

ECMA

Standardizing Information and Communication Systems

Common Language Infrastructure (CLI)

Part 2: General

Draft 1 - October 2000

This contribution is being provided “AS IS”, and the SPONSORS EXPRESSLY DISCLAIM ANY AND ALL WARRANTIES REGARDING THIS CONTRIBUTION, INCLUDING ANY WARRANTY THAT THIS CONTRIBUTION DOES NOT VIOLATE THE RIGHTS OF OTHERS OR IS FIT FOR A PARTICULAR PURPOSE.

ECMA

Standardizing Information and Communication Systems

Brief History

This ECMA Standard has been adopted by the ECMA General Assembly of

Table of contents

1	Scope	1
1.1	Overview	1
1.1.1	Structure of the Document	1
1.1.2	Text Style	1
1.2	The Execution Engine and the CLI	2
1.3	Validation and Verification	2
1.4	Common Language Specification	3
2	Introductory Examples	4
2.1	Hello World Example	4
2.2	Examples	4
3	General Syntax	4
3.1	General Syntax Notation	4
3.2	Terminals	5
3.3	Identifiers	6
3.4	Labels and Lists of Labels	6
3.5	Lists of Hex Bytes	7
3.6	Floating point numbers	7
3.7	Source Line Information	8
3.8	File Names	8
3.9	Attributes and Metadata	8
4	Assemblies, Manifests and Modules	8
4.1	Assemblies, Modules, Types and Namespaces	8
4.2	CIL Assembly Files	9
4.3	Defining an Assembly	10
4.3.1	Operational Characteristics of Assemblies	11
4.3.2	Information about the Assembly	12
4.3.3	Manifest Resources	13
4.3.4	Files in the Assembly	14
4.4	Referencing Assemblies	14
4.5	Declaring Modules	15
4.6	Referencing Modules	15
4.7	Declarations inside a Module or Assembly	16
4.8	Export Declarations	16
4.8.1	The .comtype directive	17
5	Types	17
5.1	Introduction to Types	17
5.2	The Type System	19

5.3	Types	20
5.3.1	modreq and modopt	22
5.3.2	pinned	22
5.3.3	Types in Reflection Emit	22
5.4	Built-in Types	22
5.5	Type References, Assemblies and Modules	24
5.6	Inheritance and Subtyping	25
5.6.1	Verification of Subtyping	26
5.6.2	Conformance and Subtyping at Runtime	26
5.7	Native Data Types	26
5.8	Marshaling	29
5.8.1	Marshaling with Reflection	30
6	Visibility, Accessibility and Hiding	30
6.1	Visibility	30
6.2	Hiding	31
6.3	Accessibility	31
6.3.1	Family Access	31
6.3.2	Privatescope Access	32
7	Class Types	33
7.1	Namespaces	34
7.2	Using Classes	34
7.3	Instantiating Classes	35
7.4	Defining a Class	35
7.4.1	Class Head	35
7.4.2	Built-in Class Attributes	36
7.5	Body of a Class	39
7.6	Members of Classes	39
7.6.1	Static and Instance Fields	39
7.6.2	Static and Instance Methods	40
7.6.3	Virtual Methods	40
7.6.4	Method Implementation Requirements	42
7.6.5	Instance constructors	43
7.6.6	Instance Finalizer	43
7.6.7	Type Initializers	44
7.7	Nested Classes	46
7.8	Controlling Layout and Dispatch	47
7.8.1	Layout Control of Fields	47
7.8.2	Controlling Virtual Method Dispatch	48
7.9	Global Fields and Methods	49
8	Interfaces	49
8.1	Implementing Interfaces	50

8.1.1	Implementation Requirements	51
8.1.2	MethodImpls	51
8.2	Defining Interfaces	52
9	Value Types	52
9.1	Referencing Value Types	53
9.2	Instantiating Value Types	53
9.3	Defining Value Types	54
9.4	Methods of Value Types	55
9.5	Boxing and Unboxing	56
9.6	Copy Constructors on Value Types	57
9.7	Using Value Types for C++ Classes	58
9.7.1	Representation of a Class as a Value Type	58
9.7.2	Representation of the VTable	59
10	Special Types	59
10.1	Arrays	59
10.1.1	Vectors	59
10.1.2	General Arrays	61
10.1.3	Arrays of Arrays	65
10.2	Enumerations	66
10.3	Pointer Types	68
10.3.1	Obtaining and Using an Address	69
10.3.2	Unmanaged Pointers	70
10.3.3	Managed Pointers	70
10.3.4	Transient Pointers	71
10.4	Method Pointer Types	71
10.5	Delegates	72
10.5.1	Declaring Delegates	73
10.5.2	Creating Delegates	74
10.5.3	Using Delegates	74
10.5.4	Multicast Delegates	77
10.5.5	Other Methods of Delegates	78
11	Methods	78
11.1	Method Descriptors	79
11.2	Method Signatures	80
11.3	Types of methods	80
11.3.1	Static Methods	80
11.3.2	Instance Methods	80
11.3.3	Virtual Methods	81
11.4	Method Calls	81
11.4.1	Calling Convention	82
11.4.2	Call Kinds	82
11.4.3	The call Instruction	83

11.4.4	The callvirt Instruction	84
11.4.5	Indirect Calls	84
11.4.6	Tail Calls	85
11.4.7	jmp	86
11.4.8	Calling Instance Constructors	87
11.4.9	Calling vararg Methods	87
11.5	Defining Methods	87
11.5.1	Method Head	87
11.5.2	Method Parameters	88
11.5.3	Method Body	90
11.5.4	Predefined Attributes on Methods	93
11.5.5	Implementation Attributes of Methods	95
11.5.6	Scope Blocks	97
11.5.7	vararg Methods	98
11.6	Unmanaged Methods	99
11.6.1	Calling Unmanaged Methods	99
11.6.2	Managed Native Calling Conventions (x86)	102
12	Fields	104
12.1	Predefined Attributes of Fields	105
12.1.1	Accessibility Information	105
12.1.2	Field Contract Attributes	106
12.1.3	Interoperation Attributes	106
12.1.4	Other Attributes	106
12.2	Field Init Metadata	106
12.3	Embedding Data in a PE File	107
12.3.1	Data Declaration	108
12.3.2	Accessing Data	109
12.3.3	Unmanaged Thread-local Storage	109
12.4	Initialization of Static Data	109
12.4.1	Data Known at Link Time	109
12.4.2	Data Known at Load Time	110
12.4.3	Data Known at Run Time	110
13	Properties	111
13.1	Declaring properties	111
13.1.1	Property Head	111
13.1.2	Property Members	112
14	Events	113
14.1	Implementing Events	114
14.2	Observing Events	114
14.3	Declaring Events	115
14.3.1	Event Head	115
14.3.2	Event Members	115

15	Exception Handling	118
15.1	SEH Blocks	118
15.1.1	Protected Blocks	118
15.1.2	Handlers	119
15.2	Throwing an Exception	123
16	Declarative Security	124
17	Custom Attributes	124
17.1	CLS Conventions: Custom Attribute Usage	125
17.2	Attributes Used by the Runtime	126
17.2.1	Pseudo Custom Attributes	126
17.2.2	Attributes Defined by the CLS	126
17.2.3	Custom Attributes for JIT Compiler and Debugger	127
17.2.4	Custom Attributes for Reflection	127
17.2.5	Custom Attributes for Remoting	127
17.2.6	Custom Attributes for Security	127
17.2.7	Custom Attributes for TLS	128
17.2.8	Custom Attributes for the Assembly Linker	128
17.2.9	Attributes Provided for Interoperation with COM	129
18	CIL Instructions	130
18.1	Overview	130
18.2	Numeric and Logical Operations	132
18.3	Control Flow	135
18.3.1	Unconditional Branch Instructions	136
18.3.2	Unary Compare-and-Branch and Multi-Way Branch Instructions	136
18.3.3	Binary Compare-and-Branch Instructions	136
18.3.4	Procedure Call and Related Instructions	137
18.3.5	Exception Handling	137
18.3.6	Other Control Flow Instructions	137
18.4	Moving Data	137
18.5	Object Management	139
18.6	Annotations	140
19	Overview of File Format Extensions to COFF	141
19.1	Structure of the Runtime File Format	141
19.2	Producers and Consumers of the Runtime File Format	142
19.3	Requirements Addressed by the Runtime File Format Design	142
20	Emitting A Valid CLI Image	143
20.1	File Headers	143
20.1.1	Signature	143
20.1.2	COFF Header	143
20.1.3	Optional Header	144
20.1.4	Storing Runtime Data in Sections	147

1.2.2	Runtime Header	147
20.2	Section Headers	151
20.3	Modifications to Existing PE Data	151
20.3.1	Import Address Table (IAT)	151
20.3.2	Export Section (.edata)	151
20.3.3	Thread Local Storage Table	151
20.3.4	Relocations	152
21	Common Intermediate Language	152
21.1	Local Variable Layout	152
21.2	File Format Structure Definitions	152
21.2.1	Method Body	152
21.2.2	Section Data	155
21.2.3	IMAGE_COR_ILMETHOD_SECT_EH	155
21.2.4	IMAGE_COR_ILMETHOD_SECT_EH_CLAUSE_FAT	157
22	Code Transitions	158
22.1	Call Transitions	158
22.1.1	Transition Types	158
22.2	Runtime Header Support for Transitions	159
22.2.1	VTableFixups	159
22.2.2	Export Address Table Fix-ups	159
23	Entry Points	160
23.1	Runtime API's	160
23.1.1	_CorExeMain	160
23.1.2	_CorDllMain	160
23.1.3	Entry Points for Windows CE	161
23.2	Shut Down Requirements	161
23.3	Entry Point Stubs	161
23.3.1	Runtime Aware OS Loader	161
23.3.2	Non Runtime Aware OS Loader	161
23.3.3	Sample x86 Stubs	161
24	Metadata Format	163

1 Scope

The Common Language Infrastructure (CLI) provides a powerful platform for development. The Common Language Infrastructure provides a device and language independent way to express data and behavior of applications. While the CLI primarily supports Object Oriented Programming (OOP) languages, procedural and functional languages are also supported. Through the CLI, languages can interoperate with each other and make use of a built-in garbage collector, security system, exception support, and a powerful framework.

Common Intermediate Language (CIL) is the *intermediate language* emitted by all compilers that target the CLI. The CLI converts the device independent CIL binaries into native code using CIL-to-native code compilers (also incorrectly known as JIT compilers). These compilers can be run in a *Just-In-Time* (JIT) mode, converting methods from CIL to native code before a method runs for the first time. They can also be used to convert an entire assembly (see section 4.1) to native code and then saving the native code for future use. While it is possible to interpret CIL code, the runtime never interprets CIL but always compiles it into native code.

Tools that generate CIL can benefit from the many services provided by the runtime, including the support for early and late binding, and the fact that code compiled to CIL will run on any platform supported by the CLI. CIL is simple and fast to generate, which is essential in RAD (rapid application development) environments, where speed of compilation and ease of debugging are of primary importance. The runtime manages the native code generated from CIL so that this code may benefit from features such as cross-language inheritance, code access security, garbage collection, and simplified COM programming.

1.1 Overview

This document focuses on writing programs directly in the CIL assembly language, and relies heavily on the syntax of *ilasm*, a hypothetical assembler for the CIL. A complete syntax for *ilasm* is included in Part 5.

In order to understand the process of creating programs in the CIL assembly language, it discusses

- *Execution*: The execution engine, a model of a machine that supports the execution of CIL binaries
- *Types*: The underlying type system and the declarations used to define types
- *Instructions*: The operations of the CIL instruction set
- *Deployment*: These include assemblies, manifests and modules. Assemblies are the unit of deployment in the CLI.
- *Additional Features*, such as global methods, global fields, and interoperation with existing unmanaged code.

1.1.1 Structure of the Document

This document starts with an introductory sample (section 2) and with an introduction to the grammar (section 3) followed by a discussion of assemblies, modules and manifests (section 4). The document continues with a detailed overview of the type system, including their declaration, definition, and use (sections 5 – 10) followed by a description of their various members (sections 11 – 14). After a description of the members follows the specification of exceptions (section 15), security (section 16), and custom attributes (section 17). Finally, the document concludes with the description of all CIL instructions (section 18).

Part 5 contains a number of annexes, including samples that illustrate typical uses of the CIL assembly language, keywords reserved by *ilasm*, and the complete grammar of *ilasm*.

1.1.2 Text Style

The following styles are used throughout the document:

Category	Style	Example
Body text	Times New Roman	This is some text.

Category	Style	Example
Code embedded in text	Courier New	The member func is a function.
Hyperlinks	Blue (violet if visited) and <u>underlined</u>	<u>CIL Instruction Set specification</u>
CIL Keywords	Verdana, bold	call
Important facts	<u>Underlined</u>	<u>Note</u> : This is <u>not</u> unimportant.
Sample code	Courier New, compressed and keywords are bold	ldstr "Sample code"
Terms and emphasized text	<i>Italic</i>	Types have <i>members</i> .
Tool and file names	Arial, <i>italic</i>	<i>ilasm</i>

1.2 The Execution Engine and the CLI

The *execution engine* (EE) is responsible for executing PE (*portable executable*) files which are managed by the CLI. The EE translates PE files into native code. Further, the EE provides the program with an environment to run in.

The EE is described in Part 1. It provides a number of services to applications including:

- Garbage Collection
- Code Management
- Class Loading
- CIL interpretation and/or IL to native code compilation
- Thread Support
- Type Checking
- Exception Management
- Security Engine

1.3 Validation and Verification

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

Verification refers to a check of metadata to ensure type safety and a check that the use of certain CIL instruction sequences can permit any access to memory outside the program's logical address. In conjunction with the validation tests, verification ensures that the program cannot access memory or other resources to which it is not granted access.

Part 3 specifies the rules for both valid and verifiable CIL code. A mathematical proof of soundness of the underlying type system is possible, and provides the basis for the verification requirements. Aside from these rules this standard does not specify:

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

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

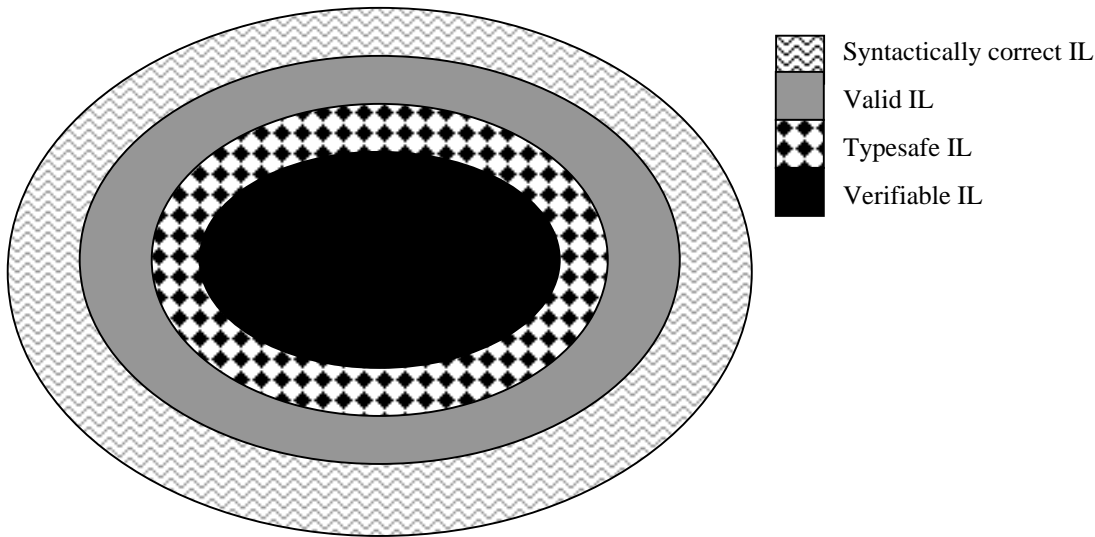


Figure 1: Relationship between valid and verifiable CIL. (Figure not drawn to scale)

In the above figure, the outer circle contains all code permitted by the CIL syntax. The next circle, which is solid gray, represents all code that is valid CIL. The dotted inner circle represents all type safe code. Finally, the innermost circle contains all code that is verifiable.

Note that even if the assembler accepts a CIL program, or a program follows the syntax described in this document, the code may still not be valid, because valid code must adhere also to other restrictions presented in this document. Also, the assembler may accept a somewhat more liberal syntax than presented in this document.

Verification is a very stringent test. There are many programs that will pass validation but will fail verification. The CLI cannot guarantee that these programs do not access memory or resources to which they are not granted access. Nonetheless, they may have been correctly constructed so that they do not access these resources. It is thus a matter of trust, rather than mathematical proof, whether it is safe to run these programs. Therefore, the CLI allows an unsafe subset of code, that is code that does not pass verification but is valid CIL, to be executed subject to administrative trust controls.

In general, CIL is used most often with a type-safe programming language whose compilers emit CIL that can be verified, but it is possible to generate CIL for unsafe languages, such as C and C++. The CIL emitted by the compilers for unsafe languages cannot, in general, be verified, but it will execute as a CLI managed application provided the correct security settings are set.

1.4 Common Language Specification

The *common language specification* (CLS) is a collection of rules and restrictions that allow interoperation between languages. Even though the CLI does not require compilers to follow CLS, code that follows the CLS rules is compatible with all other languages that follow the CLS rules.

The term *CLS consumer* refers to languages that can use all functionality provided by CLS compliant languages. This includes creating instances of classes of CLS compliant languages.

A *CLS extender* is a language which can be used to derive new classes from existing CLS classes and define new interfaces that comply with the CLS.

A *CLS framework* provides services implemented by a set of classes all of which are CLS compliant. The .NET framework is a CLS framework.

Typically a CLS compliant program will be verifiable. In general however, CLS programs may be unverifiable. The complete set of CLS rules can be found in Part 1.

2 Introductory Examples

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

2.1 Hello World Example

This section gives a simple example to illustrate the general feel of CIL. Below is code that prints the well known “Hello world” salutation. The salutation is written by calling `WriteLine`, a static method found in the .NET Frameworks class `System.Console`.

```
.assembly hello {}
.assembly extern mscorlib {}
.method static public void main() CIL managed {
    .entrypoint
    .maxstack 1
    ldstr "Hello World from CIL!"
    call void [mscorlib]System.Console::WriteLine(class System.String)
    ret
}
```

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

The method contains only three instructions. The **ldstr** instruction pushes the string constant "Hello World from CIL!" 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 must include the full signature of the called method. Finally, the last instruction returns (**ret**) from `main`.

2.2 Examples

This document contains integrated examples for most features of the CLI. Many sections conclude with an example that show a typical use of the feature. All these examples are written using the CIL assembly language. The corresponding code in other languages can be obtained from the appropriate language documentation.

Even though little samples of code make it easier to understand abstract concepts, they do not show how all the features of the runtime fit together to form a working system. As a solution this document provides a number of complete applications in Part 5. These applications show how the features of the CLI fit together to form a working system.

Many integrated examples come from parts of the samples in the appendix. Sometimes, these pieces were modified to illustrate the particular feature in a better way.

3 General Syntax

This section describes aspects of the CIL syntax that are common to many parts of the grammar.

3.1 General Syntax Notation

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

Bold items are terminals. Items placed in angle brackets (e.g. `<int64>`) are names of syntax classes and must be replaced by actual instances of the class. Items placed in square brackets (e.g. `[<float>]`) are optional, and any item followed by `*` can appear zero or more times. The character “|” means that the items

on either side of it are acceptable, and in this document each option introduced by a “|” is given on a separate line for easier reading. The options are sorted in alphabetical order (to be more specific: in ASCII order, ignoring “<” for syntax classes, and case-insensitive). If a rule starts with an optional term, the optional term is not considered for sorting purposes.

CIL is a case-sensitive language. All terminals must be used with the same case as specified in this reference.

Examples:

A grammar such as

```
<top> ::= <int32> | float <float> | floats [<float> [, <float>]*] | else <QSTRING>
```

would consider the following all to be legal:

12

float 3

float -4.3e7

floats

floats 2.4

floats 2.4, 3.7

else “Something \t weird”

but all of the following to be illegal:

else 3

3, 4

float 4.3, 2.4

float else

stuff

3.2 Terminals

The basic syntax classes used in the grammar are:

<int32> is either a decimal number or “0x” followed by a hexadecimal number, and must be represented in 32 bits.

<int64> is either a decimal number or “0x” followed by a hexadecimal number, and must be represented in 64 bits.

<hexbyte> is a hexadecimal number that fits into one byte.

<QSTRING> is a string surrounded by double quote (") marks. Within the quoted string the character “\” can be used as an escape character, with “\t” for a tab character, “\n” for a new line character, or followed by three octal digits in order to insert an arbitrary byte into the string. The “+” operator can be used to concatenate string literals. This way, a long string can be broken across multiple lines by using “+” and a new string on each line. An alternative is using “\” as the last character in a line, in which case the line break is not entered into the generated string. Any white characters (space, line feed, carriage return, and tab) between the “\” and the first character on the next line are ignored. See also examples below.

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

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

- As a label of an CIL instruction

- As an <id> which can either be an <ID> or an <SQSTRING>, so that special characters can be included.

Example:

The following examples shows breaking of strings:

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

and

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

become both "Hello World from CIL!".

3.3 Identifiers

Identifiers are used to name entities. Simple identifiers are just equivalent to an <ID>. However, the CIL syntax allows the use of any identifier that can be formed using the Unicode character set. To achieve this an identifier is placed within single quotation marks. This is summarized in the following grammar.

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

Keywords may only be used as identifiers if they appear in single quotes (see part 5 for a list of all keywords). The maximum length of identifiers is 1024 characters.

Several <id>'s may be combined to form a larger <id>. The <id>'s are separated by a dot (.). An <id> formed in this way is called a <dottedname>.

```
<dottedname> ::= <id> [ . <id> ]*
```

Note: In the syntax of ilasm, names that end with \$PST followed by a hexadecimal number have a special meaning. The assembler will automatically truncate the part starting with the \$PST. For more information see section 6.3.2.

Examples:

The following shows some simple identifiers:

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

The following shows identifiers in single quotes:

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

The following shows dotted names:

```
System.Console          A.B.C          'My Project'. 'My Component'. 'My Name'
```

3.4 Labels and Lists of Labels

A simple label is a special name that represents an address. Syntactically, a label is equivalent to an <id>. Thus, labels may be also single quoted and may contain Unicode characters.

A list of labels is comma separated, and can be any combination of these simple labels and integers. The integers are byte offsets that can be used instead of labels. However, these integers are intended for use only by a disassembler, e.g. *ildasm*, and not for use by a programmer.

```
<label> ::= <id>
```

```
<labeloroffset> ::=  
    <int32> /* For round trip use only */  
    | <label>  
  
<labels> ::= <labeloroffset> [ , <labeloroffset> ]*
```

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

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

```
<codeLabel> ::= <label> :
```

```
<dataLabel> ::= <label>
```

Example:

The following is code label that represents the address of the instruction `ldstr`:

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

3.5 Lists of Hex Bytes

A list of bytes consists simply of zero or more hex bytes. Hex bytes are pairs of characters 0 – 9, a – f, and A – F.

```
<bytes> ::= <hexbyte> [ <hexbyte>* ]
```

3.6 Floating point numbers

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

1. The regular way is to type in the floating point number with the dot (".") for the decimal point and "e" or "E" in front of the exponent. Both the decimal point and the exponent are optional. This way is primarily used to specify floats as parts of instructions
2. The second way is to convert from an integer to a floating point number by using either the keyword **float32** or **float64** and indicating the integer to be converted in parentheses. This way is used to specify floats for the metadata.

```
<float64> ::=  
    float32 ( <int32> )  
    | float64 ( <int64> )  
    | <realnumber>
```

Examples:

5.5

1.1e10

float64(128)

3.7 Source Line Information

To aid with debugging, source line information may be included in the PDB (Portable Debug) file associated with each module. There are two directives that can be used to accomplish this:

- **.line** takes a line number and an optional single quoted string that specifies the name of the file the line number is referring to
- **#line** takes both a line number and a (required) *double* quoted string that specifies the name of the file the line number is referring to

#line is only used for compatibility purposes. The **.line** directive is recommended for adding source line information.

```
<externSourceDecl> ::=  
    .line <int32> [<SQSTRING>]  
    | #line <int32> <QSTRING>
```

3.8 File Names

Some grammar elements require that a file name be supplied. A file name is like any other dotted name, with the difference that if provided the part after the last dot is interpreted as the extension of the file. The specific syntax for file names follows the specifications of the underlying operating system.

The following grammar shows the definition of file names:

<filename> ::=	Section
<dottedname>	3.3

3.9 Attributes and Metadata

The power of the CLI lies in the language independent representation of important information about types and its members in form of metadata. The term *metadata* refers to descriptive information that provides the runtime a specification of the types and their members as well as global information about the application.

In CIL, attributes of types and their members are the metadata which provides descriptive information. Many attributes are predefined and have a specific bit in the metadata associated with them. These attributes have keywords in the CIL assembly language.

Usually, the use of an attribute turns a bit on and the absence turns the same bit off. *ilasm* uses default values for attributes which are not specified. Generally, these default values correspond to the turned off bit state. However, in some cases there are exceptions. All default values are specified in this document.

4 Assemblies, Manifests and Modules

4.1 Assemblies, Modules, Types and Namespaces

It is important to understand the difference between assemblies, modules, types and namespaces, all of which are mechanisms for grouping constructs, each playing a different role in the CLI.

An *assembly* is a family of files that is deployed as a unit. Typically, it might include some image files (.exe and .dll's), resource files, online help or documentation. Although, in the extreme case, it's also perfectly feasible for an assembly to include just one file. An assembly has a strong sense of family membership; one of the files in the assembly embeds a list of all its siblings, together with their identity (name, version, culture information, etc). We call this list the assembly *manifest*. You cannot update just one file in an assembly; if you attempt to do so, the CLI will detect this *stranger* and refuse to load. If any part of an assembly needs to be updated, you build a new version of that assembly, and deploy it to customers. Types (classes, interfaces, value types) can be accessed from outside an assembly only if you explicitly mark them for export (using the *public* keyword on their definition). All other types are invisible from outside that assembly.

The assembly is also the entity treated by the security system. That's to say, a security administrator will grant rights based on the evidence of authenticity the assembly as whole unit can provide.

The assembly is also the unit of versioning. Assemblies have a version associated with them, which is used by the CLI to prevent version conflicts. In addition, the assembly provides culture information used for localization purposes.

A *module* is a single file containing executable content within an assembly, conceptually corresponding to a DLL or EXE in a native code environment. A module will typically contain a number of types and other declarations, and may itself be an assembly if the assembly only contains one module. A module contains metadata, in addition to the metadata of its enclosing assembly, that provides the runtime with necessary descriptive information about the module.

A *type* specifies a set of data and behaviors associated with each other. All values have a certain type and may only be assigned to variables that support the type. Types may themselves contain nested types. Each type is fully defined within a single module.

A *namespace* is syntactic sugar to create dotted type names, and while it may contain types (e.g. `System.Object`) and other namespaces (e.g. `System.XML`), there is no semantics associated with this containment. The CLI does not recognize namespaces as a separate entity. They become a part of the full name of a type.

4.2 CIL Assembly Files

A single file input to the CIL assembler is a sequence of declarations, defined as follows:

<ILFile> ::=	Section
<decl>*	6

The complete grammar for a top level declaration is shown below. The following sections will concentrate on the various parts of this grammar.

<decl> ::=	Section
.assembly <asmAttr>* <dottedname> { <asmDecl>* }	4.3
.assembly extern [<fullorigin>] <dottedname> { <asmRefDecl>* }	4.4
.class <classHead> { <classMember>* }	7
.comtype <comtypeHead> { <comtypeDecl>* }	4.8.1
.corflags int32	4.3
.custom <customDecl>	17
.data <datadecl>	12.3.1
.field <fieldDecl>	12
.file [<nometadata>] <filename> [.hash = (<bytes>)]	0
.manifestres [<public> <private>] <dottedname> [(<QSTRING>)] { <manResDecl>* }	4.3.3
.method <methodHead> { <methodBodyItem>* }	11
.module [<filename>]	4.5
.module extern <filename>	4.6
.namespace <dottedname> { <decl>* }	7.1
.subsystem int32	4.3
.vtfixup <vtfixupDecl>	7.8.2

<externSourceDecl>	3.7
<securityDecl>	15

4.3 Defining an Assembly

An CIL file does not necessarily define an assembly. It only does so if it contains a *manifest*. A manifest describes an assembly and contains information that is used by the assembler, other tools and the EE itself.

If a manifest is given, it should appear at the beginning of an CIL file. The manifest does not appear as a single declaration, rather it is begun by using the **.assembly** directive, and other declarations add further information. The following grammar specifies all the relevant declarations:

<decl> ::=	Section
.assembly <asmAttr>* <dottedname> { <asmDecl>* }	4.3
.corflags <int32>	4.3
.file [nometadata] <filename> [.hash = (<bytes>)]	0
.manifestres [public private] <dottedname> [(<QSTRING>)] { <manResDecl>* }	4.3.3
.subsystem int32	4.3
...	4.7

The **.assembly** directive begins the manifest and specifies to which assembly the current module belongs to. Each module may only contain one **.assembly** directive. After the **.assembly** directive any number of <asmAttr>s may be provided. A <dottedname> specifies the name of the assembly and is followed by the assembly declarations in braces.

The **.assembly** directive is required for *exe* files. However, it is optional for other modules (DLL files). The presence of the **.assembly** directive creates an assembly. At runtime, any files and modules referenced by the assembly using the **.module extern** (see section 4.6) and **.file** (see section 0) directives will become part of that assembly. Multiple assemblies may reference the same modules or files.

An CIL file may need to access other assemblies, which it does by using the **.assembly extern** directive covered in section 4.4. This information is then used by the EE to determine which assembly is to be used.

