example*]

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 operand 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 operand numbers are encoded least-significant-byte-at-smallest-address (a pattern commonly termed "little-endian"). Bytes for instruction opcodes and operands 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 having instruction stream operands, this column also assigns names to each of the operands to the instruction. For each instruction operand, 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 can 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.

 ©ISO/IEC 2006 – All rights reserved

Table 1: Opcode Encodings

Opcode	Instruction
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

©ISO/IEC 2006 – All rights reserved

Opcode	Instruction
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

Opcode	Instruction
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

 ©ISO/IEC 2006 – All rights reserved

Opcode	Instruction
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
0xA3	ldelem
0xA4	stelem
0xA5	unbox.any
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

Opcode	Instruction
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

©ISO/IEC 2006 – All rights reserved

Opcode	Instruction
0xFE 0x1C	sizeof
0xFE 0x1D	refanytype

 ©ISO/IEC 2006 – All rights reserved

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 shall 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, across 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 `&`.

Table 2 shows the result type for A `op` B—where op is add, div, mul, rem, or sub—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	×	×	×	×	×	×

Table 3 shows the result type for the unary numeric operations. 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

©ISO/IEC 2006 – All rights reserved

Result Type	int32	int64	native int	F	×	×

Table 4 shows the result type for the comparison and branch instructions. The binary comparison returns a Boolean value and the branch operations 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, beq.s, bne.un.s, 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 *end note*]

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)

Table 5 shows the result type for each possible combination of operand types in integer operations. Used for and, div.un, not, or, rem.un, xor. The div.un and rem.un instructions treat their operands as unsigned integers and produce the bit pattern corresponding to the unsigned result. As described in the CLI standard, 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 shall be of type int32 or native 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	×	×	×

 ©ISO/IEC 2006 – All rights reserved

int64	×	int64	×	×	×	×
native int	native int	×	native int	×	×	×
F	×	×	×	×	×	×
&	×	×	×	×	×	×
O	×	×	×	×	×	×

Table 6 shows the valid 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 unspecified. (e.g., 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	×	×	×	×	×	×

Table 7 shows the result type for each possible combination of operand types in the arithmetic operations with overflow checking. An exception shall be thrown if the result cannot be represented in the result type. Used for add.ovf, add.ovf.un, mul.ovf, mul.ovf.un, sub.ovf, and sub.ovf.un. For details of the exceptions thrown, see the descriptions of the specific instructions. 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	×	×	×	×	×	×

Table 8 shows the result type for the conversion operations. Conversion operations convert the top item on the evaluation stack from one numeric type to another. While converting, truncation or extension occurs as shown in the table. 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 an `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 (i.e., 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 result type is unsigned) or sign-extended (if the result type is signed). Thus, converting the value 0x1234 ABCD from the evaluation stack to an 8-bit datum yields the result 0xCD; if the result type were `int8`, this is sign-extended to give 0xFFFF FFCD; if, instead, the result type were `unsigned int8`, this is zero-extended to give 0x0000 00CD.

[2] "Truncate to zero" 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).

Rounding mode for integer to and from F conversions is the same as for arithmetic.

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 might result in an information loss through truncation or rounding) This implicit conversion

 ©ISO/IEC 2006 – All rights reserved

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])	×	×	×	×	×	×

3. [1] Passing a built-in type to a parameter that is required to be a value type is not allowed.

4. [2] The CLI's stack can contain a value type. These can only be passed if the particular value type on the stack exactly matches the class required by the corresponding parameter.

5. [3] There are special instructions to construct and pass a `Ref Any`.

6. [4] The CLI is permitted to pass floating point arguments using its internal F type, see §1.1.1. CIL generators can, 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 subclause 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, CIL 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 can 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 can appear as the result of machine-generated code and is allowed, but it shall 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. *end note*]

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

 ©ISO/IEC 2006 – All rights reserved

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 §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 stack. 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 shall be tracked by an analysis tool. *end note*]

[*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. *end rationale*]

1.7.5 Backward branch constraints

It shall 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. *end rationale*]

[*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. *end note*]

See the following for further information:

- Exceptions: Partition I
- Verification conditions for branch instructions: §3
- The tail. prefix: §3.19

1.7.6 Branch verification constraints

The *target* of all branch instruction shall be a valid branch target (see§1.7.2) within the method holding that branch instruction.

1.8 Verifiability and correctness

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.

Correct CIL is CIL that executes on all conforming implementations of the CLI, with well-defined behavior as specified in this standard. However, correct CIL need not result in identical behavior across conforming implementations; that is, the behavior might be implementation-specific.

It is perfectly acceptable to generate correct CIL code that is not verifiable, but which is known to be memory safe by the compiler writer. Thus, correct CIL might not be verifiable, even though the producing compiler might *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.

Correct 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 which are not verifiable, might violate type safety and hence might violate security or memory access constraints. See II.3 for additional information.

1.8.1 Flow control restrictions for verifiable CIL

This subclause specifies a verification algorithm that, combined with information on individual CIL instructions (see §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 might violate memory safety. Any program that is verifiable according to this standard does not violate memory safety, and a conforming implementation of the CLI shall run such programs. The implementation might 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 might 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 might 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. *end note*]

 ©ISO/IEC 2006 – All rights reserved

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 §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 shall 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 localsinit bit set, see Partition II (Flags for Method Headers). If this bit is not set, then a CLI might 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 CIL-to-native-code compiler to minimize its performance impact. Note that a CLI might optionally choose to perform definite-assignment analysis – in such a case, it might confirm that a method, even without the localsinit bit set, might in fact be verifiable (and therefore not throw a Verification exception) *end rationale*]

[*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 VES.

Performance measurements on C++ implementations (which do 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. *end note*]

The verification algorithm shall simulate all possible control flow paths through the code and ensure that a valid 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 §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.

With the exception of the unconditional branch instructions, throw, rethrow, and ret, there is a control flow path from every instruction to the subsequent instruction. There is also a control flow path from each branch instruction (conditional or unconditional) to the branch target (or targets, in the case of 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 §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 might be more than one such address) that cannot be merged with the stack state just computed. For rules of this merge operation, see §1.8.1.3.

The CLI supports the notion of a *controlled-mutability* managed pointer. (See §1.8.1.2.2, the merging rules in §1.8.1.3, the readonly. instruction prefix in §2.3, the ldfld instruction in §4.10, the stfld instruction in §4.30, and the unbox instruction in §4.32.)

The VES ensures that both special constraints and type constraints are satisfied. The constraints can be checked as early as when a closed type is constructed, or as late as when a method on the constrained generic type is

invoked, a constrained generic method is invoked, a field in a constrained generic type is accessed, or an instance of a constrained generic type is created.

To accommodate generics, the type compatibility relation is extended to deal with:

- generic parameters: a generic parameter is compatible only with itself.
- boxed generic parameters: a boxed generic parameter is compatible with the constraint types declared on the generic parameter.

In the verification semantics, boxing a value of primitive or value type on the stack introduces a value of type "boxed" type; if the value type is Nullable<T> (Partition I.8.2.4), a value of type "boxed" *T* is introduced. This notion of boxed type is extended to generic parameters. Boxing a value whose type is a generic parameter (!0, for example) introduces a value of the boxed parameter type on the stack ("boxed" !0, for example). The boxed forms of value types, and now generic parameters, are used to support efficient instance and virtual method calls on boxed values. Because the "boxed" type statically records the exact type of the underlying value, there is no need to perform a checked cast on the instance from some less informative, but syntactically expressible, reference type.

Just like the boxed forms of primitive and non-primitive value types, the boxed forms of generic parameters only occur on the verification stack (after being introduced by a box instruction). They cannot be explicitly specified using metadata signatures.

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 §1.8.1.2.1, the type compatibility rules are specified in §1.8.1.2.2, and the rules for merging stack states are in §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 §1.8.1.2.2) when considering the types of local variables, arguments, and 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&`

 ©ISO/IEC 2006 – All rights reserved

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. When returning byrefs, verification is done at the return site, not at the call site.

[*Rationale:* Some uses of returning a managed pointer are perfectly verifiable (e.g., returning a reference to a field in an object); but some not (e.g., 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. *end rationale*]

1.8.1.2.2 Controlled-mutability managed pointers

The readonly. prefix and unbox instructions can produce what is called a *controlled-mutability managed pointer*. Unlike ordinary managed pointer types, a controlled-mutability managed pointer is incompatible with ordinary managed pointers; e.g., it cannot be passed as a byref argument to a method. At control flow points, a controlled-mutability managed pointer can be merged with a managed pointer of the same type to yield a controlled-mutability managed pointer.

Controlled-mutability managed pointers can only be used in the following ways:

1. As the object parameter for an ldfld, ldflda, stfld, call, callvirt, or constrained. callvirt instruction.
2. As the pointer parameter to a ldind.* or ldobj instruction.
3. As the source parameter to a cpobj instruction.

All other operations (including stobj, stind.*, initobj, and mkrefany) are invalid.

The pointer is called a controlled-mutability managed pointer because the defining type decides whether the value can be mutated. For value classes that expose no public fields or methods that update the value in place, the pointer is read-only (hence the name of the prefix). In particular, the classes representing primitive types (such as `System.Int32`) do not expose mutators and thus are read-only.

1.8.1.2.3 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. `object := T` if `T` is an interface type.
5. `S := T` if `S` and `T` are both interfaces and the implementation of `T` requires the implementation of `S`
6. `S := null` if `S` is an object type or an interface
7. `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. (This rule deals with array covariance.)
8. If `S` and `T` are method pointers, then `S := T` if the signatures (return types, parameter types and calling convention) are the same.

9. 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 §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.

Merging a controlled-mutability managed pointer with an ordinary (that is, non-controlled-mutability) managed pointer to the same type results in a controlled-mutability managed pointer to that type.

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. *end note*]

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. *end note*]

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)
```

 ©ISO/IEC 2006 – All rights reserved

[*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. *end rationale*]

When *mthd* is a non-final virtual method on an instance other than a boxed value type, the verification also checks that the instance reference to the method being called is the result of ldarg 0, ldarg.s 0, ldarg 0 and the caller's body does not contain starg.s 0, starg 0 or ldarga.s 0, ldarga 0.

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 be a static method
   ; or a method on the class of the object on
   ; the stack or a method on any parent class
   ; of the object on the stack.  In addition, if
   ; the method is virtual, the object on the stack
   ; must be the "this" pointer of the currently
   ; executing method (the result of a ldarg.0,
   ; ldarg.s 0, or ldarg 0 instruction), and the
   ; current method body must not include any
   ; starg.s 0, starg 0, ldarga.s 0, or ldarga 0
   ; instructions.

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.

 ©ISO/IEC 2006 – All rights reserved

2.1 constrained. – (prefix) invoke a member on a value of a variable type

Format	Assembly Format	Description
FE 16 <*T*>	constrained. *thisType*	Call a virtual method on a type constrained to be type T

Stack Transition:

…, ptr, arg1, … argN → …, ptr, arg1, … argN

Description:

The `constrained.` prefix is permitted only on a callvirt instruction. The type of *ptr* must be a managed pointer (&) to *thisType*. The constrained prefix is designed to allow callvirt instructions to be made in a uniform way independent of whether *thisType* is a value type or a reference type.

When callvirt *method* instruction has been prefixed by constrained *thisType* the instruction is executed as follows.

If *thisType* is a reference type (as opposed to a value type) then

ptr is dereferenced and passed as the 'this' pointer to the callvirt of *method*

If *thisType* is a value type and *thisType* implements *method* then

ptr is passed unmodified as the 'this' pointer to a call of *method* implemented by *thisType*

If *thisType* is a value type and *thisType* does not implement *method* then

ptr is dereferenced, boxed, and passed as the 'this' pointer to the callvirt of *method*

This last case can only occur when *method* was defined on `System.Object`, `System.ValueType`, or `System.Enum` and not overridden by *thisType*. In this last case, the boxing causes a copy of the original object to be made, however since all methods on `System.Object`, `System.ValueType`, and `System.Enum` do not modify the state of the object, this fact can not be detected.

The need for the constrained prefix was motivated by the needs IL generators creating generic code. Normally the callvirt instruction is not valid on value types. Instead it is required that IL compilers effectively perform the `this' transformation outlined above at IL compile time, depending on the type of ptr and the method being called. It is not possible to do this transformation at IL compile time, however, when ptr is a generic type (which is unknown at IL compile time). This is why the constrained prefix is needed. The constrained opcode allows IL compilers to make a call to a virtual function in a uniform way independent of whether ptr is a value type or reference type. While this was targeted for the case were *thisType* is a generic type variable, constrained works for non-generic types too, and can ease the complexity of generating virtual calls in languages that hide the distinction between value and reference types.

Exceptions:

None.

Correctness:

The constrained prefix will be immediately followed by a callvirt instruction. *thisType* shall be a valid `typedef, typeref,` or `typespec` metadata token.

Verifiability:

The ptr argument will be a managed pointer (&) to *thisType*. In addition all the normal verification rules of the callvirt instruction apply after the ptr transformation as described above. This is equivalent to requiring that a boxed *thisType* must be a subclass of the class which *method* belongs to.

[*Rationale:* The goal of this instruction was to achieve uniformity of calling virtual functions, so such calls could be made verifiably in generic routines. One way of achieving this uniformity was to always box the 'this' pointer before making a callvirt. This works for both reference type (where box is a no-op), and value

types. The problem with this approach is that a copy is made in the value type case. Thus if the method being called modifies the state of the value type, this will not be reflected after the call completes since this modification was made in the boxed copy. This semantic difference (as well as the performance cost of the extra boxing), makes this alternative unacceptable.]

 ©ISO/IEC 2006 – All rights reserved

2.2 no. – (prefix) possibly skip a fault check

Format	Assembly Format	Description
FE 19 <*unsigned int8*>	no. { typecheck | rangecheck | nullcheck }	The specified fault check(s) normally performed as part of the execution of the subsequent instruction can/shall be skipped.

Description:

This prefix indicates that the subsequent instruction need not perform the specified fault check when it is executed. The byte that follows the instruction code indicates which checks can optionally be skipped. This instruction is not verifiable.

The prefix can be used in the following circumstances:

0x01: typecheck (castclass, unbox, ldelema, stelem, stelem). The CLI can optionally skip any type checks normally performed as part of the execution of the subsequent instruction. InvalidCastException can optionally still be thrown if the check would fail.

0x02: rangecheck (ldelem.*, ldelema, stelem.*). The CLI can optionally skip any array range checks normally performed as part of the execution of the subsequent instruction. IndexOutOfRangeException can optionally still be thrown if the check would fail.

0x04: nullcheck (ldfld, stfld, callvirt, ldvirtftn, ldelem.*, stelem.*, ldelema). The CLI can optionally skip any null-reference checks normally performed as part of the execution of the subsequent instruction. NullReferenceException can optionally still be thrown if the check would fail.

The byte values can be OR-ed; e.g.; a value of 0x05 indicates that both typecheck and nullcheck can optionally be omitted.

Exceptions:

None.

Correctness:

Correct IL permits the prefix only on the instructions specified above.

Verifiability:

Verifiable IL does not permit the use of no.

2.3 readonly. (prefix) – following instruction returns a controlled-mutability managed pointer

Format	Assembly Format	Description
FE 1E	readonly.	Specify that the subsequent array address operation performs no type check at runtime, and that it returns a controlled-mutability managed pointer

Description:

This prefix can only appear only immediately preceding the ldelema instruction and calls to the special *Address* method on arrays. Its effect on the subsequent operation is twofold.

1. At run-time, no type check operation is performed. (For the value class case there is never a runtime time check so this is a noop in that case).

2. The verifier treats the result of the address-of operation as a controlled-mutability managed pointer (§1.8.1.2.2).

Exceptions:

None.

Correctness:

Verifiability:

A controlled-mutability managed pointer must obey the verifier rules given in (2) of §1.8.1.2.2. See also §1.8.1.3.

[*Rationale:* The main goal of the readonly. prefix is to avoid a type check when fetching an element from an array in generic code. For example the expression

array[*i*].*method*()

where *array* has type T[] (where T is a generic parameter), and T has been constrained to have an interface with method '*method*' might compile into the following IL code.