The following picture should clarify the various forms of references:

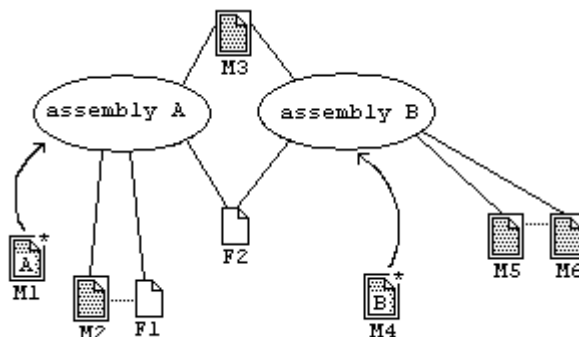


Figure 2: References

Eight files are shown in the picture. The name of each file is shown below the file. Files that declare a module have an additional border around them and have names beginning with M. The other two files have a name beginning with F. These files may be resource files, like bitmaps, or other files that do not contain CIL code. Files M1 and M4 declare an assembly in addition to the module declaration, namely assemblies

A and B, respectively. The assembly declaration in M1 and M2 references other module, shown with straight lines. A references M2 and M3. B references M3 and M5. Thus, M3 is referenced by both assemblies. Usually, a module belongs only to one assembly, but it is possible to share it across assemblies. When A is loaded at runtime, an instance of M3 will be loaded for A. When B is loaded, possibly simultaneously with A, another, independent, instance of M3 will be loaded for B. Both assemblies reference also F2, for which similar rules apply. The module M2 references F1, shown by dotted lines. As a consequence F1 will be loaded as part of the assembly A, when A is executed. Thus, the file reference must 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 must also have a module reference to M6.

Assembly names do not contain the file extension.

Note: Since some platforms may treat names in a case insensitive manner, two assemblies that have names that differ only in case should not be declared.

The following grammar shows the attributes allowed after a **.assembly** directive.

<asmAttr> ::=	Description	Section
implicitcom	Must be set in V1	
noappdomain	One instance only per application domain	4.3.1
nomachine	One instance only per machine, install time	4.3.1
noprocess	One instance only per process	4.3.1

Note that for Version 1, the CLI ignores any definitions made using the **noappdomain**, **nomachine** and **noprocess** keywords.

The **.manifestres** directive introduces a manifest resource declaration, described in section 4.3.3.

The **.subsystem** directive is used to indicate the kind of executable the assembly represents. E.g. a Windows GUI program or a Windows console program. Permissible values are defined in the Windows WinNT.h header file. See Part 5.

The **.corflags** directive can be used to specify flags specific to 64 bit architectures. Permissible values are defined in the CLI CorHdr.h header file. See Part 5.

Examples:

```
.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)
```

4.3.1 Operational Characteristics of Assemblies

The attributes **noappdomain**, **nomachine** and **noprocess** specify how many instances of the assembly are loaded. The attributes may be combined. The default is that none of them is specified. (However, note that for Version 1, the CLI ignores these settings)

An *application* is a complete program that forms its own assembly and contains its own set of threads. A *thread* is a sequential flow of execution of code. In contrast to an application, a *process* is an operating system entity that lives in a well defined memory space. Code inside a particular process cannot access any memory not allocated for that process. The CLI allows multiple applications to reside in the same process and thus share memory. The applications are separated by so called *application domains*. Each application has its own domain inside a process.

Usually, assemblies are loaded on a per application domain basis. However, if for some reason an assembly needs to be loaded only once for a process, the attribute **noappdomain** may be specified. In this case, all application domains inside the process will share the same instance of the assembly, and the assembly will have a unique state across all application domains inside the process.

If an assembly cannot support multiple instances across processes, the attribute **noprocess** may be used. This attribute is similar to **noappdomain**, but will make sure that only one instance of an assembly is loaded for a particular machine. All processes on this machine will share the same instance of the assembly with the same unique state.

In certain situations involving networking, an assembly might not support separate instances on separate machines. In this case, the **nomachine** attribute may be specified, which will guarantee that within a particular network session, only one instance of the assembly will be loaded. This instance will be shared across machines.

4.3.2 Information about the Assembly

The following grammar shows the information you may specify about an assembly. All of these are optional.

<asmDecl> ::=	Description	Section
.hash algorithm <int32>	Hash algorithm ID for hash value used in the .file directive	4.3.2.1
.title <QSTRING> [(<QSTRING>)]	Title of the assembly. The optional QSTRING is a description.	4.3.1
.custom <customDecl>	Custom attributes	17
.locale = (<bytes>) /* round trip only */	Information about the locale	4.3.2.2
.locale <QSTRING>	Information about the locale	4.3.2.2
.originator = (<bytes>)	Information about the originator. The <bytes> of the originator's public key.	4.3.1
.os <int32> .ver <int32> : <int32>	OS ID, with major version and minor version	4.3.2.3
.processor <int32>	Processor ID	4.3.2.4
.ver <int32> : <int32> : <int32> : <int32>	Major version, minor version, revision, and build	4.3.1

4.3.2.1 Hash Algorithm

The hash algorithm id is defined in the Windows header file *wincrypt.h*. The ID is the object identifier stored in one of the constants of the form CALG_<algorithm>, where <algorithm> is the name of the algorithm. This algorithm is used to calculate and compare the hash value of a file referenced with the **.file** directive (see section 0). The hash algorithm is not used for any other calculations.

In case you are not familiar with these algorithms, you may consider using the CALG_SHA1 algorithm.

See Part 5 for the permissible values.

4.3.2.2 Locale

The locale contains culture specific information. The strings that can be used in the **.locale** form are the same strings accepted by the System.Globalization.CultureInfo class. More information about this class can be found in the .NET SDK Base Class Library documentation.

The culture names follow the RFC1766 names. The format is “<language>-<country>”, where <language> is a lowercase two-letter code in ISO 639-1. <country> is an uppercase two-letter code in ISO 3166.

The list of strings is not reproduced here due to its length, but can be found with the documentation of the `CultureInfo` class.

The form of `.locale` that accepts a list of bytes rather than strings is only used by `ildasm` for round tripping purposes.

Example:

```
.locale "en-US" // locale string for the United States of America
```

4.3.2.3 Operating System

The `.OS` directive is for documentation purposes and records on what platform the assembly was built and tested. This information can be used for debugging purposes. In particular, this information does not restrict the target machine.

The OS id can be found in the Windows header file `winbase.h`. See Part 5.

4.3.2.4 Processor

Similar to the OS information, the processor information introduced by the `.processor` directive is used for documentation and debugging purposes only. The specified processor is the processor on which this assembly was built and does not restrict the target processor.

The processor id can be found in the Windows header file `winnt.h`. See Part 5.

4.3.3 Manifest Resources

A *manifest resource* is simply a named item of data associated with an assembly. As the name implies, it includes resources for the assembly, e.g. bitmaps, references to files, etc. A manifest resource is introduced using the following declaration, which adds the manifest resource to the assembly manifest begun by the `.assembly` declaration.

<decl> ::=	Section
<pre>.manifestres [public private] <dottedname> [(<QSTRING>)] { <manResDecl>* } ...</pre>	4.2

If the manifest resource is declared **public** it is exported from the assembly. If it is declared **private** it is not exported and only available from within the assembly. The <dottedname> is the name of the resource followed by an optional description in parentheses. The actual manifest resource declarations are provided in braces.

The following grammar defines a manifest resource declaration.

<manResDecl> ::=	Description	Section
<pre>.assembly extern <dottedname></pre>	Manifest resource is in external assembly with name <dottedname>.	17
<pre> .custom <customDecl></pre>	Custom attribute.	
<pre> .file <dottedname> at <int32></pre>	Manifest resource is in file <dottedname> at byte offset <int32>.	

Note that, for the **.file** directive, you must supply the byte offset at which the resource begins, even if its value (commonly) is zero.

4.3.4 Files in the Assembly

Assemblies may be associated with other files, e.g. documentation and other files that are used during execution. The declaration **.file** is used to add a reference to such a file to the manifest of the assembly:

<decl> ::=	Section
.file [nometadata] <filename> [.hash = (<bytes>)]	
...	4.2

The attribute **nometadata** is specified if the file does not contain any metadata, an example of such a file is a resource file.

The <bytes> after the optional **.hash** specify a hash value computed for the file. The EE will recompute this hash value when this file is accessed and report an error if it does not match. This ensures that the correct file is used and changes to the file do not break the code. The algorithm used to calculate this hash value is specified with **.hash algorithm** (see section 4.3.1).

If the hash value is not specified, it will be automatically computed by the assembly linker *al* when an assembly file is created using *al*. Even though the hash value is optional in the grammar for *ilasm*, it is required at runtime.

4.4 Referencing Assemblies

When you want to *refer* to constructs in an external assembly, you must first use a **.assembly extern** declaration to define some information about the assembly you wish to access, and to give the resulting assembly a name for the purposes of the rest of your CIL file. **.assembly extern** declarations are thus used for the resolution of names that refer to external entities. The declaration includes the external assembly name and a sequence of further declarations in braces. The option **fullorigin** specifies that the assembly reference holds the full (unhashed) originator.

<decl> ::=	Section
.assembly extern [fullorigin] <dottedname> { <asmRefDecl>*	
}	
...	4.2

The following is the grammar for a **.assembly extern** declaration:

<asmRefDecl> ::=	Description	Section
.hash = (<bytes>)	Hash Blob for file references	
.custom <customDecl>	Custom attributes	17
.locale = (<bytes>)	Information about the locale	4.3.2.2
/* round trip only		
*/		
.locale <QSTRING>	Information about the locale	4.3.2.2
.originator = (Information about the originator. The <bytes> are	
<bytes>)	the low 8 bytes of the SHA1 hash of the originator's	
	public key.	
.os <int32> .ver	OS ID, with major version and minor version	4.3.2.3
<int32> : <int32>		

.processor <int32>	Processor ID	4.3.2.4
.ver <int32> :	Major version, minor version, revision, and build	
<int32> : <int32>		
: <int32>		

The dotted name used in **.assembly extern** must exactly match the name of the assembly as declared with **.assembly** directive in a case sensitive manner.

These declarations are very similar to those for **.assembly** declarations (section 4.2) and some of them are described in the subsections following the description of the **.assembly** declarations, with the exception of the **.hash** value. You may determine appropriate settings for these by using *ildasm* on the assembly you wish to access.

The assembly *mscorlib* contains many of the classes and methods in the System namespace. This assembly may always be referenced implicitly, so that its classes and methods can be used without having to explicitly reference the assembly.

Example:

```
.assembly extern MyComponents {
    .originator = (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
}
```

4.5 Declaring Modules

All CIL files must be part of a module. An CIL file may be added to a module by using the **.module** directive. A module encapsulates only one file but may reference other files (see 4.6).

<decl> ::=	Section
.module [<filename>]	
...	4.2

Usually, a name will be provided with the module. The name is case sensitive and should be the same as the file name, including its file extension. The module will be referenced by this name.

If no name is provided, the name is set by *ilasm* to be the empty string (``'). If the **.module** directive is missing, *ilasm* will automatically add a **.module** directive and set the module name to be the file name, including its extension in capital letters. E.g., if the file is called foo and compiled into an exe, the module name will become "FOO.EXE".

Note: Since some platforms may treat names in a case insensitive manner, two modules that have names that differ only in case should not be declared.

Example:

```
.module CountDown.exe
```

4.6 Referencing Modules

Instead of referring to a construct via its containing assembly, you may instead need to refer to it via its module, in particular if the construct is part of another module in the same assembly. This can be done by using the **.module extern** directive with the module name. The file which contains the module must be referenced with the **.module** directive.

The module reference includes the file extension. The module name should be the same as the file name, and it is case sensitive.

<decl> ::=	Section
------------	---------

<pre> .module extern <filename> ...</pre>	4.2
--	-----

The file name used in **.module extern** must exactly match the name of the **module** as declared with **.module** directive in a case sensitive manner.

Example:

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

4.7 Declarations inside a Module or Assembly

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

<code><decl> ::=</code>	Section
.class <classHead> { <classMember>* }	7
.custom <customDecl>	17
.data <datadecl>	12.3.1
.field <fieldDecl>	12
.method <methodHead> { <methodBodyItem>* }	11
.namespace <dottedname> { <decl>* }	7.1
.vtfixup <vtfixupDecl>	7.8.2
<externSourceDecl>	3.7
<securityDecl>	15
...	

4.8 Export Declarations

.export is used to export types from a module or assembly. The **.export** declaration is used only in the main file of the assembly (i.e. the one with the manifest). This way, all exported types are specified at one place and there is no need to check all files to determine which types are exported.

As shown in the grammar below, after any number of attributes, the name under which the type is exported is specified. Finally, the export declarations follow.

<code><decl> ::=</code>	Section
.export [<exportAttr>] <dottedname> [(<compQstring>)] { <exportDecl>* }	

The following grammar shows the attributes of an **.export** declaration:

<code><exportAttr> ::=</code>	Section
nested assembly	6
nested famandassem	6
nested family	6

nested famorassem	6
nested private	6
nested public	6
public	6
private	6

The following grammar shows the body of an **.export** declaration:

<exportDecl> ::=	Description	Section
.class <int32>	Specifies a class	
.custom <customDecl>	Custom attributes	17
.file <dottedname>	Specifies a file	
.nestedtype <dottedname>	Specifies a nested type	

4.8.1 The **.comtype** directive

The **.comtype** directive is for disassembling purposes only. It should not be used. Instead, use the **.export** directive (see section 4.8).

<decl> ::=	Section
.comtype <comtypeHead> { <comtypeDecl>* } /* roundtrip only */	4.3.1

5 Types

5.1 Introduction to Types

The two key components of any software program are data and behavior. Types apply to both of these concepts. Data is stored in the form of bits and bytes. In order for humans to understand this data and do useful operations with it, the data needs to be interpreted to be of a certain *type*, e.g. an integer or a string. This form of loose typing can be made stricter. The data stored in memory is reached by using variables. In a static type model, only certain types of data can be associated with a particular variable. This is achieved by assigning a particular type to a variable.

The CLI has adopted a static typing model, in which variables have one type assigned to them. While static typing may restrict the programmer in some sense, this has many advantages. The most important advantage is *code verification*. With types, the CLI is able to detect code that handles its data incorrectly and thus in a potentially harmful way. With static typing this verification can be done before the program is executed, and any problematic code can be rejected before the execution starts. Verification does not only make code less error prone, but also opens the door for a *security system*. The security system protects the user and the machine on which the program is running from malicious attacks by applications.

However, it would be very restrictive to allow the association of a variable with only one type. E.g., it is known that an integer is a number. It is also known that a real is a number. Thus, a number is a more general type than an integer or a real. This form of a relationship between types is called *subtyping*. A type that is a special form of a general type is called a *subtype* of the general type. The general type is called the *supertype* of the specific type.

The notion of subtyping may be extended to let subtypes reuse behavior defined in their superclasses as well as affect the behavior of their superclasses. A subtype that does this needs to *inherit* behavior from its supertype. Inheritance and subtyping are defined in detail in section 5.6.

The CLI allows data that is a subtype of some general type to be assigned to a variable declared to be of that general type. Thus, at a particular time, a variable may represent a type of data other than its *declared*

type. The type of the variable which it represents at a particular time is called its *exact type*. The ability of a variable to have a variety of exact types and support different behaviors depending on its exact type is called *polymorphism*.

A type may be just a simple data item or be a container for a set of data items. Often data is associated with a particular set of behaviors, that define operations on the data. These behaviors may provide typical operations with the data and implement complicated algorithms that need to be performed on the data. This concept is called *abstraction*. Frequently, it is even desirable to protect the data from other programmers and provide a particular interface through which the data can be accessed and modified, which is called *encapsulation*. The combination of subtyping, abstraction, encapsulation, and polymorphism is referred to as an *Object Oriented Programming* (OOP) style. Even though the CLI does support procedural and functional programming styles, it has an emphasis on OOP and particularly targets this programming model.

Even though OOP is a powerful way to design large applications, Object Oriented type systems face a big challenge. On the one hand, there is the aim to fully support OOP concepts, in which there are no special types and all types conform to a common type. On the other hand, there are performance problems due to the larger overhead. Some programming languages solve this problem by distinguishing between *primitive types*, like integers, and types defined by classes. In these languages, primitive types and class types do not have a subtype/supertype relationship to each other and live in separate worlds. Other languages stay with the OOP concepts and take the performance loss.

The CLI provides an elegant solution to this problem. It does stay loyal to true OOP concepts and does not make a distinction between primitive and class types. However, at the same time, the CLI offers the same performance as a system that has primitive types. This is achieved by distinguishing two kinds of types, *reference types* and *value types*.

With the .NET Framework, the CLI provides many predefined types that define a number of behaviors, which enable the programmer to develop many applications. However, a fixed type system is of little use for a programmer. There are various ways to create new types in the CLI. New types may be defined by *classes*, *value type definitions* or *implicit type declarations*. Both the CLI and the CIL assembly language have direct support for these various forms of type declarations.

Classes of many OOP languages declare only reference types. A variable of a reference type stores a reference to the actual data. Thus, to reach the data the reference must be first *dereferenced*. The CLI supports two main kinds of reference types. These are pointer types and types defined by classes. Pointer types are fixed, while *class types*, also called self-describing types, are defined. There is yet another category of reference types called *interfaces*, which are used for type declarations.

Data that has a type defined by a class type is said to be an *instance* of the class type. Instances of class types are also called *objects*. In the CLI type system, all objects are subtypes of the class `System.Object`.

In contrast to reference types, the data of value types is stored directly in the variables themselves. This is in some sense similar to C++ structures. In the CLI type system, value types are defined by value type definitions. Value classes have certain restrictions on them as described in section 9.

In the CLI, types that correspond to primitive types of other languages are represented as value types. Further, a value type may be defined by the user, such that the CLI enables the programmer to make use of the higher performance of value types. This is especially useful for compilers of languages that have a rich set of primitive types.

The CLI supports an important feature that brings value types and reference types together. Value types may be converted into reference types. This operation is called *boxing*. A *boxed value type* behaves like any other reference type and is a subtype of `System.Object`. A boxed value type may be converted back into its unboxed form, which is called, following the naming pattern, *unboxing*.

The CIL assembly language provides keywords for certain types, like `int32` for integers. It is important to understand that even though there are keywords for these types, these types are not special. The keywords are just a more comfortable way to refer to these types and produce more efficient code. These keywords are just aliases to the value types defined in the `System` namespace. Since these types are used very often, they are encoded in a special space efficient way that is expressed at the assembly language

level by a dedicated keyword. There are also specific CIL instructions (like) that perform common operations on these types, so they may be manipulated even without the existence of the Base Class Library.

Finally, another important aspect of typing needs to be mentioned. Sometimes, it may be useful to reinterpret the data as a different type. The declared type of a variable may not change, however the type of the data may be changed. In general, reinterpreting the type of a variable is called *casting* if it does not change the data representation. If the data representation is changed as part of the reinterpretation, the change of type is called *coercion*. E.g., changing a real number into an integer is a coercion. The CLI has built-in support for casting of class types and casting or coercion of value types. Casting will explicitly declare for verification and the runtime that the data shall be treated as another type after the cast is done. The exact type of the variable must be a subtype of the type to which the variable is cast. Coercion is very similar to casting, but the exact type of the variable does not need to be a subtype of the type targeted by the coercion.

Section 7 describes reference types, section 8 discusses interfaces and section 9 describes value types. Section 10 is a description of special types, which are either types created on the fly by the runtime or features of the type system that deserve special discussion.

More information on types can also be found in the Part 1.

5.2 The Type System

The following diagrams give an overview of the CLI type system.

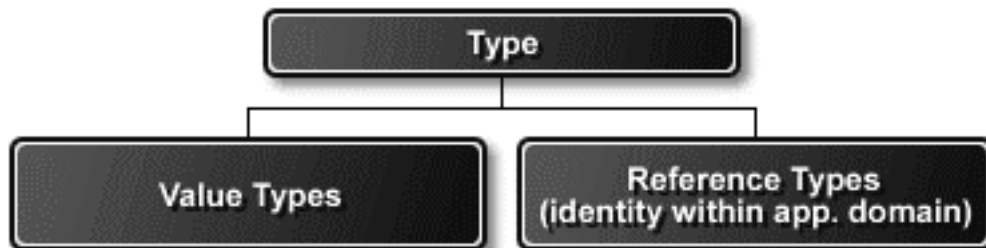


Figure 3: Value types and reference types.

Types are separated into value types and reference types. Value types have both a performance and space advantage compared to reference types. However, reference types give more flexibility to the programmer. Boxing and unboxing glue together reference and value types, by converting types defined by value classes to and from reference types and letting instances of value types enjoy the benefits of both worlds.

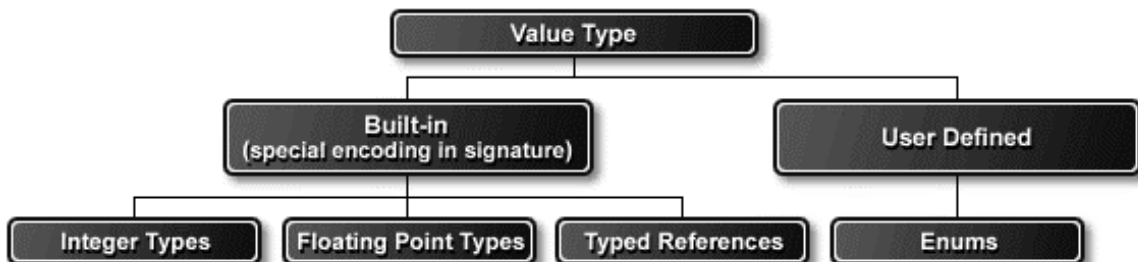


Figure 4: Value types

Value types can be further separated into built-in types, like integral types, floating point types, and typed references. Even though these types are built-in, they are not special but just predefined. Value types may also be user defined. Enumerations are a special class of user defined value types. More information on enumerations can be found in section 10.5.

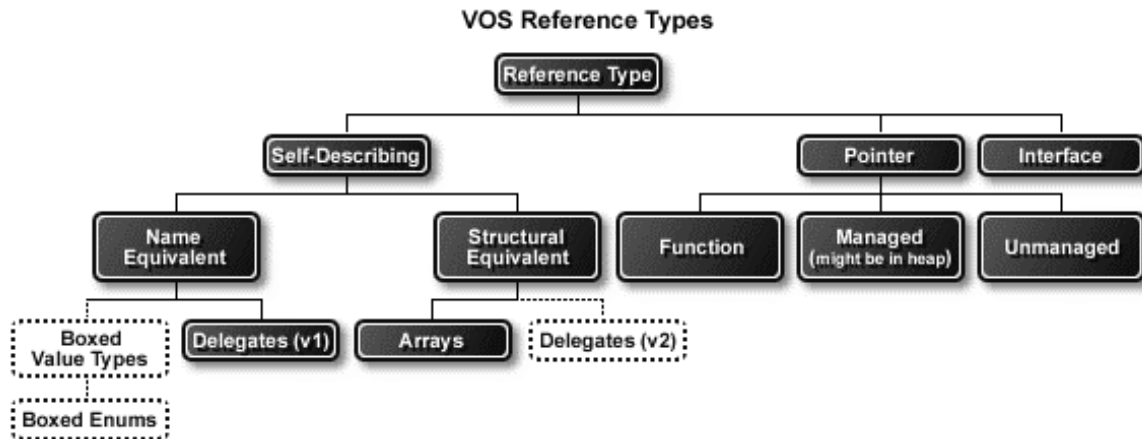


Figure 5: Reference types

Reference types are subdivided into self-describing reference types, pointer types, and interfaces. Self-describing types contain their full definition. There are name equivalent versions, which correspond to regular classes, and structural equivalent versions primarily used with arrays. The structural equivalent types are created on the fly by the CLI as needed. While in V.1 delegates will be name equivalent, in V.2 delegates may be structurally equivalent. A description of delegates can be found in section 10.2. Boxed value types, including enumerations, if implemented in V.2 will be name equivalent types.

Pointers are separated into method (or function) pointers (section 10.4) and pointers to data (section 10.3). Pointers to data may be managed by the runtime and enjoy the comfort of features like garbage collection or be unmanaged. The use of unmanaged pointers make the code in which they are used unverifiable. Unlike self-describing types, interfaces only declare a type but do not provide a full definition with it. Interfaces need to be implemented by a class.

5.3 Types

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

<type> ::=	Description	Section
bool	Boolean	
char	Unicode character	
class <typeReference>	User defined reference type.	
float32	32-bit floating point number	
float64	64-bit floating point number	
int8	Signed 8-bit integer	
int16	Signed 16-bit integer	
int32	Signed 32-bit integer	
int64	Signed 64-bit integer	
method <callConv> <type> * (<parameters>)	Method pointer	10.4
native int	Signed integer whose size varies depending on platform (32- or 64-bit)	
native unsigned int	Unsigned integer whose size varies depending on platform (32- or 64-bit)	
<type> &	Managed pointer (by-ref) to <type>. Note that <type> cannot itself be a managed pointer.	10.3
<type> *	Unmanaged pointer to <type>	10.3
<type> [[<bound> [, <bound>]*]]	Array of <type> with optional rank (number of dimensions) and bounds.	10.1.1.1
<type> modopt (<typeReference>)	Custom modifier that may be ignored by the caller.	5.3.1
<type> modreq (<typeReference>)	Custom modifier that the caller must understand.	5.3.1
<type> pinned	For local variables only. The garbage collector will not move the referenced value.	5.3.2
typedref	Typed reference, created by mkrefany and used by refanytype or refanyval .	
value class <typeReference> (value will be removed in Beta-2)	User defined value type.	
unsigned int8	Unsigned 8-bit integers	
unsigned int16	Unsigned 16-bit integers	
unsigned int32	Unsigned 32-bit integers	
unsigned int64	Unsigned 64-bit integers	
void	No type. Only allowed as a return type or as part of void *	
wchar	Unicode character	

In several situations the grammar permits the use of a slightly simpler mechanism for specifying types, by just allowing type names (e.g. “System.Object”) to be used instead of the full algebra (e.g. “**class** System.Object”). These are called *type specifications*:

<code><typeSpec> ::=</code>	Section
<code>[[.module] <dottedname>]</code>	5.5
<code> <typeReference></code>	5.4
<code> <type></code>	5.3

5.3.1 **modreq and modopt**

The use of a method or field may require certain care. The additional information is specified with a custom modifier. Typically, a custom modifier is used for a particular parameter of a method. However, it may be also used with the return type of a method or a field.

The keyword **modreq** is used with modifiers that the user of a method or field must understand and handle correctly in order to be able to use the method or field. The keyword **modopt** is used to provide additional information for tools that understand the modifier. In general, a user of a method or field does not need to understand the modifier introduced by a **modopt**.

Associated with the custom modifier is a type that is referenced in parentheses after the modifier. This type is used as a tag and qualifies the modifier. The type will tell tools and compilers what the meaning of the modifier is and distinguish the modifier from other modifiers. Even though the type is primarily intended as tag, it may implement behavior or provide fields.

The modifier with the referenced type becomes part of the signature of the method or field. This means that the same modifier with a reference to the same type needs to be used with any reference to the method or field. The type used to reference the method or field must match exactly the type of the modifier. Subclasses of this type may not be used.

5.3.2 **pinned**

The garbage collector of the runtime may move the location of objects in memory. If this is a problem, the garbage collector can be specifically instructed not to move an object referenced by a local variable. This is done by using the keyword **pinned** after the type of the local variable. **pinned** is only used with the declaration of the local variable. It is not used with any references to the local variable.

If unmanaged pointers are used to dereference managed objects, these objects must be **pinned**. Otherwise, the garbage collector may fail to operate properly.

5.3.3 **Types in Reflection Emit**

Types are primarily used as part of signatures, which can be constructed using `System.Reflection.Emit.SignatureHelper`. To construct a new type, use `System.Reflection.Emit.TypeBuilder` to define its fields, methods, properties, and events, then call the `CreateType` method to finalize the definition. Once a type has been defined, a new instance can be created using the `CreateInstance` method on the class `System.Reflection.Type`.

5.4 **Built-in Types**

The CLI built-in types have corresponding value types defined in the Base Class Library. The list of these classes must be known to all compilers because it is not legal for them to be referenced (in their unboxed form) in signatures by their value type names, they must be referenced by the `ELEMENT_TYPE` defined for that purpose (see *CorHdr.h*). When using `System.Reflection.Emit` this is taken care of by the `SignatureBuilder`.

Care must be taken when using the assembler since it will accept not only the built-in name of the type but the syntax **value class** followed by the name as used in the Base Class Library. This latter form, while it can be specified, will not execute correctly. This error is detected by *PEVerify*. (**value** keyword disappears in Beta-2)

Rather than using the name in the Base Class Library, a special keyword that the assembler provides needs to be used. E.g., rather than using `System.Int32`, the keyword **int32** must be used to refer to integers. Note that this is only required in declarations or signatures. To refer to a specific member of the class as specified in the Base Class Library, the class name needs to be used.

The table below shows various types provided by the runtime. The first column shows the syntax in *ilasm* and the second column shows the corresponding class in the Base Class Library. The third column specifies whether CLS compliant tools need to support the type. The fourth column gives a brief description and the last column shows the constant in *CorHdr.h* that corresponds to the type.

Name in <i>ilasm</i>	Type in Base Class Library	CLS Type	Description	Name from <i>CorHdr.h</i> (<code>ELEMENT_TYPE_<name></code>)
bool	<code>System.Boolean</code>	Yes	Boolean (true/false)	<code>BOOLEAN</code>
char	<code>System.Char</code>	Yes	Unicode Character	<code>CHAR</code>
<code>class System. Object¹</code>	<code>System.Object</code>	Yes	Object or boxed value type	<code>OBJECT</code>
<code>class System. String²</code>	<code>System.String</code>	Yes	Unicode String	<code>STRING</code>
float32	<code>System.Single</code>	Yes	IEEE 32-bit floating point	<code>R 4</code>
float64	<code>System.Double</code>	Yes	IEEE 64-bit floating point	<code>R 8</code>
int8	<code>System.Sbyte</code>	No	Signed 8-bit integer	<code>I 1</code>
int16	<code>System.Int16</code>	Yes	Signed 16-bit integer	<code>I 2</code>
int32	<code>System.Int32</code>	Yes	Signed 32-bit integer	<code>I 4</code>
int64	<code>System.Int64</code>	Yes	Signed 64-bit integer	<code>I 8</code>
native int	<code>System.IntPtr³</code>	No	Singed, native size integer	<code>I</code>
native unsigned int	<code>System.UIntPtr³</code>	No	Unsigned, native size integer	<code>U</code>

Name in ilasm	Type in Base Class Library	CLS Type	Description	Name from CorHdr.h (ELEMENT_TYPE_<name>)
typedref	System.TypedReference	No	Pointer to runtime type	TYPEDBYREF
unsigned int8	System.Byte	Yes	Unsigned 8-bit integer	U1
unsigned int16	System.UInt16	No	Unsigned 16-bit integer	U2
unsigned int32	System.UInt32	No	Unsigned 32-bit integer	U4
unsigned int64	System.UInt64	No	Unsigned 64-bit integer	U8
wchar ⁴	System.Char	Yes	Unicode Character	CHAR

Notes:

¹ In the .NET SDK not all mathematical operations are supported for this type. Future versions may support more operations

² In Beta-2, the ilasm keyword will change to simply **object**

³ In Beta-2, the ilasm keyword will change to simply **string**

⁴ **char** and **wchar** do the same thing. We recommend use of **wchar**

Example:

The following declares a 32 bit integer:

```
int32 x
```

The following converts the integer into a string:

```
// load x onto the stack
```

```
callvirt instance class System.String [mscorlib]System.Int32::ToString()
```

5.5 Type References, Assemblies and Modules

Type are referred to by using a dotted name. Names of nested classes (see section 7.7) are formed by using recursively the full name of the outer class, a slash (“/”) and the name of the nested class.

The use of a type within code is referred to as a *type reference*. Often, the definition of a type which is referenced is not available at compile time, but needs to be resolved by the CLI just before the code is executed. While types based on fundamental types, like **int32**, will be self-contained and appear as type specifications well known to the runtime, a type name used to specify a reference type (e.g. “**class MyClass**”) must be resolved. Once the definition of the type is found, the type reference can be replaced by a *type definition token*, which contains a pointer to an internal description of the type. These type description tokens are automatically inserted by the *ilasm* tool where possible. Type references will be used, however, if the type cannot be found in the same module in which the type is referenced.

In many cases the type definition is located in another module or even in another assembly. To help the runtime to resolve these types, a resolution scope needs to be attached in front of the type name. If the type is located in another module within the same assembly, a type reference is used and a *module reference* is attached to the type reference as part of the resolution scope. If the type is located in another assembly, an *assembly reference* is attached to the type reference. A module reference must have been declared by a prior **.module extern** directive (see section 4.5), and an assembly reference by a prior **.assembly extern** directive (see section 4.4).

The following grammar defines type references and resolution scopes. If a module reference needs to be used, then the **.module** keyword is used with the resolution scope. Otherwise an assembly reference is assumed.

```
<typeReference> ::=
    [<resolutionScope>] <dottedname> [/ <dottedname>]*
```

```
<resolutionScope> ::=
    [ .module <filename> ]
    | [ <assemblyRefName> ]
```

<pre><assemblyRefName> ::= <dottedname></pre>	Section 3.1
---	-----------------------

Examples:

The following is the proper way to refer to a class defined in the .NET SDK base class library. The name of the type is `System.Console` and it is found in the assembly named *mscorlib.dll*. A **.assembly extern** directive to define *mscorlib* must be declared before this reference:

```
class [mscorlib]System.Console
```

A reference to the named `C.D` in the module named *x.dll* in the current assembly. There must be a top-level **.module extern** directive to define *x.dll*:

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

A reference to the type named `C` nested inside of the type named `Foo.Bar` in another assembly, named *MyAssembly.exe*. Notice that there must already have been a top-level **.assembly extern** directive that defines the name *MyAssembly*:

```
class [MyAssembly]Foo.Bar/C
```

5.6 Inheritance and Subtyping

Reference types may be related to each other by inheritance and other rules. E.g. `System.String` is related to `System.Object`. We say that `System.String` is a *subtype* of `System.Object`. At the same time, `System.Object` is said to be the *supertype* of `System.String`. If one type is a subtype of another type, this means that it provides the same members, contracts, and member signatures as the other type. However, the behavior of the types may differ.

The word subtype includes the type itself. Thus, `A` is a subtype of `A`. However, `A` is not a *proper subtype* of `A`.

If class `A` defines a subtype of a type defined by a class `B`, then class `A` is a *subclass* of `B`. Similarly, class `B` is a *superclass* of `A`. The class `A` is said to *extend* the definition of class `B`. In the CLI type system every class must extend exactly one other class. The exception is the class `System.Object`, which does not extend any class.

If a class `A` defines a subtype of a type defined by an interface `I`, then class `A` is said to *implement* interface `I`. In the CLI, classes may implement any number of interfaces.

If a variable has a declared type that is a subtype of the declared type of another variable, then the first variable is said to be *assignment compatible* to the second variable.

In summary, the rules for subtyping between reference types are as follows:

- In the trivial case, a type is always a subtype of itself
- If a type B is a class type and has a superclass A, then type B is a subtype of type A
- If a type B is a class or interface type and supports interfaces I_0, \dots, I_n , then type B is a subtype of each of I_0 through I_n
- All reference types are subtypes of `System.Object`
- All array types are subtypes of `System.Array` (which is a subtype of `System.Object`)
- An array type `B[...]` is a subtype of an array type `A[...]` if type B is a subtype of type A. This applies if the rank of the two types is identical, and in the case of single dimensional arrays if one of the types has lower bound 0 then both must. The bounds are otherwise ignored. More information about arrays can be found in section 10.1.

5.6.1 Verification of Subtyping

Subtyping is most crucial for the verification of CIL instructions that call methods, perform stores, and return values from methods (e.g., **call**, **callvirt**, **ret**, **starg**, **stfld**, **stloc** etc., see section 18). The exact rules applied by verification vary (see section 18 and related specifications), but typically a variable may be assigned a value of another assignment compatible variable. E.g., if a variable has type A, and B is a subtype of A, then the variable may be assigned values of type B.

5.6.2 Conformance and Subtyping at Runtime

Types and type conformance is also relevant at runtime: the *exact type* of an object is the type of which its value is an instance. E.g. in the above case, even though the declared type is A, the exact type may be either A or B (or even some subtype of B). Exact types may be compared and checked by using CIL instructions such as **castclass** and **isinst**, as well as the facilities available in the reflection library.

5.7 Native Data Types

This section is a brief summary of native types. More information about native types can be found in the [Data Type Marshaling specification](#).

Native data types are optional items you can specify in method and field declarations. They are preceded by the **marshal** keyword, and define how a method's arguments and return value, as well as field values, should be marshalled between managed and unmanaged code.

The following table lists all native types supported by the CLI and provides a description for each of them.

<nativeType> ::=	Description
[]	Native array. Type and size are determined at runtime by the actual marshaled array.
as any	Dynamic type that determines the type of an Object at runtime and marshals the Object as that type.
bool	Boolean. 4-byte integer value where a non-zero value represents TRUE and 0 represents FALSE.
[ansi] bstr	A COM style BSTR with a prepended length and Unicode (or ANSI) characters. (Not supported for StringBuilder)
byvalstr	A string in a fixed length buffer.
custom (<QSTRING> , <QSTRING> , <QSTRING> , <QSTRING> ,	Custom marshaler. The 3 rd string is a custom marshaler type name. The 4 th string is a cookie – it may be blank. The first 2 strings must be present,

<QSTRING>)	and blank (ie ' ')
error	Return value for HRESULT methods that failed.
fixed array [<int32>]	A fixed size array of length <int32>
fixed sysstring [<int32>]	A fixed size system string of length <int32>
float32	32-bit floating point number.
float64	64-bit floating point number.
[unsigned] int	Signed or unsigned size-agnostic integer
[unsigned] int8	Signed or unsigned 8-bit integer
[unsigned] int16	Signed or unsigned 16-bit integer
[unsigned] int32	Signed or unsigned 32-bit integer
[unsigned] int64	Signed or unsigned 64-bit integer
interface	A COM interface pointer. The GUID of the interface is obtained from the class metadata.
lpstr	A pointer to a null terminated array of ANSI characters.
lpstruct	A pointer to a C-style structure. Used to marshal managed formatted classes and value types.
lpWSTR	A pointer to a null terminated array of platform characters.
lpvoid	An un-typed 4-byte pointer.
lpWSTR	A pointer to a null terminated array of Unicode characters.
method	A function pointer.
<nativeType> *	Pointer to <nativeType>.
<nativeType> []	Array of <nativeType>. The length is determined at runtime by the size of the actual marshaled array.
<nativeType> [int32]	Array of <nativeType> of size <int32>.
<nativeType> [.size .param = int32 [* int32]]	<nativeType> [.size .param = paramIndex * mult] Array of <nativeType>. The size of the array is specified by a parameter with index paramIndex (see also 11.5.6). An optional multiplier may be provided to increase the size by some factor.
safearray [<variantType>]	An OLE Automation SafeArray. The optional <variantType> supplies the unmanaged type of the elements within the array when it is necessary to differentiate among string types.
struct	A C-style structure, used to marshal managed formatted classes and value types.
tbstr	A COM style BSTR with a prepended length and platform dependent characters format (rarely used). ANSI is used on Win9x, and Unicode on WinNT and Win2K.
variant bool	Boolean. 2-byte integer value where the value -1

represents TRUE and 0 represents FALSE.

The following grammar specifies a variant type. These are used for marshalling. The native constants for variants types for older versions of Windows can be found in MSDN.

<code><variantType> ::=</code>	Description
<code>blob /* roundtrip only */</code>	Bytes prefixed with the length.
<code> blob_object /* roundtrip only */</code>	Blob contains an object.
<code> bstr</code>	A COM style BSTR with a prepended length and Unicode characters. (Not supported for StringBuilder)
<code> bool</code>	Boolean. 4-byte integer value where a non-zero value represents TRUE and 0 represents FALSE.
<code> carray /* roundtrip only */</code>	C style array.
<code> cf /* roundtrip only */</code>	Clipboard format.
<code> clsid /* roundtrip only */</code>	Class ID.
<code> currency</code>	A currency structure.
<code> date</code>	A data structure.
<code> decimal</code>	16 byte fixed point number
<code> error</code>	Return value for HRESULT methods that failed.
<code> filetime /* roundtrip only */</code>	Structure for a file time.
<code> float32</code>	32-bit single precision floating point number.
<code> float64</code>	64-bit double precision floating point number.
<code> hresult</code>	Standard return type.
<code> idispatch *</code>	COM style IDispatch interface.
<code> [unsigned] int /* roundtrip only */</code>	Signed or unsigned size-agnostic integer
<code> [unsigned] int8</code>	Signed or unsigned 8-bit integer
<code> [unsigned] int16</code>	Signed or unsigned 16-bit integer
<code> [unsigned] int32</code>	Signed or unsigned 32-bit integer
<code> [unsigned] int64 /* roundtrip only */</code>	Signed or unsigned 64-bit integer
<code> iunknown *</code>	Native pointer to COM style IUnknown interface.
<code> lpstr /* roundtrip only */</code>	A pointer to a null terminated array of ANSI characters.
<code> lpwstr /* roundtrip only */</code>	A pointer to a null terminated array of Unicode characters.
<code> null /* roundtrip only */</code>	SQL style null.
<code> record</code>	User defined type.
<code> safearray /* roundtrip only */</code>	A safe array.

storage /* roundtrip only */	Storage structure.
stored_object /* roundtrip only */	Store contains an object.
stream /* for roundtrip only */	A stream.
streamed_object /* for roundtrip only */	Stream contains an object.
userdefined /* for roundtrip only */	User defined type.
variant *	Native pointer to a variant type.
<variantType> &	Managed pointer to variant.
<variantType> []	Array of variant, size is unspecified.
<variantType> vector	A variant vector.

5.8 Marshaling

The CLI provides automatic marshaling to and from a variety of native (unmanaged) data types and corresponding managed data types.

The keyword **marshal** is used to specify how a parameter, return type or field should be marshaled to or from unmanaged code. It is used only if the implementation of a method is declared to be via COM or PInvoke (see also section 11.6.1) or if a field is passed to such a method as an argument.

The **marshal** keyword takes an optional native type. The CLI will select an appropriate marshaler according to the indicated native type and marshal the value when necessary.

Example:

The following shows a method declaration requests marshaling the string return value of the method to a **bstr**:

```
.method public static class [mscorlib]System.String marshal(bstr) MyMethod() CIL unmanaged {  
    // body  
}
```

5.8.1 Marshaling with Reflection

The marshal information can be set using the `SetMarshal` method in the appropriate builder class of the name space `System.Reflection.Emit`. The complete list of these types is in *CorHdr.h* as the enumeration `CorNativeType`.

6 Visibility, Accessibility and Hiding

The CTS (see also Common Type System Specification) makes use of three different ideas that must be mapped back to the desired language semantics:

Visibility controls whether or not a type is visible outside of the assembly in which it is defined. If a type is not visible outside its assembly, then no reference to that type can be resolved from other assemblies and the name of the type and its members do not participate in any way in name resolution at runtime.

Hiding controls which method names inherited from its super class are available during compile-time name binding. Members other than methods are not subject to hiding, since they can only be referenced using the type in which they are defined.

Accessibility does not affect name lookup directly (except for one case having to do with choosing the method implementation used to fulfill an interface method definition). Only visibility and hiding are considered when determining how a member reference should be resolved. Once resolved, the accessibility of the chosen member is examined and the lookup may fail (rather than the member being ignored) if the accessibility condition is not met.

The following sections provide more detail about these topics.

6.1 Visibility

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 allows visibility to be further refined. While a top-level type might be thought of as having either **public** or **assembly** accessibility, a nested type may have any of the 7 accessibility modes (see below) for its visibility.

The following table shows all visibility attributes.

Visibility	ilasm Keyword	Description
Public	public	The type may be exported from the assembly.
Assembly	private	The type is only visible within the assembly.
Nested	nested	The type has the same visibility as its outer type.

Note that the keyword for assembly visibility is **private**. It is assumed by default.

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 outer type is visible only within an assembly then a nested type with **public** accessibility is still only available within the assembly. By contrast, a nested class that has **assembly** accessibility is restricted to use within the assembly even if the outer class is visible outside the assembly.

6.2 Hiding

Hiding applies to individual members of a type (nested types are *not* considered to be members for this purpose). The CTS specifies two mechanisms for hiding:

hide-by-name, meaning that the introduction of a name in a given class hides all inherited members of the same kind (method or field) with the same name.

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

Hiding is a compile-time, not a runtime, issue. It specifies how the compiler should treat method references—either by generating an error or resolving to a specific method. It is only this resolved reference that is used by the runtime.

The runtime supports only the latter form of hiding. Compilers that desire the effect of hide-by-name need to emit appropriate code to achieve this functionality; this is done by the use of the **newslet** keyword and judicious choice of class name for resolution.

6.3 Accessibility

There are seven accessibility modes. These can be applied to members of a type and to nested types, which have an accessibility associated with them in addition to their **nested** visibility.

The following table shows and describes the accessibility attributes:

<accessibility> ::=	Description
assembly	Accessible only to referents in the same assembly that contains the implementation of the type.
family	Accessible only by referents whose base class (immediate or indirect) defined the member or type in question.
famandassem	Accessible only to referents that qualify for both Family and Assembly access.
famorassem	Accessible only to referents that qualify for Family or Assembly access, but not necessarily both.
private	Accessible only to referents in the implementation of the exact type that defines the member.
privatescope	Not accessible by a reference, but only with a member definition token.
public	Accessible by all referents.

With the exception of attributes involving **family** as described below, all other accessibility attributes can be verified statically (i.e. at compile time). The following two subsections give more information on **family** and **privatescope** accessibility.

6.3.1 Family Access

It is worth explaining the modes that involve **family** accessibility (**family**, **famandassem**, **famorassem**) in more detail. **family** accessibility gives a context sensitive way to control which classes may access certain members as described by the following two rules:

1. The accessing type must be a subtype of the type that declares the member.

2. If the member is associated with an instance of a type, then the type of the instance must be a subtype of the accessing type.

For **static** members, only the first rule applies. However, for **instance** (and **virtual**) members the second rule must be followed, too. The EE will always check the first rule, but not the second rule. The second rule will be checked by verification. Code that does not follow the second rule is not verifiable.

The following class diagram illustrates this:

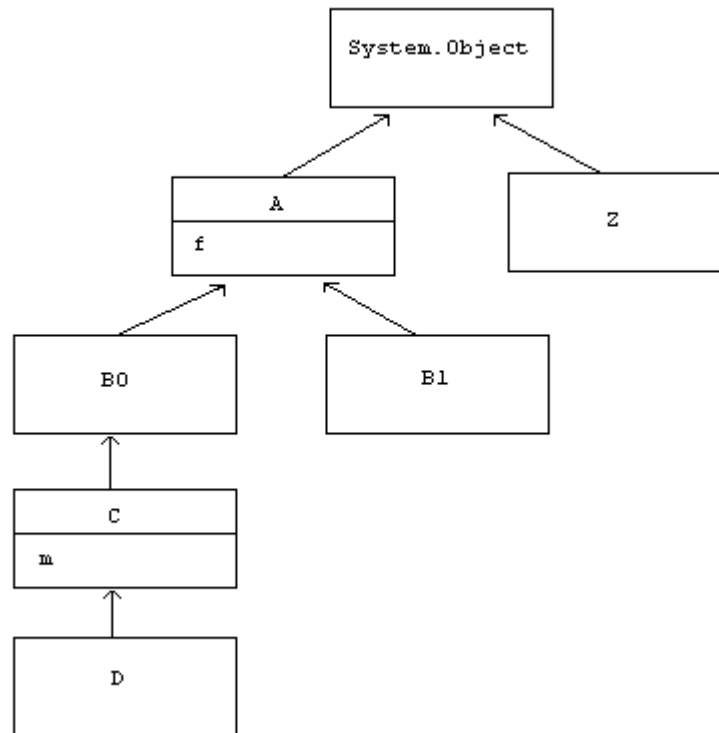


Figure 6: Family accessibility

Suppose there is a class A that defines an instance **family** member f. In the diagram, the arrows show subtyping, where, e.g., an arrow from B0 to A means that B0 is an immediate subtype of A.

Clearly Z cannot access f, since rule 1 does not apply for Z. However, types A, B0, B1, C and D all fulfill rule 1. They all are subtypes of A. Thus, the EE will not throw any exceptions if an instance of one these types is used to access f. However, not all types follow rule 2. Imagine that m, defined in C, tries to access f. To follow rule 2, m must use either an instance of C or D to access f, because those are the only types that are subtypes of the accessing type C. It is a verification error if m uses an instance of B0, B1 or A to access f.

If for some reason, m accepts an instance over a parameter that has a declared type that adheres to rule 1, but m is sure that the exact type of the parameter does follow rule 2, then a cast operation may be used with the parameter to do a verifiable access to f. The parameter needs to be cast to type C, the accessing type, and not A.

6.3.2 Privatescope Access

privatescope effectively restricts access to a member to the same compilation unit that defines it, allowing a compiler complete control over accessibility. The member can only be accessed with a member definition token. This access mode is most useful for implementing concepts like function-local static variables.

In the syntax for the assembler, a specific privatescope item can be referenced by using the postfix `$PST<token value>` after its name. The token value is a hexadecimal number that specifies the token associated with the item.

The assembler `ilasm` will automatically truncate names of the form `<name>$PST<token value>` to just `<name>`. The disassembler `ildasm` produces automatically names of the form `<name>$PST<token value>` for **privatescope** items. To determine the token value, the disassembler may need to be run on the output of the assembler after the item was declared.

Example:

The following is the declaration of a **privatescope** method, note that the actual name of the method in the metadata is `MyMethod`:

```
.method static privatescope void MyMethod$PST06000001() {  
    // method body  
}
```

The following is a call to the method:

```
call void MyMethod$PST06000001()
```

7 Class Types

Class types describe data which are heap allocated and accessed using a reference. Class types are defined by *classes*.

Classes have *members* that describe the data and behavior of a class type. The CLI supports the following groups of class members:

- fields
- methods
- properties
- events

Fields are typed memory locations that store the data of classes and their instances. Fields are described in more detail in chapter 12.

Methods implement the behavior of classes. They are described in further detail in chapter 11.

Properties are another way to represent data. While fields correspond to actual memory locations that store the data, a property is a group of methods that provide access to the data and the ability to change the data. The data represented by properties may be stored in one or more memory locations or may be computed on the fly. More information on properties can be found in chapter 13.

Events are a group of methods that communicate changes of the state of an object to other objects. Using events, an object may be notified when the state of the system changes and react to the change. More about events can be found in chapter 14.

Members may be **instance** or **static**. **instance** members are associated with a particular instance of the type, while **static** members are associated with the type itself. **static** members are shared across all instances of a type.

The CLI supports nested types. Nested types are declared inside other types. Nested types are not formal members of their outer types. However, the outer class restricts the visibility of its nested types by its visibility declaration. Nested classes are covered in more detail in section 7.7.

The CLI type system supports single class inheritance. At runtime classes may be instantiated, allocating an *object* on the heap and creating a reference to this location. As the name implies, classes are reference types and only a reference to their instances may be stored in variables or passed as arguments to other methods.

All classes inherit ultimately from the same base class, `System.Object`, which makes polymorphism between all class types possible. Any class may be treated as if it were of type `System.Object`.

[Note that in Beta-2, these will be shortened even further to use the keywords **object** and **string** respectively]

Outside of signatures both classes are referred to as any other class. The special forms provide enable a space efficient encoding in signatures.

Examples:

```
class [.module CountdownComponents.dll] CounterTextBox
```

The class CounterTextBox is referred to using the module reference *CountDownComponents.dll*.

```
class System.String[]
```

System.String is referred to without using an assembly reference. Using [mscorlib]System.String would cause a verification error.

7.3 Instantiating Classes

Classes are instantiated using the **newobj** instruction. The **newobj** instruction allocates memory for an instance of the class on the heap and calls the constructor of the class. The constructor to be called is specified as part of the **newobj** instruction. Any parameters accepted by the constructor must be pushed onto the stack before the **newobj** instruction is executed. The **newobj** instruction will return a reference to the object on the stack when it is complete.

Example:

The following code loads the parameters expected by the constructor of the class to be instantiated and then uses the **newobj** instruction to instantiate the class. The desired constructor is specified with the instruction.

```
ldloc button
ldloc count
newobj instance void [.module Counter.dll]BeepingCounter::ctor(class
[.module Counter.dll]StartStopEventSource, class [.module
Counter.dll]Count)
// reference on stack
```

7.4 Defining a Class

Classes may be defined at the top-level of an CIL file program as the following grammar shows:

```
<decl> ::= Section
    .class <classHead> { <classMember>* }
    | ...
```

7.4.1 Class Head

A class head consists of

- any number of class attributes (see section 7.4.2)
- a name (an <id>)
- an optional superclass (see section 5.6)
- an optional list of interfaces to be implemented (see also section 8)

The following grammar shows the syntax of a class declaration:

```
<classHead> ::=
    <classAttr>* <id> [extends <typeReference>] [implements
    <typeReference> [, <typeReference>]*]
```

The **extends** or **implements** clauses contain references to other classes or interfaces (see also section 7.7). A <typeReference> is simply a <dottedname> with an optional module or assembly reference, that is resolved to the appropriate class at runtime.

The **extends** keyword defines the *superclass* of a class, for a kind of inheritance called *code inheritance*. In the CLI a class always inherits code from exactly one other class. If no class is specified, ilasm will add an extend clause to make the class inherit from `System.Object`.

The **implements** keyword defines the *interfaces* of a class, a kind of inheritance called *multiple type inheritance*. Any number of interfaces may be specified. A class that implements an interface promises to provide implementation for all **abstract virtual** methods declared inside the interface. More about interfaces can be found in section 8.

The inheritance structure creates a hierarchy of classes. At the top of the hierarchy must be `System.Object`.

Example:

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

The code above 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.dll* of the current assembly. The attributes **private**, **auto** and **autochar** are described in the following sections.

7.4.2 Built-in Class Attributes

Predefined attributes of a class may be grouped into attributes that specify visibility, class layout information, class semantics information, special semantics, implementation attributes, interoperation information, and information on special handling. The following subsections provide additional information on each group of predefined attributes.

The following grammar shows and describes the attributes of a class:

<classAttr> ::=	Description	Section
abstract	Class is abstract.	7.4.2.4
ansi	Used for string marshaling across managed/unmanaged boundary.	7.4.2.6
auto	Auto layout of class.	7.4.2.2
autochar	Specifies to use platform specific char marshal across boundary.	7.4.2.6
explicit	Layout of fields are provided explicitly.	7.4.2.2
import	The class is imported from COM.	7.4.2.5
interface	The declaration is an interface declaration.	7.4.2.3
lateinit (rename to before_field_init in Beta-2)	Initialize class as late as possible.	7.4.2.7
nested assembly	Assembly accessibility for nested class.	7.4.2.1
nested famandassem	Family and Assembly accessibility for nested class.	7.4.2.1
nested family	Family accessibility for nested class.	7.4.2.1

nested famorassem	Family or Assembly accessibility for nested class.	7.4.2.1
nested private	Private accessibility for nested class.	7.4.2.1
nested public	Public accessibility for nested class.	7.4.2.1
not_in_gc_heap (remove in Beta-2)	Specifies that the class shall not be allocated in garbage collected heap.	7.4.2.3
private	Private visibility of top-level class.	7.4.2.1
public	Public visibility of top-level class.	7.4.2.1
rtspecialname	Special treatment by runtime.	7.4.2.7
sealed	The class cannot be subclassed anymore.	7.4.2.4
sequential	The class is laid out sequentially.	7.4.2.2
serializable	Specifies that the fields of the class may be serialized by the CLI serializer.	7.4.2.5
specialname	Special treatment by tools.	7.4.2.7
unicode	Used for string marshaling across managed/unmanaged boundary.	7.4.2.6

7.4.2.1 Visibility and Accessibility Attributes

The visibility of top-level classes is either **private** or **public**. Nested classes have **nested** visibility, which mean that they have the same visibility as their outer class. In addition nested classes have an accessibility attribute, which specifies the range from which they can be referenced.

The accessibility attributes are **assembly**, **famandassem**, **family**, **famorassem**, **private**, and **public**. These attributes must follow immediately the **nested** attribute.

Visibility attributes are described in section 6.1 and accessibility attributes are described in section 6.3. Nested classes are covered in section 7.7.

A class may only have one visibility and nested classes in addition one accessibility associated with them. The default visibility for top-level classes is **private**. The default accessibility for nested classes is **private**.

7.4.2.2 Class Layout Attributes

The class layout attributes are **auto**, **explicit**, and **sequential**. These attributes are used to specify how the fields of an instance of a class are arranged. Layout attributes are mutually exclusive and the default is **auto**.

auto specifies that the layout is done by the runtime.

explicit specifies that the layout of the fields is explicitly provided (see also section 7.8.1).

sequential specifies that the fields are laid out in sequential order by the runtime. The order used is the order of declaration. Otherwise, the runtime may use a different order. The attribute **sequential** needs to be specified if interoperation with the unmanaged world is desired.

7.4.2.3 Class Semantics Attributes

The class semantics attributes are **interface**, **not_in_gc_heap** (to disappear in Beta-2) and **value** (also to disappear in Beta-2)

These attributes specify what kind of type is defined. **interface** specifies that an interface is defined. The default is the definition of a reference type by a class, in which case no class semantics attribute is used.

In Beta-2, instead of marking a ValueType with the **value** attribute, you should make the class descend from `System.ValueType`.

The **not_in_gc_heap** attribute, used in conjunction with the **value** attribute, denotes that no instance of this ValueType should ever be allocated from the garbage collected heap (it should be allocated elsewhere – for example, on the stack). As a consequence, any operation that causes such an instance to be allocated, or moved into garbage collected heap is illegal. Examples of such operations are boxing or using the type as the base type of an array. Allocating an instance of a such a ValueType type on the heap causes a verification error.

Instances of class types are always allocated on garbage allocated heap and may not be marked **not_in_gc_heap**.

(the **not_in_gc_heap** keyword will be removed in Beta-2. Types with the special semantics implied by this attribute will instead be so marked by attaching one, or more, system-defined custom attributes to their definition)

7.4.2.4 Special Semantics Attributes

Attributes that specify special semantics are **abstract** and **sealed**. These attributes may be used together.

abstract specifies that this class may not be instantiated. Typically, even though not necessary, **abstract** classes contain one or more **abstract** methods (see section 11.5.4.4). If a class contains **abstract** methods, it must be declared as an **abstract** class.

sealed specifies that a class may not have any subclasses. A class that inherits from a **sealed** class cannot be loaded and will generate a `System.TypeLoadException` when an attempt is made to load such a class. **virtual** methods of **sealed** classes turn effectively into **instance** methods. They cannot be overridden since the class may not have any subclasses.

A class that is both **abstract** and **sealed** cannot have any non-**static** members. Such a class defines only a set of **static** fields and methods. It may define nested types.

7.4.2.5 Implementation Attributes

The implementation attributes are **import** and **serializable**. These attributes may be combined.

import specifies that the class (or interface) is imported from COM.

serializable indicates that the fields of the class may be serialized into a data stream by the CLI serializer. E.g., the class may be sent over the network or saved to a file. More about serialization can be found in the Base Class Library documentation.