```
ldloc array
ldloc j              // j is array index
readonly.
ldelema !0           // loads the pointer to the object
…                    // load the arguments to the call
constrained. !0
callvirt method
```

Without the readonly. prefix the ldelema would do a type check in the case that `!0` was a reference class. Not only is this type check inefficient, but it is semantically incorrect. The type check for ldelema does an exact match typecheck, which is too strong in general. If the array held derived classes of `!0` then the code above would fail the ldelema typecheck. The only reason we fetch the address of the array element instead of the element itself (which is what the source code says), is because we need a handle for *array[i]* that works both for value types and reference types that can be passed to the constrained callvirt instruction.

If the array holds elements of a reference type, in general, skipping the runtime check would be unsafe. To be safe we have to insure that no modifications of the array happen through this pointer. The verifier rules stated above insure this. Since we explicitly allow read-only pointers to be passed as the object of instance method calls, these pointers are not strictly read-only for value types, but there is no type safety problem for value types. *end rationale*]

 ©ISO/IEC 2006 – All rights reserved

2.4 tail. (prefix) – call terminates current method

Format	Assembly Format	Description
FE 14	tail.	Subsequent call terminates current method

Description:

The tail. prefix shall 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 shall be empty except for the arguments being transferred by the following call. The instruction following the call instruction shall be a ret. Thus the only valid code sequence is

tail. call (or calli or callvirt) *somewhere*
ret

Correct CIL shall not branch to the call instruction, but it is permitted to branch to the ret. The only values on the stack shall 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 can 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 can 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. *end rationale*]

Exceptions:

None.

Correctness:

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 shall be compatible with the return type of the current method.

Verifiability:

Verification requires that no managed pointers are passed to the method being called, since it does not track pointers into the current frame.

2.5 unaligned. (prefix) – pointer instruction might be unaligned

Format	Assembly Format	Description
FE 12 <*unsigned int8*>	unaligned. *alignment*	Subsequent pointer instruction might be unaligned.

Stack Transition:

…, addr → …, addr

Description:

The unaligned. prefix specifies that *addr* (an unmanaged pointer (`&`), or `native int`) on the stack mignt 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* might 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) shall 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. *end rationale*]

The unaligned. and volatile. prefixes can be combined in either order. They shall immediately precede a ldind, stind, ldfld, stfld, ldobj, stobj, initblk, or cpblk instruction.

[*Note:* See Partition I, 12.7 for information about atomicity and data alignment. *end note*]

Exceptions:

None.

Correctness and Verifiability:

An unaligned. prefix shall be followed immediately by one of the instructions listed above.

 ©ISO/IEC 2006 – All rights reserved

2.6 volatile. (prefix) – pointer reference is volatile

Format	Assembly Format	Description
FE 13	volatile.	Subsequent pointer reference is volatile.

Stack Transition:

…, addr → …, addr

Description:

The volatile. prefix specifies that *addr* is a volatile address (i.e., it can 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 shall be marked separately. Access to volatile locations need not be performed atomically. (See Partition I, "Memory Model and Optimizations")

The unaligned. and volatile. prefixes can be combined in either order. They shall immediately precede a ldind, stind, ldfld, stfld, ldobj, stobj, initblk, or cpblk instruction. Only the volatile. prefix is allowed with the ldsfld and stsfld instructions.

Exceptions:

None.

Correctness and Verifiability:

A volatile. prefix should be followed immediately 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 might be employed. Operations that are specifically related to the CTS’s object model are contained in the Object Model Instructions section.

 ©ISO/IEC 2006 – All rights reserved

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 are encapsulated in Table 2: Binary Numeric Operations.

Exceptions:

None.

Correctness and Verifiability:

See Table 2: Binary Numeric Operations.

©ISO/IEC 2006 – All rights reserved

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 are encapsulated in Table 7: Overflow Arithmetic Operations.

Exceptions:

System.OverflowException is thrown if the result cannot be represented in the result type.

Correctness and Verifiability:

See Table 7: Overflow Arithmetic Operations.

 ©ISO/IEC 2006 – All rights reserved

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 are encapsulated in Table 5: Integer Operations.

Exceptions:

None.

Correctness and 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 (having type `System.RuntimeArgumentHandle`) 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 can 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`. *end rationale*]

Exceptions:

None.

Correctness:

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.

Verifiability:

Its use is verifiable within the body of a method whose signature indicates it accepts a variable number of arguments, but verification requires that the result be an instance of the `System.RuntimeArgumentHandle` class.

 ©ISO/IEC 2006 – All rights reserved

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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

Correct CIL shall observe all of the control transfer rules specified above and shall guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §1.8 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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

Correct CIL shall observe all of the control transfer rules specified above and shall guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §1.8 for more details.

 ©ISO/IEC 2006 – All rights reserved

3.7 bge.un.<length> – branch on greater than or equal to, 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 floating-point values).

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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

Correct CIL shall observe all of the control transfer rules specified above and shall guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §1.8 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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

Correct CIL shall observe all of the control transfer rules specified above and shall guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §1.8 for more details.

 ©ISO/IEC 2006 – All rights reserved

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 floating-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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

Correct CIL shall observe all of the control transfer rules specified above and shall guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §1.8 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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

Correct CIL shall observe all of the control transfer rules specified above and shall guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §1.8 for more details.

 ©ISO/IEC 2006 – All rights reserved

3.11 ble.un.<length> – branch on less than or equal to, 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 floating-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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

Correct CIL shall observe all of the control transfer rules specified above and shall guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §1.8 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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

Correct CIL shall observe all of the control transfer rules specified above and shall guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §1.8 for more details.

 ©ISO/IEC 2006 – All rights reserved

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 floating-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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

Correct CIL shall observe all of the control transfer rules specified above and shall guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §1.8 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 floating-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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

Correct CIL shall observe all of the control transfer rules specified above and shall guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §1.8 for more details.

 ©ISO/IEC 2006 – All rights reserved

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 shall 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 might 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 valid. *end rationale*]

Exceptions:

None.

Correctness:

Correct CIL shall observe all of the control transfer rules specified above.

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §1.8 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:

…, → …

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 might trap to a debugger, do nothing, or raise a security exception: the exact behavior is implementation-defined.

Exceptions:

None.

Correctness:

Verifiability:

The break instruction is always verifiable.

 ©ISO/IEC 2006 – All rights reserved

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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

Correct CIL shall observe all of the control transfer rules specified above and shall guarantee there is a minimum of one item on the stack.

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §1.8 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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

Correct CIL shall observe all of the control transfer rules specified above and shall guarantee there is a minimum of one item on the stack.

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §1.8 for more details.

 ©ISO/IEC 2006 – All rights reserved

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 (a `methodref`, `methoddef`, or `methodspec`;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 can be immediately preceded by a tail. prefix to specify that the current method state should be released before transferring control (see §2.3).

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 this with the callvirt instruction for calling virtual methods, where the destination address also depends upon the exact 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 base class" behavior. *end rationale*]

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, and so on. There are three important special cases:

1. Calls to an instance (or virtual, see below) method shall 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) (For calls to methods on value types, the `this` pointer is a managed pointer, not an instance reference.)
2. It is valid 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 can be called with either the call or callvirt instruction.

Exceptions:

`System.SecurityException` can be thrown if system security does not grant the caller access to the called method. The security check can occur when the CIL is converted to native code rather than at runtime.

`System.MethodAccessException` can be thrown when there is an invalid attempt to access a private or protected method inside a class.

`System.MissingMethodException` can be thrown when there is an attempt to dynamically access a method that does not exist.

Correctness:

Correct CIL ensures that the stack contains the correct number and type of arguments for the method being called.

Verifiability:

For a typical use of the call instruction, verification checks that (a) *method* refers to a valid `methodref`, `methoddef`, or `methodspec` 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 call site, and (d) the method is not abstract (i.e., it has an implementation).

The call instruction can also be used to call an object's base class 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 shall be static.

When using the call opcode to call a non-final virtual method on an instance other than a boxed value type, verification checks that the instance reference to the method being called is the result of ldarg.s 0, ldarg 0 and the caller's body does not contain starg.s 0, starg 0 or ldarga.s 0, ldarga 0.

[*Rationale*: This means that non-virtually calling a non-final virtual method is only verifiable in the case where the subclass methods calls one of its superclasses using the same `this` object reference, where "same" is easy to verify. This means that an override implementation effectively "hides" the superclass' implementation, and can assume that the override implementation cannot be bypassed by code outside the class hierarchy.

For non-sealed class hierarchies, malicious code can attempt to extend the class hierarchy in an attempt to bypass a class' override implementation. However, this can only be done on an object of the malicious type, and not of the class with the override, which mitigates much of the security concern. *end rationale*]

 ©ISO/IEC 2006 – All rights reserved

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 can 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 originating 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 could 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, and so on. The argument-building code sequence for an instance or virtual method shall push that instance reference (the this pointer, which shall not be null) before any of the user-visible arguments. (For calls to methods on value types, the this pointer is a managed pointer, not an instance reference.)

Exceptions:

System.SecurityException can be thrown if the system security does not grant the caller access to the called method. The security check can occur when the CIL is converted to native code rather than at runtime.

Correctness:

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.

Verifiability:

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 *value1* equals *value2*, else push 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.

Correctness:

Correct CIL provides two values on the stack whose types match those specified in Table 4: Binary Comparison or Branch Operations

Verifiability:

There are no additional verification requirements.

 ©ISO/IEC 2006 – All rights reserved

3.22 cgt – compare greater than

Format	Assembly Format	Description
FE 02	Cgt	Push 1 (of type int32) if *value1* > *value2*, else push 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 with 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.

Correctness:

Correct CIL provides two values on the stack whose types match those specified in Table 4: Binary Comparison or Branch Operations

Verifiability:

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 *value1* > *value2*, unsigned or unordered, else push 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.

Correctness:

Correct CIL provides two values on the stack whose types match those specified in Table 4: Binary Comparison or Branch Operations

Verifiability:

There are no additional verification requirements.

 ©ISO/IEC 2006 – All rights reserved

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 behavior is unspecified if *value* is not a floating-point number.

Exceptions:

`System.ArithmeticException` is thrown if *value* is a NaN or an infinity.

Correctness:

Correct CIL guarantees that *value* is a floating-point number.

Verifiability:

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 *value1* < *value2*, else push 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.

Correctness:

Correct CIL provides two values on the stack whose types match those specified in Table 4: Binary Comparison or Branch Operations

Verifiability:

There are no additional verification requirements.

 ©ISO/IEC 2006 – All rights reserved

3.26 clt.un – compare less than, unsigned or unordered

Format	Assembly Format	Description
FE 05	clt.un	Push 1 (of type int32) if *value1* < *value2*, unsigned or unordered, else push 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.

Correctness:

Correct CIL provides two values on the stack whose types match those specified in Table 4: Binary Comparison or Branch Operations

Verifiability:

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 might 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 when 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, or if the floating-point value being converted to an integer is a NaN, 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`.

The acceptable operand types and their corresponding result data type is encapsulated in Table 8: Conversion Operations.

Exceptions:

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.

Correctness:

Correct CIL has at least one value, of a type specified in Table 8: Conversion Operations, on the stack.

Verifiability:

The table Table 8: Conversion Operations specifies a restricted set of types that are acceptable in verified code.

 ©ISO/IEC 2006 – All rights reserved

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 an unsigned int8 (on the stack as int32) and throw an exception on overflow.
B6	conv.ovf.u2	Convert to an unsigned int16 (on the stack as int32) and throw an exception on overflow.
B8	conv.ovf.u4	Convert to an unsigned int32 (on the stack as int32) and throw an exception on overflow
BA	conv.ovf.u8	Convert to an unsigned int64 (on the stack as int64) and throw an exception on overflow.
D4	conv.ovf.i	Convert to a 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 result cannot be represented in 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:

`System.OverflowException` is thrown if the result can not be represented in the result type.

Correctness:

Correct CIL has at least one value, of a type specified in Table 8: Conversion Operations, on the stack.

Verifiability:

The table Table 8: Conversion Operations 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 on the top of the stack is treated as an unsigned value before the conversion.

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 are encapsulated in Table 8: Conversion Operations.

Exceptions:

`System.OverflowException` is thrown if the result cannot be represented in the result type.

Correctness:

Correct CIL has at least one value, of a type specified in Table 8: Conversion Operations, on the stack.

Verifiability:

The table Table 8: Conversion Operations specifies a restricted set of types that are acceptable in verified code.

 ©ISO/IEC 2006 – All rights reserved

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 operation of the cpblk instruction can be altered by an immediately preceding volatile. or unaligned. prefix instruction.

[*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. *end rationale*]

Exceptions:

`System.NullReferenceException` can be thrown if an invalid address is detected.

Correctness:

CIL ensures the conditions specified above.

Verifiability:

The cpblk instruction is never verifiable.

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 are encapsulated in Table 2: Binary Numeric Operations.

Exceptions:

Integral operations throw `System.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
```

Correctness and Verifiability

See Table 2: Binary Numeric Operations.

 ©ISO/IEC 2006 – All rights reserved

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:

System.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
```

Correctness and Verifiability

See Table 5: Integer Operations.

3.33 dup – duplicate the top value of the stack

Format	Assembly Format	Description
25	Dup	Duplicate the value on the top of the stack.

Stack Transition:

…, value → …, value, value

Description:

The dup instruction duplicates the top element of the stack.

Exceptions:

None.

Correctness and Verifiability:

No additional requirements.

 ©ISO/IEC 2006 – All rights reserved

3.34 endfilter – end exception handling filter clause

Format	Assembly Format	Description
FE 11	Endfilter	End an exception handling filter clause.

Stack Transition:

…, value → …

Description:

Used to return from the filter clause of an exception (see the Exception Handling subclause of Partition I for a discussion of exceptions). *value* (which shall be of type `int32` and 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.

The result of using any other integer value is unspecified.

The entry point of a filter, as shown in the method's exception table, shall be the (lexically) first instruction in the filter's code block. The endfilter shall 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 valid to execute a ret or leave instruction within a `filter` block. It is not valid 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.

Correctness:

Correct CIL guarantees the control transfer restrictions specified above.

Verifiability:

The stack shall 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 subclause 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 can 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 valid 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.

endfinally empties the evaluation stack as a side-effect.

Exceptions:

None.

Correctness:

Correct CIL guarantees the control transfer restrictions specified above.

Verifiability:

There are no additional verification requirements.

 ©ISO/IEC 2006 – All rights reserved

3.36 initblk – initialize a block of memory to a value

Format	Assembly Format	Description
FE 18	initblk	Set all bytes in a block of memory to a given byte value.

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. *end rationale*]

The operation of the initblk instructions can be altered by an immediately preceding volatile. or unaligned. prefix instruction.

Exceptions:

`System.NullReferenceException` can be thrown if an invalid address is detected.

Correctness:

Correct CIL code ensures the restrictions specified above.

Verifiability:

The initblk instruction is never verifiable.

3.37 jmp – jump to method

Format	Assembly Format	Description
27 <*T*>	jmp *method*	Exit current method and jump to the specified method.

Stack Transition:

… → …

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 shall be empty when this instruction is executed. The calling convention, number and type of arguments at the destination address shall 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.

Correctness:

Correct CIL code obeys the control flow restrictions specified above.

Verifiability:

The jmp instruction is never verifiable.

 ©ISO/IEC 2006 – All rights reserved

3.38 ldarg.<length> – load argument onto the stack

Format	Assembly Format	Description
FE 09 <*unsigned int16*>	ldarg *num*	Load argument numbered *num* onto the stack.
0E <*unsigned int8*>	ldarg.s *num*	Load argument numbered *num* onto the stack, short form.
02	ldarg.0	Load argument 0 onto the stack.
03	ldarg.1	Load argument 1 onto the stack.
04	ldarg.2	Load argument 2 onto the stack.
05	ldarg.3	Load argument 3 onto the stack.

Stack Transition:

… → …, value

Description:

The ldarg *num* instruction pushes onto the evaluation stack, the *num*'th incoming argument, where arguments are numbered 0 onwards (see Partition I). 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 one of the first 4 arguments. The ldarg.s instruction is an efficient encoding for loading argument numbers 4–255.

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.

Correctness:

Correct CIL guarantees that *num* is a valid argument index.

Verifiability:

See §1.8 for 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 0–255.

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 byref parameter passing (see *Partition I*). In other cases, ldarg and starg should be used. *end rationale*]

Exceptions:

None.

Correctness:

Correct CIL ensures that *argNum* is a valid argument index.

Verifiability:

See §1.8 for details on how verification determines the type of the value loaded onto the stack.

 ©ISO/IEC 2006 – All rights reserved

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* or some constant 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. For constants that shall be expressed in more than 32 bits, use the ldc.i8 instruction.
2. For constants that require 9–32 bits, use the ldc.i4 instruction followed by a conv.i8.
3. For constants that can be expressed in 8 or fewer bits, use a short form instruction followed by a conv.i8.

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.

Correctness:

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 can 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).

[*Note*: There are many options for implementing this instruction. Conceptually, this instruction places on the virtual machine's evaluation stack a representation of the address of the method specified. In terms of native code this can be an address (as specified), a data structure that contains the address, or any value that can be used to compute the address, depending on the architecture of the underlying machine, the native calling conventions, and the implementation technology of the VES (JIT, interpreter, threaded code, etc.). *end note*]

Exceptions:

None.

Correctness:

Correct CIL requires that method is a valid `methoddef` or `methodref` token.

Verifiability:

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.

 ©ISO/IEC 2006 – All rights reserved

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 an 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* shall be to a location with the natural alignment of `<`*type*`>` or a `NullReferenceException` might occur (but see the unaligned. prefix instruction). (Alignment is discussed in Partition I.) The results of all CIL instructions that return addresses (e.g., ldloca and ldarga) are safely aligned. For data types larger than 1 byte, the byte ordering is dependent on the target CPU. Code that depends on byte ordering might not run on all platforms.

The operation of the ldind instructions can 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. *end rationale*]

Exceptions:

`System.NullReferenceException` can be thrown if an invalid address is detected.

Correctness:

Correct CIL only uses an ldind instruction in a manner consistent with the type of the pointer.

Verifiability:

For verifiable code, the address on the stack shall be a managed pointer, and the instruction form used shall be consistent with the type of the pointer, as specified by the table above.

 ©ISO/IEC 2006 – All rights reserved

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 localsinit 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 4 local variables. The ldloc.s instruction provides an efficient encoding for accessing local variables 4–255.

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 are expanded to type `int32` when they are loaded onto the stack. Floating-point values are expanded to their native size (type `F`).

Exceptions:

`System.VerificationException` is thrown if the the localsinit bit for this method has not been set, and the assembly containing this method has not been granted `System.Security.Permissions.SecurityPermission.SkipVerification` (and the CIL does not perform automatic definite-assignment analysis)

Correctness:

Correct CIL ensures that *indx* is a valid local index.

For the *ldloc indx* instruction, *indx* shall lie in the range 0–65534 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 valid, it would require a wider integer to track the number of locals in such a method. *end rationale*]

Verifiability:

For verifiable code, this instruction shall guarantee that it is not loading an uninitialized value – whether that initialization is done explicitly by having set thelocalsinit bit for the method, or by previous instructions (where the CLI performs definite-assignment analysis).

See §1.8 for more details on how verification determines the type of a local variable.

3.44 ldloca.<length> – load local variable address

Format	Assembly Format	Description
FE 0D <*unsigned int16*>	ldloca *indx*	Load address of local variable with index *indx*.
12 <*unsigned int8*>	ldloca.s *indx*	Load address of local variable with index *indx, short form*.

Stack Transition:

… → …, address

Description:

The ldloca instruction pushes the address of the local variable number *indx* 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 0–255. (Local variables that are the subject of ldloca shall be aligned as described in the ldind instruction, since the address obtained by ldloca can be used as an argument to ldind.)

Exceptions:

`System.VerificationException` is thrown if the the localsinit bit for this method has not been set, and the assembly containing this method has not been granted `System.Security.Permissions.SecurityPermission.SkipVerification` (and the CIL does not perform automatic definite-assignment analysis)

Correctness:

Correct CIL ensures that *indx* is a valid local index.

For the ldloca *indx* instruction, i*ndx* shall lie in the range 0–65534 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 valid, it would require a wider integer to track the number of locals in such a method. *end rationale*]

Verifiability:

See §1.8 for details on how verification determines the type of a local variable.

For verifiable code, this instruction shall guarantee that it is not loading the address of an uninitialized value – whether that initialization is done explicitly by having set the localsinit bit for the method, or by previous instructions (where the CLI performs definite-assignment analysis)

 ©ISO/IEC 2006 – All rights reserved

3.45 ldnull – load a null pointer

Format	Assembly Format	Description
14	ldnull	Push a null reference on the stack.

Stack Transition:

… → …, 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 an ldc.i instruction, which does not exist. However, even if CIL were to include an ldc.i instruction it would still benefit verification algorithms to retain the ldnull instruction because it makes type tracking easier. *end rationale*]

Exceptions:

None.

Correctness:

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:

…, →

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 the former 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 valid to use a leave instruction to exit a `finally` block. To ease code generation for exception handlers it is valid from within a `catch` block to use a leave instruction to transfer control to any instruction within the associated `try` block.

The leave instruction can be used to exit multiple nested blocks (see Partition I).

If an instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Exceptions:

None.

Correctness:

Correct CIL requires the computed destination lie within the current method.

Verifiability:

See §1.8 for details.

 ©ISO/IEC 2006 – All rights reserved

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` or `U4`) bytes from the local dynamic memory pool and returns the address (an unmanaged pointer, type native int) of the first allocated byte. If the localsinit flag on the method is true, the block of memory returned is initialized to 0; otherwise, the initial value of that block of memory is unspecified. 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 shall be computed at runtime. It can be used for C's intrinsic alloca method. *end rationale*]

Exceptions:

`System.StackOverflowException` is thrown if there is insufficient memory to service the request.

Correctness:

Correct CIL requires that the evaluation stack be empty, apart from the *size* item

Verifiability:

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.

Correctness and Verifiability:

See Table 2: Binary Numeric Operations.

 ©ISO/IEC 2006 – All rights reserved

3.49 mul.ovf.<type> – multiply integer values with overflow check

Format	Assembly Format	Description
D8	mul.ovf	Multiply signed integer values. Signed result shall fit in same size.
D9	mul.ovf.un	Multiply unsigned integer values. Unsigned result shall 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:

System.OverflowException is thrown if the result can not be represented in the result type.

Correctness and 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.

Correctness and Verifiability:

See Table 3: Unary Numeric Operations.

 ©ISO/IEC 2006 – All rights reserved

3.51 nop – no operation

Format	Assembly Format	Description
00	Nop	Do nothing.

Stack Transition:

…, → …,

Description:

The nop instruction does nothing. It is intended to fill in space if bytecodes are patched.

Exceptions:

None.

Correctness:

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:

The not instruction computes the bitwise complement of the integer value on top of the stack and leaves 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 are encapsulated in Table 5: Integer Operations.

Exceptions:

None.

Correctness and Verifiability:

See Table 5: Integer Operations.

 ©ISO/IEC 2006 – All rights reserved

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 are encapsulated in Table 5: Integer Operations.

Exceptions:

None.

Correctness and Verifiability:

See Table 5: Integer Operations.

3.54 pop – remove the top element of the stack

Format	Assembly Format	Description
26	pop	Pop *value* from the stack.

Stack Transition:

…, value → …

Description:

The pop instruction removes the top element from the stack.

Exceptions:

None.

Correctness:

Verifiability:

No additional requirements.

 ©ISO/IEC 2006 – All rights reserved

3.55 rem – compute remainder

Format	Assembly Format	Description
5D	rem	Remainder when dividing one value by another.

Stack Transition:

…, value1, value2 → …, result

Description:

The rem instruction divides *value1* by *value2* and pushes the remainder *result* on the stack.

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

$0 \leq |result| < |value2|$, and

$sign(result) = sign(value1)$,

where div is the division instruction, which truncates towards zero.

For floating-point operands

rem is defined similarly as for integer operands, except that, if *value2* is zero or *value1* is infinity, *result* is NaN. If *value2* is `infinity`, *result* is *value1*. 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 `System.DivideByZeroException` if *value2* is zero.

Integral operations can throw `System.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)
```

Correctness and Verifiability:

See Table 2: Binary Numeric Operations.

©ISO/IEC 2006 – All rights reserved

3.56 rem.un – compute integer remainder, unsigned

Format	Assembly Format	Description
5E	rem.un	Remainder when dividing one unsigned value by another.

Stack Transition:

…, value1, value2 → …, result

Description:

The rem.un instruction divides *value1* by *value2* and pushes the remainder *result* on the stack. (rem.un treats its arguments as unsigned integers, while rem treats them as signed integers.)

result = *value1* rem.un *value2* satisfies the following conditions:

result = *value1* – *value2*×(*value1* div.un *value2*), and

$0 \leq result < value2$,

where div.un is the unsigned division instruction. 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 `System.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)
```

Correctness and Verifiability:

See Table 5: Integer Operations.

 ©ISO/IEC 2006 – All rights reserved

3.57 ret – return from method

Format	Assembly Format	Description
2A	Ret	Return from method, possibly with 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 shall 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.

Correctness:

Correct CIL obeys the control constraints describe above.

Verifiability:

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` 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.

Correctness and Verifiability:

See Table 5: Integer Operations.

 ©ISO/IEC 2006 – All rights reserved

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 → …, 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` 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 *result*. See Table 6: Shift Operations for details of which operand types are allowed, and their corresponding result type.

Exceptions:

None.

Correctness and 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.

Correctness and Verifiability:

See Table 5: Integer Operations.

 ©ISO/IEC 2006 – All rights reserved

3.61 starg.<length> – store a value in an argument slot

Format	Assembly Format	Description
FE 0B <*unsigned int16*>	starg *num*	Store *value* to the argument numbered *num.*
10 <*unsigned int8*>	starg.s *num*	Store *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 shall 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.

Correctness:

Correct CIL requires that *num* is a valid argument slot.

Verifiability:

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 value *val* at address *addr* (an unmanaged pointer, type `native int`, or managed pointer, type `&`). The address specified by *addr* shall be aligned to the natural size of *val* or a `NullReferenceException` can 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 data types larger than 1 byte, the byte ordering is dependent on the target CPU. Code that depends on byte ordering might not run on all platforms. All of the stind instructions are shortcuts for an stobj instruction that specifies the corresponding built-in value class.

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 can be altered by an immediately preceding volatile. or unaligned. prefix instruction.

Exceptions:

`System.NullReferenceException` is thrown if *addr* is not naturally aligned for the argument type implied by the instruction suffix.

Correctness:

Correct CIL ensures that *addr* is a pointer whose type is assignment-compatible with that of *val,* subject to implicit conversion as specified in §1.6.

Verifiability:

For verifiable code, *addr* shall be a managed pointer, and the type of *val* shall be assignment compatible with *addr*: if *addr* has type S& and *val* has type T, then S := T according to the rules in §1.8.1.2.2.

 ©ISO/IEC 2006 – All rights reserved

3.63 stloc – pop value from stack to local variable

Format	Assembly Format	Description
FE 0E <*unsigned int16*>	stloc *indx*	Pop a value from stack into local variable *indx*.
13 <*unsigned int8*>	stloc.s *indx*	Pop a value from stack into local variable *indx*, short form.
0A	stloc.0	Pop a value from stack into local variable 0.
0B	stloc.1	Pop a value from stack into local variable 1.
0C	stloc.2	Pop a value from stack into local variable 2.
0D	stloc.3	Pop a value from stack into local variable 3.

Stack Transition:

…, value → …

Description:

The stloc *indx* instruction pops the top value off the evaluation stack and moves it into local variable number *indx* (see Partition I), where local variables are numbered 0 onwards. The type of *value* shall 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 4 local variables; the stloc.s instruction provides an efficient encoding for local variables 4–255.

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.

Correctness:

Correct CIL requires that *indx* be a valid local index. For the stloc *indx* instruction, i*ndx* shall lie in the range 0–65534 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 valid, it would require a wider integer to track the number of locals in such a method. *end rationale*]

Verifiability:

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 are encapsulated in Table 2: Binary Numeric Operations.

Exceptions:

None.

Correctness and Verifiability:

See Table2: Binary Numeric Operations.

 ©ISO/IEC 2006 – All rights reserved

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 shall fit in same size.
DB	sub.ovf.un	Subtract native unsigned int from a native unsigned int. Unsigned result shall 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 are 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:

System.OverflowException is thrown if the result can not be represented in the result type.

Correctness and Verifiability:

See Table 7: Overflow Arithmetic Operations.

3.66 switch – table switch based 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, and so on). 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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

Correct CIL obeys the control transfer constraints listed above.

Verifiability:

Verification requires the type-consistency of the stack, locals and arguments for every possible way of reaching all destination instructions. See §1.8 for more details.

 ©ISO/IEC 2006 – All rights reserved

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.

Correctness and 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 CTS. 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*. *end rationale*]

 ©ISO/IEC 2006 – All rights reserved

4.1 box – convert a boxable value to its boxed form

Format	Assembly Format	Description
8C <*T*>	box *typeTok*	Convert a boxable value to its boxed form

Stack Transition:

…, val → …, obj

Description:

If *typeTok* is a value type, the box instruction converts *val* to its boxed form. When *typeTok* is a non-nullable type (§1.8.2.4), this is done by creating a new object and copying the data from *val* into the newly allocated object. If it is a nullable type, this is done by inspecting *val*'s HasValue property; if it is false, a null reference is pushed onto the stack; otherwise, the result of boxing *val*'s Value property is pushed onto the stack. If *typeTok* is a reference type, the box instruction does nothing.

typeTok is a metadata token (a `typedef, typeref,` or `typespec`) indicating the type of *val*. *typeTok* can represent a value type, a reference type, or a generic parameter.

Exceptions:

`System.OutOfMemoryException` is thrown if there is insufficient memory to satisfy the request.

`System.TypeLoadException` is thrown if *typeTok* cannot be found. (This is typically detected when CIL is converted to native code rather than at runtime.)

Correctness:

typeTok shall be a valid `typedef, typeref,` or `typespec` metadata token. The type operand *typeTok* shall represent a boxable type.

Verifiability:

The top-of-stack shall be assignment compatible with the type represented by *typeTok*. When *typeTok* represents a non-nullable value type or a generic parameter, the resulting type is "boxed" *typeTok*; when *typeTok* is Nullable<T>, the resulting type is "boxed" *T*. When *typeTok* is a reference type, the resulting type is *typeTok*. The type operand *typeTok* shall not be a byref, byref-like, or `void` type.

[*Rationale*: Based on the rules above, when *typeTok* represents a generic parameter whose runtime type is Nullable<*T*> the verifier states that the result is of type boxed Nullable<*T*>. This does not match what the implementation of box produces dynamically. At runtime, boxing a value of type Nullable<*T*> will actually produce an instance of boxed *T*, not boxed Nullable<*T*> as the verifier claims. For this to be type-safe, there must be no operation the verifier permits on a boxed Nullable<*T*> that isn't also permissible on a boxed *T*. This requirement holds the Nullable<*T*> does not have any constrainable features, e.g. interface implementations. *end rationale*]

©ISO/IEC 2006 – All rights reserved

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 an object.

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 exact 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 can 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`, `methodref` or `methodspec` 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 base class 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, and so on.

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) shall 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 indicate whether the method requires passing the this pointer (see Partition II)

Note that a virtual method can also be called using the call instruction.

Exceptions:

`System.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 base classes. This is typically detected when CIL is converted to native code, rather than at runtime.

`System.NullReferenceException` is thrown if *obj* is null.

`System.SecurityException` is thrown if system security does not grant the caller access to the called method. The security check can occur when the CIL is converted to native code rather than at runtime.

Correctness:

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.

Verifiability:

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 call site. A callvirt annotated by tail. has additional considerations – see §1.8.

 ©ISO/IEC 2006 – All rights reserved

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* (of type O) to the *class*. *class* is a metadata token (a typeref, typedef or typespec), indicating the desired class. When *class* is the type System.Nullable<*T*> and the object's class is *T*, castclass succeeds. Otherwise, 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 derived class 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 note 2 above, enums are treated as their underlying type: thus E1[] can be 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:

System.InvalidCastException is thrown if *obj* cannot be cast to *class*.

System.TypeLoadException is thrown if *class* cannot be found. This is typically detected when CIL is converted to native code rather than at runtime.

Correctness:

Correct CIL ensures that *class* is a valid typeRef, typeDef or typeSpec token, and that *obj* is always either null or an object reference.

Verifiability:

There are no additional verification requirements.

4.4 cpobj – copy a value from one address to another

Format	Assembly Format	Description
70 <*T*>	cpobj *typeTok*	Copy a value type from *src* to *dest*.

Stack Transition:

…, dest, src → …,

Description:

The cpobj instruction copies the value at the address specified by *src* (an unmanaged pointer, `native int`, or a managed pointer, `&`) to the address specified by *dest* (also a pointer). *typeTok* can be a `typedef`, `typeref`, or `typespec`. The behavior is unspecified if the type of the location referenced by *src* is not assignment-compatible with the type of the location referenced by *dest*.

If *typeTok* is a reference type, the cpobj instruction has the same effect as ldind.ref followed by stind.ref.

Exceptions:

`System.NullReferenceException` can be thrown if an invalid address is detected.

`System.TypeLoadException` is thrown if *typeTok* cannot be found. This is typically detected when CIL is converted to native code rather than at runtime.

Correctness:

typeTok shall be a valid `typedef`, `typeref`, or `typespec` metadata token.

Verifiability:

The static types of the destination (*dest*) and source (*src*) values shall both be managed pointers (`&`) to values whose types we denote *destType* and *srcType*, respectively. Finally, *srcType* shall be assignment-compatible with *typeTok*, and *typeTok* shall be assignment-compatible with *destType*. In the case of an Enum, its type is that of the underlying, or base, type of the Enum.

 ©ISO/IEC 2006 – All rights reserved

4.5 initobj – initialize the value at an address

Format	Assembly Format	Description
FE 15 <*T*>	initobj *typeTok*	Initialize the value at address *dest*.

Stack Transition:

…, dest → …,

Description:

The initobj instruction initializes an address with a default value. *typeTok* is a metadata token (a `typedef`, `typeref`, or `typespec`). *dest* is an unmanaged pointer (`native int`), or a managed pointer (`&`). If *typeTok* is a value type, the initobj instruction initializes each field of *dest* to null or a zero of the appropriate built-in type. If *typeTok* is a value type, then after this instruction is executed, the instance is ready for a constructor method to be called. If *typeTok* is a reference type, the initobj instruction has the same effect as ldnull followed by stind.ref.

Unlike newobj, the initobj instruction does not call any constructor method.

Exceptions:

None.

Correctness:

typeTok shall be a valid `typedef`, `typeref`, or `typespec` metadata token.

Verifiability:

The type of the destination value on top of the stack shall be a managed pointer to some type *destType*, and *typeTok* shall be a subtype of *destType*. If *typeTok* is a non-reference type, the definition of subtyping implies that *destType* and *typeTok* shall be equal.

©ISO/IEC 2006 – All rights reserved

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, typedef or typespecsee 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 derived class of *class* (if *class* is a regular class), or if *class* is System.Nullable<*T*> and the object's class is of type *T*, 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 note 2, enums are treated as their underlying type: thus E1[] can cast to E2[] if E1 and E2 share an underlying type.

Exceptions:

System.TypeLoadException is thrown if *class* cannot be found. This is typically detected when CIL is converted to native code rather than at runtime.

Correctness:

Correct CIL ensures that *class* is a valid typeref or typedef or typespec token indicating a class, and that *obj* is always either null or an object reference.

Verifiability:

There are no additional verification requirements.

 ©ISO/IEC 2006 – All rights reserved

4.7 ldelem – load element from array

Format	Assembly Format	Description
A3 <*T*>	ldelem *typeTok*	Load the element at *index* onto the top of the stack.

Stack Transition:

…, array, index → …, value

Description:

The ldelem instruction loads the value of the element with index *index* (of type `native int` or `int32`) in the zero-based one-dimensional array *array*, and places it on the top of the stack. The type of the return value is indicated by the type token *typeTok* in the instruction.

Exceptions:

`System.IndexOutOfRangeException` is thrown if *index* is larger than the bound of *array*.

`System.NullReferenceException` is thrown if array is null.

Correctness:

typeTok shall be a valid `typedef`, `typeref`, or `typespec` metadata token.

array shall be either null or a single dimensional, zero-based array.

Verifiability:

The static type of *array* is either the special reference type `Null` or a proper zero-based, one-dimensional array type *elem*[], for some type *elem*. If the array type is `Null`, take *elem* to be the type represented by operand *typeTok*. The value *index* shall have type `native int`. The type *elem* shall be a subtype of the type operand *typeTok*. The type of the value left on the stack is *typeTok*.

©ISO/IEC 2006 – All rights reserved

4.8 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 a native int.
9A	ldelem.ref	Load the element at *index* onto the top of the stack as an O. The type of the O is the same as the element type of the array pushed on the CIL stack.

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:

`System.NullReferenceException` is thrown if *array* is null.

`System.IndexOutOfRangeException` is thrown if *index* is negative, or larger than the bound of *array*.

Correctness:

 ©ISO/IEC 2006 – All rights reserved

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)

Verifiability:

The type of *index* shall be `int32` or `native int`. The element type of *array* shall match the type specified by the suffix, as described above.

4.9 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 → …, 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 an `Address` method.

If this instruction is prefixed by the readonly. prefix, it produces a controlled-mutability managed pointer (§1.8.1.2.2).

Exceptions:

`System.NullReferenceException` is thrown if *array* is null.

`System.IndexOutOfRangeException` is thrown if *index* is negative, or larger than the bound of *array*.

`System.ArrayTypeMismatchException` is thrown if *array* doesn't hold elements of the required type.

Correctness:

Correct CIL ensures that *class* is a `typeref` or `typedef` or `typespec` 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.

Verifiability:

The type of *index* shall be `int32` or `native int`. The element type of *array* shall match *class* exactly.

 ©ISO/IEC 2006 – All rights reserved

4.10 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* shall 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 shall 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 can be either an instance field (in which case *obj* shall not be null) or a static field.

The ldfld instruction can be preceded by either or both of the unaligned. and volatile. prefixes.

Exceptions:

System.FieldAccessException is thrown if *field* is not accessible.

System.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.

System.NullReferenceException is thrown if *obj* is null and the field is not static.

Correctness:

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.

Verifiability:

For verifiable code, *obj* shall not be an unmanaged pointer.

It is not verifiable to access an overlapped object reference field.

A field is accessible only if every field that overlaps it is also accessible.

©ISO/IEC 2006 – All rights reserved

4.11 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 shall refer to a field member. The field can be either an instance field (in which case *obj* shall not be null) or a static field.

Exceptions:

System.FieldAccessException is thrown if *field* is not accessible.

System.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.

System.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.

System.NullReferenceException is thrown if *obj* is null and the field isn't static.

Correctness:

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.

Verifiability:

For verifiable code, *field* cannot be init-only.

It is not verifiable to access an overlapped object reference field.

A field is accessible only if every field that overlaps it is also accessible.

Remark:

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 might lead to unpredictable behavior.

 ©ISO/IEC 2006 – All rights reserved

4.12 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:

`System.NullReferenceException` is thrown if *array* is null.

Correctness:

Correct CIL ensures that *array* is indeed always null or a zero-based, one dimensional array.

Verifiability:

4.13 ldobj – copy a value from an address to the stack

Format	Assembly Format	Description
71 <*T*>	ldobj *typeTok*	Copy the value stored at address *src* to the stack.

Stack Transition:

…, src → …, val

Description:

The ldobj instruction copies a value to the evaluation stack. *typeTok* is a metadata token (a typedef, typeref, or typespec). *src* is an unmanaged pointer (native int), or a managed pointer (&). If *typeTok* is a reference type, the ldobj instruction has the same effect as ldind.ref.

[Rationale: The ldobj instruction can be used to pass a value type as an argument. *end rationale*]

The operation of the ldobj instruction can be altered by an immediately preceding volatile. or unaligned. prefix instruction.

Exceptions:

System.NullReferenceException can be thrown if an invalid address is detected.

System.TypeLoadException is thrown if *typeTok* cannot be found. This is typically detected when CIL is converted to native code rather than at runtime.

Correctness:

typeTok shall be a valid typedef, typeref, or typespec metadata token.

Verifiability:

The static type of the source value on top of the stack shall be a managed pointer to some type *srcType,* and *srcType* shall be a subtype of *typeTok*. The static type of the value remaining on the stack is t*ypeTok*.

 ©ISO/IEC 2006 – All rights reserved

4.14 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 can have a volatile. prefix.

Exceptions:

`System.FieldAccessException` is thrown if *field* is not accessible.

`System.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.

Correctness:

Correct CIL ensures that *field* is a valid metadata token referring to a static field member.

Verifiability:

There are no additional verification requirements.

4.15 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* can 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:

`System.FieldAccessException` is thrown if *field* is not accessible.

`System.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.

Correctness:

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.

Verifiability:

For verifiable code, *field* cannot be init-only.

Remark:

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 can lead to unpredictable behavior.

 ©ISO/IEC 2006 – All rights reserved

4.16 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* (which is a string literal).

By default, 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"). This behavior can be controlled using the `System.Runtime.CompilerServices.CompilationRelaxationsAttribute` and the `System.Runtime.CompilerServices.CompilationRelaxations.NoStringInterning` (see Partition IV).

Exceptions:

None.

Correctness:

Correct CIL requires that *string* is a valid string literal metadata token.

Verifiability:

There are no additional verification requirements.

4.17 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 shall be one of:

A `methoddef`, `methodref` or `methodspec`: pushes a `RuntimeMethodHandle`

A `typedef`, `typeref`, or `typespec` : 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.

Correctness:

Correct CIL requires that *token* describes a valid metadata token of the kinds listed above

Verifiability:

There are no additional verification requirements.

 ©ISO/IEC 2006 – All rights reserved

4.18 ldvirtftn – load a virtual method pointer

Format	Assembly Format	Description
FE 07 <*T*>	ldvirtftn *method*	Push address of virtual method *method* 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 *method* (a metadata token, a `methoddef`, `methodref` or `methodspec`; 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) if that routine expects the corresponding calling convention. Note that the address computed by this instruction can 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:

`System.NullReferenceException` is thrown if *object* is null.

Correctness:

Correct CIL ensures that *method* is a valid `methoddef`, `methodref` or `methodspec` token. Also that *method* references a non-static method that is defined for *object*.

Verifiability:

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.

4.19 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* shall 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 valid 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:

`System.TypeLoadException` is thrown if *class* cannot be found. This is typically detected when CIL is converted to native code rather than at runtime.

Correctness:

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.

Verifiability:

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*.

 ©ISO/IEC 2006 – All rights reserved

4.20 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 *etype***,** a metadata token (a `typeref`, `typedef` or `typespec`; see Partition II). *numElems* (of type `native int` or `int32`) specifies the number of elements in the array. Valid array indexes are 0 ≤ 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:

`System.OutOfMemoryException` is thrown if there is insufficient memory to satisfy the request.

`System.OverflowException` is thrown if *numElems* is < 0

Correctness:

Correct CIL ensures that *etype* is a valid `typeref`, `typedef` or `typespec` token.

Verifiability:

.*numElems* shall be of type `native int` or `int32`.

4.21 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 shall 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 *ctor* 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:

`System.InvalidOperationException` is thrown if *ctor*'s class is abstract.

`System.OutOfMemoryException` is thrown if there is insufficient memory to satisfy the request.

`System.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.

Correctness:

Correct CIL ensures that *ctor* is a valid `methodref` or `methoddef` token, and that the arguments on the stack are compatible with those expected by the constructor.

Verifiability:

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.

 ©ISO/IEC 2006 – All rights reserved

4.22 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.

Correctness:

Correct CIL ensures that *TypedRef* is a valid typed reference (created by a previous call to mkrefany).

Verifiability:

The refanytype instruction is always verifiable.

4.23 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* shall match the type specified by *type* (a metadata token, either a `typedef` or a `typeref`; see Partition II). See the mkrefany instruction.

Exceptions:

`System.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*)

`System.TypeLoadException` is thrown if *type* cannot be found.

Correctness:

Correct CIL ensures that *TypedRef* is a valid typed reference (created by a previous call to mkrefany).

Verifiability:

The refanyval instruction is always verifiable.

 ©ISO/IEC 2006 – All rights reserved

4.24 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. A rethrow does not change the stack trace in the object.

Exceptions:

The original exception is thrown.

Correctness:

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, an exception will be thrown, but precisely which exception, is undefined

Verifiability:

There are no additional verification requirements.

©ISO/IEC 2006 – All rights reserved

4.25 sizeof – load the size, in bytes,of a type

Format	Assembly Format	Description
FE 1C <*T*>	sizeof *typeTok*	Push the size, in bytes, of a type as an `unsigned int32`.

Stack Transition:

…, → …, size (4 bytes, unsigned)

Description:

Returns the size, in bytes, of a type. *typeTok* can be a generic parameter, a reference type or a value type.

For a reference type, the size returned is the size of a reference value of the corresponding type, not the size of the data stored in objects referred to by a reference value.

[*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 type – including any padding the implementation chooses to add. Specifically, array elements lie `sizeof` bytes apart. *end rationale*]

Exceptions:

None.

Correctness:

typeTok shall be a `typedef`, `typeref`, or `typespec` metadata token.

Verifiability:

It is always verifiable.

 ©ISO/IEC 2006 – All rights reserved

4.26 stelem – store element to array

Format	Assembly Format	Description
A4 <*T*>	stelem *typeTok*	Replace array element at *index* with the *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 `native int` or `int32`) in the one-dimensional array *array*, with *value*. Arrays are objects and hence are represented by a value of type `O`. The value has the type specified by the token *typeTok* in the instruction.

Exceptions:

`System.NullReferenceException` is thrown if *array* is null.

`System.IndexOutOfRangeException` is thrown if *index* is larger than the bound of *array*.

`System.ArrayTypeMismatchException` is thrown if *array* doesn't hold elements of the required type.

Correctness:

typeTok shall be a valid `typedef`, `typeref`, or `typespec` metadata token.

array shall be null or a single dimensional array.

Verifiability:

The static type of array shall either be the special reference type `Null` or a proper zero-based, one-dimensional array type `elem[]`, for some type *elem*. If the array type is `Null`, take *elem* to be the type represented by type operand *typeTok*. The value *index* shall have type `native int`. The type of *elem* shall be a supertype of the type operand *typeTok*. The type of *value* shall be assignment-compatible with *typeTok*.

©ISO/IEC 2006 – All rights reserved

4.27 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 can throw the `ArrayTypeMismatchException`.

For one-dimensional arrays that aren't zero-based and for multidimensional arrays, the array class provides a `StoreElement` method.

Exceptions:

`System.NullReferenceException` is thrown if *array* is null.

`System.IndexOutOfRangeException` is thrown if *index* is negative, or larger than the bound of *array*.

`System.ArrayTypeMismatchException` is thrown if *array* doesn't hold elements of the required type.

Correctness:

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 (e.g., stelem.r4 can only be applied to a zero-based, one dimensional array of `float32`s).

Verifiability:

The static type of *array* shall either be the special reference type Null or a proper zero-based, one-dimensional array type *elem*[], for some type *elem*. Both the type of *array* and the type of *value* shall be consistent with the instruction suffix. For the stelem.ref instruction, it is required only that the value and array element type are both reference types.