7.4.2.6 Interoperation Attributes

These attributes are for interoperation with unmanaged code and define how managed Strings should be marshalled between managed and unmanaged code. Specify none or one of **ansi**, **autochar**, **unicode** (default is **ansi**).

ansi says that arguments whose type is managed String (array of 2-byte Unicode characters) will be marshalled to ANSI strings (arrays of 1-byte ANSI characters) before calling the target, unmanaged routine (and vice-versa)

unicode says that arguments whose type is managed String will be marshalled to UNICODE strings before calling the target, unmanaged routine (and vice-versa). [in effect, a simple copy]

autochar says that Strings will be marshalled to either Unicode or ANSI, depending upon the platform at runtime (Windows-NT => Unicode, Windows-9X => ANSI)

7.4.2.7 Special Handling Attributes

The three attributes that are used for special handling are **lateinit** (becomes `before_field_init` in Beta-2), **rtspecialname** and **specialname**. These attributes may be combined.

lateinit instructs the runtime to initialize the class as late as possible, rather than initializing classes at load time or soon after that (see also section 7.6.7).

rtspecialname signals a special name to the runtime, while **specialname** signals a special name to some other tool.

7.5 Body of a Class

A class may contain any number of further declarations. The following grammar shows the grammar for these declarations and provides a description for each item.

<code><classMember> ::=</code>	Description	Section
<code>.class <classHead> { <classMember>* }</code>	Defines a nested class.	7.4
<code> .comtype <comtypeHead> { <comtypeDecl>* } /* for round trip only */</code>	Exports a COM type, use .export instead. (For round trip only).	4.8.1
<code> .custom <customDecl></code>	Custom attribute.	17
<code> .data <dataDecl></code>	Defines static data associated with the class.	12.3
<code> .event <eventHead> { <eventMember>* }</code>	Declares an event.	14
<code> .export [public private] <dottedname> { <exportDecl>* }</code>	Specifies entities to export. Public exports it from the assembly. Private exports it only inside the same assembly.	4.8
<code> .field <fieldDecl></code>	Declares a field belonging to the class.	12
<code> .method <methodHead> { <methodBodyItem>* }</code>	Declares a method of the class.	11
<code> .override <typeSpec> :: <methodName> with <callConv> <type> <typeSpec> :: <methodName> (<parameters>)</code>	Specifies that the first method is overridden by the definition of the second method.	7.6.3.1
<code> .pack <int32></code>	Used for explicit layout of fields.	7.8.1
<code> .property <propHead> { <propMember>* }</code>	Declares a property of the class.	13
<code> .size <int32></code>	Used for explicit layout of fields.	7.8.1
<code> <externSourceDecl></code>	.line or #line	3.7
<code> <securityDecl></code>	.permission or .capability	16

The directives **.event**, **.field**, **.method**, and **.property** are used to declare members of a class. These members are discussed in more detail in the following sections and chapters.

The directive **.class** inside a class declaration is used to create a nested type, which is discussed in further detail in section 7.7.

7.6 Members of Classes

This section gives an introduction into some of the members of classes. This discussion primarily concentrates on the members and their relationship to a class or to an instance of a class. Most of the members are also discussed in other chapters of this document.

7.6.1 Static and Instance Fields

static fields are associated with the type which declares them. They are similar to global variables in the sense that they are shared by all objects of the type. Only one copy of the field is created each time

the type is loaded. A **static** variable is created when a class is loaded and initialized by the type initializer. (see section 7.6.7)

instance fields are not only associated with the type but with a particular instance of the type. To access an **instance** field, a reference to the instance is needed. Each instance of the type has its unique copy of the **instance** field. **instance** fields are created and initialized when an instance of the class is created.

7.6.2 Static and Instance Methods

static methods are very similar to procedures of procedural languages. They are associated with a class or interface, but are not associated with any particular instance of the type.

In contrast to **static** methods, **instance** methods are associated with an instance of the class. Typically, to call an **instance** method an instance of the class is needed. However, it is possible to use a *null* reference to call an **instance** method.

The CLI automatically adds a *hidden* parameter to instance methods. This new parameter becomes the first parameter of the method and has the type of the *declaring* class or interface. This parameter is not explicitly specified in the signature of the method, unless it is specified otherwise by the calling convention (see section 11.4.1).

Even though not explicitly specified, the hidden parameter still needs to be passed to the method. Thus, for all calls of **instance** methods, a reference that points to an object that is a (not necessarily proper) subtype of the declaring type of the method. The passed in hidden parameter, which has argument index 0, can be used in calls to other **instance** methods declared by the type of the hidden parameter or it can be used to access the **instance** fields of that type. However, an **instance** method needs to be prepared for the case when the first argument is *null*.

Except for their special hidden parameter, **instance** methods are the same as **static** methods.

More about **static** and **instance** methods can be found in section 11.3.

7.6.3 Virtual Methods

virtual methods are similar to instance methods. They are also associated with a particular instance of a class. However, **virtual** methods may be overridden by subclasses, such that the implementation of a **virtual** method is determined at runtime. This gives subclasses the ability to modify their behavior compared to that of their superclasses.

Similar to **instance** methods (see section 7.6.2) the first argument of **virtual** methods must be a reference to an instance of the class. When called as a virtual method (see below) the hidden parameter must not be **null**.

Introducing a new **virtual** method in a **sealed** class is possible and is precisely the same as introducing the method as an **instance** method.

The **final** attribute on a **virtual** method prohibits any subclass from providing its own implementation of this **virtual** method. However, a new **virtual** method with the same signature may be introduced via the **newslot** attribute. This can be thought of as creating a new method that has no relationship to the method of the superclass.

A **virtual** method of a superclass is overridden by providing a direct implementation of the **virtual** method using a definition in the class and not specifying it to be **newslot** (see section 11.1). A method may also be overridden using the **override** directive, which is described in section 7.6.3.1.

When a **virtual** method is introduced for the first time in the inheritance hierarchy, it can be done in either of two ways. The preferred method is to use a definition that provides the location of the code that implements the method and is marked as **newslot**. This causes the CLI to create a new method that has no relationship with any method of its superclass. If later a superclass defines a **virtual** method with the same signature, this method will not override the implementation of the superclass.

It is also possible not to mark the method as **newslot** which will cause the CLI to create a new method only if no existing **virtual** method with that signature is provided by its superclass. This is not

recommended for the first introduction of a **virtual** method, however, since it allows a superclass to capture this implementation by introducing a **virtual** method with the same signature.

When computing whether a **virtual** method overrides a **virtual** method of its superclass, given that the **virtual** method is not marked **newslot** the EE will try to find an inherited **virtual** method to override. The EE will continue to search up the chain of superclasses looking for a **virtual** method with the same signature as the introduced **virtual** method. This search ignores intervening **static** methods with the same signature. If the EE is unable to locate an existing **virtual** method in any of its parents a new method is created, exactly as though the definition had been marked **newslot**.

7.6.3.1 The **.override** Directive

Usually, the runtime will automatically determine which method overrides which method by matching signatures.

However, it is possible to explicitly specify which method overrides another method using **MethodImpls** (see also section 11.1). A **MethodImpl** takes two **MethodRefs**. The first **MethodRef** specifies the method to be implemented and the second specifies the method that implements the first method. The implementing method must be declared in the class that specifies the **MethodImpl**. Its definition may be deferred to a subclass by declaring the implementing method **abstract** in the class that defines the **MethodImpl**. Both methods must have a matching signature, which means that except for their type and name their signature must be equal to each other.

In the *ilasm* grammar a **MethodImpl** is defined with the **.override** directive. However, the **.override** directive makes a simplification and requires only the type and name of the first method and a full **MethodRef** for the implementing method. The calling convention, return type and parameter types of the first **MethodRef** are inferred from the second **MethodRef** since they must be the same.

The syntax for **.override** is as follows:

	Section
<classMember> ::=	
.override <typeSpec> :: <methodName> with <callConv>	
<type> <typeSpec> :: <methodName> (<parameters>)	
...	7.5

The first method specified by the partial **MethodRef** will be overridden by the method specified by the **MethodRef** after the **with** keyword.

The **.override** directive may appear inside a class or inside a method. If it appears inside a method, the implementing method is the method inside which the directive appears and the second **MethodRef** is omitted from the **.override** directive (see also section 11.5.3).

Example:

The following example shows a typical use of the **.override** directive. A method implementation is provided for a method declared in an interface. Interfaces are described in detail in section 8.

Suppose the interface, call it **I**, declares the following method:

```
.method public virtual abstract void m() CIL managed {}
```

And suppose a class, call it **C**, implements **I** and provides the following method, notice the different name:

```
.method virtual public void m2() {  
    // body of m2  
}
```

Then the **.override** directive below will associate the method from the interface with the implementation of the class:

```
.override I::m with instance void C::m2()
```

7.6.4 Method Implementation Requirements

A class (concrete or **abstract**) may provide

- implementations for **instance**, **static**, and **virtual** methods that it introduces
- implementations for methods declared in interfaces that it has specified it will implement, or that its superclass has specified it will implement
- alternative implementations for **virtual** methods inherited from its parent
- implementations for **virtual** methods inherited from an **abstract** superclass that did not provide an implementation

A concrete (i.e. non-**abstract**) class must provide either directly or by inheritance an implementation for

- all methods declared by the class itself
- all **virtual** methods of interfaces implemented by the class
- all **virtual** methods that the class inherits from its superclass

If a class overrides an inherited method, it may *widen*, but it cannot *narrow*, the accessibility of that method. As a principle, if a client of a class is allowed to access a method of that class, then it should also be able to access that method (identified by name and signature) in any derived class. Table 7.1 defines the precise meaning of *narrow* and *widen* – a “Yes” denotes that the subclass can apply that accessibility, a “No” denotes it is illegal.

Table 7.1: Legal Widening of Access to a Virtual Method

Subclass		Superclass Accessibility					
		private	family	assembly	famandassem	famorassem	public
private		Yes	No	No	No	No	No
family		Yes	Yes	No	No	If <u>not</u> in same assembly	No
assembly		Yes	No	Same assembly	No	No	No
famandassem		Yes	No	No	Same assembly	No	No
famorassem		Yes	Yes	Same assembly	Yes	Same assembly	No
public		Yes	Yes	Yes	Yes	Yes	Yes

Notice that a method may be overridden even if it may not be accessed by the subclass. If a method has **assembly** accessibility, then it must have **public** accessibility if it is being overridden by a method in a different assembly. A similar rule applies to **famandassem**, where also **famorassem** is allowed outside the assembly. In both cases **assembly** or **famandassem**, respectively, may be used inside the same assembly.

A special rule applies to **famorassem**. Here is the only case where the accessibility is apparently narrowed by the subclass. A **famorassem** method may be overridden with **family** accessibility by a class in another assembly. This way, the implementer is not forced to use **public**. The EE will handle this case in a special way. For such a method, the EE will grant access to any type inside the assembly of the superclass, even though **family** is specified.

For CLS compatibility, the accessibility of a **virtual** method must not be changed when it is overridden.

7.6.5 Instance constructors

Instance constructors initialize an instance of a class or value type. An instance constructor is called when an instance of a class is created.

An instance constructor must not be **static** or **virtual**. It must be named **.ctor** and marked with **rtspecialname** and **specialname**. Instance constructors may take parameters, but may not return a value. Instance constructors may be overloaded, (i.e. a class may have several instance constructors). Each instance constructor must have a unique signature. At instantiation time, this signature is specified and determines which constructor is called.

Instance constructors are a kind of **instance** method. However, instance constructors have a special privilege to write into fields of the class that are marked with the **initonly** attribute.

An instance constructor should call one of the instance constructors of its superclass. If an instance constructor of the superclass is not called, the instance of the class will be not be completely initialized. A program that has an instance constructor which does not call an instance constructor of its superclass is not verifiable.

Example:

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

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

7.6.6 Instance Finalizer

An instance finalizer allows objects to execute some final code before they are reclaimed by the garbage collector. The finalize method is invoked when the GC determines that the current object is no longer being referenced by any other object.

The finalize method is defined in the class `System.Object` as follows:

```
family virtual void Finalize() { }
```

As you can see, the `Finalize` method in `System.Object` does nothing. If you want to use a `Finalize` method to, for example, release resources before that object is reclaimed by GC, you must override the `Finalize` method. Finalizers should be kept short: there is a timeout (of a few seconds) for running Finalizers during shutdown – after that time, the process is simply shutdown without completing the running of any remaining Finalizers.

Note that CLI does not guarantee *when* a Finalizer will be run, nor the *order* in which it is run, compared with other Finalizers. In the extreme, a Finalizer may not actually be run at all. For example, when the application shuts down, the CLI does **not**, by default, execute Finalizers at all – after all, the process and its entire memory space is about to be reclaimed by the Operating System. However, if you want to change this default, then call the method `System.GC.RequestFinalizeOnShutdown` – this can be called at any time, and it applies to the whole process. Thus, if one `AppDomain` calls `RequestFinalizeOnShutdown()`, then all `AppDomains` in that process are affected. Note that there is *no* method provided to cancel the request to finalize on shutdown – it's a once-off decision for that process.

Finalization of a specific object may be explicitly prevented by using the `System.GC.SuppressFinalize(Object obj)` method. By notifying GC, we avoid burning cycles needlessly;

it also allows that object to be reclaimed sooner by the GC, since it need never be added to the internal “finalize” queue.

Note that ‘nothing comes free’ – if you request Finalization, then GC will take longer to complete. `Finalize` may take any action, including resurrecting an object (that is, making the object accessible again) after it has been cleaned up by the GC. (However, an object that is resurrected will never have its Finalizer run again, even if it becomes garbage)

Because Finalizers are not guaranteed to run, you should follow the Dispose design pattern (see [Design Guidelines spec](#)) if you really need objects to release resources promptly. [in effect, devise a “Dispose” method on the class, and call it explicitly when the last reference to a particular object of that class is about to be released]

In Beta2, the CLI will default to always running Finalizers on shutdown. The `System.GC.RequestFinalizeOnShutdown` will be withdrawn. And there will be no means provided to change that default behavior. The `System.GC.SuppressFinalize` method will be retained – its usefulness still applies.

Example:

```
.method virtual family void Finalize() {
    .maxstack 3
    // remove the onClick event
    ldarg.0
    dup
    ldfld class [mscorlib]System.EventHandler StartStopButton::
        onClickEventHandler
    call instance void [System.Windows.Forms]System.Windows.Forms.Button::
        remove_Click(class [mscorlib]System.EventHandler)
    ret
}
```

7.6.7 Type Initializers

Classes may contain special methods called *type initializers* to initialize the class itself.

Classes, interfaces, and value types may all have type initializers. This method must be **static**, take no parameters, return no value, be marked with **rtspecialname** and **specialname**, and be named **.cctor**. Thus a class may only have one type initializer. Most type initializers are simple methods that initialize static fields of the type from stored constants or via simple computations. There are, however, no direct limitations on what code is permitted in a type initializer.

Type initializers are a kind of **static** method. Thus, they may only access **static** fields. Type initializers have a special privilege to write into **static** fields of the class that are marked with the **initonly** attribute.

Example:

The following shows the definition of a type initializer:

```
.method static public rtspecialname specialname void .cctor() CIL managed
{
    // do other initialization work
    ret
}
```

7.6.7.1 Type Initialization Guarantees

There are three fundamental guarantees about Type initialization.

1. The Type initializer (`.cctor`) always starts running before

- any instance of that Type is created
 - any static member (method or field) of that Type is referenced
2. A Type initializer is run exactly once for any given Type, unless explicitly called by user code (few languages allow the user to call a Type initializer explicitly, but it *is* supported by ILASM)
 3. No method other than these called directly or indirectly from the Type initializer will be able to access members of a Type before its initializer completes execution.

7.6.7.2 Delaying Type Initialization

The default behavior (which we informally describe as “precise”) of a Type Initializer (.cctor) is as follows:

- CLI guarantees to have started running the .cctor, by the first occurrence of : access of any static or instance field (of that Type), or invocation of any static or instance method (of that Type). Moreover, it is one of these four events that triggers execution of the .cctor -- so, in effect, running the .cctor is left as late as possible

You can request another behavior, by specifying the **lateinit** attribute on the Type definition (in Beta-2, this name will change to **before_field_init**). This provides improved performance, particularly when the Type is included in a ‘shared’ or ‘domain-neutral’ assembly. The behavior in this case is as follows:

- CLI guarantees to have started running the .cctor, by the first access of any static field (of that Type); but CLI is allowed to start running the .cctor earlier, if it so chooses
- Note that, if you select this option for a particular Type, it's entirely possible that code might execute a static method of this Type before the .cctor runs. Also, it's possible that code might create objects of the Type, and call methods on these objects, before the .cctor runs

Classes, ValueTypes and Interfaces can be marked as either “late initialize required” or “early initialize allowed” based on the **lateinit** attribute (see also section 7.4.2.7) (note that this attribute will be renamed to **before_field_init** in Beta-2) Requiring them to be initialized late ensures that the type initializer will be called no earlier than absolutely required to meet the first of the guarantees. Allowing early initialization relaxes this requirement and allows the JIT and the execution engine to initialize the class at any earlier time, allowing them to optimize performance but at the possible cost of predictable behavior.

7.6.7.3 Races and Deadlocks

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, any attempt to reference any of the **static** variables causes a problem, since the type initializer for each of A and B requires that the type initializer of the other be invoked first. If we required that no access to a type was permitted until its initializer had completed we 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 class to be visible, which would make it difficult to guarantee repeatable results.

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

The CLI deals with these problems by ensuring that in addition to the three type initialization guarantees (see 7.6.7.1) two further guarantees for code that is called from a class initializer are met:

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

The check of these guarantees is optimized by the JIT. Once the class is initialized, the check is not done anymore.

A rough outline of the algorithm is as follows:

1. At class load time (hence prior to initialization time) store zero or null into all **static** variables of the class.
2. If the type is initialized you are done.
3. If the type is not yet initialized, try to take an initialization lock.
 - If successful, record this thread as responsible for initializing the type and proceed to step 4.
 - If not, see whether this thread or any thread waiting for this thread to complete already holds the lock.
 - a. 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.
 - b. If not, block until the type is initialized then return.
4. Initialize the parent type and then all interfaces implemented by this type.
5. Execute the type initialization code for this type.
6. Mark the type as initialized, release the initialization lock, awaken any threads waiting for this type to be initialized, and return.

7.7 Nested Classes

One class may be nested within another. All references to the class are through its *enclosing* class. The nested class has its own accessibility, and references to the nested class must therefore have access to both the enclosing class and the nested class itself. Nested classes have the same visibility as their enclosing class. Their visibility must be declared to be **nested**.

A nested class is not associated with an instance of its enclosing class. The nested class has its own superclass and may be instantiated independent of the enclosing class. This means that the **instance** members of the enclosing class are not accessible using the *this* pointer of the nested class.

A nested class may access any members of its enclosing class, including **private** members, as long as the member is **static** or the nested class has a reference to an instance of the enclosing class. Thus, by using nested classes a class may give access to its **private** members to another class.

On the other side, the enclosing class may not access any **private** or **family** members of the nested class. Only members with **assembly**, **famorassem**, or **public** accessibility can be accessed by the enclosing type.

Example:

The following example shows a class declared inside another class. Both classes declare a field. The nested class may access both fields, while the enclosing class does not have access to the field b.

```
.class private auto autochar CounterTextBox extends
[System.WinForms]System.WinForms.TextBox implements [.module
Counter.dll]ICountDisplay {
    .field static private int32 a
    /* Nested class. Declares the NegativeNumberException */
    .class nested assembly NonPositiveNumberException extends
[mscorlib]System.Exception {
        .field static private int32 b
        // body of nested class
    } // end of nested class NegativeNumberException
}
```

7.8 Controlling Layout and Dispatch

In some cases, it may be useful to control the layout of fields of an instance. E.g., layout control becomes necessary to enable interoperation with unmanaged code. A set of directives make layout control of fields possible. Further, it may be also desirable to directly influence the dispatch of **virtual** methods.

7.8.1 Layout Control of Fields

The CLI supports **sequential** and **explicit** layout control. If **sequential** layout control is specified, the runtime will allocate space for the fields in the order they are declared and one next to the other.

If the exact position of the fields needs to be specified, explicit layout control must be used and the class marked with the attribute **explicit** (see section 7.4.2.2).

The following grammar shows the declaration of a field as it appears after the **.field** directive:

```
<fieldDecl> ::=
    [[ <int32> ]] <fieldAttr>* <type> <id> [= <fieldInit> | at
        <dataLabel>]
```

The optional **int32** specified in brackets at the beginning of the declaration specifies the byte offset from the beginning of the instance of the class. This form of **explicit** layout control cannot be used with **static** fields.

Any offset may be specified. It is possible to overlap fields in this way, even though it is not recommended. The field may be accessed using pointer arithmetic and **ldind** to load the field indirectly or **stind** to store the field indirectly (see section 18.4).

The directive **.pack** specifies to put fields at multiples of the specified number or the natural word boundary of the native machine, whichever is smaller. E.g., **.pack 4** on a 64 bit or 32 bit machine has no effect, while on a 16 bit machine the variables are put at offsets divisible by 4. The integer following **.pack** must be one of the numbers 1, 2, 4, 8 or 16.

The directive **.size** specifies that a memory block of the specified amount of bytes shall be allocated for an instance of the class. E.g., **.size 32** would leave a blob of 32 bytes for the instance. This can be

used to store values in the blob by using pointers into the blob. However, this is only possible with unverifiable code.

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. Suppose the natural alignment of variables on this platform is 32 bit.

```
.class value sealed explicit public MyClass extends 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 blob of size 16:

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

7.8.2 Controlling Virtual Method Dispatch

In some cases when interoperation with unmanaged code is required, the exact address of a **virtual** method is important. In addition, the unmanaged code might not use the correct calling convention to invoke a managed **virtual** method.

Both of these problems are solved by the **.vtfixup** directive. This directive may appear several times only at the top level of an CIL assembly file, as shown by the following grammar:

<pre><decl> ::= .vtfixup <vtfixupDecl> ...</pre>	Section 4.7
--	---

The **.vtfixup** directive declares that at a certain memory location there is a table that contains MethodDefs which needs to be converted into method pointers. The CLI will do this conversion automatically.

The table does not need to be a contiguous block of memory. It may be separated into several chunks of memory. Each chunk is called an entry, and has an entry number associated with it. Each entry must capture a contiguous block of memory and is divided into slots. The slots will contain a pointer that points to the actual code.

The entries are numbered by the order of their declaration within a file. The syntax does not provide a way for explicit numbering, so a tool that uses this feature must keep track of the entry numbers.

Each entry occupies a certain size at a certain memory location. This memory location with the desired size must be reserved using the **.data** directive (see section 12.3).

The syntax for **.vtfixup** takes the desired number of slots for the virtual method table entry in brackets. Note that the number of slots is different from the size of the memory block required for **.data** directive and must be calculated. Following the number of slots are any number of attributes and the label at which the memory block was reserved. This is shown by the following grammar:

```
<vtfixupDecl> ::=
    [ <int32> ] <vtfixupAttr>* at <dataLabel>
```

The following grammar shows the attributes that can be used with the **.vtfixup** directive:

```
<vtfixupAttr> ::=
    fromunmanaged
    | int32
    | int64
```

The attributes **int32** and **int64** are mutually exclusive. **int32** is the default. These attributes specify the width of each slot. If **int32** is used, the slots are 32 bits wide, if **int64** is used the slots are 64 bits wide. If **int64** is used and the pointers on the target machine are only 32 bits wide, the high order bits of the slot will be filled with zeros and ignored.

If **fromunmanaged** is specified, the runtime will automatically 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.

An application needs to emit the desired method definition tokens with the **.data** directive. The CLI will convert these into method pointers and at runtime, the slots in the reserved memory will have pointers to the desired methods.

7.9 Global Fields and Methods

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

It is simplest to understand global fields and methods in the CLI by imagining that they are simply members of an invisible **abstract public** class. In fact, the CLI defines such a special class, called '**<Module>**', that does not have a superclass and does not implement any interfaces. The only noticeable difference is in how definitions of this special class are treated by the metadata merge code, the CLI class loader, and Reflection, all of which follow the same rules.

For an ordinary type, if the metadata merges two different 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 class is loaded. For the special class that holds global members, however, members are unioned across all compilation units at merge time. If the same name appears to be defined for cross-compilation-unit use in multiple compilation units 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 **privatescope**, then add them all in the output class.
- If there are duplicates and two or more have an accessibility other than **privatescope** an error is reported.

8 Interfaces

Interfaces define a contract that classes may implement. Similar to a class, an interface defines a reference type. Unlike classes however, interfaces cannot be instantiated. They are just constructs to specify type information.

Even though interfaces may have **static** fields and methods, they may not have **instance** fields or methods. However, interfaces typically define **abstract virtual** methods.

Classes may *implement* interfaces. A class that implements an interface promises to provide an implementation for each **abstract virtual** method declared in the interface. A class may propagate this promise to its subclasses without providing the method implementations, but must then be declared **abstract** and cannot be instantiated since it has unimplemented methods. If a class implements an interface, all classes that inherit from it also implement that interface. They may use the original implementation or override individual methods.

While a class must always extend another class, a class may implement any number of interfaces. Even though single inheritance is easy to understand and use, it has a problem describing objects which are combinations of various types. E.g., a seaplane has the properties of a swimming object and a flying object. However, single inheritance forces the implementation to choose one of them. Even though interfaces do not solve the problem, they provide a work around without introducing the complexities of multiple inheritance. A class may agree to provide implementations for a set of other methods. The class will not be able to inherit any code, but the type checker will be able to verify that a class follows a special contract. E.g. in the case of a seaplane, the class *Seaplane* may implement the interfaces *IAircraft* and *IWatercraft*. Some user who is only interested in watercraft may treat the seaplane as an *IWatercraft*, because *Seaplane* implements methods that a watercraft must have. The same applies to *IAircraft*.

The negative side of interfaces is that a class does not inherit any code and must provide its own implementation, although it may inherit an implementation from the topmost class.

Interfaces may be nested inside interfaces. Interfaces may also be nested inside classes and classes may be nested inside interfaces. This works fine, since the nested type is not associated with an instance of the outer type.

8.1 Implementing Interfaces

A class may implement an interface by adding the interface name after the optional **implements** keyword in the class declaration. Any number of interface names separated by comma may be listed after the **implements** keyword.

```
<classHead> ::=  
    <classAttr>* <id> [extends <typeReference>] [implements  
        <typeReference> [, <typeReference>]*]
```

A class that declares that it implements an interface but does not provide an implementation for all the **virtual** methods declared by the interface must be declared to be **abstract**.

Interfaces are referenced in the same way as class types.

Interfaces cannot extend other classes, but they may require a class that implements the interface to also implement a set of other interfaces. To do this, an interface declares that it implements one or more other interfaces. This is done in the same way as for classes using the **implements** keyword. Note that the word *implements* is a little bit misleading. It refers to the class that eventually implements the interface. An interface can not provide any implementation itself. The class implementing the interface does not need to declare that it also implements the interfaces required by the interface it implements, although it may do so. The difference is subtle. (see section 8.1.1)

Example:

The following class implements the interface *IStartStopEventSource* defined in the module *Counter.dll*.