 ©ISO/IEC 2006 – All rights reserved

4.28 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 *value*.

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 can have a prefix of either or both of unaligned. and volatile..

Exceptions:

System.FieldAccessException is thrown if *field* is not accessible.

System.NullReferenceException is thrown if *obj* is null and the field isn't static.

System.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.

Correctness:

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, subject to implicit conversion as specified in §1.6.

Verifiability:

For verifiable code, *obj* shall 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 can lead to unpredictable behavior. It cannot, however, compromise memory integrity or type safety so it is not tested by verification. *end note*]

It is not verifiable to access an overlapped object reference field.

A field is accessible only if every field that overlaps it is also accessible.

4.29 stobj – store a value at an address

Format	Assembly Format	Description
81 <*T*>	stobj *typeTok*	Store a value of type *typeTok* at an address.

Stack Transition:

…, dest, src → …,

Description:

If *typeTok* is a value type, the stobj instruction copies the value *src* to the address *dest*. If *typeTok* is a reference type, the stobj instruction has the same effect as stind.ref.

The operation of the stobj instruction can be altered by an immediately preceding volatile. or unaligned. prefix instruction.

Exceptions:

System.NullReferenceException can be thrown if an invalid address is detected.

System.TypeLoadException is thrown if *typeTok* cannot be found. This is typically detected when CIL is converted to native code rather than at runtime.

Correctness:

typeTok shall be a valid typedef, typeref, or typespec metadata token.

Verifiability:

Let the static type of the value on top of the stack be some type *srcType*. The value shall be initialized (when *srcType* is a reference type). The static type of the destination address *dest* on the preceding stack slot shall be a managed pointer (of type destType &) to some type *destType*. Finally, *srcType* shall be assignment-compatible with *typeTok*, and *typeTok* shall be a subtype of *destType*. When *typeTok* is a non-reference type, the definition of subtyping implies that *srcType* shall be assignment-compatible with *destType* (which, itself, shall be equal to *typeTok*).

 ©ISO/IEC 2006 – All rights reserved

4.30 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 shall refer to a static field member. stsfld pops the value off the stack and updates the static field with that value.

The stsfld instruction can have a volatile. prefix.

Exceptions:

`System.FieldAccessException` is thrown if *field* is not accessible.

`System.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.

Correctness:

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, subject to implicit conversion as specified in §1.6.

Verifiability:

[*Note:* Using stsfld to change the value of a static, init-only field outside the body of the class initializer can lead to unpredictable behavior. It cannot, however, compromise memory integrity or type safety so it is not tested by verification. *end note*]

4.31 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 and empties the stack. For details of the exception mechanism, see Partition I.

[*Note:* While the CLI permits any object to be thrown, the CLS describes a specific exception class that shall be used for language interoperability. *end note*]

Exceptions:

System.NullReferenceException is thrown if *obj* is null.

Correctness:

Correct CIL ensures that *object* is always either null or an object reference (i.e., of type O) .

Verifiability:

There are no additional verification requirements.

 ©ISO/IEC 2006 – All rights reserved

4.32 unbox – convert boxed value type to its raw form

Format	Assembly Format	Description
79 <*T*>	unbox *valuetype*	Extract a value-type 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 controlled-mutability managed pointer (§1.8.1.2.2), type `&`), its unboxed form. *valuetype* is a metadata token (a `typeref`, `typedef` or `typespec`). The type of *value type* contained within *obj* must be assignment compatible with *valuetype*. [*Note*: This effects the behavior with enum types, see Partition II.14.3. *end note*]

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.

[*Note*: Typically, unbox simply computes the address of the value type that is already present inside of the boxed object. This approach is not possible when unboxing nullable value types. Because Nullable<*T*> values are converted to boxed *T*s during the box operation, an implementation often must manufacture a new Nullable<*T*> on the heap and compute the address to the newly allocated object. *end note*]

Exceptions:

`System.InvalidCastException` is thrown if *obj* is not a boxed value type, *valuetype* is a Nullable<*T*> and *obj* is not a boxed *T*, or if the type of the value contained in *obj* is not assignment compatible with *valuetype*.

`System.NullReferenceException` is thrown if *obj* is null and *valuetype* is a non-nullable value type (Partition I.8.2.4).

`System.TypeLoadException` is thrown if the class cannot be found. (This is typically detected when CIL is converted to native code rather than at runtime.)

Correctness:

Correct CIL ensures that *valueType* is a `typeref`, `typedef` or `typespec` metadata token for some value type, and that *obj* is always an object reference (i.e., of type `O`). If *valuetype* is the type Nullable<*T*>, the boxed instance shall be of type *T*.

Verifiability:

There are no additional verification requirements.

4.33 unbox.any – convert boxed type to value

Format	Assembly Format	Description
A5 <*T*>	unbox.any *typeTok*	Extract a value-type from *obj*, its boxed representation

Stack Transition:

…, obj → …, value or obj

Description:

When applied to the boxed form of a value type, the unbox.any instruction extracts the value contained within *obj* (of type O). (It is equivalent to unbox followed by ldobj.) When applied to a reference type, the unbox.any instruction has the same effect as castclass *typeTok*.

If *typeTok* is a *GenericParam*, the runtime behavior is determined by the actual instantiation of that parameter.

Exceptions:

System.InvalidCastException is thrown if *obj* is not a boxed value type or a reference type, *typeTok* is Nullable<*T*> and *obj* is not a boxed *T*, or if the type of the value contained in *obj* is not assignment compatible with *typeTok*.

System.NullReferenceException is thrown if *obj* is null and *typeTok* is a non-nullable value type (Partition I.8.2.4).

Correctness:

obj shall be of reference type.

Verifiability:

The *obj* object reference shall be initialized. The type operand *typeTok* shall not be a byref, byref-like, or void type. The type of the value left on the stack is *typeTok*.

Rationale:

There are two reasons for having both unbox.any and unbox instructions:

1. Unlike the unbox instruction, for value types, unbox.any leaves a value, not an address of a value, on the stack.
2. The type operand to unbox has a restriction: it can only represent value types and instantiations of generic value types.

 ©ISO/IEC 2006 – All rights reserved

5 Index

ArithmeticException....57
array covariance....21
binary format....1
controlled-mutability managed pointer....**21**
ELEMENT_TYPE_BYREF....5
ELEMENT_TYPE_PTR....4
endianess....6
evaluation stack....11
exception_continue_search....67
exception_execute_handler....67
Execution Engine....1
ExecutionEngineException....1, 23
IMAGE_COR_ILMETHOD....17
instruction description....**1**
  assembly language notation....1
  binary format....*See* binary format
  exception list....1
  operand type table....1
  stack transition diagram....1
  variants....1
  verifiability....1
instruction format....**16**
instruction prefix....24
  constrained....*See* constrained.
  no....*See* no.
  readonly....*See* readonly.
  tail....*See* tail.
  unaligned....*See* unaligned.
  volatile....*See* volatile.
Instruction Variant Table....5
integer
  4-byte....2
  8-byte....2
  native-size....2
  short....2
  unsigned....3
localsinit flag....19, 77, 78, 81
maxstack....17
metadata token....**23**
null....**1**
opcode....5, **6**
  experimental....**6**
  standard....**6**
operand type....11
OutOfMemoryException....1
pointer....4
  controlled-mutability managed....See controlled-mutability managed pointer
  managed....**5**
  unmanaged....**4**
profile....1
RuntimeArgumentHandle....36
RuntimeFieldHandle....120
RuntimeHandle....120
RuntimeMethodHandle....120
RuntimeTypeHandle....120
SecurityPermission.SkipVerification....77, 78
stack transition....1, **11**
StackOverflowException....1
string interning....119
System.ArgIterator....36
System.Array....105, 108
System.Math.IEEERemainder....89
System.RuntimeArgumentHandle....*See* RuntimeArgumentHandle
thunk....74, 121
type
  Boolean....4
  floating-point....3
  numeric....2
  pointer....*See* pointer

©ISO/IEC 2006 – All rights reserved

reference 4
User String heap 23
verifiability 1, **18**

 ©ISO/IEC 2006 – All rights reserved

Common Language Infrastructure (CLI)
Partition IV:
Profiles and Libraries

Table of contents

1	**Overview**	**1**
2	**Libraries and Profiles**	**2**
2.1	Libraries	2
2.2	Profiles	2
2.3	The relationship between Libraries and Profiles	3
3	**The Standard Profiles**	**4**
3.1	The Kernel Profile	4
3.2	The Compact Profile	4
4	**Kernel Profile feature requirements**	**5**
4.1	Features excluded from the Kernel Profile	5
4.1.1	Floating point	5
4.1.2	Non-vector arrays	5
4.1.3	Reflection	5
4.1.4	Application domains	6
4.1.5	Remoting	6
4.1.6	Vararg	6
4.1.7	Frame growth	6
4.1.8	Filtered exceptions	6
5	**The standard libraries**	**7**
5.1	General comments	7
5.2	Runtime infrastructure library	7
5.3	Base Class Library (BCL)	7
5.4	Network library	7
5.5	Reflection library	7
5.6	XML library	7
5.7	Extended numerics library	8
5.8	Extended array library	8
5.9	Vararg library	8
5.10	Parallel library	8
6	**Implementation-specific modifications to the system libraries**	**10**
7	**The XML specification**	**11**

©ISO/IEC 2006 – All rights reserved

7.1	Semantics	11
7.1.1	Value types as objects	19
7.1.2	Exceptions	19
7.2	XML signature notation issues	19
7.2.1	Serialization	19
7.2.2	Delegates	19
7.2.3	Properties	20
7.2.4	Nested types	20

 ©ISO/IEC 2006 – All rights reserved

1 Overview

[*Note:* While compiler writers are most concerned with issues of file format, instruction set design, and a common type system, application programmers are most interested in the programming library that is available to them in the language they are using. The Common Language Infrastructure (CLI) specifies a Common Language Specification (CLS, see Partition I) that shall be used to define the externally visible aspects (such as method signatures) when they are intended to be used from a wide range of programming languages. Since it is the goal of the CLI Libraries to be available from as many programming languages as possible, all of the library functionality is available through CLS-compliant types and type members.

The CLI Libraries were designed with the following goals in mind:

- To support for a wide variety of programming languages.
- To have consistent design patterns throughout.
- To have features on parity with the ISO/IEC C Standard library of 1990.
- To support more recent programming paradigms, notably networking, XML, runtime type inspection, instance creation, and dynamic method dispatch.
- To be factored into self-consistent libraries with minimal interdependence.

end note]

This partition provides an overview of the CLI Libraries, and a specification of their factoring into Profiles and Libraries. A companion file, considered to be part of this Partition but distributed in XML format, provides details of each type in the CLI Libraries. While the normative specification of the CLI Libraries is in XML form, it can be processed using an XSL transform to produce easily browsed information about the Class Libraries.

[*Note:* Partition VI contains an informative annex describing programming conventions used in defining the CLI Libraries. These conventions significantly simplify the use of libraries. Implementers are encouraged to follow them when creating additional (non-standard) Libraries. *end note*]

©ISO/IEC 2006 – All rights reserved

2 Libraries and Profiles

Libraries and Profiles, defined below, are constructs created for the purpose of standards conformance. They specify a set of features that shall be present in an implementation of the CLI, and a set of types that shall be available to programs run by that CLI.

[*Note:* There need not be any direct support for Libraries and Profiles in the Virtual Execution System (VES). They are not represented in the metadata and they have no impact on the structure or performance of an implementation of the CLI. Libraries and Profiles can span assemblies (the deployment unit), and the names of types in a single Library or Profile are not required to have a common prefix ("namespace"). *end note*]

In general, there is no way to test whether a feature is available at runtime, nor a way to enquire whether a particular Profile or Library is available. If present, however, the Reflection Library makes it possible to test, at runtime, for the existence of particular types and members.

2.1 Libraries

A Library specifies three things:

1. A set of types that shall be available, including their grouping into assemblies. (The standard library types are contained in three assemblies: mscorlib, System, and System.Xml. The specification for each type indicates the assembly in which it resides.)
2. A set of features of the CLI that shall be available.

   [*Note*: The set of features required for any particular Library is a subset of the complete set of CLI features. Each Library described in §5 has text that defines the CLI features that are required for implementations that support that Library. *end note*]
3. Modifications to types defined in *other* Libraries. These modifications typically involve the addition of methods and interfaces to types belonging to some other Library, and additional exception types that can be thrown by methods of that other Library's types. These modifications shall provide only additional functionality or specify behavior where it was previously unspecified; they shall not be used to alter previously specified behavior.

[*Example*: Consider the Extended Numerics Library. Since it provides a base data type, `Double`, it also specifies that the method `ToDouble` be added to the `System.Convert` class that is part of the Base Class Library. It also defines a new exception type, `System.NotFiniteNumberException`, and specifies existing methods in other Libraries methods that throw it (as it happens, there are no such methods). *end example*]

In the XML specification of the Libraries, each type specifies the Library to which it belongs. For those members (e.g., `Console.WriteLine(float)`) that are part of one Library (such as Extended Numerics (§5.7)), but whose type is in another Library (such as Base Class Library (§5.3)), the XML specifies the Library that defines the method. See §7.

2.2 Profiles

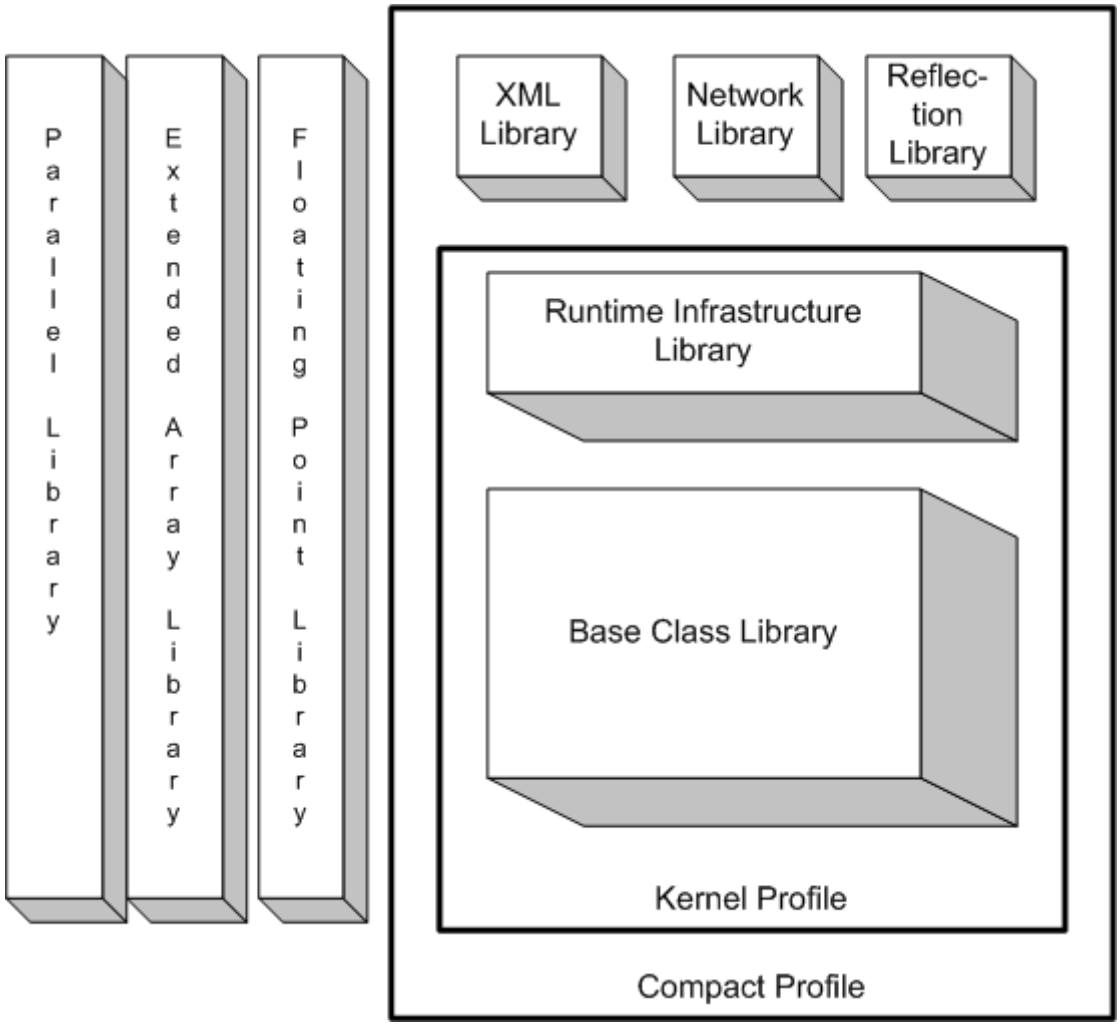
A Profile is simply a set of Libraries, grouped together to form a consistent whole that provides a fixed level of functionality. A conforming implementation of the CLI shall specify the Profile it implements, as well as any additional Libraries that it provides. The Kernel Profile (§3.1) shall be included in all conforming implementations of the CLI. Thus, all Libraries and CLI features that are part of the Kernel Profile are available in all conforming implementations. This minimal feature set is described in §4.

[*Rationale:* The rules for combining Libraries together are complex, since each Library can add members to types defined in other libraries. By standardizing a small number of Profiles the interaction of the Libraries that are part of each Profile are specified completely. A Profile provides a consistent target for vendors of devices, compilers, tools, and applications. Each Profile specifies a trade-off of CLI feature and implementation complexity against resource constraints. By defining a very small number of Profiles, market for each Profile is increased, making each a desirable target for a class of applications across a wide range of implementations and tool sets. *end rationale*]

 ©ISO/IEC 2006 – All rights reserved

2.3 The relationship between Libraries and Profiles

This standard specifies two *Standard Profiles* (§3) and seven *Standard Libraries* (§5). The following diagram shows the relationship between the Libraries and the Profiles:



The Extended Array and Extended Numerics Libraries are not part of either Profile, but can be combined with either of them. Doing so adds the appropriate methods, exceptions, and interfaces to the types specified in the Profile.

©ISO/IEC 2006 – All rights reserved

3 The Standard Profiles

There are two Standard Profiles. The smallest conforming implementation of the CLI is the *Kernel Profile*, while the *Compact Profile* contains additional features useful for applications targeting a more resource-rich set of devices.

A conforming implementation of the CLI shall throw an appropriate exception (e.g., `System.NotImplementedException`, `System.MissingMethodException`, or `System.ExecutionEngineException`) when it encounters a feature specified in this Standard but not supported by the particular Profile (see Partition III).

[*Note:* Implementers should consider providing tools that statically detect features they do not support so users have an option of checking programs for the presence of such features before running them. *end note*]

[*Note:* Vendors of compliant CLI implementations should specify exactly which configurations of Standard Libraries and Standard Profiles they support. *end note*]

[*Note:* "Features" can be something like the use of a floating-point CIL instruction in the implementation of a method when the CLI upon which it is running does not support the Extended Numerics Library. Or, the "feature" might be a call to a method that this Standard specifies exists only when a particular Library is implemented and yet the code making the call is running on an implementation of the CLI that does not support that particular library. *end note*]

3.1 The Kernel Profile

This Profile is the minimal possible conforming implementation of the CLI. It contains the types commonly found in a modern programming language class library, plus the types needed by compilers targeting the CLI.

Contents: Base Class Library, Runtime Infrastructure Library

3.2 The Compact Profile

This Profile is designed to allow implementation on devices with only modest amounts of physical memory yet provides more functionality than the Kernel Profile alone. It also contains everything required to implement the proposed ECMAScript compact Profile.

Contents: Kernel Profile, XML Library, Network Library, Reflection Library

 ©ISO/IEC 2006 – All rights reserved

4 Kernel Profile feature requirements

All conforming implementations of the CLI support at least the Kernel Profile. Consequently, all CLI features required by the Kernel Profile shall be implemented by all conforming implementations. This clause defines that minimal feature set, by enumerating the set of features that are not required; i.e., a minimal conforming implementation shall implement all CLI features except those specified in the remainder of this clause. The feature requirements of individual Libraries as specified in §5 are defined by reference to restricted items described in this clause. For ease of reference, each feature has a name indicated by the name of the clause or subclause heading. Where Libraries do not specify any additional feature requirement, it shall be assumed that only the features of the Kernel Profile as described in this clause are required.

4.1 Features excluded from the Kernel Profile

The following internal data types and constructs, specified elsewhere in this Standard, are *not* required of CLI implementations that conform only to the Kernel Profile. All other CLI features are required.

4.1.1 Floating point

The **floating point feature set** consists of the user-visible floating-point data types `float32` and `float64`, and support for an internal representation of floating-point numbers.

If omitted: The CIL instructions that deal specifically with these data types throw the `System.NotImplementedException` exception. These instructions are: ckfinite, conv.r.un, conv.r4, conv.r8, ldc.r4, ldc.r8, ldelem.r4, ldelem.r8, ldind.r4, ldind.r8, stelem.r4, stelem.r8, stind.r4, stind.r8. Any attempt to reference a signature including the floating-point data types shall throw the `System.NotImplementedException` exception. The precise timing of the exception is not specified.

[*Note:* These restrictions guarantee that the VES will not encounter any floating-point data. Hence the implementation of the arithmetic instructions (such as add) need not handle those types. *end note*]

Part of Library: Extended Numerics (§5.7)

4.1.2 Non-vector arrays

The **non-vector arrays feature set** includes support for arrays with more than one dimension or with lower bounds other than zero. This includes support for signatures referencing such arrays, runtime representations of such arrays, and marshalling of such arrays to and from native data types.

If omitted: Any attempt to reference a signature including a non-vector array shall throw the `System.NotImplementedException` exception. The precise timing of the exception is not specified.

[*Note:* The type `System.Array` is part of the Kernel Profile and is available in all conforming implementations of the CLI. An implementation that does not provide the non-vector array feature set can correctly assume that all instances of that type are vectors. *end note*]

Part of Library: Extended Arrays (see §5.8).

4.1.3 Reflection

The **reflection feature set** supports full reflection on data types. All of its functionality is exposed through methods in the Reflection Library.

If omitted: The Kernel Profile specifies an opaque type, `System.Type`, instances of which uniquely represent any type in the system and provide access to the name of the type.

[*Note:* With just the Kernel Profile there is no requirement, for example, to determine the members of the type, dynamically create instances of the type, or invoke methods of the type given an instance of `System.Type`. This can simplify the implementation of the CLI compared to that required when the Reflection Library is available. *end note*]

Part of Library: Reflection (see §5.5).

©ISO/IEC 2006 – All rights reserved

4.1.4 Application domains

The **application domain feature set** supports multiple application domains. The Kernel Profile requires that a single application domain exist.

If omitted: Methods for creating application domains (part of the Base Class Library, see §5.3) throw the `System.NotImplementedException` exception.

Part of Library: (none)

4.1.5 Remoting

The **remoting feature set** supports remote method invocation. It is provided primarily through special semantics of the class `System.MarshalByRefObject` as described in Partition I.

If omitted: The class `System.MarshalByRefObject` shall be treated as a simple class with no special meaning.

Part of Library: (none)

4.1.6 Vararg

The **vararg feature set** supports variable-length argument lists and runtime-typed pointers.

If omitted: Any attempt to reference a method with the `vararg` calling convention or the signature encodings associated with vararg methods (see Partition II) shall throw the `System.NotImplementedException` exception. Methods using the CIL instructions arglist, refanytype, mkrefany, and refanyval shall throw the `System.NotImplementedException` exception. The precise timing of the exception is not specified. The type `System.TypedReference` need not be defined.

Part of Library: Vararg (see §5.9).

4.1.7 Frame growth

The **frame growth feature set** supports dynamically extending a stack frame.

If omitted: Methods using the CIL localloc instruction shall throw the `System.NotImplementedException` exception. The precise timing of the exception is not specified.

Part of Library: (none)

4.1.8 Filtered exceptions

The **filtered exceptions feature set** supports user-supplied filters for exceptions.

If omitted: Methods using the CIL endfilter instruction or with an **`exceptionentry`** that contains a non-null **`filterstart`** (see Partition I) shall throw the `System.NotImplementedException` exception. The precise timing of the exception is not specified.

Part of Library: (none)

 ©ISO/IEC 2006 – All rights reserved

5 The standard libraries

The detailed content of each Library, in terms of the types it provides and the changes it makes to types in other Libraries, is provided in XML form. This clause provides a brief description of each Library's purpose as well as specifying the features of the CLI required by each Library beyond those required by the Kernel Profile.

5.1 General comments

Unless stated otherwise in the documentation of a method, all copy operations are shallow, not deep.

Some methods traffic in "default values". For a reference type, the default value is null; for a nullable value type, the default value is `HasValue` returns false; for a non-nullable value type, the default value is all-bits-zero (which for Boolean represents false, and for all arithmetic types represents zero).

5.2 Runtime infrastructure library

The Runtime Infrastructure Library is part of the Kernel Profile. It provides the services needed by a compiler to target the CLI and the facilities needed to dynamically load types from a stream in the file format specified in Partition II. For example, it provides `System.BadImageFormatException`, which is thrown when a stream that does not have the correct format is loaded.

Name used in XML: RuntimeInfrastructure

CLI Feature Requirement: None

5.3 Base Class Library (BCL)

The Base Class Library is part of the Kernel Profile. It is a simple runtime library for modern programming languages. It serves as the Standard for the runtime library for the language C# as well as one of the CLI Standard Libraries. It provides types to represent the built-in data types of the CLI, simple file access, custom attributes, security attributes, string manipulation, formatting, streams, collections, among other things.

Name used in XML: BCL

CLI Feature Requirement: None

5.4 Network library

The Network Library is part of the Compact Profile. It provides simple networking services including direct access to network ports as well as HTTP support.

Name used in XML: Networking

CLI Feature Requirement: None

5.5 Reflection library

The Reflection Library is part of the Compact Profile. It provides the ability to examine the structure of types, create instances of types, and invoke methods on types, all based on a description of the type.

Name used in XML: Reflection

CLI Feature Requirement: Must support Reflection, see §5.1.

5.6 XML library

The XML Library is part of the Compact Profile. It provides a simple "pull-style" parser for XML. It is designed for resource-constrained devices, yet provides a simple user model. A conforming implementation of the CLI that includes the XML Library shall also implement the Network Library (see §5.4).

Name used in XML: XML

CLI Feature Requirement: None

©ISO/IEC 2006 – All rights reserved

5.7 Extended numerics library

The Extended Numerics Library is not part of any Profile, but can be supplied as part of any CLI implementation. It provides the support for floating-point (`System.Single`, `System.Double`) and extended-precision (`System.Decimal`) data types. Like the Base Class Library, this Library is directly referenced by the C# Standard.

[*Note:* Programmers who use this library will benefit if implementations specify which arithmetic operations on these data types are implemented primarily through hardware support. *end note*]

[*Rationale:* The Extended Numerics Library is kept separate because some commonly available processors do not provide direct support for the data types. While software emulation can be provided, the performance difference is often so large (1,000-fold or more) that it is unreasonable to build software using floating-point operations without being aware of whether the underlying implementation is hardware-based. *end rationale*]

Name used in XML: ExtendedNumerics

CLI Feature Requirement: Floating Point, see §4.1.1.

5.8 Extended array library

This Library is not part of any Profile, but can be supplied as part of any CLI implementation. It provides support for non-vector arrays. That is, arrays that have more than one dimension, or arrays that have non-zero lower bounds.

CLI Feature Requirement: Non-vector Arrays, see § 4.1.2.

5.9 Vararg library

The Vararg Library is not part of any Profile. It provides support for dealing with variable-length argument lists.

Name used in XML: Vararg

CLI Feature Requirement: None

5.10 Parallel library

This Library is not part of any Profile, but can be supplied as part of any CLI implementation. The purpose of the extended threading library is twofold:

1. Provide easy parallelism for non-expert programmers, so that multithreaded CPUs can be exploited. The Profile stresses simplicity over large scalability.
2. Not require changing the virtual machine or source languages. All features of the Profile can be implemented as a library on top of the existing CLI. The Profile can be used in conjunction with any CLI language that supports delegates.

The loop class hierarchy is summarized below:

```
ParallelLoop
   ParallelWhile
   ParallelForEach
   ParallelFor
```

The base class `ParallelLoop` factors out common functionality for parallel looping over a collection of values. The three derived classes distinguish three common kinds of parallel looping. If the collection might grow while being processed, then use `ParallelWhile`. Otherwise, if the collection implements `IEnumerable`, use `ParallelForEach`. If the collection or collections are indexible by `int32`, use `ParallelFor`.

To choose the kind of loop to use in a specific situation, consider how the loop could be written sequentially. If the loop could be written using "for (int *i*=0; *i*<*n*; ++*i*)", and *n* is known before the loop executes, use `ParallelFor`. If the loop could be written with a `foreach` statement, over collection that does not change

 ©ISO/IEC 2006 – All rights reserved

while the `foreach` is running, use ParallelForEach. If the loop could be written "while (collection is not yet empty) {remove item from collection and process it}", use `ParallelWhile`. When there is a choice, use `ParallelFor` if possible, because it is significantly more efficient.

Name used in XML: Parallel

CLI Feature Requirement: BCL

6 Implementation-specific modifications to the system libraries

Implementers are encouraged to extend or modify the types specified in this Standard to provide additional functionality. Implementers should notice, however, that type names beginning with "`System.`" and bearing the special Standard Public Key are intended for use by the Standard Libraries: such names not currently in use might be defined in a future version of this Standard.

To allow programs compiled against the Standard Libraries to work when run on implementations that have extended or modified the Standard Libraries, such extensions or modifications shall obey the following rules:

- The contract specified by virtual methods shall be maintained in new classes that override them.
- New exceptions can be thrown, but where possible these should be derived classes of the exceptions already specified as thrown rather than entirely new exception types. Exceptions initiated by methods of types defined in the Standard Libraries shall be derived from `System.Exception`.
- Interfaces and virtual methods shall not be added to an existing interface. Nor shall they be added to an abstract class unless that class provides an implementation.

  [*Rationale:* An interface or virtual method can be added only where it carries an implementation. This allows programs written when the interface or method was not present to continue to work. *end rationale*]
- Instance methods shall not be implemented as virtual methods.

  [*Rationale:* Methods specified as instance (non-static, non-virtual) in this standard are not permitted to be implemented as virtual methods in order to reduce the likelihood of creating non-portable files by using implementation-supplied libraries at compile time. Even though a compiler need not take a dependence on the distinction between virtual and instance methods, it is easy for a user to inadvertently override a virtual method and thus create non-portable code. The alternative of providing special files corresponding to this Standard for use at compile time is prone to user error. *end rationale*]
- The accessibility of fields and non-virtual methods can be widened from than specified in this Standard.

[*Note:* The following common extensions are permitted by these rules.

- Adding new members to existing types.
- Concrete (non-abstract) classes can implement interfaces not defined in this standard.
- Adding fields (values) to enumerations.
- An implementation can insert a new type into the hierarchy between a type specified in this standard and the type specified as its base type. That is, this standard specifies an inheritance relation between types but does not specify the immediate base type.
- Implementations can add overrides to existing virtual methods, provided the new overrides satisfy the existing contract.

end note]

[*Rationale:* An implementation might wish to split functionality across several types in order to provide non-standard extension mechanisms, or might wish to provide additional non-standard functionality through the new base type. As long as programs do not reference these non-standard types, they will remain portable across conforming implementations of the CLI. *end rationale*]

 ©ISO/IEC 2006 – All rights reserved

7 The XML specification

7.1 Semantics

The XML specification conforms to the Document Type Definition (DTD) in Figure 7-1. Only types that are included in a specified library are included in the XML.

There are three types of elements/attributes:

- Normative: An element or attribute is normative such that the XML specification would be incomplete without it.
- Informative: An element or attribute is informative if it specifies information that helps clarify the XML specification, but without it the specification still stands alone.
- Rendering/Formatting: An element or attribute is for rendering or formatting if it specifies information to help an XML rendering tool.

Unless explicitly stated otherwise, the text associated with an element or an attribute (e.g., #PCDATA, #CDATA) is normative or informative depending on the element or attribute with which it is associated, as described in the figure.

[*Note*: Many of the elements and attributes in the DTD are for rendering purposes. *end note*]

Figure 7-1: XML DTD

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!ELEMENT AssemblyCulture (#PCDATA)>
```

(Normative) Specifies the culture of the assembly that defines the current type. Currently this value is always “none”. It is reserved for future use.

```
<!ELEMENT AssemblyInfo (AssemblyName, AssemblyPublicKey,
AssemblyVersion, AssemblyCulture, Attributes)>
```

(Normative) Specifies information about the assembly of a given type. This information corresponds to sections of the metadata of an assembly as described in Partition II, and includes information from the AssemblyName, AssemblyPublicKey, AssemblyVersion, AssemblyCulture and Attributes elements.

```
<!ELEMENT AssemblyName (#PCDATA)>
```

(Normative) Specifies the name of the assembly to which a given type belongs. For example, all of the types in the BCL are members of the “mscorlib” assembly.

```
<!ELEMENT AssemblyPublicKey (#PCDATA)>
```

(Normative) Specifies the public key of the assembly. The public key is represented as a 128-bit value.

```
<!ELEMENT AssemblyVersion (#PCDATA)>
```

(Normative) Specifies the version of the assembly in the form 2.0.x.y, where x is a build number and y is a revision number.

```
<!ELEMENT Attribute (AttributeName, Excluded, ExcludedTypeName?,
ExcludedLibraryName?)>
```

(Normative) Specifies the text for a custom attribute on a type or a member of a type. This includes the attribute name and whether or not the attribute type itself is contained in another library.

```
<!ELEMENT AttributeName (#PCDATA)>
```

(Normative) Specifies the name of the custom attribute associated with a type or member of a type. Also contains the data needed to instantiate the attribute.

```
<!ELEMENT Attributes (Attribute*)>
```

(Normative) Specifies the list of the attributes on a given type or member of a type.

<!ELEMENT Base (BaseTypeName?, ExcludedBaseTypeName?, ExcludedLibraryName?)>

(Normative) Specifies the information related to the base type of the current type. Although the **ExcludedBaseTypeName** and **ExcludedLibraryName** elements are rarely found within this element, they are required when a type inherits from a type not found in the current library.

<!ELEMENT BaseTypeName (#PCDATA)>

(Normative) Specifies the fully qualified name of the class from which a type inherits (i.e., the type's base class).

<!ELEMENT Docs (summary?, altmember?, altcompliant?, param*, returns?, value?, exception*, threadsafe?, remarks?, example?, permission?)>

(Normative) Specifies the textual documentation of a given type or member of a type.

<!ELEMENT Excluded (#PCDATA)>

(Normative) Specifies, by a '0' or '1', whether a given member can be excluded from the current type in the absence of a given library. '0' specifies that it cannot be excluded.

<!ELEMENT ExcludedBaseTypeName (#PCDATA)>

(Normative) Specifies the fully qualified name of the type that the current type must inherit from if a given library were present in an implementation. The library name is specified in the **ExcludedLibraryName** element. An example is the System.Type class that inherits from System.Object, but if the Reflection library is present, it must inherit from System.Reflection.MemberInfo.

<!ELEMENT ExcludedLibrary (#PCDATA)>

(Normative) Specifies the library that must be present in order for a given member of a type to be required to be implemented. For example, System.Console.WriteLine(double) need only be implemented if the ExtendedNumerics library is available.

<!ELEMENT ExcludedLibraryName (#PCDATA)>

(Normative) This element appears only in the description of custom attributes. It specifies the name of the library that defines the described attribute. For example, the member that is invoked when no member name is specified for System.Text.StringBuilder (in C#, this is the indexer) is called "Chars". The attribute needed for this is System.Reflection.DefaultMemberAttribute. This is found in the RuntimeInfrastructure library. This element is used with the **ExcludedTypeName** element.

<!ELEMENT ExcludedTypeName (#PCDATA)>

(Normative) Specifies the fully qualified name of the attribute that is needed for a member to succesfully specify the given attribute. This element is related to the **ExcludedLibraryName** element and is used for attributes.

<!ELEMENT Interface (InterfaceName, Excluded)>

(Normative) Specifies information about an interface that a type implements. This element contains sub-elements specifying the interface name and whether another library is needed for the interface to be required in the current library.

<!ELEMENT InterfaceName (#PCDATA)>

(Normative) Represents the fully-qualified interface name that a type implements.

<!ELEMENT Interfaces (Interface*)>

(Normative) Specifies information on the interfaces, if any, a type implements. There is one **Interface** element for each interface implemented by the type.

<!ELEMENT Libraries (Types+)>

(Normative) This is the root element. Specifies all of the information necessary for all of the class libraries of the standard. This includes all of the types and all children elements underneath.

 ©ISO/IEC 2006 – All rights reserved

<!ELEMENT Member (MemberSignature+, MemberType, Attributes?, ReturnValue, Parameters, MemberValue?, Docs, Excluded, ExcludedLibrary*)>

(Normative) Specifies information about a member of a type. This information includes the signatures, type of the member, parameters, etc., all of which are elements in the XML specification.

<!ATTLIST Member

MemberName NMTOKEN #REQUIRED

(Normative) **MemberName** specifies the name of the current member.

>

<!ELEMENT MemberOfLibrary (#PCDATA)>

(Normative) **PCDATA** is the name of the library containing the type.

<!ELEMENT MemberSignature EMPTY>

(Normative) Specifies the text (in source code format) for the signature of a given member of a type.

<!ATTLIST MemberSignature

Language CDATA #REQUIRED

(Normative) **CDATA** is the programming language in which the signature is written. All members are described in both ILAsm and C#.

Value CDATA #REQUIRED

(Normative) **CDATA** is the text of the member signature in a given language.