```
.class private auto autochar StartStopButton extends  
    [System.WinForms]System.WinForms.Button implements [.module  
    Counter.dll]IStartStopEventSource {  
    // body of class  
}
```

8.1.1 Implementation Requirements

When a class declares that it **implements** an interface the EE follows a simple set of rules to determine which method definition will be used. A `MethodImpl` (see also section 8.1.2) can be used when the default behavior does not capture the programmer's intention.

The implementation of **virtual** methods may be provided by:

- directly specifying an implementation
- inheritance from its parent class
- use of an explicit `MethodImpl` (see section 8.1.2).

A method definition is said to be a *matching method definition* for a method declared by an interface, if it is a **virtual** method with the same calling convention, return type, and the same order of parameter types as the method declared by the interface.

The detailed rules for determining which method body implements a particular interface method are as follows:

Suppose an interface `I` requires the implementation of method `Foo()`, and class `A` implements `I`. Then,

1. if `A` provides a `MethodImpl` for `I::Foo()`, then the `MethodImpl` specifies the method body to use. If a `MethodImpl` is supplied but the body is not a matching method definition, a `System.TypeLoadException` is generated.
2. *otherwise*, if `A` itself provides a matching method definition for a **public** method named `Foo`, then use that method
3. *otherwise*, if the immediate parent of `A` implements the same interface and provides an implementation for `I::Foo()`, then use whatever implementation `A`'s parent provides,
4. *otherwise*, if any parent of `A` provides a matching method definition for a **public** method named `Foo`, then use the method from the closest parent, even if that parent does not implement the interface,
5. *otherwise*: leave the slot empty if class `A` is **abstract** or generate a `System.TypeLoadException` if class `A` is concrete.

8.1.2 MethodImpls

MethodImpls are a mechanism used to explicitly specify, for a given class, what method definition should be used to implement a **virtual** method.

A `MethodImpl` consists of two parts, both of which are `MethodRefs`. The first `MethodRef` specifies the method to be implemented and the second specifies the implementing method. Both methods must have a matching signature, which means that except for their names and declaring type their signatures must be equivalent.

`MethodImpls` can be used to specify the implementation of a method declared by an interface. This way of specifying a method implementation is especially appropriate if a class implements several interfaces and each declares a method with the same name. If `MethodImpls` are not used, both method declarations will be implemented by the same method. If this is not desired, `MethodImpls` can be used to provide methods with different names for the two interfaces.

In V1 of the CLI, the implementing `MethodRef` cannot refer to an inherited method implementation. It must refer to a method declaration inside the class that declares the `MethodRef`. However, the method may override a method of its superclass or may be overridden by subclasses. The `MethodDecl` inside the class may also be an **abstract** method which is implemented by a subclass.

In the *ilasm* syntax, `MethodImpls` are declared using the **.override** directive which is discussed in section 7.6.3.1. `MethodImpls` are also described in section 11.1.

8.2 Defining Interfaces

Interface definitions are very similar to class definitions (see 7.4). They are introduced with the **.class** directive but contain the **interface** attribute.

Compared to classes, interfaces are subject to various constraints:

- all methods must be either **virtual** or **static**
- all **virtual** methods must be **abstract** and **public**
- no **instance** fields are allowed
- interfaces must be **abstract** and cannot be instantiated
- interfaces may not inherit from a class

Interfaces may have **static** methods, which must have an implementation for them. This is used, for example, to provide a type initializer for the interface to initialize its **static** fields.

However, it is not *CLS compatible* for interfaces to have **static** methods other than a type initializer (named **.cctor** and marked with both **specialname** and **rtspecialname**).

Example:

```
.class interface public auto autochar CountDisplay {  
    .method public abstract virtual void SetCount(int32 count) {}  
    .method public abstract virtual int32 GetCount() {}  
}
```

The interface `CountDisplay` is defined with two members. The description of the attributes can be found in section 7.4.2.

9 Value Types

In contrast to reference types (section 7), value types are not accessed by using a reference, but are stored directly in the location occupied by the variable that declares the value type.

Typically, value types are used to describe the type of small data items. Often, value types will be used as the type of a local variable. Value types are useful for stack allocated data, since their creation is fast. However, they can also be useful as the type of fields in heap allocated data. Compared to reference types, value types are accessed faster since there is no additional indirection involved.

Value types are inappropriate to use if inheritance is desired, if changes to local data need to be visible across method boundaries, or if the size of the data is fairly large.

Typical value types are complex numbers or dates. A typical non-value type would be a `Window`.

Value types have the same set of members as reference types:

- fields
- methods
- properties
- events

Similar to reference types, all members may be **instance** or **static**. Value types may have **virtual** methods. These are used to override **virtual** methods defined in `System.ValueType` or `System.Object` or to provide implementations for methods declared by interfaces.

The CLI does not support subtyping of value types. Value types may not have any subtypes and must be declared **sealed**.

As the name implies, value types are passed by value in method calls, which means that a new copy of the value will be created for the called method.

A value type may be converted into a reference type by a process called *boxing*. In this representation it is said to be in its *boxed form*. A boxed value type may be converted back into its value type representation, the *unboxed form*, by a process called *unboxing*. More about boxing and unboxing can be found in section 9.5.

All boxed value types inherit from the base class `System.ValueType`, which is a subclass of `System.Object`. This relationship between value types and reference types creates a unified type system in which all types may be treated as subtypes of `System.Object`.

Similar to reference types, value types may be nested. Value types may have nested classes and interfaces and classes and interfaces may have nested value types. This does not cause any problems since a nested type is not associated with any instance of the outer type.

Value types may declare that they implement interfaces, but the implementation is only effective in their boxed form. Only the boxed form can be used as an instance of the interface type.

When a value type is defined, a corresponding type that describes the boxed form of the value type will be automatically created by the runtime. In version 1 of the CLI, this type cannot be directly represented in metadata, so `System.Object` or any interface implemented by the value type has to be used as the type of all boxed value types and only members of `System.Object` or a member of an implemented interface can be used with the boxed form of a value type.

Unboxed value types are not considered subtypes of another type and it is not valid to use the **isinst** instruction on unboxed value types. The **isinst** instruction may be used for boxed value types. E.g., the **isinst** instruction may be used with `System.ValueType` to check whether the top of the stack is a reference to a boxed value type.

Unboxed value types may not be assigned the value *null* and they may not be compared to *null*.

Value types support layout control in the same way as reference types do (see section 7.8). This is especially important when values are imported from native code.

9.1 Referencing Value Types

The unboxed form of a value type is referred to by using the **value class** keyword followed by a type reference. The type reference is resolved to a type definition token either at load time, or if possible at compile time. The following grammar defines value type references:

```
<valueTypeReference> ::=  
    value class <typeReference>
```

The boxed form of a value type is referenced using **class** `System.Object`. (becomes simply **object** in Beta-2)

Example:

The following example declares the variable `size` of type `System.Drawing.Size` which is a value type:

```
value class [System.Drawing]System.Drawing.Size size
```

9.2 Instantiating Value Types

A value itself is already an instance of a value type. Instances of value types are stored in local variables, fields, and method arguments. The value type is instantiated when memory is allocated for the local, field, or argument. In addition, instances of value types may be also be directly created on the stack (e.g. by using **ldc**).

Value types are initialized with the **initobj** instruction. The **initobj** instruction zeroes out all **instance** fields of a value type. The **static** fields are initialized when the value type is loaded.

The **initobj** instruction expects a managed pointer to an instance of the value type. **initobj** specifies the value type to be initialized as part of the instruction. The instruction does not return anything on the stack. If a value type is used without calling **initobj** first, the values of its instance fields may have any value.

Value types may have any number of initializers. Similar to instance constructors of classes, an initializer is named **.ctor** and has the attributes **rtspecialname** and **specialname**, which mark a special name for the runtime and for other compilers and tools, respectively. The initializer is not automatically called by the **initobj** instruction and should be explicitly called after the **initobj** instruction.

Verification requires that all fields of a value type be written before they are read or a pointer to an instance of the value type is passed to a method other than an initializer of the value type. Every initializer of the value type is required to store into every field of the value type (this applies recursively, when one value type is embedded as a field of another value type).

Tools are urged to insert calls to the initializer, but this is not required. Programmers who expect their value types to be used from another language must be prepared to handle the case where the state has been zeroed but no initializer has been called.

If a value type has an initializer, an instance of its unboxed type can be created as is done with classes. The **newobj** instruction is used along with the initializer and its parameters to allocate and initialize the instance. The **newobj** does not require any pointer to a variable of the value type. The instance of the value type will be allocated on the stack. Unlike the **initobj** instruction, the **newobj** does call the initializer. It takes the a reference to the initializer as part of the instruction.

The Base Class Library provides the method `System.Array.Initialize()` to zero out all instances in an array of unboxed value types.

Example:

The following code initializes the value type variable declared in the example in section 9.2:

```
ldloca      size      // load address of local variable
initobj     value class [System.Drawing]System.Drawing.Size
ldloca      size      // load address for initializer (argument 0)
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)
// instance of value type is initialized
```

9.3 Defining Value Types

Similar to class types, the definition of value types is introduced by the **.class** directive and follows the same rules as for class types (see section 7.3) with the difference that the **value** attribute must be used. (the value attribute will be removed for type definitions in Beta-2) In addition, all value types must explicitly inherit from `System.ValueType` defined in the DLL *mscorlib*. The only exception are enumerations, which are described in section 10.2.

Example:

The following example defines a value type to represent complex numbers. The value type provides an implementation of the `ToString` method that overrides the implementation given in `System.Object`.

```
.class value public sealed Complex {
    .field public int32 re    // the real part
    .field public int32 im    // the imaginary part

    // create string of the form 1 + 2i, where 1 is re and 2 is im
    .method virtual public hidebysig instance class System.String ToString()
CIL managed {
    .maxstack 4
    ldarg.0
    ldflld int32 Complex::re
    call class System.String System.Int32::ToString(int32)
    ldstr " + "
```

```
ldarg.0
ldfld int32 Complex::im
call class System.String System.Int32::ToString(int32)
ldstr "i"
// combine the pieces on the stack
call class System.String System.String::Concat(class
    System.String, class System.String)
call class System.String System.String::Concat(class
    System.String, class System.String)
call class System.String System.String::Concat(class
    System.String, class System.String)
ret          // return combined string on stack
}
}
```

9.4 Methods of Value Types

Value types may have **static**, **instance** and **virtual** methods. **static** methods of value types are defined and called the same way as **static** method of class types.

While **instance** methods of class types expect a reference to an instance of the class as the first argument, all **instance** methods of value types, boxed or unboxed, expect a managed pointer to an unboxed instance of the value type as the first argument to the method. The same applies for **virtual** methods. This argument corresponds to the *this* pointer for value types and can be used to access the fields and call **instance** or **virtual** methods of the value type.

Both **instance** and **virtual** methods of a boxed or unboxed value type are called using the **call** instruction. The **callvirt** instruction may not be used with unboxed value types.

However, since methods implemented for an interface can only be called using the **callvirt** instruction, all methods in a value type defined for an interface and thus declared **virtual** must be called using the **callvirt** instruction. This requires the value type to be in its boxed form, since **callvirt** is not allowed with its unboxed form. If the **callvirt** instruction is used, it is illegal to use the value type to refer to the method to be called. Only a superclass of the value class, e.g. `System.Object`, or an interface type may be used with the **callvirt** instruction as a type reference to specify the method to be called. Since value types do not have subclasses, a **callvirt** that would explicitly reference the value type would never make sense and is considered illegal.

There is a problem for **virtual** methods that override a **virtual** method of a superclass or that implement a method declared in an interface. Callers of the method that make the call using the superclass or the interface declaration do not know that the actual implementation is defined inside a value type. As a consequence, they will pass a reference to the boxed version of the value type as the first argument to the method. However, since the **virtual** method is in a value type, it expects a managed pointer to the unboxed instance of the value type as the first argument. This conflict is resolved by the EE. When a **virtual** method of a value type is called using the **callvirt** instruction, the caller is required to pass a reference to an instance of the boxed value type. Thus, from the callers point of view, there is no difference between calling a **virtual** method of a value type or class with the **callvirt** instruction. The EE will automatically unbox the *this* pointer reference and pass a managed pointer to the unboxed instance to the **virtual** method.

The following table summarizes the discussion in this section. It shows what the caller needs to push onto the stack for the first argument in order to call an **instance** or **virtual** method with the **call** instruction, or a **virtual** method with the **callvirt** instruction. The method of the value type will always expect an unmanaged pointer to an unboxed instance of the value type. The first column shows the kind of type reference the caller uses.

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

	Value Type (Boxed or Unboxed)	Interface	Class Type
call	managed pointer to value type	illegal	object reference
callvirt	illegal	object reference	object reference

Examples:

The following converts an integer of the value type **int32** into a string. 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 class System.String System.Int32::ToString()
```

However, if **System.Object** is used as the type reference rather than **System.Int32**, the code becomes:

```
ldloc boxed_x
callvirt instance class System.String System.Object::ToString()
```

9.5 Boxing and Unboxing

If data of a value type needs to be assigned to a memory location of type **System.Object** or an interface type, e.g. as part of a method call, it first needs to be converted into a reference type by boxing it.

An instance of an unboxed value type can be converted to an instance of the corresponding boxed type using the **box** instruction. The **box** instruction takes the address of an instance of a value type, allocates space on the heap, copies the instance fields of the value type to the allocated space and returns a reference to the location on the heap. Since the **box** instruction requires an address, the instance of the value type must be stored in a local variable, argument, or field before the **box** instruction can be used. This requirement follows from the restriction that it is not possible to obtain an address to a value on the stack. The boxed version of the value type behaves like any other reference type. However, since its exact type is not accessible, only members of **System.Object** are accessible in the boxed version. In order to access members specific to the value type, the boxed form needs to be unboxed first. The **unbox** instruction takes a type reference to the value type as part of the instruction. The following picture illustrates the **box** operation:

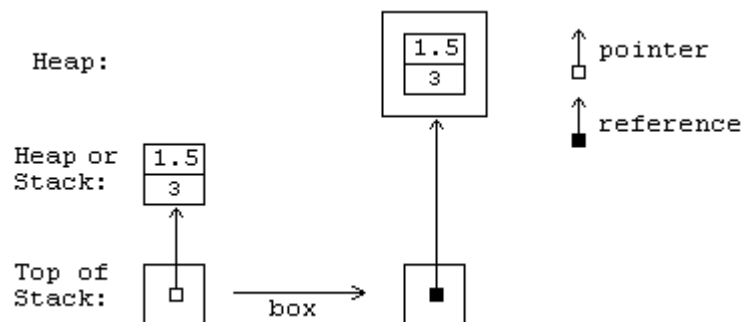


Figure 7: box

An instance of a boxed value type can be converted to an instance of the corresponding unboxed type using the instruction **unbox**. The **unbox** instruction takes a reference to the boxed version of a value type and returns a managed pointer to the actual data of the value type. The returned pointer will point to data of the actual value type, such that all members of the value type are accessible and can be modified. Notice that the **unbox** instruction does not do a copy. The result of **unbox** shares state with the original object. The **unbox** instruction takes a type reference to the value type as part of the instruction. The following picture illustrates the **unbox** operation:

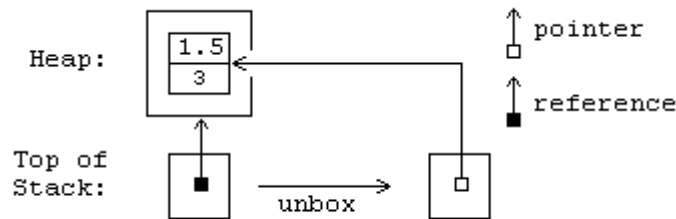


Figure 8: unbox

The result of **unbox** can be stored in a local variable using the **cpobj** instruction. **cpobj** copies a value type from a source to a destination, both of which must be declared to be value types. It expects the address of the source on top of the stack and the address of the destination as a second argument on the stack. Similar to **unbox** and **box**, the **cpobj** takes a type reference to the value type as part of the instruction.

There is one case in which the runtime will do the unboxing implicitly, which is discussed in section 9.4 and has to do with the **callvirt** instruction.

Both boxed and unboxed value types are marshal by value (not reference) and are not contextful. This means that value types, boxed or unboxed, are copied in remoting scenarios (across application domain boundaries) and thus do not have a strong notion of identity. Within a single application domain, but across contexts, boxed value types retain their identity.

For CLS compliance, the reference returned by the box instruction must be treated as a reference to an instance of `System.Object`. The reference needs to be explicitly cast to an interface type before assigning it to a variable that is declared to be of that interface type.

Examples:

The following code boxes the local `myInt` of type `int32` and stores it in the local `boxedInt` of type `System.Object`:

```
ldloca myInt           // load address of myInt
box int32
stloc boxedInt
```

The following unboxes the boxed version of `myInt` and copies it back to `myInt`:

```
ldloca myInt           // load address of destination
ldloc boxedInt         // load reference of boxed type
unbox int32            // address of unboxed type on stack
cpobj int32            // copy into myInt
```

9.6 Copy Constructors on Value Types

Copy constructors are necessary if certain behavior needs to be executed when an instance of a value type is copied from one memory location to another.

A value type can have a copy constructor, which is simply an initializer (i.e. it is named `.ctor` and has the attributes `specialname` and `rtspecialname`). The copy constructor will receive the pointer as a argument. Typically, copy constructors do not declare other parameters. Further, a value type with a copy constructor should have the attribute `not_in_gc_heap` associated with it.

The copy constructor is not called automatically by the runtime, and is not supported on boxed instances, i.e. the garbage collector will not call it.

Since the garbage collector only manages objects on the GC heap, the `not_in_gc_heap` attribute will prevent any problems associated with movement of the value type. However, it has also some restrictions associated with it. Instances of value types marked with `not_in_gc_heap`

- may not be assigned to fields of classes, fields of value types that aren't also marked **not_in_gc_heap**, or static variables
- may not be boxed

since both of the above would move the instance to the GC heap.

For classes that have a copy constructor, the compiler must insert code to do the copy construction as appropriate. The x86 managed calling convention is that the caller makes the copy and then passes the address of that copy to the callee. This is unlike C++ where methods that take a copy-constructed parameter appear to be called by value.

In order to interoperate with unmanaged code that passes the copy constructed values on the stack, PInvoke (platform invoke) may call the copy constructor an additional time to construct a copy on the unmanaged stack. This behavior is triggered (if needed) by a particular custom modifier applied to the parameter which must be copy constructed. This modifier, *Microsoft.VisualBasic.NeedsCopyConstructorModifier*, may only be placed in front of a parameter whose type is a managed pointer to a value type.

For CLS compliancy, copy constructors may not be used.

Example:

The following defines a copy constructor for the value type `Complex` given in example 9.7.2:

```
.class value not_in_gc_heap public sealed Complex {  
    .method public hidebysig rtspecialname specialname instance void  
    .ctor() CIL managed {  
        // body of copy constructor  
    }  
}
```

9.7 Using Value Types for C++ Classes

Where possible, languages that provide features in their object system not directly supported by the CLI should use value types with visible field members to represent their classes. The following rules, designed for compiling C++ to CIL, expose as many features of the underlying object model to other languages as is possible with the CTS.

For C++, the trick is that managed languages will import C++ classes as value types and will see C++ virtual and instance methods on those types as static methods with an explicit unmanaged *this* pointer. The C++ compiler is fully responsible for handling multiple inheritance, so other languages can use methods that are defined in C++ using multiple inheritance even though they could not define such methods themselves. In addition, because value types are **sealed**, it is not possible to extend frameworks created in C++ in another language.

Similar rules can be devised for many other languages.

9.7.1 Representation of a Class as a Value Type

First, a custom attribute needs to be defined, possibly in a reserved part of the System namespace, to indicate that a value type is a CTS representation of a language-specific class. The custom attribute may have fields that provide more information of use to a browser that understands the language-specific semantics. All value types that represent unmanaged classes need to be marked with instances of this custom attribute.

Then the object layout has to be described in form of a value type, possibly with explicit layout control. Static methods, static fields, and instance fields can be defined on the value class directly. There will be necessarily a field for a pointer to the VTable and all constructors for the class must store the appropriate value (see below) into this field.

Instance and virtual methods have to be converted to static methods with one more parameter than the original C++ method had. This additional parameter is the first parameter, functions as the *this* parameter, and its type is an unmanaged pointer to the value type. These methods should have a custom

attribute attached so that browsers (and the C++ compiler itself) can distinguish *true* static methods from these introduced static methods.

The method body for instance and virtual methods has access to the pointer to the original object through its first parameter, through which it can access the VTable, and via the VTable the function pointers to the virtual methods. The compiler can use explicit address arithmetic to adjust the address of the object or VTable to handle multiple inheritance, etc. The **calli** instruction is used to call through the function pointers in the VTable.

Classes that have user supplied copy constructors or destructors cannot be boxed, cannot be fields of managed types, and cannot be passed by value directly. For this reason the **not_in_gc_heap** attribute should be used to prevent the instance of the value type to be moved to GC heap. C++ classes that do not have a copy constructor are usable by other languages just as any other value type would be and should not set this bit. Copy constructors are also further described in section 9.6.

9.7.2 Representation of the VTable

The VTable itself is represented as the value of a static variable with specified RVA (relative virtual address). The type of this variable is a class with a unique mangled name (or a type nested within the object's type), explicit layout, and with named fields that are function pointer types to native method implementations.

The data for the VTable is stored in the data section of the PE file and the appropriate fixups must be stored in the CLI header within the PE file. If unmanaged compatibility is required the entries may be forced to be 32 bits wide and the PE file header marked for 32-bit architectures only. Otherwise, the entries should be 64 bits wide. See also the description of **vtfixup** in section 7.8.2.

10 Special Types

10.1 Arrays

An array is a contiguous memory block that stores an indexed collection of values of the same type. The CLI has partially built-in support for arrays, including multidimensional arrays.

Arrays that have only one dimension and are zero based (i.e. the first element is at index zero) are called *vectors*. Even though the runtime has support for multidimensional arrays and arrays with a specified lower and upper bound, the execution engine has only built in instructions for dealing with vectors. The .NET Framework provides services to create arrays which are not vectors.

The following sections discuss vectors, general arrays, and arrays of arrays.

CLS Note: CLS-compliant tools are only required to support arrays whose elements are of types supported by the CLS and which have zero lower bounds for all dimensions. For a CLS consumer there is no need to accept arrays of other types. For a CLS extender there is no need to provide syntax to define other types of arrays or to extend interfaces or classes that use other array types. For a CLS framework other array types may not appear in exposed members.

10.1.1 Vectors

10.1.1.1 Declaring Vectors

Vectors are declared by providing the type of the elements followed by brackets.

<code><type> ::=</code>	Section
<code><type> []</code>	
<code> ...</code>	5.3

The type of the elements may be any subtype of `System.Object` (including `System.Object` itself) and any boxed or unboxed value type. This includes Delegates, and function pointers (see example below). Pointer types are not allowed as an element type in the first version of the CLI.

The vector itself is an instance of a special array type that defines the vector. If necessary, the VES (Virtual Execution System, see [CTS Spec](#)) creates this special array type automatically.

The special array type is an object type. It is always a subtype of `System.Array`, which is a subtype of `System.Object`. The operations on an array type are defined by the CTS. These include indexing the array to read and write a value and computing the address, obtained in form of a managed pointer, of an element of the array. The `System.Array` type supports additional operations which are discussed in section 10.1.1.4.

Example:

A vector of Strings:

```
class System.String[] errorStrings
(becomes: string[] errorStrings in Beta-2 if a signature)
```

Example:

A vector of function pointers:

```
.field method instance void*(int32) [0..5] myVec
```

10.1.1.2 Creating Vectors

Vectors are created using the **newarr** instruction. Note that the **newarr** instruction can only be used to create vectors and not other types of arrays. The **newarr** instruction expects the type of the elements as part of the instruction and an unsigned 32 bit integer specifying the size of the vector on the stack. The instruction returns a reference to the array on the stack.

Example:

```
ldc.i4.4
newarr    class System.String
stfld     class System.String[] CountdownForm::errorStrings
( in Beta-2, becomes:
ldc.i4.4
newarr    class System.String // no change!
stfld     string[] CountdownForm::errorStrings // change! )
```

The instructions above create a zero based array with size 4. The maximum index in the array is 3. The next instruction stores this array in a dedicated memory location.

10.1.1.3 Using Vectors

The most common operations on arrays is to index them to retrieve an element or to set the value of an element. The CLI has special instructions that implement these operations for vectors.

The instruction **ldelem** is used to load an element of the vector onto the stack. **ldelem** expects the type of the elements as a postfix and a reference to the vector and an unsigned 32 bit integer specifying the index of the element to load on the stack. The instruction returns the element on the stack. The complete list of **ldelem** instructions can be found in section 18.

The instruction **stelem** is used to store a value in the vector. Similar to **ldelem**, **stelem** expects the type as a postfix and a reference to the vector, the index, and value to store on the stack. The complete list of **stelem** instructions can be found in section 18.

The instruction **ldelem** has a corresponding instruction **ldelema** that loads the address of an element rather than the element itself.

There is a special case for value types. To access a value type in an array, the address of the array element needs to be loaded via **ldelema** and then the value type can be loaded using **ldobj**. To store a value type, first the address of the element is loaded with **ldelema** and then **stobj** is used. More about these instructions can be found in section 18.5.

All three instructions throw a `System.NullReferenceException` if the array on the stack is *null*. The instructions throw a `System.IndexOutOfRangeException` if the index is greater or equal to the size of the vector. An `System.ArrayTypeMismatchException` is thrown if the

array does not hold elements of the required type. Note that the index value is treated as an unsigned value.

The instruction **ldlen** may be used to obtain the length of a vector. It takes the array on the stack and returns the length as a 32 bit integer on the stack. The instruction throws a `System.NullReferenceException` if the array on the stack is *null*.

Examples:

```
ldfld    class System.String[] CountdownForm::errorStrings
ldloc    errorCode
ldelem.ref
```

The instructions above load a vector, an index and use **ldelem** to retrieve the element. The postfix **.ref** is used since strings are reference types.

```
ldfld    class System.String[] CountdownForm::errorStrings
ldc.i4.1
ldstr    "Number must be positive!"
stelem.ref
```

The code fragment above loads a vector, the index of the second element, a string, and stores the string in the vector using **stelem.ref**.

10.1.1.4 Methods of Vectors

Array types form a hierarchy, with all array types inheriting from the type `System.Array`. This is an abstract class in *mscorlib.dll* that represents all arrays.

The class `System.Array` defines a number of methods that can be used to apply operation to the array. E.g., the class defines methods to retrieve the length of the array, copy the array, reverse the array, or sort the elements of the array among other operations.

More information can be found in the Base Class Library documentation.

Example:

```
ldloc    myVector
call     instance int32 [mscorlib]System.Array::get_Length()
call     void [mscorlib]System.Console::WriteLine(int32)
```

The code above loads a reference to a vector and calls the method `get_Length()` to obtain the length of the array.

10.1.2 General Arrays

There are no CIL instructions to handle arrays which are not vectors, but they are still supported by the runtime. In contrast to vectors, general arrays may have any bounds. The following sections discuss how general arrays are declared, created, and used.

10.1.2.1 Declaring General Arrays

The *rank* of an array is defined to be the number of dimensions of the array. The CLI does not support arrays with rank 0.

General arrays are defined in a way similar to vectors. However, they must declare their rank and optionally may restrict their bounds.

<code><type> ::=</code>	Section
<code><type> [[<bound> [,<bound>]*]]</code>	
...	5.3

The rank of the array is declared by using commas (",") between the brackets. The number of commas plus one is equal to the rank of the array. For example, no commas means that the rank

equals to one, one comma means that the rank equals to two, five commas means the rank equals to six, etc.

Array declarations have the following choices to declare the bounds for each dimension:

- No bounds declaration, lower bound is assumed to be zero.
- Declaration of upper bound only, lower bound is assumed to be zero.
- Declaration of lower bound only
- Declaration of both lower and upper bounds.

<bound> ::=	Description
...	lower and upper bound are unspecified
<int32>	zero lower bound, <int32> upper bound
<int32> ...	lower bound only specified
<int32> ... <int32>	both bounds specified

While vectors have a type based on the type of the elements in the array, regardless of the upper bound, arrays with more than one dimension or one dimension but with non-zero lower bound have the same type if they have the same element type and rank, regardless of lower and upper bound of the array. This array type is created on the fly by the execution engine as required.

Note that ILASM allows you to declare array variables with this rich syntax, providing some lower bounds, missing out others, etc. However, in Version 1, CLI largely ignores this information – in effect, all it pays attention to is the *rank* of that array variable. Just to be clear – when you *define* an instance of an array (using the **newobj** instruction, detailed later), you must supply the actual upper and lower bounds for each dimension – you cannot miss any out. Thereafter, when you bind this new object to your array variable, the CLI ignores all the bounds information you gave to that variable – so long as its rank is the same (and the array <type> is assignment-compatible of course), the bind will be accepted.

The following table shows examples of array declarations – note carefully that only the first results in a *vector* – all the others, despite what you might have guessed, result in a general *array*

Array Declaration	Array Type
int32[]	vector of int32
int32[0 . . . 5]	array of int32, rank 1
int32[. . .]	array of int32, rank 1
int32[0 . . .]	array of int32, rank 1
int32[5]	array of int32, rank 1
int32[,]	array of int32, rank 2
int32[0 . . . 3 , . . .]	array of int32, rank 2
int32[1 . . . , 0 . . .]	array of int32, rank 2

Only arrays which have zero lower bounds in all their dimensions are CLS compliant. [*vectors* are therefore, of course, all CLS compliant]

10.1.2.2 Creating General Arrays

The **newarr** instruction can not be used to create arrays other than vectors. However, when an array is declared the execution engine will create a type for the array which can be used to construct the array.

The declared array type may be used to access the members of the array type. The execution engine defines two constructors for each array type. One of them takes the same number of unsigned 32 bit integers as arguments as the rank of the array. Each of the arguments specifies the number of elements for each dimension of the array, starting with the first dimension.

The other constructor takes twice as many arguments as the rank of the array. The first argument is the lower bound for the first dimension and the second argument is the length of the first dimension. The third argument is the lower bound of the second dimension and the fourth is the length of the second dimension, and so on. Every even numbered argument (starting at zero) specifies the lower bound, while every odd numbered argument specified the length for a certain dimension.

While the second constructor declares the lower bounds, the first constructor sets the lower bounds automatically to zero.

Note: The declaration of the bounds is for documentation purposes only and will not affect the runtime or verification. Only the choice of the constructor determines if the array has lower bounds zero or some specified lower bounds.

The following table shows some constructors:

Array Declaration	Constructors
int32[]	void int32[]::ctor(int32)
	void int32[]::ctor(int32,int32)
float32[1...5]	void float32[1...5]::ctor(int32)
	void float32[1...5]::ctor(int32, int32)
class System.String[,]	void class System.String[,]::ctor(int32, int32)
	void class System.String[,]::ctor(int32, int32, int32, int32)
int32[0...,1...5,...]	void int32[0...,1...,...]::ctor(int32, int32, int32)
	void int32[0...,1...,...]::ctor(int32, int32, int32, int32, int32, int32)

Example:

Assume the following array is declared:

```
int32[5...10,3...7] myArray
```

The following code creates an instance of the array. Note that the length for each dimension has to be computed according to the formula upper bound – lower bound + 1. This formula assumes that the bounds are inclusive.

```
ldc.i4.5      // load lower bound, dim 1
ldc.i4.6      // load upper bound - lower bound + 1, dim 1
ldc.i4.3      // load lower bound, dim 2
ldc.i4.5      // load upper bound - lower bound + 1, dim 2
newobj instance void int32[5...10,3...7]::ctor(int32, int32)
stloc myArray
```

10.1.2.3 Using General Arrays

In addition to the constructor, the execution engine defines the instance methods `Get` and `Set` for the declared array type. These methods take the same number of arguments as the rank of the array, each of them a 32-bit integer specifying the index of the element for each dimension starting with the first dimension. The method `Set` takes an additional argument that specifies the value to store and has the same type as the type of the elements.