>

<!ELEMENT MemberType (#PCDATA)>

(Normative) Specifies the kind of the current member. The member kinds are: method, property, constructor, field, and event.

<!ELEMENT MemberValue (#PCDATA)>

(Normative) Specifies the value of a static literal field.

<!ELEMENT Members (Member*)>

(Normative) Specifies information about all of the members of a given type.

<!ELEMENT PRE EMPTY>

(Rendering/Formatting) This element exists for rendering purposes only to specify, for example, that future text should be separated from the previous text

<!ELEMENT Parameter (Attributes?)>

(Normative) Specifies the information about a specific parameter of a method or property.

<!ATTLIST Parameter

Name NMTOKEN #REQUIRED

(Normative) Specifies the name of the parameter.

Type CDATA #REQUIRED

(Normative) Specifies the fully-qualified name of the type of the parameter.

>

<!ELEMENT Parameters (Parameter*)>

©ISO/IEC 2006 – All rights reserved

(Normative) Specifies information for the parameters of a given method or property. The information specified is included in each **Parameter** element of this element. This element will contain one **Parameter** for each parameter of the method or property.

```
<!ELEMENT ReturnType (#PCDATA)>
```

(Normative) Specifies the fully-qualified name of the type that the current member returns.

```
<!ELEMENT ReturnValue (ReturnType?)>
```

(Normative) Specifies the return type of a member. **ReturnType** shall be present for all kinds of members except constructors.

```
<!ELEMENT SPAN (#PCDATA | para | paramref | SPAN | see | block)*>
```

(Rendering/Formatting) This element specifies that the text should be segmented from other text (e.g., with a carriage return). References to parameters, other types, and even blocks of text can be included within a **SPAN** element.

```
<!ELEMENT ThreadingSafetyStatement (#PCDATA)>
```

(Normative) Specifies a thread safety statement for a given type.

```
<!ELEMENT Type (TypeSignature+, MemberOfLibrary, AssemblyInfo,
ThreadingSafetyStatement?, Docs, Base, Interfaces, Attributes?,
Members, TypeExcluded)>
```

(Normative) Specifies all of the information for a given type.

```
<!ATTLIST Type
  Name CDATA #REQUIRED
```

(Informative) Specifies the simple name (e.g., “String” rather than “System.String”) of a given type.

```
  FullName CDATA #REQUIRED
```

(Normative) Specifies the fully-qualified name of a given type. For generic types, this includes the spelling of generic parameter names.

```
  FullNameSP CDATA #REQUIRED
```

(Informative) Specifies the fully-qualified name with each ‘.’ of the fully qualified name replaced by an ‘_’.

```
>
<!ELEMENT TypeExcluded (#PCDATA)>
```

(Normative) **PCDATA** shall be ‘0’.

```
<!ELEMENT TypeSignature EMPTY>
```

(Normative) Specifies the text for the signature (in code representation) of a given type.

```
<!ATTLIST TypeSignature
  Language CDATA #REQUIRED
```

(Normative) Specifies the language the specified type signature is written in. All type signatures are specified in both ILAsm and C#.

```
  Value CDATA #REQUIRED
```

(Normative) **CDATA** is the type signature in the specified language.

```
>
<!ELEMENT Types (Type+)>
```

(Normative) Specifies information about all of the types of a library.

```
<!ATTLIST Types
```

 ©ISO/IEC 2006 – All rights reserved

Library NMTOKEN #REQUIRED

(Normative) Specifies the library in which all of the types are defined. An example of such a library is “BCL”.

>

<!ELEMENT altcompliant EMPTY>

(Informative) Specifies that an alternative, CLS compliant method call exists for the current non-CLS compliant method. For example, this element exists in the System.IO.TextWriter.WriteLine(ulong) method to show that System.IO.TextWriter.WriteLine(long) is an alternative, CLS compliant method.

<!ATTLIST altcompliant

cref CDATA #REQUIRED

(Informative) Specifies the link to the actual documentation for the alternative CLS compliant method. [**Note:** In this specification, **CDATA** matches the documentation comment format specified in Appendix E of the C# language standard.]

>

<!ELEMENT altmember EMPTY>

(Informative) Specifies that an alternative, equivalent member call exists for the current method. This element is used for operator overloads.

<!ATTLIST altmember

cref CDATA #REQUIRED

(Informative) Specifies the link to the actual documentation for the alternative member call. [**Note:** In this specification, **CDATA** matches the documentation comment format specified in Appendix E of the C# language standard.]

>

<!ELEMENT block (#PCDATA | see | para | paramref | list | block | c | subscript | code | sup | pi)*>

(Rendering/Formatting) Specifies that the children should be formatted according to the **type** specified as an attribute.

<!ATTLIST block

subset CDATA #REQUIRED

(Rendering/Formatting) This attribute is reserved for future use and currently only has the value of ‘none’.

type NMTOKEN #REQUIRED

(Rendering/Formatting) Specifies the type of block that follows, one of: usage, overrides, note, example, default, behaviors.

>

<!ELEMENT c (#PCDATA | para | paramref | code | see)*>

(Rendering/Formatting) Specifies that the text is the output of a code sample.

<!ELEMENT code (#PCDATA)>

(Informative) Specifies the text is a code sample.

<!ATTLIST code

lang CDATA #IMPLIED

(Informative) Specifies the programming language of the code sample. This specification uses C# as the language for the samples.

```
>
<!ELEMENT codelink EMPTY>
```

(Informative) Specifies a piece of code to which a link might be made from another sample. **[Note:** the XML format specified here does not provide a means of creating such a link.**]**

```
<!ATTLIST codelink
  SampleID CDATA #REQUIRED
```

(Informative) SampleID is the unique id assigned to this code sample.

```
  SnippetID CDATA #REQUIRED
```

(Informative) SnippetID is the unique id assigned to a section of text within the sample code.

```
>
<!ELEMENT description (#PCDATA | SPAN | paramref | para | see | c |
permille | block | sub)*>
```

(Normative) Specifies the text for a description for a given term element in a list or table. This element also specifies the text for a column header in a table.

```
<!ELEMENT example (#PCDATA | para | code | c | codelink | see)*>
```

(Informative) Specifies that the text will be an example on the usage of a type or a member of a given type.

```
<!ELEMENT exception (#PCDATA | paramref | see | para | SPAN | block)*>
```

(Normative) Specifies text that provides the information for an exception that shall be thrown by a member of a type, unless specified otherwise. This element can contain just text or other rendering options such as blocks, etc.

```
<!ATTLIST exception
  cref CDATA #REQUIRED
```

(Rendering/Formatting) Specifies a link to the documentation of the exception. [**Note:** In this specification, **CDATA** matches the documentation comment format specified in Appendix E of the C# language standard.]

```
>
<!ELEMENT i (#PCDATA)>
```

(Rendering/Formatting) Specifies that the text should be italicized.

```
<!ELEMENT item (term, description*)>
```

(Rendering/Formatting) Specifies a specific item of a list or a table.

```
<!ELEMENT list (listheader?, item*)>
```

(Rendering/Formatting) Specifies that the text should be displayed in a list format.

```
<!ATTLIST list
  type NMTOKEN #REQUIRED
```

(Rendering/Formatting) Specifies the type of list in which the following text will be represented. Values in the specification are: bullet, number and table.

```
>
<!ELEMENT listheader (term, description+)>
```

(Rendering/Formatting) Specifies the header of all columns in a given list or table.

```
<!ELEMENT onequarter EMPTY>
```

 ©ISO/IEC 2006 – All rights reserved

(Rendering/Formatting) Specifies that text, in the form of ¼, is to be displayed.

```
<!ELEMENT para (#PCDATA | see | block | paramref | c | onequarter |
superscript | sup | permille | SPAN | list | pi | theta | sub)*>
```

(Rendering/Formatting) Specifies that the text is part of what can be considered a paragraph of its own.

```
<!ELEMENT param (#PCDATA | c | paramref | see | block | para | SPAN)*>
```

(Normative) Specifies the information on the meaning or purpose of a parameter. The name of the parameter and a textual description will be associated with this element.

```
<!ATTLIST param
  name CDATA #REQUIRED
```

(Nomrative) Specifies the name of the parameter being described.

```
>
<!ELEMENT paramref EMPTY>
```

(Rendering/Formatting) Specifies a reference to a parameter of a member of a type.

```
<!ATTLIST paramref
  name CDATA #REQUIRED
```

(Rendering/Formatting) Specifies the name of the parameter to which the **paramref** element is referring.

```
>
<!ELEMENT permille EMPTY>
```

(Rendering/Formatting) Represents the current text is to be displayed as the '‰' symbol.

```
<!ELEMENT permission (#PCDATA | see | paramref | para | block)*>
```

(Normative) Specifies the permission, given as a fully-qualified type name and supportive text, needed to call a member of a type.

```
<!ATTLIST permission
  cref CDATA #REQUIRED
```

(Rendering/Formatting) Specifies a link to the documentation of the permission. [**Note:** In this specification, **CDATA** matches the documentation comment format specified in Appendix E of the C# language standard.]

```
>
<!ELEMENT pi EMPTY>
```

(Rendering/Fomatting) Represents the current text is to be displayed as the 'π' symbol

```
<!ELEMENT pre EMPTY>
```

(Rendering/Formatting) Specifies a break between the preceding and following text.

```
<!ELEMENT remarks (#PCDATA | para | block | list | c | paramref | see |
pre | SPAN | code | PRE)*>
```

(Normative) Specifies additional information, beyond that supplied by the **summary**, on a type or member of a type.

```
<!ELEMENT returns (#PCDATA | para | list | paramref | see)*>
```

(Normative) Specifies text that describes the return value of a given type member.

```
<!ELEMENT see EMPTY>
```

(Informative) Specifies a link to another type or member.

```
<!ATTLIST see
  cref CDATA #IMPLIED
```

(Informative) **cref** specifies the fully-qualified name of the type or member to link to. [**Note:** In this specification, **CDATA** matches the documentation comment format specified in Appendix E of the C# language standard.]

```
  langword CDATA #IMPLIED
```

(Informative) **langword** specifies that the link is to a language agnostic keyword such as “null”.

```
  qualify CDATA #IMPLIED
```

(Informative) Qualify indicates that the type or member specified in the link must be displayed as fully-qualified. Value of this attribute is ‘true’ or ‘false’, with a default value of ‘false’

```
>
<!ELEMENT sub (#PCDATA | paramref)*>
```

(Rendering/Formatting) Specifies that current piece of text is to be displayed in subscript notation.

```
<!ELEMENT subscript EMPTY>
```

(Rendering/Formatting) Specifies that current piece of text is to be displayed in subscript notation.

```
<!ATTLIST subscript
  term CDATA #REQUIRED
```

(Rendering/Formatting) Specifies the value to be rendered as a subscript.

```
>
<!ELEMENT summary (#PCDATA | para | see | block | list)*>
```

(Normative) Specifies a summary description of a given type or member of a type.

```
<!ELEMENT sup (#PCDATA | i | paramref)*>
```

(Rendering/Formatting) Specifies that the current piece of text is to be displayed in superscript notation.

```
<!ELEMENT superscript EMPTY>
```

(Rendering/Formatting) Specifies that current piece of text is to be displayed in superscript notation.

```
<!ATTLIST superscript
  term CDATA #REQUIRED
```

(Rendering/Formatting) Specifies the value to be rendered as a superscript.

```
>
<!ELEMENT term (#PCDATA | block | see | paramref | para | c | sup | pi
| theta)*>
```

(Rendering/Formatting) Specifies the text is a list item or an item in the primary column of a table.

```
<!ELEMENT theta EMPTY>
```

(Rendering/Formatting) Specifies that text, in the form of ‘θ’, is to be displayed.

```
<!ELEMENT threadsafe (para+)>
```

(Normative) Specifies that the text describes additional detail, beyond that specified by **ThreadingSafetyStatement**, the thread safety implications of the current type. For example, the text will describe what an implementation must do in terms of synchronization.

```
<!ELEMENT value (#PCDATA | para | list | see)*>
```

 ©ISO/IEC 2006 – All rights reserved

(Normative) Specifies description information on the “value” passed into the set method of a property.

7.1.1 Value types as objects

Throughout the textual descriptions of methods in the XML, there are places where a parameter of type `object` or an interface type is expected, but the description refers to passing a value type for that parameter. In these cases, the caller shall box the value type before making the call.

7.1.2 Exceptions

Many members of types defined in the XML have associated exception conditions. Unless it is stated otherwise in a member’s definition, the exceptions listed for any given member shall be thrown when the stated conditions occur.

7.2 XML signature notation issues

For each type and member described in the XML, there is an ILAsm and C# signature pair. These are intended to be equivalent and to provide sufficient information to allow these types and members to be implemented correctly. Each signature pair shows both the low-level and one high-level view of these signatures. However, as written in the XML, the members of a given pair of signatures are not always written in an equivalent manner, even though they are intended to produce identical behavior. The differences in signature notation are described in this subclause.

7.2.1 Serialization

As shown in the ILAsm signatures, many of the types in the standard library have the predefined attribute `serializable` attached. A type that is marked with this attribute is to be serialized as part of the persistent state of a value of the type. This standard does not require that a conforming implementation provide support for serialization (or its counterpart, deserialization), nor does it specify the mechanism by which these operations might be accomplished.

Consider the ILAsm and C# signatures in the XML for `System.String`:

```
[ILAsm]
.class public sealed serializable String …

[C#]
public sealed class String …
```

Although the C# standard does not address the issue of serialization, if this library type is written in C#, when the C# declaration above is compiled, the intent is that the code generated for the class contains the `serializable` attribute as shown. [*Note*: Some implementations provide an attribute type, `System.SerializableAttribute`, for this purpose. *end note*]

7.2.2 Delegates

The standard library contains a number of delegate types. However, as recorded in the XML, their ILAsm signatures are incomplete. Consider `System.EventHandler` as an example; its ILAsm signature is defined in the XML as follows:

```
.class public sealed serializable EventHandler extends System.Delegate {
.method public hidebysig newslot virtual instance void Invoke(object
   sender, class System.EventArgs e) }
```

However, this type also has a constructor and two optional asynchronous methods, `BeginInvoke` and `EndInvoke`, all of which are described in Partition II, “Delegates”. The signatures for these three members for `System.EventHandler` are as follows:

```
.method public hidebysig specialname rtspecialname void .ctor(object
   'object', native int 'method')
.method public hidebysig newslot virtual class System.IAsyncResult
```

```
    BeginInvoke(object sender, class System.EventArgs e, class
    System.AsyncCallback callback, object 'object')
.method public hidebysig newslot virtual void EndInvoke(
    class System.IAsyncResult result)
```

The other standard delegate types have a corresponding constructor and method pair whose signatures can be deduced from the ILAsm in the XML and the information in Partition II, "Delegates".

Unless stated otherwise, a standard delegate type provides the two optional asynchronous methods, `BeginInvoke` and `EndInvoke`.

7.2.3 Properties

The standard library contains many types that have properties. However, as recorded in the XML, their ILAsm signatures are incomplete. Consider the read-write instance property `System.Collections.ArrayList.Capacity`. Its ILAsm signature is defined in the XML as follows:

```
.property int32 Capacity {
  public hidebysig virtual specialname int32 get_Capacity()
  public hidebysig virtual specialname void set_Capacity(int32 value)
}
```

However, this is an abbreviation of the ILAsm syntax. The complete (and correct) signature for this property is as follows:

```
.property instance int32 Capacity() {
  .get instance int32 ArrayList::get_Capacity()
  .set instance void ArrayList::set_Capacity(int32)
}
.method public hidebysig newslot specialname virtual instance int32
   get_Capacity() { … }
.method public hidebysig newslot specialname virtual instance void
   set_Capacity(int32 'value') { … }
```

As a second example, consider the readonly static property `System.DateTime.Now`; its ILAsm signature is defined in the XML as follows:

```
.property valuetype System.DateTime Now {
  public hidebysig static specialname valuetype System.DateTime
    get_Now()
}
```

However, the complete (and correct) signature for this property is:

```
.property valuetype System.DateTime Now() {
  .get valuetype System.DateTime DateTime::get_Now()
}
.method public hidebysig specialname static valuetype System.DateTime
   get_Now() { ... }
```

All other properties (including those that are indexed) are formatted in the XML in a similar abbreviated manner.

7.2.4 Nested types

With one exception, the definitions of all members of any given type are contained in the XML for that type. The exception is for nested types. Nested types have their own definition in the XML, where their names are qualified by the name of the type in which they are nested. [*Example*: The type `System.Collections.Generic.List<T>` contains the nested type `Enumerator`. These types are described in the BCL library of the XML under the names `List<T>` and `List<T>.Enumerator`, respectively. *end example*]

 ©ISO/IEC 2006 – All rights reserved

Common Language Infrastructure (CLI)
Partition V:
Debug Interchange Format

Table of contents

1	**Portable CILDB files**	**1**
1.1	Encoding of integers	1
1.2	CILDB header	1
1.2.1	Version GUID	1
1.3	Tables and heaps	1
1.3.1	SymConstant table	2
1.3.2	SymDocument table	2
1.3.3	SymMethod table	2
1.3.4	SymSequencePoint table	3
1.3.5	SymScope table	3
1.3.6	SymVariable table	4
1.3.7	SymUsing table	4
1.3.8	SymMisc heap	4
1.3.9	SymString heap	4
1.4	Signatures	4

©ISO/IEC 2006 – All rights reserved

1 Portable CILDB files

Portable CILDB files provide a standard way to interchange debugging information between CLI producers and consumers. This partition serves to fill in gaps not covered by metadata, notably the names of local variables and source line correspondences.

1.1 Encoding of integers

All integers are stored in little-endian format, except for those in signatures, which are encoded as described in Partition II.

1.2 CILDB header

A CILDB file starts with a 72-byte header, whose layout is as follows:

Offset	Size	Field	Description
0	16	**Signature**	Magic signature for CILDB "_ildb_signature\0"
16	16	**GUID**	Version GUID
32	4	**UserEntryPoint**	MethodDef token of the entry point.
36	4	**CountOfMethods**	Number of rows in the SymMethod table.
40	4	**CountOfScopes**	Number of rows in the SymScopes table.
44	4	**CountOfVars**	Number of rows in the SymVariable table.
48	4	**CountOfUsing**	Number of rows in the SymUsing table.
52	4	**CountOfConstants**	Number of rows in the SymConstant table.
56	4	**CountOfDocuments**	Number of rows in the SymDocument table.
60	4	**CountOfSequencePoints**	Number of rows in the SymSequencePoint table.
64	4	**CountOfMiscBytes**	Number of bytes in the SymMisc heap.
68	4	**CountOfStringBytes**	Number of bytes in the SymString heap.

1.2.1 Version GUID

The version GUID is the 16-byte sequence shown below:

0x7F	0x55	0xE7	0xF1	0x3C	0x42	0x17	0x41
0x8D	0xA9	0xC7	0xA3	0xCD	0x98	0x8D	0xF1

1.3 Tables and heaps

The CILDB header is immediately followed by various tables and heaps, in the following order:

1. SymConstant
2. SymMethod
3. SymScopes
4. SymVariable
5. SymUsing
6. SymSequencePoint
7. SymDocument

©ISO/IEC 2006 – All rights reserved

8. SymMisc
9. SymString

Some of the tables contain CIL offsets. These offsets are in bytes, and the offset of the first instruction is zero. The offsets do not necessarily match the beginning of a CIL instruction. For example, offsets denoting the end of a range of bytes often refer to the last byte of an instruction. Lengths are also in bytes.

The rows in each of the tables 3–7 above that belong to the same method must be contiguous within their parent table.

1.3.1 SymConstant table

Each row of the SymConstant table describes a constant, as follows:

Offset	Size	Field	Description
0	4	**Scope**	Index of parent scope
4	4	**Name**	Index of the name in the SymString heap
8	4	**Signature**	Index of the signature in the SymMisc heap
12	4	**SignatureSize**	Length of the signature
16	4	**Value**	Index of the value in the SymMisc heap
20	4	**ValueSize**	Length of the value.

The value of the constant is encoded just like a Blob for the *Value* column of a Constant metadata table in Partition II, except that there is no length prefix.

1.3.2 SymDocument table

Each row of a SymDocument describes a source document, as shown below. The document can either be referred to indirectly (by its URL) or incorporated directly into the CILDB file as part of the SymMisc heap. The GUID values referred to in this subclause are not defined by this Standard; space is simply reserved for them.

Offset	Size	Field	Description
0	16	**Language**	GUID for the language.
16	16	**LanguageVendor**	GUID for the language vendor.
32	16	**DocumentType**	GUID for the document type.
48	16	**AlgorithmId**	GUID of the checksum algorithm; or 0 if there is no checksum.
64	4	**CheckSumSize**	Size of the checksum; or 0 if there is no checksum.
68	4	**CheckSumEntry**	Index of the checksum in the SymMisc heap; or 0 if there is no checksum.
72	4	**SourceSize**	Size of the source in the SymMisc heap; or 0 if the source document is not directly incorporated into the file.
76	4	**SourceEntry**	Index of the source in the SymMisc heap; or 0 if the source document is not directly incorporated into the file.
80	4	**UrlEntry**	Index of the document URL in the SymString heap.

1.3.3 SymMethod table

Each row of a SymMethod table has the following format:

Offset	Size	Field	Description

 ©ISO/IEC 2006 – All rights reserved

0	4	**MethodToken**	A MethodDef metadata token.
4	8	**Scopes**	[Start,Stop) range of SymScope table.
12	8	**Vars**	[Start,Stop) range of SymVariable table.
20	8	**Using**	[Start,Stop) range of SymUsing table.
28	8	**Constant**	[Start,Stop) range of SymConstant table.
36	8	**Documents**	[Start,Stop) range of SymDocument table.
44	8	**SequencePoints**	[Start,Stop) range of SymSequencePoint table.

Each [Start,Stop) range is represented as two 4-byte integers. The first integer is the index of the first related table row; the second integer is the index of one past the last related table row.

The rows of a SymMethod table are sorted in ascending order of the **MethodToken** field. There is at most one row for each method.

1.3.4 SymSequencePoint table

Each row of a SymSequencePoint table describes a sequence point, as follows:

Offset	Size	Field	Description
0	4	**Offset**	CIL offset of the sequence point.
4	4	**StartLine**	Starting line of the source document.
8	4	**StartColumn**	Starting column, or 0 if not specified.
12	4	**EndLine**	Ending line of the source document, or 0 if not specified.
16	4	**EndColumn**	Ending column, or 0 if not specified.
20	4	**Doc**	Index of the source document in the SymString heap.

Together, EndLine and EndColumn specify the column "one past" the last byte position associated with the sequence point. In other words, they specify the end of a half-open interval [start,end).

Rows of the SymSequencePoint belonging to the same Method must be contiguous and sorted in ascending order of Offset.

1.3.5 SymScope table

Each row of a SymScope table describes a scope, as follows:

Offset	Size	Field	Description
0	4	**Parent**	Index of parent SymScope row, or –1 if scope has no parent.
4	4	**StartOffset**	CIL offset of the first byte in the scope.
8	4	**EndOffset**	CIL offset of the last byte in the scope.
12	4	**HasChildren**	1 if scope has child scopes; 0 otherwise
16	4	**HasVars**	1 if scope has variables; 0 otherwise

The scopes belonging to a method must form a tree, with the following constraints:

- A parent scope must precede its child scopes.
- The StartOffset and EndOffset of a child scope must be within the (inclusive) range of offsets specified by its parent's scope.

©ISO/IEC 2006 – All rights reserved

1.3.6 SymVariable table

Each row of a SymVariable table describes a local variable.

Offset	Size	Field	Description
0	4	**Scope**	Index of the parent scope
4	4	**Name**	Index of the variable's name in the SymString heap.
8	4	**Attributes**	Flags describing the variable (see below).
12	4	**Signature**	Index of the signature in the SymMisc heap.
16	4	**SignatureSize**	Length of the signature.
20	4	**AddressKind**	Always 1.
24	4	**Address1**	Local variable number.
28	4	**Address2**	Always 0.
32	4	**Address3**	Always 0.
36	4	**StartOffset**	CIL offset where the variable is first visible .
40	4	**EndOffset**	CIL offset where the variable is last visible.
44	4	**Sequence**	Always 0.
48	4	**IsParam**	Always 0.
52	4	**IsHidden**	1 if variable should be hidden from debugger; 0 otherwise.

The least-significant bit of Attributes indicates whether the variable is user-generated (0) or compiler-generated (1). The other bits are reserved and should be set to zero.

Because parameters are fully described by the Metadata, they do not appear in the SymVariable table.

1.3.7 SymUsing table

Each row of the SymUsing table describes importation of a namespace, as follows:

Offset	Size	Field	Description
0	4	**Scope**	Index of the parent scope
4	4	**Namespace**	Index of the namespace in the SymString heap

1.3.8 SymMisc heap

The SymMisc heap holds various byte sequences (e.g., signatures and checksums).

1.3.9 SymString heap

The stream of bytes in the SymString heap has the same form as those for the #Strings heap (see Partition II).

1.4 Signatures

Signatures of variables and constants are encoded as an index into the SymMisc heap, and a signature size. The values of the bytes are similar to those for a FieldSig (see Partition II), and include the prefix FIELD (0x6), even though the variables are not fields. Because the length of the signature is encoded in the tables, it is not included in the SymMisc heap. For example, type int32 is encoded as "0x06 0x08".

 ©ISO/IEC 2006 – All rights reserved

Common Language Infrastructure (CLI)

Partition VI:
Annexes

Annex A Introduction **1**

Annex B Sample programs **2**
B.1 Mutually recursive program (with tail calls) 2
B.2 Using value types 3
B.3 Custom attributes 6
B.4 Generics code and metadata 9
B.4.1 ILAsm version 9
B.4.2 C# version 10
B.4.3 Metadata 11

Annex C CIL assembler implementation **12**
C.1 ILAsm keywords 12
C.2 CIL opcode descriptions 15
C.3 Complete grammar 26
C.4 Instruction syntax 41
C.4.1 Top-level instruction syntax 42
C.4.2 Instructions with no operand 42
C.4.3 Instructions that refer to parameters or local variables 44
C.4.4 Instructions that take a single 32-bit integer argument 45
C.4.5 Instructions that take a single 64-bit integer argument 45
C.4.6 Instructions that take a single floating-point argument 45
C.4.7 Branch instructions 45
C.4.8 Instructions that take a method as an argument 46
C.4.9 Instructions that take a field of a class as an argument 46
C.4.10 Instructions that take a type as an argument 47
C.4.11 Instructions that take a string as an argument 47
C.4.12 Instructions that take a signature as an argument 47
C.4.13 Instructions that take a metadata token as an argument 47
C.4.14 Switch instruction 48

Annex D Class library design guidelines **49**

Annex E Portability considerations **50**
E.1 Uncontrollable behavior 50
E.2 Language- and compiler-controllable behavior 50
E.3 Programmer-controllable behavior 50

Annex F Imprecise faults **52**
F.1 Instruction reordering 52

©ISO/IEC 2006 – All rights reserved

F.2 Inlining 52
F.3 Finally handlers still guaranteed once a try block is entered 52
F.4 Interleaved calls 53
F.4.1 Rejected notions for fencing 53
F.5 Examples 54
F.5.1 Hoisting checks out of a loop 54
F.5.2 Vectorizing a loop 54
F.5.3 Autothreading a loop 55

Annex G Parallel library 56
G.1 Considerations 56
G.2 ParallelFor 56
G.3 ParallelForEach 57
G.4 ParallelWhile 57
G.5 Debugging 57

 ©ISO/IEC 2006 – All rights reserved

Annex A Introduction

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 can 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 F contains information of interest to implementers with respect to relaxed fault handling.

Annex G shows several complete examples written using the parallel library.

©ISO/IEC 2006 – All rights reserved

Annex B Sample programs

This clause 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.

[*Example:*

```
.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)
```

 ©ISO/IEC 2006 – All rights reserved

```
    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'
```

end example]

B.2 Using value types

The following program shows how rational numbers can be implemented using value types.

[*Example:*

```
.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
```

 ©ISO/IEC 2006 – All rights reserved

```
  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
}
```

end example]

B.3 Custom attributes

[*Example:*

This subclause includes many example uses of custom attributes to help clarify the grammar and rules described above. The examples are written in C#, and each one shows a collection of one or more attributes, applied to a class (called "App"). The hex and 'translation' of the custom attribute blobs are shown as comments. The following abbreviations are used:

- FIELD = ELEMENT_TYPE_FIELD
- PROPERTY = 0x54
- STRING = ELEMENT_TYPE_STRING
- SZARRAY = ELEMENT_TYPE_SZARRAY
- U1 = ELEMENT_TYPE_U1
- I4 = ELEMENT_TYPE_I4
- OBJECT = 0x51

```
//
*****************************************************************
***************
// CustomSimple.cs
using System;
[AttributeUsage(AttributeTargets.Class, AllowMultiple=true)]
class B : Attribute { public B(int i, ushort u) {} }

[B(7,9)]     // 01 00          // Prolog
             // 07 00 00 00    // 0x00000007
             // 09 00          // 0x0009
             // 00 00          // NumNamed
class App { static void Main() {} }

//
*****************************************************************
***************
// CustomString.cs
using System;
[AttributeUsage(AttributeTargets.Class, AllowMultiple=true)]
class A : Attribute {
   public  string field;        // field
   private string back;         // backing field for property
   public  string prop {        // property
      get { return back;  }
      set { back = value; }
   }
   public  A(string x) {}       // ctor
}
[A(null)]    // 01 00            // Prolog
             // FF               // null
             // 00 00            // NumNamed

[A("")]      // 01 00            // Prolog
             // 00               // zero-length string
             // 00 00            // NumNamed

[A("ab",field="cd",prop="123")]  // 01 00             // Prolog
                                 // 02 61 62          // "ab"
                                 // 02 00             // NumNamed
                                 // 53 0e             // FIELD,
STRING
                                 // 05 66 69 65 6c 64 // "field"
as counted-UTF8
```

 ©ISO/IEC 2006 – All rights reserved

```
                                    // 02 63 64          // "cd" as
counted-UTF8
                                    // 54 0e             //
PROPERTY, STRING
                                    // 04 70 72 6f 70    // "prop"
as counted-UTF8
                                    // 03 31 32 33       // "123" as
counted-UTF8
class App { static void Main() {} }

//
*****************************************************************
***************
// CustomType.cs
using System;
[AttributeUsage(AttributeTargets.Class, AllowMultiple=true)]
class C : Attribute {
   public C(Type t) {}
}
[C(typeof(C))]
// 01 00                                                  // Prolog
// 01 43                                                  // "C" as
counted-UTF8
// 00 00                                                  // NumNamed

[C(typeof(string))]
// 01 00                                                  // Prolog
// 0d 53 79 73 74 65 6d 2e 53 74 72 69 6e 67              //
"System.String" as counted-UTF8
// 00 00                                                  // NumNamed

[C(typeof(System.Windows.Forms.Button))]
// 01 00                                                  // Prolog
// 76 53 79 73 74 65 6d 2e 57 69 6e 64 6f 77              //
"System.Window
// 73 2e 46 6f 72 6d 73 2e 42 75 74 74 6f 6e 2c 53        //
s.Forms.Button,S
// 79 73 74 65 6d 2e 57 69 6e 64 6f 77 73 2e 46 6f        //
ystem.Windows.Fo
// 72 6d 73 2c 20 56 65 72 73 69 6f 6e 3d 32 2e 30        // rms,
Version=2.0
// 2e 33 36 30 30 2e 30 2c 20 43 75 6c 74 75 72 65        // .3600.0,
Culture
// 3d 6e 65 75 74 72 61 6c 2c 20 50 75 62 6c 69 63        //
=neutral, Public
// 4b 65 79 54 6f 6b 65 6e 3d 62 37 37 61 35 63 35        //
KeyToken=b77a5c5
// 36 31 39 33 34 65 30 38 39 00 00                       //
61934e089"
// 00 00                                                  // NumNamed
class App { static void Main() {} }
```

Notice how various types are ‘stringified’: if the type is defined in the local assembly, or in mscorlib, then only its full name is required; if the type is defined in a different assembly, then its fully-qualified assembly name is required, includeing Version, Culture and PublicKeyToken, if non-defaulted.

```
//
*****************************************************************
***************
// CustomByteArray.cs
```

```
using System;
class D : Attribute {
   public  byte[] field;                                  // field
   private byte[] back;                                   // backing
field for property
   public  byte[] prop {                                  // property
      get { return back;  }
      set { back = value; }
   }
   public D(params byte[] bs) {}                          // ctor
}
[D(1,2, field=new byte[]{3,4},prop=new byte[]{5})]
// 01 00                                                  // Prolog
// 02 00 00 00                                            // NumElem
// 01 02                                                  // 1,2
// 02 00                                                  // NumNamed
// 53 1d 05                                               // FIELD,
SZARRAY, U1
// 05 66 69 65 6c 64                                      // "field"
as counted-UTF8
// 02 00 00 00                                            // NumElem =
0x00000002
// 03 04                                                  // 3,4
// 54 1d 05                                               // PROPERTY,
SZARRAY, U1
// 04 70 72 6f 70                                         // "prop" as
counted-UTF8
// 01 00 00 00                                            // NumElem =
0x00000001
// 05                                                     // 5
class App { static void Main() {} }

//
*****************************************************************
***************
// CustomBoxedValuetype.cs
using System;
[AttributeUsage(AttributeTargets.Class, AllowMultiple=true)]
class E : Attribute {
   public object obj;                              // field called
"obj"
   public object o {                               // property
called "o"
      get { return o; }
      set { o = value; }
   }
   public E() {}                                   // default ctor
   public E(object x) {}
}

[E(42)]                                            // boxed 42
// 01 00                                           // Prolog
// 08                                              // I4
// 2a 00 00 00                                     // 0x0000002A
// 00 00                                           // NumNamed

[E(obj=7)]                                         // named field
// 01 00                                           // Prolog
// 01 00                                           // NumNamed
// 53 51                                           // FIELD, OBJECT
```

 ©ISO/IEC 2006 – All rights reserved

```
// 03 6f 62 6a                                  // "obj" as
counted-UTF8
// 08                                           // I4
// 07 00 00 00                                  // 0x00000007

[E(o=0xEE)]                                     // named property
// 01 00                                        // Prolog
// 01 00                                        // NumNamed
// 54 51                                        // PROPERTY,
OBJECT
// 01 6f                                        // "o" as
counted-UTF8
// 08                                           // I4
// ee 00 00 00                                  // 0x000000EE
class App { static void Main() {} }
```

This example illustrates how to construct blobs for a custom attribute that accepts a `System.Object` in its constructor, as a named field, and as a named property. In each case, the argument given is an `int32`, which is boxed automatically by the C# compiler.

Notice the OBJECT = 0x51 byte. This is emitted for "named" fields or properties of type `System.Object`.

```
//
*****************************************************************
***************
// CustomShortArray.cs
using System;
[AttributeUsage(AttributeTargets.Class, AllowMultiple=true)]
class F : Attribute {
   public F(params short[] cs) {}    // ctor
}
//[F()]
// 01 00                                // Prolog
// 00 00 00 00                          // NumElem
// 00 00                                // NumNamed

//[F(null)]
// 01 00                                // Prolog
// ff ff ff ff                          // NumElem = -1 => null
// 00 00                                // NumNamed

[F(1,2)]
// 01 00                                // Prolog
// 02 00 00 00                          // NumElem
// 01 00 02 00                          // 0x0001, 0x0002
// 00 00                                // NumNamed
class App { static void Main() {} }
```

end example]

B.4 Generics code and metadata

The following informative text, shows a partial implementation for a naive phone-book class. It shows the source first, as written in ILAsm, followed by the equivalent (much shorter) code, written in C#. The section then goes on to examine the metadata generated for this code.

B.4.1 ILAsm version

```
.assembly extern mscorlib {}
.assembly Phone {}
```

```
.class private Phone`2<([mscorlib]System.Object) K,
([mscorlib]System.Object) V>
    extends  [mscorlib]System.Object {
   .field private int32 hi
   .field private !0[]  keys
   .field private !1[]  vals
   .method public instance void Add(!0 k, !1 v) {
      .maxstack  4
      .locals init (int32 temp)
      ldarg.0
      ldfld      !0[] class Phone`2<!0,!1>::keys
      ldarg.0
      dup
      ldfld      int32 class Phone`2<!0,!1>::hi
      ldc.i4.1
      add
      dup
      stloc.0
      stfld      int32 class Phone`2<!0,!1>::hi
      ldloc.0
      ldarg.1
      stelem     !0
      ldarg.0
      ldfld      !1[] class Phone`2<!0,!1>::vals
      ldarg.0
      ldfld      int32 class Phone`2<!0,!1>::hi
      ldarg.2
      stelem     !1
      ret
   }  // end of Method Add
}  // end of class Phone

.class App extends [mscorlib]System.Object {
   .method static void Main() {
      .entrypoint
      .maxstack  3
      .locals init (class Phone`2<string,int32> temp)
      newobj     instance void class
        Phone`2<string,int32>::.ctor()
      stloc.0
      ldloc.0
      ldstr      "Jim"
      ldc.i4.7
      callvirt   instance void class
        Phone`2<string,int32>::Add(!0, !1)
      ret
   }  // end of method Main
}  // end of class App
```

B.4.2 C# version

```
using System;

class Phone<K,V> {
   private int hi = -1;
   private K[] keys;
   private V[] vals;
   public Phone() { keys = new K[10]; vals = new V[10]; }
   public void Add(K k, V v) { keys[++hi] = k; vals[hi] = v; }
}

class App {
```

 ©ISO/IEC 2006 – All rights reserved

```
    static void AddOne<KK,VV>(Phone<KK,VV> phone, KK kk, VV vv) {
       phone.Add(kk, vv);
    }
    static void Main() {
       Phone<string, int> d = new Phone<string, int>();
       d.Add("Jim", 7);
       AddOne(d, "Joe", 8);
    }
}
```

B.4.3 Metadata

As detailed in §23.2.12 of Partition II, the *Type* non-terminal now includes a production for generic instantiations, as follows:

```
Type ::= . . .
     | GENERICINST (CLASS | VALUETYPE) TypeDefOrRefEncoded GenArgCount Type *
```

Following this production, the `Phone<string,int>` instantiation above is encoded as:

```
0x15  ELEMENT_TYPE_GENERICINST
0x12  ELEMENT_TYPE_CLASS
0x08  TypeDefOrRef coded index for class "Phone<K,V>"
0x02  GenArgCount = 2
0x0E     ELEMENT_TYPE_STRING
0x08     ELEMENT_TYPE_I4
```

Similarly, the signature for the field `vals` is encoded as:

```
0x06  FIELD
0x1D  ELEMENT_TYPE_SZARRAY
0x13  ELEMENT_TYPE_VAR
0x01  1, representing generic argument number 1 (i.e., "V")
```

Similarly, the signature for the (rather contrived) static method `AddOne` is encoded as:

```
0x10  IMAGE_CEE_CS_CALLCONV_GENERIC
0x02  GenParamCount = 2 (2 generic parameters for this method:
KK and VV
0x03  ParamCount = 3 (phone, kk and vv)
0x01  RetType = ELEMENT_TYPE_VOID
0x15  Param-0:  ELEMENT_TYPE_GENERICINST
0x12            ELEMENT_TYPE_CLASS
0x08            TypeDefOrRef coded index for class
"Phone<KK,VV>"
0x02            GenArgCount = 2
0x1e               ELEMENT_TYPE_MVAR
0x00               !!0 (KK in AddOne<KK,VV>)
0x1e               ELEMENT_TYPE_MVAR
0x01               !!1 (VV in AddOne<KK,VV>)
0x1e  Param-1   ELEMENT_TYPE_MVAR
0x00            !!0 (KK in AddOne<KK,VV>)
0x1e  Param-2   ELEMENT_TYPE_MVAR
0x01            !!1 (VV in AddOne<KK,VV>)
```

Notice that the above example uses indenting to help denote loops over the three method parameters, and the two generic parameters on `Phone`.