The following tables shows some examples of the `Get` and `Set` methods:

Array Declaration	Get Method
<code>int32[]</code>	<code>void int32[]::Get(int32)</code>
<code>float32[1...5]</code>	<code>void float32[1...5]::Get(int32)</code>
<code>class System.String[,]</code>	<code>void class System.String[,]::Get(int32, int32)</code>
<code>int32[0...,1...5,...]</code>	<code>void int32[0...,1...,...]::Get(int32, int32, int32)</code>

Array Declaration	Set Method
<code>int32[]</code>	<code>void int32[]::Set(int32, int32)</code>
<code>float32[1...5]</code>	<code>void float32[1...5]::Set(int32, float32)</code>
<code>class System.String[,]</code>	<code>void class System.String[,]::Set(int32, int32, class System.String)</code>
<code>int32[0...,1...5,...]</code>	<code>void int32[0...,1...,...]::Set(int32, int32, int32)</code>

The index must be within the range specified at construction time.

The next section will introduce more methods that are defined for arrays.

Examples:

Assume the following array is declared:

```
int32[5...10,3...7] myArray
```

The following example stores the value 25 at position [6,7], i.e. `myArray[6,7] = 25`.

```
ldlocmyArray    // load instance to array
ldc.i4 10       // desired index for dim 1
ldc.i4 7        // desired index for dim 2
ldc.i4 25       // value to be stored in element
int32[5...10,3...7]::Set(int32,int32,int32)
```

The following example retrieves the value at position [6,7], i.e. `myArray[6,7]`.

```
ldloc myArray    // reference to an instance of the array on stack
ldc.i4 6         // desired index for dim 1
ldc.i4 7         // desired index for dim 2
call instance int32 int32[5...10,3...7]::Get(int32,int32)
// The value retrieved from the
array is on stack
```

10.1.2.4 Methods of General Arrays

Similar to vectors, all general arrays are subtypes of `System.Array` (see also 10.1.1.4). All members of `System.Array` are inherited to the specific array types created by the execution

engine on the fly. These members can be used to obtain further information on the array, e.g. the size of a specific dimension, or do certain operations on the array, e.g. sort the elements.

More information on these members can be found in the Base Class Library documentation.

10.1.3 Arrays of Arrays

Arrays of arrays are different then multi dimensional arrays. While multi dimensional arrays form one memory block, arrays of arrays are arrays that reference other arrays.

The following graphs illustrate this:

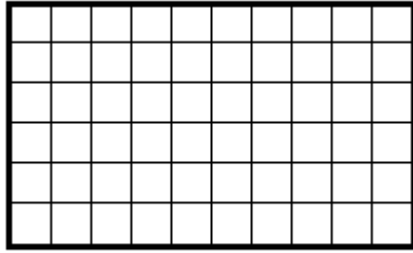


Figure 10.1: A 2 Dimensional array

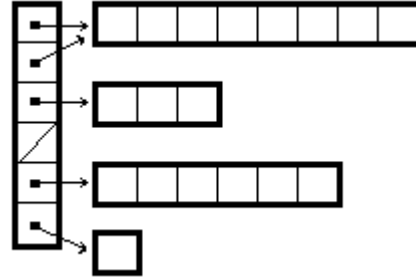


Figure 10.2: An array of arrays

In figure 10.2, not one but 5 arrays were created. The vertical array is an array of arrays and references the horizontal arrays. As can be seen in the figure, several references may exist to the same array. In figure 10.1, there is only one memory block. Verifiable code cannot reference a sub-array inside the array.

While in the case of arrays of arrays an additional indirection is needed to reach the final element, the location of the element can be calculated directly in multi dimensional arrays. This has performance advantages, especially for large dimensions. Further, multi dimensional arrays make pointer arithmetic possible, since the range of memory that contains all of the elements is well known.

On the other hand, all dimensions of a multi dimensional array must be of the same size. In the case of arrays of arrays, it is possible to reference arrays with different sizes.

The syntax for arrays of arrays easily follows from the previous sections. The only difference is that the elements are arrays themselves. The following sections give a summary and some examples.

The execution engine creates a special type for arrays of arrays on the fly, in the same way it is done for regular arrays.

10.1.3.1 Declaring Arrays of Arrays

The syntax for the declaration of an array of arrays follows recursively from the syntax for general arrays. An array of arrays is declared by using an additional pair of brackets. This may be repeated to declare an array of arrays of arrays, etc.

Example:

```
int32[][][] arrayOfArraysOfIntArray
```

In the example above, an array is declared that has elements of type array, which also have elements of type array, which have element type int32.

10.1.3.2 Creating Arrays of Arrays

Arrays of arrays can be treated as vectors as appropriate or as general arrays. The syntax for creating arrays of arrays follows from the discussion of general arrays in section 10.1.2.2 and 10.1.2.3.

The following example illustrates this using the syntax for general arrays.

Example:

Suppose the following array is declared:

```
int32[][][] myArray
```

Then the following code creates my Array, sets the first element to be an array of array of integers, and the second element of the latter array to be an array of integers.

```
ldc.i4.5          // Size of myArray
newobj instance void int32[][][]::ctor(int32)
stloc myArray      // myArray is created
ldloc myArray      // start creating an int32[][] to store in myArray
ldc.i4.0           // index to store int32[][]
ldc.i4.3           // size of int32[][]
newobj instance void int32[][]::ctor(int32)
// int32[][] created, now store it in the int32[][][]
call instance void int32[][][]::Set(int32, int32[][])
ldloc myArray      // start creating an int32[] to store in int32[][]
ldc.i4.0           // retrieve reference to int32[][]
call instance int32[][] int32[][]::Get(int32)
ldc.i4.1           // index to store int32[]
ldc.i4 10          // size of int32[]
newobj instance void int32[]::ctor(int32)
// int32[] created, now store it in the int32[][]
call instance void int32[][]::Set(int32, int32[])
```

10.1.3.3 Using Arrays of Arrays

When using arrays of arrays, they may be treated like vectors or general arrays, whichever is more appropriate. However, they should be treated in a consistent way. Arrays of arrays inherit from the class `System.Array` and may use all of its members to do the operations defined by that class.

The following example uses the syntax of general arrays to store and retrieve an integer in the array created in the example in section 10.1.3.2.

Example:

The following represents the operation `myArray[0][1][5] = 100`

```
ldloc myArray // load reference to myArray
ldc.i4.0      // retrieve myArray[0]
call instance int32[][] int32[][]::Get(int32)
ldc.i4.1      // retrieve myArray[0][1]
call instance int32[] int32[][]::Get(int32)
ldc.i4.5      // store the value at myArray[0][1][5]
ldc.i4 100
call instance void int32[]::Set(int32, int32)
```

The following represents the expression `myArray[0][1][5]`

```
ldloc myArray // load reference to myArray
ldc.i4.0      // retrieve myArray[0]
call instance int32[][] int32[][]::Get(int32)
ldc.i4.1      // retrieve myArray[0][1]
call instance int32[] int32[][]::Get(int32)
ldc.i4.5      // retrieve myArray[0][1][5]
call instance int32 int32[]::Get(int32)
```

10.2 Enumerations

An *enumeration* defines a set of symbols that all have the same type. A variable declared to be of the enumeration type should only be assigned a symbol that belongs to the set of symbols of the enumeration.

E.g., an enumeration days of week would contain the names of the days as symbols, i.e. Monday, Tuesday, etc.

All enumerations inherit from the class `System.Enum` and must explicitly indicate this in their definition. `System.Enum` inherits from `System.ValueType`. As a consequence, enumerations are value types and all rules that apply for value types (see section 9) also apply for enumerations. E.g., enumerations have boxed and unboxed forms and may not have any subclasses.

The symbols of an enumeration are represented by an integral type, like `int32`. Any of the integer types can be selected as the *underlying representation* of the enumeration. The first version of the CLI does not support an underlying representation for enumerations other than integers.

The CLI does not provide a guarantee that values of the enumeration type are integers corresponding to one of the symbols (unlike Pascal).

Enumerations are subject to a series of restrictions in addition to the restrictions imposed by the fact that they are value types:

- Enumerations may not contain any members other than fields, not even constructors or type initializers
- As a consequence of the above, enumerations may not implement any interfaces
- Enumeration may not have **sequential** or **explicit** field layout, they must have **auto** field layout (see 7.4.2.2)
- All enumerations must have an instance field of the underlying type as described below
- All fields of an enumerations, except the value field, must be **static** and **literal**
- Enumerations may not be initialized with the `initobj` instruction, they need to be explicitly assigned a value.

All enumerations must declare a single instance field. This field must be of the underlying type of the enumeration. In addition the field must be marked with `rtspecialname` and `specialname`. To be useful, the field should be accessible by all users of the enum and typically is **public**.

CLS note: The field must be named `value__` and all fields must be **public**.

The instance field stores the current value of an instance of the enumeration. This field is never directly accessed. The runtime will automatically take care of setting or retrieving this value. To set or retrieve the value of an enumeration, the instance of the enumeration is used and not the field.

The **static**, **literal** fields of an enumeration declare the mapping of the symbols of the enumeration to the underlying integral type. Recall that the runtime does not allocate any memory for **literal** fields. **literal** fields are only metadata information and can only be queried by using reflection. All of these fields should have the type of the enumeration and should all have a field init metadata part that assigns a value of the underlying type to them (see section 12.2 for a description of field init metadata).

When used in a signature, enumerations must be marked as **value class**, since they are value types. For binding purposes enums are distinct from their underlying type. This is used, for example, during the mapping from a *methodref* to its corresponding *methoddef*.

For verification purposes and all uses within the execution engine, an unboxed enum automatically coerces to and from its underlying type.

Enums can be boxed to a corresponding boxed instance type. This type is not the same as the boxed type of the underlying type, so that the enum remains type distinct.

Because the unboxed form can be coerced to its underlying type and from there to any other enum with the same underlying type, an enum can be boxed to any corresponding type. The choice is dictated by the type used in the box instruction. Similarly, a boxed enum can be unboxed to the enum, its underlying type, or any enum that has the same underlying type.

Examples:

The following is a declaration of an enumeration:

```
.class value sealed serializable auto autochar public ErrorCodes extends
[mscorlib]System.Enum {
    .field public specialname rtspecialname unsigned int8 value__
    .field public static literal value class ErrorCodes no_error = int8(0)
    .field public static literal value class ErrorCodes format_error = int8(1)
    .field public static literal value class ErrorCodes overflow_error =
int8(2)
    .field public static literal value class ErrorCodes nonpositive_error =
int8(3)
}
```

The enumeration was declared **serializable** and **autochar**, but it does not need to be. However, it must be declared **sealed**, since it is a value type, and **auto**. The underlying type of the enumeration is an **unsigned int8**. Thus, it declares an instance field of that type named `value__`. Then, the symbols of the enumeration are listed as **static, literal** fields with the corresponding mapping to the underlying type.

The following is a declaration of a variable named `errorCode` of the enumeration shown above:

```
value class ErrorCodes errorCode
```

The code below stores a value in the enumeration:

```
ldc.i4.1                // store the value 1, (= format_error in metadata)
stloc errorCode
```

The code below loads the value of the enumeration

```
ldarg errorCode
```

10.3 Pointer Types

A *pointer* contains the address of a memory location. Usually, this memory location contains a data item of a specific type. It follows that pointers need to be typed, too, in order to guarantee type safety. The CLI allows the use of generic pointers only in unverifiable code.

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. A *location signature* contains the type of the data item and has special syntax that marks it as a pointer type.

While pointer types are reference types, values of a pointer type are not objects, 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:

- loading the value from the location referenced by the pointer
- storing an assignment compatible value into the location referenced by the pointer

The CTS also provides three type unsafe operations on pointer types (byte-based address arithmetic):

- adding integers to pointers
- subtracting integers from pointers
- subtracting one pointer from another

The results of the first two operations are pointers to the same type signature as the original pointer. See the [CIL Instruction Set](#) specification for details.

CLS note: Unmanaged pointer types are not part of the CLS. For CLS consumers there is no need to support pointer types. For CLS extender there is no need to provide syntax to define or access unmanaged pointer types. In CLS framework, pointer types must not be externally exposed.

The syntax for declaring a pointer's unmanaged type is as follows:

<type> ::=	Section
<type> &	10.3.3
<type> *	10.3.4
...	5.3

The **&** indicates a *managed pointer*, while the ***** indicates an *unmanaged pointer*.

For pointers into the same array or object (see the [EE Architecture Specification](#)), the following arithmetic operations are defined:

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

None of these operations is allowed in verifiable code.

It is important to understand the impact on the garbage collector of using arithmetic on the different kinds of pointers. Since unmanaged pointers 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. Similarly, transient pointers are not reported to the garbage collector and arithmetic can be performed without impact on garbage collection. (see section 10.3.4)

Managed pointers, however, are reported to the garbage collector. As part of garbage collection both the contents of the location to which they point *and* the pointer itself can be modified. The garbage collector will ignore managed pointers if they point into memory that is not under its control (the evaluation stack, the call stack, static memory, or memory under the control of another allocator). If, however, a managed pointer refers to memory controlled by the garbage collector it *must* point to either a field of an object, an element of an array, or the address of the element just past the end of an array. If address arithmetic is used to create a managed pointer that refers to any other location (an object header or a gap in the allocated memory) the garbage collector's operation is unspecified.

Pointers are compatible with the **int64** and, on 32-bit architectures, with the **int32** type. They are best considered as **unsigned int**, whose sites vary based on architecture.

10.3.1 Obtaining and Using an Address

Most of the load instructions have a corresponding instruction that has the same name as the load instruction but ends with an "a" and loads the address of the data item which the load instruction would load. In particular these instructions are:

Instruction	Description
ldarga	Load address of argument
ldelema	Load address of array element
ldflda	Load address of field
ldloca	Load address of local variable
ldsfla	Load address of static field

Once a pointer is loaded onto the stack, the instruction **ldind** may be used to load data indirectly. **ldind** takes an address on the stack and return the value located at the address. **ldind** also takes the type of the instruction as a postfix. **Ldind** is never verifiable.

Note that the runtime may throw an **InvalidOperationException** for a **ldflda** instruction if the *obj* is not within the current application domain. (An example would be where the object derives from `System.MarshalByRefObject`, where its field values are retrieved by a proxy from the actual target object)

More detail about these instructions can be found in section 18.

Example:

```
ldloc pString      // load pointer to string reference
ldind.ref
// reference to string is on stack
```

10.3.2 Unmanaged Pointers

Unmanaged pointers are the traditional pointers used in languages like C and C++. There are no restrictions on their use, although for the most part they result in code that cannot be verified. While it is perfectly legal to mark locations that contain unmanaged pointers as though they were unsigned integers (and this is, in fact, how they are treated by the EE), 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.
- It is best to think of unmanaged pointers as unsigned.
- Verifiable code cannot use unmanaged pointers to reference memory (i.e. it treats them as integers, not 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:
 1. The unmanaged pointer refers to memory that is not in memory managed by the garbage collector
 2. The unmanaged pointer refers to a field within an object
 3. The unmanaged pointer refers to an element within an array
 4. The unmanaged pointer refers to the location where the element following the last element in an array would be located

Example:

```
int32* pInt
```

10.3.3 Managed Pointers

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

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

- Managed pointers can be passed as arguments, stored in local variables, and returned as values.
- If 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 EE. This conversion is only safe if one of the following is known to be true:
 1. The managed pointer does not point into the garbage collector's memory area
 2. The memory referred to has been pinned for the entire time that the unmanaged pointer is in use
 3. A garbage collection cannot occur while the unmanaged pointer is in use, or the managed pointer refers to a pinned local variable

Example:

```
int32& pInt
```

10.3.4 Transient Pointers

Transient pointers are intermediate between managed and unmanaged pointers. They are created within the EE by certain CIL instructions, but users cannot declare locations of this type. When a transient pointer is passed as an argument, returned as a value, or stored into a user-visible location it is converted either to a managed pointer or an unmanaged pointer depending on the type specified for the destination.

- The CIL instructions that create transient pointers (**ldloca**, **ldarga**, **ldsflda** when the type of the field is not an object) are guaranteed to produce pointers to data that is not in managed memory.
- Transient pointers need not be reported to the garbage collector, and they are automatically converted to managed or unmanaged pointers when necessary (during method calls or when stored into a local or argument that requires a managed pointer).
- Transient pointers can exist only on the evaluation stack within a single method.
- Verification treats transient pointers as managed pointers.

Example:

Assume that the local variable `myLoc` is declared

```
ldloca myLoc
// transient pointer is on stack
```

10.4 Method Pointer Types

The CLI supports method pointers. A *method pointer* has a type, which is the signature of the method including its calling convention. Unlike other pointers, a method pointer points to the beginning of a method and not to a data item.

The following grammar shows the syntax for a method pointer type.

<type> ::=	Section
method <callConv> <type> * (<parameters>)	
...	5.3

Variables that have the type of the method pointer may store the address of the entry point to a method with compatible signature, which may be used to call the method.

A pointer to a **static** or (non-virtual) **instance** method is obtained with the **ldftn** instruction (see section 18.4). A pointer to a **virtual** method may be obtained by using the **ldvirtftn** instruction (see

section 18.4). Both instructions take the method definition token as part of the instruction and return a method pointer on the stack. In addition, the **ldvirtftn** instruction expects a reference to an instance of the class that defines the method, or one of its subclasses. In contrast to **ldftn**, **ldvirtftn** will return a pointer to an overridden version of the method if applicable.

A method may be called by using a method pointer with the **calli** instruction (see section 18.3.4). The **calli** instruction may be used with all method pointers independent of whether they were loaded with **ldftn** or **ldvirtftn**. The **calli** instruction expects all arguments to the method on the stack as defined by the method signature, including any instance references for **instance** and **virtual** methods. The first argument needs to be pushed first onto the stack. **calli** also expects the method pointer on the stack. The method pointer must be pushed last, i.e. it must be on top of the stack when **calli** is executed. If the method has a return value, it will be on top of the stack after the execution of the **calli** instruction.

Like other pointers, method pointers are compatible with the **int64** and, on 32-bit architectures, with the **int32** type. The preferred usage, however, is **unsigned int**.

Example:

```
ldarg.0          // load this pointer, method is defined by this class
ldvirtftn instance void StartStopButton::onClick(class System.Object, class
[mscorlib]System.EventArgs)
// Method pointer is on stack
```

The code above load a pointer to a virtual method defined in the same class. If a subclass overrides the method, the overridden version will be loaded.

10.5 Delegates

A Delegate is perhaps best thought of as a *self-describing* function pointer. Recall that in unmanaged code, a function pointer is just a *naked* address. To call the target function, your assembler code must push the appropriate arguments, and then branch to that target address. But there's no way to check that where you *land* expects the particular argument types you just pushed. All you provided was that *naked* address – you did not supply a *signature* that describes the function whose entry point lies at that address. In the managed world, such a call is inherently *unverifiable*.

A Delegate, on the other hand, associates a signature with a target address. Therefore, we can invoke the target method, and be sure we 'land safely'.

Before launching into the details of Delegates in ILASM, here's a simple example, written in a high-level managed language (C#) that should clarify the situation:

```
class Foo {
    private int val;
    public Foo(int v) {val = v;}
    public int Inc(int i) {return i + val;}
}

class Test {
    delegate int Calc(int i);
    static int Double(int i) {return 2 * i;}
    public static void Main() {
        int x;
        Calc a = new Calc(Double); x = a(7);
        Foo f = new Foo(5);
        Calc c = new Calc(f.Inc); x = c(7);
    }
}
```

We define a delegate called *Calc* – it takes a single *int* argument and returns an *int*. Then we invent a couple of methods with that same signature – a static method called *Double* and an instance method on a class *Foo* called *Inc*.

To use the delegate, we create an instance (called *a*) and *attach* it to the static method *Double*. We then invoke that method, giving it the argument 7, via our delegate.

Similarly, we create an instance of *Foo*, create another delegate instance (called *b*), and *attach* it to the instance method *f.Inc*. And again, we call *Inc* via our delegate.

Notice that we can *attach* our delegate to any old method we can access! – just so long as its signature matches that of our delegate. But unlike the use of function pointers in unmanaged code, the delegate technique ensures our code can be verified as type-safe.

In this example, we have called the delegate in a synchronous fashion – that's to say, we call the method via the delegate, that method executes, and control then returns to the caller. But it's possible to define delegates to provide asynchronous calls too – that's to say, one thread calls the delegate and continues execution; the target method executes asynchronously (by a different thread) and informs the caller later, when it's finished its work.

With this introduction to delegates over, the following sections discuss how delegates are declared, created and used.

10.5.1 Declaring Delegates

Delegates are reference types, and are declared in form of Classes. All delegates must inherit from `System.Delegate`. Delegates may not have subclasses, and so must be declared **sealed**. The only kind of members a Delegates may have are methods – they are not allowed to have fields, properties or events.

All Delegates must declare either two or four methods:

1. An instance constructor
2. An `Invoke` method (virtual)
3. A `BeginInvoke` method (optional, virtual)
4. An `EndInvoke` method (if `BeginInvoke` is declared, virtual)

All methods must be declared **runtime** and **managed** (see 11.5.5). The methods must not provide a body. The body will be automatically created by the CLI.

The constructor must take exactly two parameters. The first parameter is of type `System.Object` and the second parameter is of type `void*`. The first argument is an instance of the class (or one of its subclasses) that defines the target method. The reference encapsulates the environment of the method. The second argument is a method pointer to the method to be called.

The `Invoke` method must have the same signature as the target method – ie, it must have the same return type, the same parameter types, the same calling convention, and the same modifiers associated with the return type or parameters. The `Invoke` method defines what methods this Delegate represents.

The `BeginInvoke` method must always have the return type `System.IAsyncResult`. It must take the same parameters as `Invoke`, including all associated modifiers, with two additional parameters of types `System.AsyncCallback` and `System.Object`. The `BeginInvoke` method has the same calling convention as the `Invoke` method.

The `EndInvoke` method must always have the same return type as the `Invoke` method. It always takes the BYREF subset of the `Invoke` signature, plus one additional parameter of type `System.IAsyncResult`.

More information on `BeginInvoke` and `EndInvoke` and their parameters can be found in section 10.5.3.2.

Example:

The following example declares a Delegate used to call functions that take a single integer and return void --

```
.class private sealed auto autochar StartStopEventHandler extends
    [mscorlib]System.Delegate {
    .method public hidebysig specialname rtspecialname void .ctor(class
        System.Object object, void* 'method') runtime managed {}
```

```
.method public hidebysig virtual void Invoke(int32 action) runtime
managed {}

.method public hidebysig newslot virtual class
[mscorlib]System.IAsyncResult BeginInvoke(int32 action, class
[mscorlib]System.AsyncCallback callback, class System.Object object)
runtime managed {}

.method public hidebysig newslot virtual void EndInvoke(class
[mscorlib]System.IAsyncResult result) runtime managed {}

}
```

10.5.2 Creating Delegates

Like any regular Class, Delegates need to be instantiated before they can be used to call a method. Delegates are instantiated in the same way as other reference types (see section 7.3). The **newobj** instruction is used and the constructor of the delegate is specified with this instruction.

First, a method that follows the declaration of the delegate must be defined. This method is declared the same way as the **Invoke** method of the delegate, except it may have a different name, a different accessibility and may be **static**, **instance** or **virtual**.

The two parameters required by the constructor of the delegate need to be loaded onto the stack before the **newobj** instruction is executed. Recall that the constructor takes a parameter of type **System.Object** and another one of type **void***.

If the method to be abstracted by the delegate is a **static** method, then the first argument to the constructor has to be *null* (*null* is loaded with the instruction **ldnull**, see section 18.5). If the method is an **instance** or **virtual** method, the first argument has to be a reference to the object that represents the environment in which the method shall be called, i.e. the instance that is passed as the first argument to the method.

The second argument to the constructor is a managed pointer to the method. A method pointer may be obtained with the **ldftn** or **ldvirtftn** instruction (see section 10.4).

Example:

Suppose the delegate given in the example in section 10.5.2 is declared. Then the following method would follow the declaration of the delegate:

```
.method public void onStartStop(int32 action) {
    // body
}
```

The code below creates and instance of the delegate for the method above.

```
ldarg.0          // load this instance as environment for method
ldftn instance void Counter::onStartStop(int32)    // load method pointer
// the next line creates an instance of the delegate
newobj instance void StartStopEventHandler::.ctor(class System.Object,
void*)
// instance of delegate on stack
```

10.5.3 Using Delegates

As explained earlier, there are two ways to ‘call’ delegates: synchronously or asynchronously. For each delegate call, you must select synchronous or asynchronous.

10.5.3.1 Synchronous Calls

The synchronous mode of calling delegates corresponds to regular method calls. When a delegate call is made, the caller blocks until the called method returns. The called method is executed on the same thread as the caller.

To make a synchronous call the **Invoke** method of the delegate has to be used. Since the **Invoke** method is a virtual method, it needs to be called using the **callvirt** instruction (see section 11.4). The **callvirt** instruction requires that first an instance of the delegate is loaded onto the stack and

then all arguments of the `Invoke` method. If the method has a return value, it will be available on the stack after the call.

Example:

Continuing the example introduced in section 10.5.1, the following shows how the **Invoke** method of delegates is used:

```
ldloc startStopEventHandler    // load the delegate instance
ldloc state                   // load the argument
callvirt instance void StartStopEventHandler::Invoke(int32)
```

10.5.3.2 Asynchronous Calls

In the asynchronous mode, the call is dispatched, and the caller continues execution, without waiting for the method to return. The called method will be executed on a separate thread. There is no control over *when* the called method complete – the exact timing is determined by the thread scheduler.

Note: if the caller thread terminates before the callee completes, the callee thread will throw an exception. A thread that invokes an asynchronous call should not terminate until the call returns.

Note that the callee may throw exceptions. If such an exception is not caught by the callee, it is handed back to the caller.

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

10.5.3.2.1 The `BeginInvoke` Method

The asynchronous call is done using the `BeginInvoke` method of the delegate. The `BeginInvoke` method will enqueue request to execute the target method, and return control immediately to the caller. (there is no control over when that target method will be executed – it's a function of the thread scheduler, other requests enqueued, possible thread pooling, etc)

The `BeginInvoke` method is very similar to the `Invoke` method (10.5.3.1), but has three differences:

1. It has an additional parameter of type `System.AsyncCallback`
2. It has an additional parameter of type `System.Object`
3. The return type of the method is `System.IAsyncResult`

The `BeginInvoke` method has the same calling convention as the `Invoke` method. The two additional parameters appear after the parameters that the `BeginInvoke` method has in common, including their modifiers, with the `Invoke` method.

Once the call is made, the caller usually wants to know when the called method returns. This is especially important when the called method has a return value. This problem is traditionally solved with *callback* methods. A callback method is a method that is called when a certain asynchronous event occurs. In this case, the callback method is called when the delegate method returns. In order to call the callback method, a pointer to the method is needed. This is a typical case when the use of delegates is appropriate. Rather than passing a type unsafe pointer to the outside, the caller of the delegate method may instantiate yet another delegate that encapsulates the call to the callback method. In this particular case, the delegate is an instance of the delegate type `System.AsyncCallback`.

The caller needs to define a method that follows the declaration of `System.AsyncCallback`. The `Invoke` method of `System.AsyncCallback` has no return values and has one parameter of type `System.IAsyncResult`. The method defined by the caller needs to follow exactly this declaration.

Once the callback method is defined, the `System.AsyncCallback` delegate can be created in exactly the same way as described in section 10.5.2.

The second additional parameter of the `BeginInvoke` method is of type `System.Object`. The argument passed in for this parameter must be a reference to the calling object.

The return value of the `BeginInvoke` method is of type `System.IAsyncResult`. Usually, this return value may be ignored, but it may be useful in special circumstances. The interface defines a number of properties, including a wait handle that can be used to suspend the current thread until the called thread returns. Further, it has properties to query the state of the call, i.e. whether completed or not, and the object that is executed asynchronously, which in this case is the delegate of the called method.

Note: If the delegate accepts **out** parameters (& or * pointers to variables passed in as arguments), the variable to which the pointers refer to will be asynchronously updated by the called method. Any objects shared with the called method by references may asynchronously change their state. Beware of possible deadlocks in the latter case if the object has **synchronized** methods.

Example:

The following method follows the declaration of the `System.AsyncCallback` delegate:

```
.method private hidebysig instance void callback(class
[mscorlib]System.IAsyncResult result) {
    // body
}
```

The following piece of code executes an asynchronous call continuing the example of section 10.5.1:

```
ldloc startStopEventHandler    // load the delegate
ldloc state                    // load the argument to the method
// the next three instruction create the AsyncCallback delegate
ldloc caller                    // load reference to caller
// load method pointer to callback method
ldftn instance void StartStopButton::callback(class
[mscorlib]System.IAsyncResult result)
// create the AsyncCallback delegate
newobj instance void [mscorlib]System.AsyncCallback::.ctor(class
System.Object, int32)
ldloc caller                    // load the caller for BeginInvoke
method
// make the asynchronous call
callvirt instance class [mscorlib]System.IAsyncResult
StartStopEventHandler::BeginInvoke(int32, class
[mscorlib]System.AsyncCallback callback, class System.Object
object)
pop                             // ignore the IAsyncResult
```

10.5.3.2.2 The EndInvoke Method

When the method called via the delegate returns, the caller is notified through its callback method. The caller does not need to implement any logic in the callback method, but typically for delegate calls with return value, the caller may be interested in this return value.

The return value of the method called through the delegate may be obtained by calling `EndInvoke`. The name is a little bit misleading. When `EndInvoke` is called, the called method already completed. It is only used to obtain the return value.

Note: If the delegate has **out** parameters, these parameters were already asynchronously updated at the time `EndInvoke` is called.

Recall that the `EndInvoke` method expects one parameter of type `System.IAsyncResult`. The callback method receives an argument of the same type. In fact, the argument of the callback method may be passed on to the `EndInvoke` method.