©ISO/IEC 2006 – All rights reserved

Annex C CIL assembler implementation

This clause contains only informative text

This clause 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, §C.1 and §C.3 might be 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 subclause 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	.locals	.vtentry	bge
.addon	.manifestres	.vtfixup	bge.s
.assembly	.maxstack	.zeroinit	bge.un
.cctor	.method	^THE_END^	bge.un.s
.class	.module	abstract	bgt
.corflags	.mresource	add	bgt.s
.ctor	.namespace	add.ovf	bgt.un
.custom	.other	add.ovf.un	bgt.un.s
.data	.override	algorithm	ble
.emitbyte	.pack	alignment	ble.s
.entrypoint	.param	and	ble.un
.event	.pdirect	ansi	ble.un.s
.export	.permission	any	blob
.field	.permissionset	arglist	blob_object
.file	.property	array	blt
.fire	.publickey	as	blt.s
.get	.publickeytoken	assembly	blt.un
.hash	.removeon	assert	blt.un.s
.imagebase	.set	at	bne.un
.import	.size	auto	bne.un.s
.language	.subsystem	autochar	bool
.line	.try	beforefieldinit	box
.locale	.ver	beq	br
.localized	.vtable	beq.s	br.s

 ©ISO/IEC 2006 – All rights reserved

break
brfalse
brfalse.s
brinst
brinst.s
brnull
brnull.s
brtrue
brtrue.s
brzero
brzero.s
bstr
bytearray
byvalstr
call
calli
callmostderived
callvirt
carray
castclass
catch
cdecl
ceq
cf
cgt
cgt.un
char
cil
ckfinite
class
clsid
clt
clt.un
const
constrained.
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
cpobj
currency
custom
date
decimal
default
default

demand
deny
div
div.un
dup
endfault
endfilter
endfinally
endmac
enum
error
explicit
extends
extern
false
famandassem
family
famorassem
fastcall
fastcall
fault
field
filetime
filter
final
finally
fixed
float
float32
float64
forwardref
fromunmanaged
handler
hidebysig
hresult
idispatch
il
illegal

implements
implicitcom
implicitres
import
in
inheritcheck
init
initblk
initobj
initonly
instance
int
int16
int32
int64
int8
interface
internalcall
isinst
iunknown
jmp
lasterr
lcid
ldarg
ldarg.0
ldarg.1
ldarg.2
ldarg.3
ldarg.s
ldarga
ldarga.s
ldc.i4
ldc.i4.0
ldc.i4.1
ldc.i4.2
ldc.i4.3
ldc.i4.4
ldc.i4.5

ldc.i4.6	ldlen	newarr	prejitdeny
ldc.i4.7	ldloc	newobj	prejitgrant
ldc.i4.8	ldloc.0	newslot	preservesig
ldc.i4.M1	ldloc.1	noappdomain	private
ldc.i4.m1	ldloc.2	no.	privatescope
ldc.i4.s	ldloc.3	noinlining	protected
ldc.i8	ldloc.s	nomachine	public
ldc.r4	ldloca	nomangle	readonly.
ldc.r8	ldloca.s	nometadata	record
ldelem	ldnull	noncasdemand	refany
ldelem.i	ldobj	noncasinheritance	refanytype
ldelem.i1	ldsfld	noncaslinkdemand	refanyval
ldelem.i2	ldsflda	nop	rem
ldelem.i4	ldstr	noprocess	rem.un
ldelem.i8	ldtoken	not	reqmin
ldelem.r4	ldvirtftn	not_in_gc_heap	reqopt
ldelem.r8	leave	notremotable	reqrefuse
ldelem.ref	leave.s	notserialized	reqsecobj
ldelem.u1	linkcheck	null	request
ldelem.u2	literal	nullref	ret
ldelem.u4	localloc	object	rethrow
ldelem.u8	lpstr	objectref	retval
ldelema	lpstruct	opt	rtspecialname
ldfld	lptstr	optil	runtime
ldflda	lpvoid	or	safearray
ldftn	lpwstr	out	sealed
ldind.i	managed	permitonly	sequential
ldind.i1	marshal	pinned	serializable
ldind.i2	method	pinvokeimpl	shl
ldind.i4	mkrefany	pop	shr
ldind.i8	modopt	prefix1	shr.un
ldind.r4	modreq	prefix2	sizeof
ldind.r8	mul	prefix3	special
ldind.ref	mul.ovf	prefix4	specialname
ldind.u1	mul.ovf.un	prefix5	starg
ldind.u2	native	prefix6	starg.s
ldind.u4	neg	prefix7	static
ldind.u8	nested	prefixref	stdcall

 ©ISO/IEC 2006 – All rights reserved

stdcall	stind.ref	switch	unmanagedexp
stelem	stloc	synchronized	unsigned
stelem.i	stloc.0	syschar	unused
stelem.i1	stloc.1	sysstring	userdefined
stelem.i2	stloc.2	tail.	value
stelem.i4	stloc.3	tbstr	valuetype
stelem.i8	stloc.s	thiscall	vararg
stelem.r4	stobj	thiscall	variant
stelem.r8	storage	throw	vector
stelem.ref	stored_object	tls	virtual
stfld	stream	to	void
stind.i	streamed_object	true	volatile.
stind.i1	string	typedref	wchar
stind.i2	struct	unaligned.	winapi
stind.i4	stsfld	unbox	with
stind.i8	sub	unbox.any	wrapper
stind.r4	sub.ovf	unicode	xor
stind.r8	sub.ovf.un	unmanaged	

C.2 CIL opcode descriptions

This subclause 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 can 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 can vary by instruction), next is an object reference

m. PopRef+PopI+PopI – Top of stack has two integers (size can 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

 ©ISO/IEC 2006 – All rights reserved

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 2-byte encoding, or 0xFF if single byte instruction.

9. One byte encoding, or second byte of 2-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)
```

 ©ISO/IEC 2006 – All rights reserved

```
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)
```

 ©ISO/IEC 2006 – All rights reserved

```
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)
```

 ©ISO/IEC 2006 – All rights reserved

```
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)
```

 ©ISO/IEC 2006 – All rights reserved

```
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,
0x34FFFFFFFFF)
    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
```

 ©ISO/IEC 2006 – All rights reserved

```
                        | '.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 '=' '('
                        ;
```

 ©ISO/IEC 2006 – All rights reserved

```
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 ')'
```

 ©ISO/IEC 2006 – All rights reserved

```
                        | 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
```

 ©ISO/IEC 2006 – All rights reserved

```
                        ;


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
```

 ©ISO/IEC 2006 – All rights reserved

```
                        | 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'
```

 ©ISO/IEC 2006 – All rights reserved

```
                        | '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 '=' '('
```

 ©ISO/IEC 2006 – All rights reserved

```
                        ;

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'
```

 ©ISO/IEC 2006 – All rights reserved

```
                        | 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 subclause 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 §C.1 with the information in the following table:

Table 1: Instruction Syntax classes

Grammar Class	Format(s) Specified in §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.

 ©ISO/IEC 2006 – All rights reserved

```
<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
```

 ©ISO/IEC 2006 – All rights reserved

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
```

 ©ISO/IEC 2006 – All rights reserved

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)
```

 ©ISO/IEC 2006 – All rights reserved

Annex D Class library design guidelines

This clause contains only informative text

Information on this topic can be found at the following location:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconnetframeworkdesignguidelines.asp

Annex E Portability considerations

This clause 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 clause 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 can 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 might 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 might not be sufficient to find all

 ©ISO/IEC 2006 – All rights reserved

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.

Annex F Imprecise faults

This clause contains only informative text

Some CIL instructions perform implicit run-time checks that ensure memory and type safety. Originally, the CLI guaranteed that exceptions were precise, meaning that program state was preserved when an exception was thrown. However, enforcing precise exceptions for implicit checks makes some important optimizations practically impossible to apply. Programmers can now declare, via a custom attribute, that a method is "relaxed", which says that exceptions arising from implicit run-time checks need not be precise.

Relaxed checks preserve verifiability (by preserving memory and type safety) while permitting optimizations that reorder instructions. In particular, it enables the following optimizations:

- Hoisting implicit run-time checks out of loops.
- Reordering loop iterations (e.g., vectorization and automatic multithreading)
- Interchanging loops
- Inlining that makes an inlined method as least as fast as the equivalent macro

F.1 Instruction reordering

Programs that always perform explicit checks and explicit throws, instead of relying on implicit run-time checks, are never visibly affected by relaxation, except for variations already permitted by the existing CLI standard (e.g., non-determinism of cross-thread non-volatile reads and writes). Furthermore, only control dependences induced by implicit run-time checks are relaxed. Nonetheless, *data* dependences must be respected.

Authors of strict methods can reason about their behavior without knowing details about whether their callers or callees are relaxed, because strict instructions act as a fence. On the other hand, we want calls from E-relaxed methods to E-relaxed methods to be inlinable "as if" they were inlined by hand at the source level. That is why an E-relaxed sequence is allowed to span between methods.

F.2 Inlining

Inliners must be careful when dealing with a call to a method of different strictness. A call from a method to a more relaxed method can be inlined, conservatively, by treating the callee as strict as the caller; i.e., by ignoring any additional latitude granted the callee. Otherwise, if the strictness of the caller and callee differ, inlining the call requires either careful tracking of whether each check is relaxed or strict, or demoting the entire caller and inlined copy of the callee to a strictness that is at least as strict as the strictnesses of the caller and callee.

F.3 Finally handlers still guaranteed once a try block is entered

Because relaxed sequences cannot span across protected non-trivial region boundaries, this guarantee still holds. This is essential for preserving the usual idiom for acquiring and releasing a resource: [*Example*:

```
bool acquired = false;
try {
       acquire(ref acquired);
       S1;
} finally {
       if (acquired) release resource;
}
```

end example]

Quite often, the programmer knows little about how S1 might fail. If the "acquire", S1, and "release" were allowed to be part of the same relaxed sequence, and S1 failed, then the acquire and/or release portions could be suppressed at whim (by the rest of the rules). By forcing the

 ©ISO/IEC 2006 – All rights reserved

three parts to be in three separate sequences, we eliminate problems with regard to S1 failing. Of course, we do not eliminate problems that might arise if something else in the sequence for "acquire" fails, but that is a problem that can't be dealt with at the CLI level, and must be left to the programmer.

Relaxed sequences are allowed to span trivial region boundaries because optimizers were already allowed to remove such regions even when strict exception handling is specified.

F.4 Interleaved calls

One potential hazard that users should look out for is that when a relaxed method calls another relaxed method, checks can appear to migrate from callee to caller and vice versa. Thus, methods that enforce program invariants that must be maintained in spite of faults should be marked as being strict for faults whose retiming may break the invariant.

For example, the method `T.M` below keeps `x+y` invariant.

[*Example*:

```
.class M {
  .field public int32 x;
  .field public int32 y;

  .method public void T() cil managed {
        .maxstack 2
        ldarg.0                 // Compute x=x-1

        dup
        ldfld x
        ldc.i4.1
        sub
        stfld x

        ldarg.0                 // Compute y=y+1

        dup
        ldfld y
        ldc.i4.1
        add
        stfld y
  }
  ...
}
```

end example]

If this method is relaxed, and the caller is also relaxed, then the caller would be allowed, in the absence of constraining data or control dependences, to interleave the call with other instructions in the caller. If one of those other interleaved instructions faults, then any or all of `M`'s side effects might be suppressed. Thus, method `M` should be marked as strict if it is important to prevent a fault from destroying the invariant.

This "interference" from the caller is potentially annoying, but seems to be intrinsic to any definition of relaxed exceptions that permits both:

1. instruction reordering and
2. inlined method calls are at least as fast as manual inlining.

F.4.1 Rejected notions for fencing

This subclause explains why some alternative ideas for "check fence" rules that were rejected.

Volatile operations were a candidate, since they already prevent some kinds of reordering. Treating volatile memory operations as check fences would prevent interference in critical sections. However, there are two arguments against this. First, not all situations that need check

fences have anything to do with volatile operations. Second, it would penalize volatile references, which exist for sake of fast cross-thread communication.

F.5 Examples

This subclause shows some classic optimizations, and how relaxed exceptions make them much easier to apply than strict exceptions.

F.5.1 Hoisting checks out of a loop

In a relaxed method, bounds checks for arithmetically progressing indices can be hoisted out of a loop, and only the extremes are checked. For example, consider:

```
for( int i=lower; i<upper; ++i ) {
  a[i] = b[i];
  c[i] = d[i];
}
```

In a strict method, the bounds checks on `a` and `b` are difficult to hoist, because the assignment to `c[i]` is control-dependent on success of all the bounds checks in the loop. If a fault causes the loop to end prematurely, the initial prefixes of `a` and `c` must be written up to where the fault occurred. The hoisting can be of course done via "two versioning", but that would double the size of the generated code.

In relaxed methods, the bounds checks can easily be hoisted without resorting to two-versioning, so that the code executes as if written:

```
if(lower < upper) {
  // "Landing pad" in compiler parlance.
  if( lower < 0 || upper < a.Length || upper < b.Length || upper
< c.Length
      || upper < d.Length)
        throw IndexOutOfRangeException;

  int i=lower;
  do {
        a[i] = b[i];       // Unchecked
        c[i] = d[i];       // Unchecked
  } while( ++i<upper );
}
```

Notice that the rewrite implicitly hoists the check for `NullReferenceException` too. With strict exceptions, that hoisting would not be valid, because perhaps `a[0]=b[0]` succeeds but then `c` is null. For similar reasons, relaxed exceptions (specifically, with the exceptions indicated by `CompilationRelaxations.RelaxedArrayExceptions` and `CompilationRelaxations.RelaxedNullReferenceException` relaxed) enables the hoisting of the checks for `ArrayTypeMismatchException` for both assignments. Notice that relaxation allows the checks to be hoisted, not removed.

F.5.2 Vectorizing a loop

Vectorizing a loop usually requires knowing two things:

1. The loop iterations are independent
2. The number of loop iterations is known.

In a method relaxed for the checks that might fault, part 1 is frequently false, because the possibility of a fault induces a control dependence from each loop iteration to succeeding loop iterations. In a relaxed method, those control dependences can be ignored.

In most cases, relaxed methods simplify vectorization by allowing checks to be hoisted out of a loop. Nevertheless, even when such hoisting is not possible, ignoring cross-iteration dependences implied by faults can be crucial to vectorization for "short vector" SIMD hardware such as IA-32 SSE or PowerPC Altivec. For example, consider this loop:

 ©ISO/IEC 2006 – All rights reserved

```
for (k = 0; k < n; k++) {
  x[k] = x[k] + y[k] * s[k].a;
}
```

where `s` is an array of references. The checks for null references cannot be hoisted out of the loop, even in a relaxed context. But relaxed does allow "unroll-and-jam" to be applied successfully. The loop can be unrolled by a factor of 4 to create aggregate iterations, and the checks hoisted to the top of each aggregate iteration.

F.5.3 Autothreading a loop

Below is a C# rendition of the key routine for a sparse matrix multiply from the SciMark 2.0 suite:

```
int M = row.Length - 1;

for (int r=0; r<M; r++) {
  double sum = 0.0;
  int rowR = row[r];
  int rowRp1 = row[r + 1];
  for (int i = rowR; i < rowRp1; i++)
        sum += x[ col[i] ] * val[i];
  y[r] = sum;
}
```

This is an attractive candidate for parallelizing the outer loop. In a strict method, doing so is quite difficult; either we have to know `x[col[i]]` never faults, or have a way to make the writes to `y[r]` speculative.

If the method is relaxed for the possible faults, parallelizing the outer loop is only a matter of solving the usual data dependence problem ("Does `y[r]` ever alias `x[col[i]]`"). If any iteration of the loop faults, the relaxed exceptions allows the other iterations to quit early or keep going without concern for what state they leave `y` in.

Annex G Parallel library

This clause contains only informative text

This Annex shows several complete examples written using the parallel library

The classes in `System.Threading.Parallel` enable you to write parallel loops that take advantage of hardware that can execute multiple threads in parallel, without having to get involved in the details of dispatching and synchronizing multiple threads. [*Example*: The library lets you take an ordinary sequential loop like this:

```
for( int i=0; i<n; ++i ) {
    loop body
}
```

and rewrite it as a parallel loop like this:

```
new ParallelFor().Run( delegate( int i ) {
    loop body
});
```

end example]

G.1 Considerations

The programmer is responsible for ensuring that the loop iterations are *independent* (for sake of correctness) and have sufficient *grain size* (for sake of efficiency.) Loop iterations are *independent* if they can be carried out in arbitrary order, or concurrently and still produce the right answer. The *grain size* is the amount of work performed by a loop iteration. If the grain size is too small, the overhead (calling the delegate, synchronizing with other threads, etc.) may overwhelm the intended work. The ideal is to make the grain size large and uniform, but not so large as to make it difficult to distribute work evenly across physical threads.

For efficiency, `ParallelFor` is the preferred loop class when there is a choice. It tends to be the most efficient because it has the least general iteration space.

G.2 ParallelFor

`ParallelFor` should be used when parallelizing a loop whose index takes on values from 0 to *`n-1`*. Below is an example of how `ParallelFor` might be used in C# to parallelize the iterations in a cellular automaton on a grid. The variables `oldState` and `newState` are two-dimensional arrays the respectively hold the old and new states of the cells. [*Example*:

```
int n = oldState.GetLength(0);
new ParallelFor(n-2).Run(delegate(int iteration) {
    int i = iteration+1;
    for (int j = 1; j < n-1; j++){
        int count =
            (oldState[i-1,j-1] + oldState[i-1,j] + oldState[i-1,j+1] +
             oldState[  i,j-1] +                   oldState[  i,j+1] +
             oldState[i+1,j-1] + oldState[i+1,j] + oldState[i+1,j+1]);
        byte s = (count | oldState[i, j]) == 3 ? Live : Dead;
        newState[i, j] = s;
    }
});
```

end example]

There are two key points to notice. First, the outer loop logically iterates `i` from `1` to `n-1`. However, the `ParallelFor` class always iterates starting at 0. Hence the desired logical value of `i` is computed from the physical loop iteration number `iteration`. Second, outer loop is parallel; the inner loop is sequential. In general, if the loop iterations are independent for both inner and outer loops, it is better to parallelize the outer loop because doing so yields the largest grain size.

 ©ISO/IEC 2006 – All rights reserved

G.3 ParallelForEach

`ParallelForEach` should be used to parallelize a loop that iterates over a collection that supports the usual enumerator pattern. Below is an example that iterates over a list of file names.
[*Example:*

```
List<string> files = ...;
new ParallelForEach(files).Run( delegate(filename) {
    FileStream f = new FileStream( filename, FileMode.Open );
    ...read file f and process it...
    f.Close();
});
```

end example]

G.4 ParallelWhile

Use `ParallelWhile` to parallelize a loop over a collection that grows while it is being processed. Below is an excerpt showing how `ParallelWhile` might be used for parallel update of cells in a spreadsheet. Each cell is presumed to have a set `Successors` of cells that depend upon it, and a field `PredecessorCount` that is initially zero. Each cell must be updated before any of its successors is updated.

[*Example:*

```
void UpdateAll() {
    // Phase 1: Count predecessors
    foreach (Cell c in SetOfAllCells)
        foreach (Cell dependent in currentCell.Sucessors)
            ++dependent.PredecessorCount

    // Phase 2: Find cells with no predecessors
    ParallelWhile<Cell> parallelWhile = new ParallelWhile<Cell>();
    foreach (Cell c in SetOfAllCells)
        if (c.PredecessorCount]==0)
            parallelWhile.Add(c);

    // Phase 3: Do the updating
    parallelWhile.Run( delegate(Cell c) {
        ....update value of cell c...
       foreach (Cell dependent in c.Sucessors)
           if (Interlocked.Decrement(ref
dependent.PredecessorCount)==0)
               parallelWhile.Add(dependent);
    });
}
```

end example]

The example is structured as a classic topological sort. Phases 1 and 2 are sequential code. Because they are sequential, they do not have to update `PredecessorCount` in a thread-safe manner. Phase 3 is parallel: it starts processing all cells that phase 2 found were ready to update, and any cells found by phase 3 itself that were found ready to run. Because phase 3 is parallel, it updates `PredecessorCount` in a thread-safe manner.

G.5 Debugging

During initial debugging, set `System.Threading.Parallel.ParallelEnvironment.MaxThreads` to 1, which causes sequential execution of the parallel loop classes. Once your code runs correctly sequentially, experiment with setting `System.Threading.Parallel.ParallelEnvironment.MaxThreads` to higher values. In final production code, it is preferable to not set it at all, because it affects parallel loops everywhere in the application domain.

ICS 35.060

Price based on 555 pages

© ISO/IEC 2006 – All rights reserved