However, since the `EndInvoke` method is a **virtual** method, it also has a hidden argument that expects a reference to the delegate that represented the called method. The argument passed to the

callback function contains a reference to this delegate. It is stored in the `AsyncObject` property of the argument. This property has type `System.Object`. Thus, it will be necessary to do an explicit cast to the correct delegate after the value of the property is retrieved.

The `EndInvoke` method has the same return type as the method represented by the delegate. The `EndInvoke` method will leave the return value of the method on the stack when it returns.

The `EndInvoke` method does not need to be called if the delegate method has no return value. In this case, the `EndInvoke` will also have no return value.

Example:

This example retrieves the return value from the delegate call of the example in section 10.5.3.2.1:

```
ldarg asyncResult                // the argument passed to callback
    method
// obtain the AsyncObject
callvirt class System.Object
    [mscorlib]System.IAsyncResult::get_AsyncObject()
// cast the AsyncObject to the used delegate type
castclass StartStopEventHandler
ldarg asyncResult                // also needed by EndInvoke
callvirt instance int32 StartStopEventHandler::EndInvoke(class
    [mscorlib]System.IAsyncResult result)
// return value on stack
```

10.5.4 Multicast Delegates

The delegates discussed so far are called *single cast* delegates, since they only abstract one method. However, in some cases it may be desirable to abstract a set of methods with the same signature and trigger a call to all these methods at the same time. Delegates that represent more than one method are called *multicast delegates*.

To create a multicast delegate, two or more delegates need to be combined. This is done using the `Combine` method that combines either two delegates or an array of delegates. The `Combine` delegate accepts both single and multicast delegates as arguments. The rules are applied recursively if multicast delegates are combined.

Only delegates which are subclasses of `System.MulticastDelegate` may be combined. `System.MulticastDelegate` is a subclass of `System.Delegate`. The `Combine` method will throw a `System.MulticastNotSupportedException` for attempts to combine subclasses of `System.Delegate`. Notice, that despite this fact the `Combine` method is defined in the class `System.Delegate`.

Only delegates of the same type may be combined. If the delegates do not have the same type, the `Combine` method will throw a `System.ArgumentException`.

The `Combine` method returns a new multicast delegate that represents the combined delegates.

When a multicast delegate is invoked, each delegate that was added with the `Combine` method will be invoked. The order of invocation corresponds to the order in which the single delegates were added. The methods represented by the first argument of the `Combine` method will be called first. If an array was passed to the `Combine` method, the first array element will be invoked first.

Delegates that are combined typically have no return values, but may have a return value. If they have a return value, only the return value of the last delegate added to the multicast delegate will be returned. All other return values will be discarded.

A particular delegate may be added multiple times to a multicast delegate. If the same delegate is added several times to a multicast delegate, it is called as many times it is added, using the order in which it was added compared to other method.

If an asynchronous call needs to be made to the delegate, the delegate must have a void return type. Further, the delegate must be marked with the custom attribute

`System.Runtime.Remoting.OneWayAttribute`. This attribute specifies that the caller is not interested in the return value of the method.

If a method represented by one of the delegates inside the multicast delegate throws an uncaught exception, no further calls will be made. The process will be terminated at the point where the exception was thrown.

A delegate may be removed from a multicast delegate by calling the `Remove` method. The `Remove` method accepts a multicast delegate as its first argument and a delegate as its second argument. It returns a new multicast delegate that does not contain the second argument. The original multicast delegate passed in as the first argument remains unmodified.

An interesting question arises about the order in which delegates that were added multiple times to the multicast delegate are removed. The `Remove` method removes the delegates in the opposite order they were added. E.g., if a particular delegate was added multiple times to a multicast delegate, the last version will be removed first.

Example:

The following declares a multicast delegate:

```
.class private sealed auto autochar StartStopEventHandler extends
    [mscorlib]System.MulticastDelegate {
    // insert the body of the delegate here
}
```

The following shows how delegates are combined:

```
ldloc handler1           // some StartStopEventHandler delegate
ldloc handler2           // some other StartStopEventHandler delegate
call class [mscorlib]System.Delegate
    [mscorlib]System.Delegate::Combine(class [mscorlib]System.Delegate,
    class [mscorlib]System.Delegate)
// multicast delegate is on stack
```

10.5.5 Other Methods of Delegates

In addition to the methods discussed above, the `System.Delegate` and `System.MulticastDelegate` classes have a number of other functions. A detailed description of these functions can be found in the Base Class Library documentation.

It should be pointed out that the `System.Delegate` class implements the interfaces `System.Runtime.ISerializable` and `System.IClonable` and thus supports serialization and cloning.

11 Methods

Methods implement the behavior of types. They contain the CIL instructions executed by the execution engine (EE).

One of the most important features of the CLI is that it offers various services to support the execution of a method. Code that makes use of these services is called *managed code*. Managed code provides enough information to allow the CLI to provide a set of core services, which include

- Given an address inside the code of a method, locating the metadata describing the method
- Walking the stack
- Handling exceptions
- Storing and retrieving security information

Only managed code may access managed data. To produce verifiable CIL code a compiler must produce managed code.

Code written for the CLI will be managed code. However, it is also possible to make calls to unmanaged methods from managed methods.

Methods may be defined at the global level as well as inside classes:

<code><decl> ::=</code>	Section
<code> .method <methodHead> { <methodBodyItem>* }</code>	
<code> ...</code>	4.2
<code><classMember> ::=</code>	Section
<code> .method <methodHead> { <methodBodyItem>* }</code>	
<code> ...</code>	7.5

The CLI distinguishes between three kinds of methods:

- **static** methods (section 11.3.1)
- **instance** methods (section 11.3.2)
- **virtual** methods (section 11.3.3)

The CLI has some special methods. These are the type initializers (**.cctor**) and instance constructors (**.ctor**) of types. The type initializers are called automatically by the runtime before a type is used (see section 7.6.7). Instance constructors are called when an instance of a class is created (section 7.6.5). In some cases, like value types, instance constructors can be called explicitly (see 9.2).

All methods have a common syntax as described in this chapter. A method definition consists of the keyword **.method**, a method head, and the body surrounded by braces, which contains the actual instructions to be executed.

```
<method> ::= .method <methodHead> { <methodBodyItem>* }
```

The following sections will give more details on these parts of a method definition. But first, the next section will specify the various method descriptors used in the CLI.

11.1 Method Descriptors

The runtime distinguishes between *MethodDecls*, *MethodDefs*, *MethodRefs*, and *MethodImpls*; each of them represented by tokens and describing methods.

A *MethodDecl* is simply the declaration of a method that contains all the information in the method head. A *MethodDecl* contains enough information to specify a contract that a caller can use to interact with the method. The *MethodDecl* specifies the name of the method, its parameters, return value, calling convention, accessibility, method attributes, and implementation information. A *MethodDecl* also includes a reference to the definition of the type that declares the method. *MethodDecls* are used in interfaces and for the declaration of **abstract** methods in classes.

A *MethodDef* is the definition of a method. It contains the body of the method, which contains all of its instructions, exception handlers, local variable information, and additional runtime or custom metadata about the method. A *MethodDef* includes a *MethodDecl*.

A *MethodRef* is a reference to a *MethodDecl*. It is used when a method is called or a pointer to a method is obtained. A *MethodRef* needs to be resolved into a *MethodDef* before the method is called. This can be either done at compile time, if the method is defined in the same module, or at runtime, when the class loader resolves all references. If a matching *MethodDef* cannot be found, the EE will throw a *System.MissingMethodException*.

A *MethodImpl* associates a *MethodDecl* with a *MethodDef*. It consists of two *MethodRefs*, the first of which specifies the *MethodDecl* to be implemented and the second specifies the implementing method. The *MethodRefs* will be resolved by the loader. A *MethodImpl* may be used to specify that a particular *MethodDecl* from an implemented interface is defined by a particular *MethodDef* of the implementing class. A *MethodImpl* can also be used to specify the definition of an **abstract** method, or to specify that

a particular method overrides another `MethodDef`. In any case, the signatures of the implemented and implementing method must match, which means that except their name and declaring type the signatures must be the same.

11.2 Method Signatures

The signature of a method is the part of a method declaration that uniquely identifies that method. A signature includes the following information:

- The name of the method
- The type that declares the method
- The calling convention of the method
- The return type of the method, including any **modreq** or **modopt** specifications
- The types of the parameters of the method in the order in which they are declared, including any **modreq** or **modopt** specifications

To reference a method, all of the information above must be present. Two methods may not have the same value for all items in the list. They must differ in at least one item.

If in two method declarations the first two items are the same, i.e. they have the same name and are declared in the same class, the methods are said to *overload* each other. Two methods also overload each other if they have the same name but are declared in different types, as long as one of the types is a subtype of the other. Note that unlike in many higher level languages the return type of the method belongs to this list. Thus in the CLI, it is possible to overload a method with a different return type.

11.3 Types of methods

As already pointed out, the CLI supports **static**, **instance** and **virtual** methods. These kinds of methods are described in the following sections.

11.3.1 Static Methods

static methods are methods that are associated with a type, but not with its instances. **static** methods are similar to procedures in procedural languages.

static methods may be defined in classes, value types, interfaces, and at the global level. In fact, all global methods must be declared as **static** methods. **static** methods must always have a body associated with them. **static** methods are referenced using a type reference to the type in which they are declared.

11.3.2 Instance Methods

In contrast to **static** methods, **instance** methods are associated with an **instance** of a type. It follows that **instance** methods may only be defined in classes or value types. In version 1 of the CLI they may not be defined inside interfaces.

While **static** methods only represent stand-alone behavior, **instance** methods have behavior that is associated with an environment of data that exists beyond the duration of the method call. The same environment may be re-used with the same method or another method that expects an instance of a compatible type.

instance methods accept a *this* pointer that has the same type as the type that declares the method and thus requires that an instance of that type or one of its subtypes must be created in order to be able to call the method. The instance referenced by the *this* pointer represents the environment in which the method is called. The *this* pointer is not included in the signature, but it is automatically added to the signature of a method as the first parameter. However, when an **instance** method is called, an instance of the class must be passed to the method explicitly as the first argument, even though this argument is not apparent from the signature.

Even though usually not practiced, the *this* pointer may be a *null* reference. Thus, **instance** and **virtual** methods must be prepared to handle the possibility that the first argument is *null*.

As an exception to the above, an **explicit** calling convention allows the explicit use of the *this* pointer in the signature (see 11.4.1).

For classes, the *this* pointer is a reference to an instance of the class, while for value types the *this* pointer is a managed pointer to the value type.

Note: Unlike **static** or **virtual**, **instance** is not a method attribute but part of the calling convention (see 11.4.1) of a method.

11.3.3 Virtual Methods

Similar to **instance** methods, **virtual** methods are associated with an **instance** of a class. However, unlike **instance** methods, the implementation of the method is not fixed. The MethodDef that implements a **virtual** method can be determined dynamically at runtime. **virtual** methods may be called with two instructions, **call** (see 11.4.3) and **callvirt** (see 11.4.4).

When the **call** instruction is used, **virtual** methods behave like **instance** methods. However, when the **callvirt** instruction is used, the notion of subclassing comes into play. A subclass may *override* a **virtual** method of its superclass, providing a new definition of the method. A **callvirt** to the method of the parent will result in a call to the new method, as long as an **instance** of the subclass is used. Since the subclass is a subtype of the original class, the declared types used in the code does not need to be changed. This allows significant modification of behavior with little changes to the code.

The method attribute **newslot** causes the runtime not to override the **virtual** method definition of the superclass with the new definition, but to treat the new definition as an independent **virtual** method definition. The first declaration of a **virtual** method in a class hierarchy should be marked **newslot**. More about **newslot** can be found in section 11.5.4.3.

Similar to **instance** methods, the *this* pointer must be passed to **virtual** methods, too.

11.3.3.1 Virtual Method Interop

Some languages, like C++, have more complicated inheritance rules. For these languages, the implementation of **virtual** methods in the CLI is not sufficient. Rather than using the implementation of the runtime, these languages may explicitly emit code to implement their own algorithm to call **virtual** methods.

The CLI has some support for these languages. It provides the **.vtfixup** directive (discussed in section 7.8.2) that declares a block of memory to contain a table in which method definition tokens may be inserted. After the methods are loaded into memory, the runtime will automatically convert the entries in the table into method pointers. Thus, compilers may emit code to obtain the addresses of specific methods at runtime. These addresses may be used to call the appropriate method.

Even though intended to allow virtual method interop, the **.vtfixup** directive can also be used to declare other tables in which MethodRefs need to be converted into method pointers.

11.4 Method Calls

The following subsections discuss in detail the various forms of method calls. This section gives a brief overview.

There are three call instructions to make a method call:

- **call**
- **callvirt**
- **calli**

There are two additional instruction that call methods in a special way:

- **jmp**
- **newobj**

Every method has its own *evaluation stack*, which is called just stack for short in this document. Data is loaded onto this stack and instructions take their arguments from the stack. The result of evaluating an

expression can be found on the stack after the appropriate instruction has executed. The runtime automatically creates and maintains this stack when a method is called. When a method exits, the stack is automatically released.

In addition to the evaluation stack there is also a *call stack* that keeps track of the method calls and is also maintained by the runtime.

A method usually returns to the caller using the **ret** instruction. If the method has a return value, it must be on the stack when the **ret** instruction is executed. The stack may not contain any value other than the return value when a **ret** is executed.

The CLI supports exception handling (see section 15). As a consequence, a method may exit also by throwing an exception.

The CLI does support tail calls, which can be found primarily in functional programming languages. A tail call is done using the prefix instruction **tail**. with any of the three call instructions above. More about tail calls can be found in section 11.4.6.

In a normal (non-tail) call, the current stack frame is kept intact and a new frame is allocated for the called method. For a tail call, on the other hand, the current frame is replaced with a frame for the called procedure.

The CLI also supports various kinds of calls as discussed in the next sections.

11.4.1 Calling Convention

A calling convention specifies how a method expects its arguments to be loaded onto the stack.

A calling convention includes a *call kind* (see section 11.4.2). In addition it has the optional keywords **instance** or **instance explicit**, which have to do with the *this* parameter.

```
<callConv> ::= [instance [explicit]] [<callKind>]
```

By default, the method call will be treated as a call to a **static** method. If **instance** is specified, the method call will be treated as a method call that accepts a *this* pointer as its first argument.

explicit applies only to **instance** methods. If **explicit** is specified, the *this* pointer is included explicitly in the method signature. This is typically used in the declaration of method pointers in connection with the **calli** instruction. **explicit** is not generally used with the **call** or **callvirt** instruction, except when interoperating with unmanaged code.

Example:

Suppose a method is defined in class **Bar** and takes a single parameter of type **int32**. The following type declaration of a variable of type method pointer uses an **explicit** signature:

method instance explicit void * (class Bar, int32)

Bar appears explicitly in the signature and specifies the *this* pointer type.

11.4.2 Call Kinds

The CLI supports various kinds of method calls as part of the calling convention (see section 11.4.1) . The two managed kinds of calls are **default** and **vararg**. As the name implies, **default** is the call kind used by default by the CLI. **vararg** specifies that the method accepts a variable number of arguments and thus needs to be called in a special way. More information about calling **vararg** methods can be found in section 11.4.9. Defining **vararg** method is covered in section 11.5.7.

The unmanaged kinds of calls are primarily intended to support languages which are similar to C++ and cannot produce managed methods for arbitrary code permitted by the language. These calling conventions are specific to Microsoft Windows and are generally needed to call methods of unmanaged DLLs.

There are four unmanaged call kinds:

- **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

The following grammar summarizes the call kinds.

```
<callKind> ::=  
    default  
    | unmanaged cdecl  
    | unmanaged fastcall  
    | unmanaged stdcall  
    | unmanaged thiscall  
    | vararg
```

11.4.3 The call Instruction

static and **instance** methods are called using the **call** instruction. The **call** instruction may also be used to call **virtual** methods. However, there is also another instruction for **virtual** methods which is discussed in section 11.4.4. The **call** instruction takes a **MethodRef** as part of the instruction and expects the arguments of the method on the stack. The first argument is pushed first onto the stack.

The only two differences between a call to a **static** and a call to an **instance** method is the use of the keyword **instance** in the call convention and that the first argument must be the *this* pointer when **instance** methods are called.

The general syntax for a **call** instruction is as follows:

```
call <callConv> <type> [<typeSpec>::]<methodName>( <parameters> )
```

The <typeSpec> specifies the type that declares the method. If it is missing, then the type in which the method call is made is assumed. The use of <typeSpec> is recommended in all cases.

For static methods the syntax simply becomes:

```
call <callKind> <type> [<typeSpec>::]<methodName>( <parameters> )
```

And for **instance** methods the syntax becomes:

```
call instance <callKind> <type> [<typeSpec>::]<methodName>( <parameters> )
```

To resolve the **MethodRef** of a **call** instruction the EE searches for a **MethodDef** that matches the **MethodRef**. The EE starts in the type referenced by the **MethodRef**. If the method cannot be found in this type, then the EE will go up step by step the hierarchy of superclasses until a match is found. E.g., if a **MethodRef** specifies a particular type in its <typeSpec> and this type does not provide a **MethodDef** for the **MethodRef**, then its superclass is searched. This continues until **System.Object** has been queried. Only then the EE will throw a **System.MissingMethodException**.

Example:

The following is a call to a static method:

```
call void [mscorlib]System.GC::RequestFinalizeOnShutdown()
```

The following is a call to an instance method:

```
ldloc timer // load the this pointer (argument #0)  
ldloc timerEventHandler // load argument #1
```

```
call instance void [System.Timers]System.Timers.Timer::add_Tick(class
[mscorlib]System.EventHandler)
```

Notice that the *this* pointer is not apparent from the signature of the method.

11.4.4 The callvirt Instruction

Since a call to **virtual** methods involves a runtime resolution of the `MethodRef`, there is a special instruction to call **virtual** methods, **callvirt**. Even though the **call** instruction may be used with **virtual** methods, only the **callvirt** instruction will consider overridden versions of the method.

Since virtual methods always expect a *this* pointer as the first parameter, the call convention must always include the keyword **instance**. The *this* pointer passed to a virtual method with a **callvirt** instruction cannot be *null*. (It may be *null* if a **call** instruction was used.) Thus, programmers of a **virtual** method must be prepared to get a *null* value for the first argument.

The **callvirt** instruction has the same syntax as the **call** instruction for **instance** methods:

```
callvirt instance <callKind> <type> [<typeSpec>::]<methodName>(
<parameters> )
```

Example:

The following code makes a call to a virtual method:

```
ldarg.0          // load this pointer
callvirt instance void Counter::HandleTick()
```

11.4.4.1 Super Calls

In some cases, it may be desirable to re-use code defined in the superclass. E.g., an overriding **virtual** method may want to call its previous version. This kind of re-use is called a *super call*, since the overridden method of the superclass is called. However, there is a problem. The **callvirt** instruction will consider overridden versions of the method, such that a **callvirt** to the method of the superclass will either result in a call to this method, causing an infinite recursion, or if the method was overridden by subclasses and the *this* pointer passed to the method is an instance of one of the subclasses, then an overridden version of the method depending on the exact type of *this* pointer will be called. However, the method wants to call exactly the version of the superclass. It turns out, that in this case the **virtual** method must be called with the **call** instruction and treated like an **instance** method.

Even though not a necessity, it is strongly recommended that the super call is made to the immediate superclass of the **virtual** method. As described in section 11.4.3 the **call** instruction will automatically search the hierarchy until an appropriate `MethodDef` is found. It is important that the call is made to the immediate super class, because this will enable a class to support newer versions of its super class. E.g., the superclass may add or remove the appropriate `MethodDef`, without breaking the implementation of the subclass.

Example:

The following is a super call to a virtual method (embedded in method of a class):

```
.class public auto autochar BeepingCounter extends Counter {
    .method virtual family void HandleTick() CIL managed {
        ldarg.0          // load the this pointer
        // call version in superclass
        call instance void Counter::HandleTick()
    }
}
```

11.4.5 Indirect Calls

A method may also be called using a pointer to the method with the **calli** instruction. In addition to the arguments, the **calli** instruction expects a method pointer at the top of the stack. A method pointer to

any kind of method can be loaded with the **ldftn** instruction, while method pointers to **virtual** methods can also be loaded with the **ldvirtftn** instruction, which will consider overloaded versions of the **virtual** method. More about method pointers can be found in section 10.4.

The syntax for **calli** does not include the method name:

```
calli <callConv> <type> ( <parameters> )
```

Example:

The following is an indirect call to a virtual method:

```
ldarg.0           // load the this pointer
ldvirtftn instance void Counter::HandleTick()
calli instance void ()
```

11.4.6 Tail Calls

Normally calls return to the caller when the called method is done. Tail calls, however, do not return to the caller. They effectively continue with the called method.

Non-tail calls, since they return to the caller, need to save who the caller is. This takes memory on the stack and may limit the depth of a recursion.

In contrast, tail calls do not need to save any return information on the stack. This gives recursions the possibility to continue practically forever without causing a stack overflow and may have significant performance advantages. In particular, tail calls may be used to efficiently implement various looping constructs.

The CLI has support for tail calls which may be marked with the prefix instruction **tail.. tail.** must be followed by one of the call instructions. Since in some cases, as explained below, the runtime may ignore a tail call, the **ret** instruction is required after the **call** instruction of a tail call. This will guarantee that the control flow of the method will always be the same independent of whether a tail or non-tail call was executed. If a tail call is executed, the **ret** instruction is ignored.

Methods called with a tail call may accept parameters like any other method and they may return a value.

Tail calls may only be used if the following conditions are met:

1. When the call instruction is executed, the stack may only contain the arguments of the called method and for the **calli** instruction a method pointer.
2. The return type of the called method must be the same or a subtype of the return type of the caller.
3. The caller may not pass the address of an argument or a local variable to the called method.
4. The call is not made from a protected block (try block) or from a handler of a protected block. The CLI does not support exception handling for tail calls.

If the four rules above are met, a **tail.** prefix may be used in front of the call. However, this does not guarantee that a tail call will actually be performed. In general, tail calls cannot be done if the calling method must do work to exit. In such a case, the runtime reserves the right to ignore the **tail.** prefix. The following lists the two additional conditions that must be met in order for a tail call to be actually executed:

1. The called method is a **managed** method and contains save CIL as determined by runtime security checks.
2. The calling method must not be a **synchronized** method.

If the above conditions are not met, the CLI will ignore the **tail.** prefix and execute a regular call.

The tail call behaves like a jump from one method to another. The EE will release the stack of the current method and create a new one for the called method. If a value is returned, the value will be directly returned to the original caller of the method making the tail call.

Compilers are recommended to emit the **tail.** prefix whenever it can be determined that a tail call is appropriate.

Example:

The following example shows a program that calculates the factorial of a number using tail calls. It does not require the use of the call stack.

```
/* num is the number for which the fact shall be computed (see also  
previous examples), result must be one, the return value is the final  
result */  
.method static public int64 fact(int64 num, int64 result) {  
    .maxstack 3  
    ldarg      num  
    ldc.i8     0  
    bgt compute  
    ldarg      result      // base case  
    ret  
compute:      // recursion  
    ldarg      num          // calculate new number  
    ldc.i8     1  
    sub  
    ldarg      result      // calculate result for this iteration  
    ldarg      num  
    mul  
    tail.      // make the call  
    call int64 fact(int64, int64)  
    ret  
}
```

11.4.7 jmp

The **jmp** instruction executes a jump across methods. A method may jump only to the beginning of a method and only to a method that has the same signature as the original method (including parameter types, return type and calling convention)

The **jmp** instruction is not verifiable. The **jmp** instruction is a special case of a tail call. It transfers control to a method that accepts the same parameters and has the same arguments as the caller.

Unlike in a call instruction, the **jmp** instruction passes the arguments of the current method to the next method. Thus, the arguments may not be loaded explicitly. If the arguments were mutated before the instruction is executed, the new values will be used. The arguments may be modified using the **starg** instruction (see section 18.4).

Similar to tail calls, if the method has a return value, the return value of the called method will be returned to the original caller.

Methods called with a **jmp** instruction are subject to even stronger restrictions than tail calls:

1. When the **jmp** is executed, the stack must be empty.
2. The caller must have the same signature as the called method. (In a future version, we might relax this to allow covariance)
3. The **jmp** may not be executed in a protected block or a handler block. There is no exception support for the **jmp** instruction.

In addition, to guarantee correct execution of code, the following restrictions should be followed:

1. The called method should contain code that is trusted by the security policy.
2. The caller should not be a **synchronized** method.

Example:

The following examples call the factorial method given in example 11.4.6.

```
.method private static int64 MyJump(int64, int64) CIL managed {  
    // do not load any arguments  
    jmp int64 fact(int64, int64)  
}
```

11.4.8 Calling Instance Constructors

Even though instance constructors (see also section 7.6.5) may be called directly, they are usually called with the **newobj** instruction.

The **newobj** instruction is used to instantiate a class. It creates a new object, but also calls the constructor of a class, which is a special **instance** method. Thus, the **newobj** instruction is very similar to an **instance** method call. The major difference is that no instance is passed to the constructor. Only its parameters as they are visible in the signature need to be passed to the constructor. The instance is automatically created by the runtime and passed to the constructor, such that the argument at index zero still ends up to be the *this* pointer.

More information about the **newobj** instruction can be found in section 7.3.

Example:

```
ldloc button  
ldloc count  
newobj instance void [.module Counter.dll]BeepingCounter::.ctor(class  
    [.module Counter.dll]StartStopEventSource, class [.module  
    Counter.dll]Count)  
// reference on stack
```

11.4.9 Calling vararg Methods

This section explains how **vararg** methods are called. Defining **vararg** methods is covered in section 11.5.7.

Since **vararg** is a calling convention, and thus part of the signature of the method, it must be specified also in the call. If a particular call includes no optional arguments, you specify the signature exactly the same as in its definition (specifically, do **not** include any ellipsis) . Conversely, if a particular call includes optional arguments, insert an ellipsis into the signature after the fixed parameters, followed by the additional parameter types for this call.

When the method is called, all the additional arguments must be on the stack.

Example:

The following example calls a **static vararg** method that takes one required integer. The definition of the **vararg** method can be found in section 11.5.7.

```
ldc.i4.1          // required  
ldc.i4.2          // these are all optional  
ldstr            "Hello vararg"  
ldc.r8 1.1  
call vararg void MyMethod(int32,...,int32,class System.String,float64)
```

11.5 Defining Methods

11.5.1 Method Head

The method head contains important information for the identification and correct handling of a method by the runtime. The head of a method also functions as an interface to other methods.

The method head consists of

- any number of predefined method attributes (section 11.5.4)
- an optional indication that this method is an **instance** method
- an optional description of the kind of call to use
- a return type with optional attributes (section 11.2)
- optional marshalling information (see also section 5.7)
- a method name
- a signature in brackets
- and any number of implementation attributes (section 11.5.4.4)

as also shown by the following syntax rule:

```
<methodHead> ::=  
    <methAttr>* [<callConv>] [<paramAttr>*] <type> [marshal (  
        [<nativeType>] )] <methodName> ( <parameters> ) <implAttr>*
```

Methods not marked as **static** or **virtual** are **instance** methods. Even though not necessary, the **instance** keyword may be used in the signature after the attributes. With the call kind together this is shown as a call convention in the above grammar. However, note that the keyword **explicit** has no meaning in a method declaration.

In the CLI, there are no attributes that can be used with the return type. However, future versions may have return type attributes, which is reflected in the grammar. Existing parameter attributes should not be used with the return type.

Methods that do not have a return value must use the keyword **void** as the return type.

Example:

The following declares a virtual method in a class:

```
.class public auto autochar Counter extends [mscorlib]System.Object {  
    .method virtual newslot hidebysig family instance void HandleTick()  
        synchronized CIL managed {  
        /* insert instructions and directives here */  
    }  
}
```

11.5.1.1 Method Name

Most method names are a <dottedname>. The exception are constructors because they would otherwise require quotation marks. Instance constructors of a type always have the name **.ctor**, while static (class) constructors of a type always have the name **.cctor**.

```
<methodName> ::=  
    .cctor  
    | .ctor  
    | <dottedname>
```

11.5.2 Method Parameters

Method *parameters* let the method accept information from the caller. Method parameters are type safe and force the caller to pass the required information to the called method. The contract between caller and called method specified through method parameters is enforced by the EE.

When a method is called, the caller must pass an *argument* for each required parameter of the method.

Method parameters are specified in parentheses after the method name and specify the types of all parameters to the method.

As shown by the following grammar, a method parameter declaration consists of any number of parameters:

```
<parameters> ::= [<param> [, <param>]*]
```

An individual parameter must either be the special token “...” or have a defined type and optionally have additional information.

```
<param> ::=  
    ...  
    | [<paramAttr>]* <type> [marshal ( [<nativeType>] )] [<id>]
```

The <id>, if present, is the name of the parameter. A parameter may be referenced either by using its name or the zero based index of the paramter.

The special value “...” can only occur once in a parameter declaration and it must be the last parameter.

The parameter attributes specify special handling of certain parameters:

```
<paramAttr> ::=  
    [in]  
    | [lcid]  
    | [opt]  
    | [out]  
    | [retval]
```

None of the parameter attributes is part of the method signature. Methods that have the same declaration but with different parameter attributes are considered to be duplicate method declarations (which is invalid CIL).

in and **out** specify whether a managed reference or pointer parameter is used to supply input to the method, return a value from the method, or both. If neither is specified **in** is assumed.

Note: These attributes are used by Interop marshalling code: for example, if a parameter is marked both **pdIn** and **pdOut**, then its value is marshalled to the callee, and its value is marshalled back to the caller (else, pinned and referenced as an optimization)

retval should only appear on one parameter of a method, and that parameter must be a pointer type. It is used only on interfaces that are being exposed to unmanaged COM clients, and is the parameter through which the CLI return value will be made visible to those clients.

opt indicates that this parameter is optional.

Note: The **opt** parameter attribute is for documentation purposes only. Compilers may mark a parameter with **opt** such that the compiler itself and other tools understand that the parameter is optional. However, from the point of view of the runtime optional parameters are not at all optional, they are treated the same way as required parameters. A corresponding argument must be provided in a method call.

lcid indicates that this parameter provides the locale ID to unmanaged COM clients.

Examples:

A single input parameter of type **int32**:

(int32 i)

A pair of parameters, with the first one an input parameter of type `System.String` and the second one an in/out parameter with type managed pointer to **int32**:

(class System.String myString, [in][out]int32& ptrInt)

A parameter declaration that takes one required input parameter, one optional input parameter and then any number of further parameters:

(int32 required, [opt]int32 optional, ...)

11.5.2.1 Method Parameters with Reflection

Signatures can be created using `System.Reflection.Emit.SignatureHelper`. They are most easily accessed using the method `GetParameters` on `System.Reflection.MethodBase` and `GetIndexParameters` on `System.Reflection.PropertyInfo`.

The information about an individual parameter can be seen through a `System.Reflection.ParameterInfo` and can be created using `System.Reflection.Emit.ParameterBuilder`.

The attributes in `<paramAttr>*` can be found under `CorParamAttr` in *CorHdr.h*.

11.5.3 Method Body

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

The following grammar shows the syntax for the body of a method and describes each item. More information about some of the directives can be found in the following subsections.

<code><methodBodyItem> ::=</code>	Description	Section
<code>.custom <customDecl></code>	Definition of custom attributes.	17
<code>.data <datadecl></code>	Emits data to the data section of the method.	12.3
<code>.emitbyte <unsigned int8></code>	Emits a byte to the code section of the method.	11.5.3.1
<code>.entrypoint</code>	Specifies that this method is the entry point to the application (only one such method is allowed).	11.5.3.2
<code>.locals [init] (<localsSignature>)</code>	Defines a set of local variables for this method.	11.5.3.3
<code>.maxstack <int32></code>	int32 specifies the maximum number of elements on the evaluation stack during the execution of the method	11.5.3
<code>.override <typeSpec>::<methodName></code>	Sets this method definition as the implementation for the method specified in the instruction.	7.6.3.1
<code>.param [<int32>] [= <fieldInit>]</code>	For parameter number <int32>, stores the constant defined by <fieldInit> as its default value	11.5.3.4

.ventry <int32> : <int32>	.ventry <entry> : <slot>	11.5.3.5
.zeroinit	Specifies that all local variables are initialized to zero in this method.	11.5.3
<externSourceDecl>	.line or #line	3.7
<instr>	An instruction	18.1
<codeLabel> :	A label	3.4
<scopeBlock>	See below	11.5.6
<securityDecl>	.permission or .capability	16
<sehBlock>	An exception block	15.1

The following sections describe some of the directives above in more detail.

11.5.3.1 .emitbyte

The **.emitbyte** directive emits an unsigned 8 bit value directly into the code section of the method. The value is emitted at the position where the directive appears.

The **.emitbyte** directive can be used to emit the opcode of a certain instruction. However, typically the **.emitbyte** directive is used to emit compiler specific data, which is not intended to be executed by the execution engine. In the latter case, the compiler should also emit code to jump over the emitted value. It should be considered whether it is necessary to emit data into the code section rather than the data section. Emitting data into the data section is covered in section 12.3.

Examples:

The following code emits a breakpoint, which can also be done with the **break** instruction.

```
.emitbyte 1
```

The following example shows how custom data can be emitted into the code section.

```
// some code
br.s continue      // jump over data
.emitbyte 123      // custom data
continue:
// some other code
```

11.5.3.2 .entrypoint

The **.entrypoint** directive marks the current method as the entry point to an application. The execution engine will call this method to start the application.

Every executable must have exactly one entry point method. This entry point method may be a global method or may appear inside a type. However, it must be a **static** method.

The entry point method may either accept no arguments or may accept a vector of strings. If it accepts a vector of strings, the strings inside the vector will represent the arguments to the executable, with index 0 containing the first argument.

The return value of the entry point method must be either **void**, **int32**, or **unsigned int32**. If an **int32** or **unsigned int32** is returned, the executable can return an exit code to the operating system. A value of 0 indicates that the application terminated ordinarily. A non-zero return value may indicate various abnormal termination conditions. This value may be queried by other applications.

Entry point methods may have any accessibility. The CLI will always have access to this method.

Example:

The following example prints the first argument and return successfully to the operating system:

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

11.5.3.3 .locals

.locals is used to define local variables for this method. *Local variables* are variables that may only be accessed by the method in which they are defined. They may be used to store data that is only needed during a method call. Once the method returns, the local variable will be discarded. Memory for local variables can be allocated faster than memory for fields.

The local variables of a method are described by a signature, although the syntax is slightly different from that for methods, since it is not possible to specify attributes (**in**, **out**, etc.) or marshaling for local variables.

A `<localsSignature>` is simply a comma separated list of one or more local variable descriptions.

`<localsSignature> ::= <local> [, <local>]*`

`<local> ::= [[<int32>]] <type> [<id>]`

The assembler allows nested local variable scopes to be provided and allows locals in nested scopes to share the same location as those in the outer scope. The information about local names, scoping, and overlapping of scoped locals is persisted to the PDB (debugger symbol) file rather than the PE file itself.

The integer in brackets that precedes the `<type>`, if present, specifies the local number (starting with 0) being described. This allows nested locals to reuse the same location as a local in the outer scope. It is not legal to overlap two local variables unless they have the same type. The identifier, if present, is the name of the local within the current scope. When no explicit index is specified, the next unused index is chosen. That is, two locals never share an index unless the index is given explicitly.

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. The `<localsSignature>` (see 11.5.3.3) lists the local variables. Each local variable receives a zero based index that is unique within the method.

Using **init** in a **.locals** directive has the same effect as using the **.zeroinit** directive at the method level. Thus, if **init** is used, all local variables will be initialized to their default values, even variables in another **.locals** directive in the same method, which does not have the **init** directive.

Example:

The following declares two local variables for the method:

```
.locals init (int32 myCount,
    value class [System.Drawing]System.Drawing.Point point)
```

11.5.3.3.1 Local Variables with Reflection

Information about local variables can be created using the `System.Reflection.Emit.LocalBuilder` class.

11.5.3.4 .param

Stores a *default* value with method parameter number <int32>. The value is stored into metadata. And that value may be interrogated when importing that metadata. But the CLI itself does not retrieve the value or insert it automatically into method calls. We call it a *default* value, because that is how compilers often use the feature – but the CLI attaches no meaning to the constant.

Note: Since it emits metadata, **.param** uses the indexing of the metadata engine. Index 0 specifies the return value of the method. Index 1 is the first parameter of the method. This is different from the index used in CIL instructions (eg **ldarg**), where 0 indicates the first argument for a method – not its return value.

11.5.3.5 .vtentry

This directive is used within the body of a virtual method definition. It directs ILASM to insert the token for this method into a specified slot in an unmanaged vtable. For example,

```
.vtentry 0:2
```

will insert this method's token into slot number 2 of the 0th unmanaged vtable. This feature is used by compilers who want to control their own virtual method dispatch (for example, in implementing multiple inheritance). For a fuller description, see the section on **.vtfixup** (section 7.8.2).

11.5.4 Predefined Attributes on Methods

Predefined attributes of a method are attributes which provide important information for the caller of a method. Predefined attributes of a method specify information about accessibility, contract information, virtual method table information, implementation attributes, interoperation attributes, as well as information on special handling.

In addition to predefined attributes, the CLI supports custom attributes which are described in further detail in section 17.

The following subsections contain additional information on each group of predefined attributes of a method.

<methAttr> ::=	Description	Section
abstract	Specifies that the method is an abstract method.	11.5.4.4
assembly	Assembly accessibility	11.5.4.1
famandassem	Family and Assembly accessibility	11.5.4.1
family	Family accessibility	11.5.4.1
famorassem	Family or Assembly accessibility	11.5.4.1
final	Specifies that this method cannot be overridden by subclasses.	11.5.4.2
hidebySIG	Hide by signature. Ignored by the runtime.	11.5.4.2
newslot	Specifies that this method shall get a new slot in the virtual method table.	11.5.4.3
pinvokeimpl (<QSTRING> [as <QSTRING>] <pinvAttr>*)	pinvokeimpl (<DLL name> [as <alias>]	11.5.4.5

		<attribute>*)	
private	Private accessibility	11.5.4.1	
privatescope	Privatescope accessibility.	11.5.4.1	
public	Public accessibility.	11.5.4.1	
rtsspecialname	The method name needs to be treated in a special way by the runtime.	11.5.4.6	
specialname	The method name needs to be treated in a special way by some tool.	11.5.4.4	
static	Specifies that this method is a static method of a type.	11.5.4.2	
unmanagedexp	Marks for exports to unmanaged world.	11.5.4.5	
virtual	Specifies that this method is a virtual method.	11.5.4.2	

11.5.4.1 Accessibility Information

The accessibility attributes are **assembly**, **famandassem**, **family**, **famorassem**, **private**, **privatescope** and **public**. These attributes are exclusive. If you do not specify an accessibility for a method, then ILASM inserts one for you, depending upon the visibility of its owner Type, as follows:

- global, **private** => **privatescope**
- global, **public** => **public**
- non-global, top-level, **public** => **public**
- non-global, top-level, **private** => **private**
- nested, **XXX** => **XXX** (where XXX is any of the above)

Accessibility attributes are described in section 6.3.

11.5.4.2 Method Contract Attributes

Method contract attributes are **final**, **hidebysig**, **static**, and **virtual**. These attributes may be combined, except a method may not be **static** and **virtual** at the same time. Only **virtual** methods may be **final** and **abstract** methods may not be **final**.

final methods may not be overridden by subclasses of this class. This makes sure the functionality provided by the implementing class is not modified by other implementations.

hidebysig specifies that the declared method hides all methods of the parent classes that have a matching method signature. Note that this attribute is ignored by the runtime. The runtime always matches methods by signature. This attribute is intended to let compilers emit additional information, that can be read by the same or other compilers or some other tools. If a compiler intends to hide methods by signature, the **hidebysig** attribute should be emitted. Some languages use hide by name, where only the name of the method must match for hiding purposes. These languages should not emit **hidebysig**. If However, since the runtime still uses hiding by signature, compilers of

these languages must emit explicit code to reference the intended methods. More about hiding can be found in section 6.2.

You can define a method, including the keyword **static**. When called, such a method does not expect an implicit *this* reference as its first argument. Or, you can defined a method, including the **virtual** keyword. (If neither **static** nor **virtual** is specified, the method is called an *instance* method). **Virtual** or *instance* methods *do* expect an implicit *this* reference as their first argument. Do not specify the contradictory combination **virtual** and **static**.

11.5.4.3 Overriding Behavior

The only attribute in this group is **newslot**. **newslot** can only be used with **virtual** methods.

By default, a **virtual** method will override the implementation of a matching **virtual** method in the superclass. This can be prevented by using the attribute **newslot**. If **newslot** is used, the method will not override any method of the superclass.

Calls to the method of the superclasses of this method will be redirected to the implementation of the superclass of this class. Calls to method in this class and its subclasses will be redirected to the new implementation, or to an overriding version of one of the subclasses.

The attribute **newslot** should be used if a virtual method is declared for the first time in a class hierarchy.

The implementation of the superclass may still be overridden by subclasses by using a `MethodImpl` with an explicit reference to the implementation of the superclass.

11.5.4.4 Implementation Attributes

The two implementation attributes are **abstract** and **specialname**. **abstract** can only be used with non-**final** virtual methods. These two attributes may be combined.

abstract specifies that the method is not provided and needs to be defined by a subclass. **abstract** methods can only appear in **abstract** classes (see section 7.2).

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

11.5.4.5 Interoperation Attributes

These attributes are for interoperation with Windows or classical COM applications. The attributes in this category are **pinvokeimpl** and **unmanagedexp**. These two attributes may be combined.

pinvokeimpl instructs the runtime to use the platform invoke functionality to invoke an unmanaged method in the specified DLL with the specified export name (see also section 11.6.1.1).

unmanagedexp marks this managed method as exported for use by unmanaged callers, via the legacy Export Address Table in the PE image.

11.5.4.6 Other Attributes

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

11.5.5 Implementation Attributes of Methods

Implementation attributes of a method are attributes that provide important additional information to the runtime. They contain information about required special handling by the runtime or more information on the code implemented by the method. Implementation attributes may also contain additional information on interoperation with classical COM.

The following subsections contain additional information on each group of implementation attributes..

<code><implAttr> ::=</code>	Description	Section
forwardref	Specifies that the body of this method is not specified with this declaration.	11.5.5.3

CIL	Specifies that the method contains standard CIL code.	11.5.5.1
internalcall /* round trip only */	Used only for disassembling purposes.	11.5.5.3
managed	Specifies that the method is a managed method.	11.5.5.2
native	Specifies that the method contains native code.	11.5.5.1
noinlining	Specifies that the runtime shall not attempt to inline the method.	11.5.5.3
ole	Indicates method signature is mangled to return HRESULT, with the return value as a parameter.	11.5.5.4
optil	Specifies that the method contains OptIL code.	11.5.5.1
runtime	The body of the method is not defined but produced by the runtime.	11.5.5.1
synchronized	The method will be executed in a single threaded fashion.	11.5.5.3
unmanaged	Specifies that the method is unmanaged.	11.5.5.2

11.5.5.1 Code Implementation Attributes

The code implementation attributes are **CIL**, **native**, **optil**, and **runtime**. These attributes are exclusive. The default is **CIL**.

These attributes 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 must be provided if **CIL** is used.

native indicates that a method was implemented using native code, rather than the device independent CIL. **native** methods only execute on the platform their code targets. **native** methods may not have a body with instructions but must refer to a native method that declares the body. Typically, the PInvoke functionality (see 11.6.1.1) of the CLI is used to refer to a native method. **native** method declarations are used to define the signature of the method important for callers and the runtime. In addition, **native** methods may have custom attributes and other metadata information associated with them.

optil marks that the method contains optimized CIL. Optimized CIL follows certain restrictions that allow the JIT to compile the CIL to native code faster.

runtime specifies that the implementation of the method is automatically provided by the runtime and is primarily used for the constructor and invoke method of delegates.

11.5.5.2 Managed or Unmanaged

The options are either **managed** or **unmanaged**. The default is **managed**.

As the names imply, if **managed** is specified, the execution of the method will be managed by the CLI. If **unmanaged** is specified, the code will not be managed by the CLI.

11.5.5.3 Implementation Information

The attributes in this group are **forwardref**, **internalcall**, **synchronized**, and **noinlining**. The attributes may be combined.

forwardref specifies that the body of the method is provided elsewhere –in another module of the same assembly. (This corresponds to a C++ forward declaration)

internalcall is a special token used by the disassembler for some methods and must not be used. The explicit use of **internalcall** will cause the execution engine to throw a `System.Security.SecurityException`.

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 will be obtained before the method is entered. If this method is a static method a lock on the class will be obtained before the method is entered. If a lock cannot be obtained the requesting thread will be suspended and placed on a waiting queue until it is granted the lock. This may cause dead locks.

noinlining specifies that the runtime shall not inline this method. Inlining refers to the process of replacing the call instruction with the body of the called method. This may be done by the runtime for optimization purposes.

11.5.5.4 Interoperation

The attribute **ole** is used for compatibility with unmanaged COM. It instructs the runtime to convert the signature of a method for calls in both directions unmanaged to managed and managed to unmanaged.

The conversion from managed to unmanaged appends the return value of a method to its parameter list as an **out**, **retval** parameter with the corresponding pointer type of the return type. The new return type of the method becomes HRESULT. Instead of throwing an exception, the HRESULT value will indicate success or failure.

The conversion from unmanaged to managed is the opposite way.

11.5.6 Scope Blocks

Scope blocks are syntactic sugar and primarily serve for readability and debugging purposes.

Syntactically, a scope block is enclosed inside braces and recursively contains a `<methodBodyItem>`.

```
<scopeBlock> ::= { <methodBodyItem>* }
```

A scope block defines the scope in which a local variable is accessible by its name. Scope blocks may be nested, such that a reference of a local variable will be first tried to resolve in the innermost scope block, than at the next level, and so on until the top-most level of the method, is reached. A declaration in an inner scope block hides declarations in the outer layers.

If duplicate declarations are used, the reference will be resolved to the first occurrence. Even though valid CIL, duplicate declarations are not recommended.

Scoping does not affect the lifetime of a local variable. All local variables are created (and if specified initialized) when the method is entered. They stay alive until the execution of the method is completed.

The scoping does not affect the accessibility of a local variable by its zero based index. All local variables are accessible from anywhere within the method by their index.

The index is assigned to a local variable in the order of declaration. Scoping is ignored for indexing purposes. Thus, each local variable is assigned the next available index starting at the top of the method. This behavior can be altered by specifying an explicit index, as described by a `<localsSignature>` as shown in section 11.5.3.3.

Example:

```
{
    .locals (int32 a)           // declares local 0
    {
        .locals (int32 a)       // declares local 1
        ldloc a                 // loads local 1
    }
}
```

```
        pop
    }
    ldloc    a                // loads local 0
    pop
    ldloc.1                // loads local 1, which is still alive
    pop
}
```

11.5.7 vararg Methods

vararg methods are methods that may accept a variable number of arguments.

Methods that want to accept a variable number of arguments must have the calling convention **vararg**. This becomes part of their signature. When such a method is called, you must separate the fixed from the additional parameters in the signature with an ellipsis (see section 11.4.9)

Any number and type of arguments may be passed after the last required argument. The **vararg** arguments may be accessed by obtaining a handle to the argument list. This is done using the **arglist** instruction. The handle can be used to create an instance of the value type `System.ArgIterator`. The iterator can be used to obtain the arguments using the `GetNextArg` method. This method returns a **typedref**. Finally, the **typedref** can be used to obtain a reference to the argument. This is also illustrated in the example at the end of this section.

The `GetRemainingCount` method of `System.ArgIterator` can be used to obtain the number of arguments left. If `GetNextArg` is called after the last **vararg** argument was returned, `System.ArgIterator` will throw a `System.InvalidOperationException`.

Calling **vararg** methods is described in section 11.4.9.

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 (value class System.ArgIterator it,
        int32 x)
    ldloca    it                // initialize the iterator
    initobj   value class System.ArgIterator
    ldloca    it
    arglist                                // obtain the argument handle
    call instance void System.ArgIterator::.ctor(value class
System.RuntimeArgumentHandle) // call constructor of iterator
    /* argument value will be stored in x when retrieved, so load
        address of x */
    ldloca    x
    ldloca    it
    // retrieve the argument, the argument for required does not matter
    call instance typedref System.ArgIterator::GetNextArg()
    call class System.Object System.TypedReference::ToObject(typedref)
        // retrieve the object
    castclass System.Int32 // cast and unbox
    unbox     int32
    cpobj     int32                // copy the value into x
    // first vararg argument is stored in x
    ret
}
```

11.6 Unmanaged Methods

11.6.1 Calling Unmanaged Methods

There are two primary mechanisms to call unmanaged methods, *It Just Works* and *PInvoke*.

It Just Works (IJW) scenarios are designed for programmers who wish to use existing unmanaged data types for interoperation with unmanaged code. The CLI provides very little marshaling support and, since the data types are unmanaged, the programmer is required to deal directly with lifetime and memory management. Users can write their own custom marshaling code to wrap existing unmanaged code if they wish to provide a managed view.

The primary issue in IJW is to guarantee that execution cannot transfer from managed code to unmanaged code (or vice versa) without first executing transition code supplied by the CLI. This transition primarily deals with exception handling and garbage collection. To support this, IJW relies on the following:

- Instances of the type `System.ArgIterator` are marshaled specially across the managed/unmanaged boundary, so that they appear to unmanaged code as the type required by the C++ `va_*` macros or functions.
- Function pointers are not marshaled across the boundary. It is the responsibility of the user to convert pointers as needed across the boundary, and the CLI provides a mechanism for doing this conversion. By considering managed/unmanaged to be part of the type of a function pointer, this work can be handled automatically by a compiler.

Platform invoke (PInvoke) is a combination of the transition management provided by IJW with data marshaling similar to that provided by COM Interop. It allows existing APIs to be called from managed code, with automatic conversion between some managed types and their unmanaged equivalents.

11.6.1.1 Platform Invoke

Methods defined in a native DLL may be invoked using the *PInvoke* (platform invoke) functionality of the CLI. PInvoke will handle everything needed to make the call work. It will automatically switch from managed to unmanaged state and back and also handle necessary conversions. Methods that need to be called using PInvoke are marked as **pinvokeimpl**. In addition, the methods must have the implementation attributes **native** and **unmanaged** (see 11.5.4.4).

pinvokeimpl takes in parentheses the name of the DLL with any number of PInvoke attributes. After the name of the DLL, an **as** clause may be inserted that specifies an alias that may be used to call the method. While the name of the method must exactly match the name of the method as declared in the DLL, the alias may be used as an alternative name for the method in CIL. This may be useful, since some compilers use cryptic names for methods in DLLs.

<methAttr> ::=	Description	Section
pinvokeimpl (<QSTRING> [as <QSTRING>] <pinvAttr>*)	pinvokeimpl (<DLL name> [as <alias>] <attribute>*)	
...		11.5.4

A method declared with **pinvokeimpl** may not have a body, since it was declared to be **native**.

pinvokeimpl methods may be global methods as well as methods of classes. Only **static** methods may be **pinvokeimpl**. The CLI PInvoke feature does not support **instance** or **virtual** methods.

A call to a **pinvokeimpl** method is just like any other call to a **static** method.

Note: The disassembler *ildasm* may output **pinvokeimpl** declarations with no DLL name, e.g. when VC++ code is disassembled, and the assembler *ilasm* does accept **pinvokeimpl** declaration with no DLL. However, this is invalid CIL and the produced code will fail to execute. *PEVerify* will report **pinvokeimpl** declarations without a specified DLL as errors.

The following grammar shows the attributes of a **pinvokeimpl** instruction.

<code><pinvAttr> ::=</code>	Description
<code>ansi</code>	ANSI character set.
<code> autochar</code>	Determine character set automatically.
<code> cdecl</code>	Standard C style call.
<code> fastcall</code>	C style fastcall.
<code> lasterr</code>	Indicates that method supports C style last error querying.
<code> nomangle</code>	Use the member name as specified – do not search for ‘A’ (ascii) or ‘W’ (wchar) variants in the Win32 DLLs
<code> ole</code>	Indicates method signature is mangled to return HRESULT, with the return value as a parameter.
<code> stdcall</code>	Standard C++ style call.
<code> thiscall</code>	The method accepts an implicit this pointer.
<code> unicode</code>	Unicode character set.
<code> winapi</code>	Pinvoke will use native call convention appropriate to target windows platform.

Example:

The following shows the declaration of the method `MessageBeep` located in the Windows DLL `user32.dll`:

```
.method public static pinvokeimpl("user32.dll" cdecl) int8  
MessageBeep(unsigned int32) native unmanaged {}
```

The above method may be called like any other static method from CIL code.

11.6.1.1.1 Name Mangling in DLLs

Some compilers, like VC++, convert human readable names into cryptic names that contain additional information like types, etc. This process is called *name mangling*.

ilasm does not support automatic name mangling for PInvoke calls. Thus, it is important to use the exact mangled name of the method. The mangled name of the method can be obtained using the *dumpbin* tool applied to the DLL:

```
dumpbin /symbol <dllname>
```

The alias used with **pinvokeimpl** gives an opportunity to declare explicitly the unmangled name of the method.

11.6.1.2 Via Function Pointers

Unmanaged functions can also be called via function pointers. There is no difference between calling managed or unmanaged functions with pointers. However, the unmanaged function needs to be declared with **pinvokeimpl** as described in section 11.6.1.1. Calling managed methods with function pointers is described in section 11.4.

11.6.1.3 COM Interop

Unmanaged COM operates primarily by publishing uniquely identified interfaces and then sharing them between implementers (traditionally called “servers”) and users (traditionally called “clients”) of a given interface. It supports a rich set of types for use across the interface, and the interface itself can supply named constants and static methods, but it does not supply instance fields, instance methods, or virtual methods.

The CLI provides mechanisms useful to both implementers and users of existing classical COM interfaces. The goal is to permit programmers to deal with managed data types (thus eliminating the need for explicit memory management) while at the same time allowing interoperability with existing unmanaged servers and clients. COM Interop does not support the use of global functions (i.e. methods that are not part of a managed class), static functions, or parameterized constructors.

- Given an existing classical COM interface definition as a type library, the *tlbimp* tool produces a file that contains the metadata describing that interface. The types it exposes in the metadata are managed counterparts of the unmanaged types in the original interface.
- *Implementers* of an *existing* classical COM interface can import the metadata produced by *tlbimp* and then write managed classes that provide the implementation of the methods required by that interface. The metadata specifies the use of managed data types in many places, and the CLI provides automatic marshaling (i.e. copying with reformatting) of data between the managed and unmanaged data types.
- *Implementers* of a new service can simply write a managed program whose publicly visible types adhere to a simple set of rules. They can then run the *tlbexp* tool to produce a type library for classical COM users. This set of rules guarantees that the data types exposed to the classical COM user are unmanaged types that can be marshaled automatically by the CLI.
- *Implementers* need to run the RegAsm tool to register their implementation with classical COM for location and activation purposes – if they wish to expose managed services to unmanaged code
- *Users* of *existing* classical COM interfaces simply import the metadata produced by *tlbimp*. They can then reference the (managed) types defined there and the CLI uses the assembly mechanism and activation information to locate and instantiate instances of objects implementing the interface. Their code is the same whether the implementation of the interfaces is provided using classical COM (unmanaged) code or the CLI (managed) code: the interfaces they see use managed data types, and hence do not need explicit memory management.
- For some existing classical COM interfaces, the CLI execution engine provides an implementation of the interface. In some cases the EE allows the user to specify all or parts of the implementation; for others it provides the entire implementation.

11.6.1.4 Calling from Managed to Unmanaged

From the point of view of an CIL code generator, both IJW and PInvoke are handled in the same way as calls to other named methods. There is a call to a method using the ordinary CIL mechanisms (a **call**, **callvirt**, or **jmp** instruction) that specifies a destination by way of a metadata token. When resolved at runtime, the metadata token is discovered to be associated with a *methoddef* that is specially marked that its implementation is unmanaged code. This definition effectively provides two signatures: one for the managed side (indicating how it is being called) and one for the unmanaged side (indicating how it is implemented).

It is the job of the CLI execution engine and any CIL-to-native-code compilers to cooperate to make sure that the transition is done correctly, including any possible data marshaling. When the data types are identical in both of the signatures, no marshaling occurs. Where the type as passed by the (managed) caller differs from the type expected by the (unmanaged) receiver, the PInvoke marshaling rules are invoked to convert the data types.

For calls or jumps via a function pointer, the mechanism is slightly different. The **ldftn** and **ldvirtftn** instructions construct a pointer to an entry point and the type conveys whether it is a managed or unmanaged entry point. There is a Base Class Library routine (unsafe but known to verification) that takes a function pointer and converts it from any given calling convention to any other, by producing a transition stub as needed.

11.6.1.5 Calls from Unmanaged to Managed

Just as there are two ways to call from managed to unmanaged (direct and via a pointer), there are two ways to call from unmanaged to managed. Since the call is arising in unmanaged code, however,

there is no simple way to arrange for a direct call to a managed method. For IJW, the VC compiler and linker arrange that any unmanaged code that tries to call managed code will do so by one of two mechanisms:

- If the managed code is in the same module as the call site, the linker arranges for an entry in a table that represents the managed address, and forces the jump or call to go via that table entry. When the module is loaded, the CLI execution engine is started (because the module has managed code in it) and this table is updated to contain transition thunks for use when calling from unmanaged to managed code (see also 7.8.2).
- If the managed code is in a different module than the call site, the linker uses its existing mechanism to make an entry in the Import Address Table requesting the appropriate *unmanaged* entry point. The exporting module will have exported this entry point, and made it to point to a table entry (also fixed up by the CLI execution engine) to perform the transition.

The situation is somewhat easier for function pointers. The assumption is that the function pointer is already pointing to a transition function. This will have been generated either because

- The marshaling code saw a managed function pointer or delegate in the managed signature and a pointer to an unmanaged function in the unmanaged signature and so produced the necessary stub, or
- The compiler saw a type mismatch between an attempt to pass or store a pointer to a managed function where a pointer to an unmanaged function was required, so it called the Base Class Library function mentioned earlier to produce the transition function.

11.6.2 Managed Native Calling Conventions (x86)

This section is intended for an advanced audience. It describes the details of a native method call from managed code on the x86 architecture. The information provided in this section may be important for optimization purposes. This section is not important for the further understanding of the CLI and may be skipped.

There are two managed native calling conventions used on the x86. They are described here for completeness and because knowledge of these conventions allows an unsafe mechanism for bypassing the overhead of a managed to unmanaged code transition.

11.6.2.1 Standard 80x86 Calling Convention

The standard native calling convention is a variation on the **fastcall** convention used by VC. It differs primarily in the order in which arguments are pushed on the stack.

The only values that can be passed in registers are managed and unmanaged pointers, object references, and the built-in integer types I1, U1, I2, U2, I4, U4, I and U. Enums are passed as their underlying type. All floating point values and 8-byte integer values are passed on the stack. When the return type is a value type that can't be passed in a register, the caller must create a buffer to hold the result and passes the address of this buffer as a hidden parameter.

Arguments are passed in left-to-right order, starting with the *this* pointer (for instance and virtual methods), followed by the return buffer pointer if needed, followed by the user-specified argument values. The first of these that can be placed in a register is put into ECX, the next in EDX, and all subsequent ones are passed on the stack.

The return value is handled as follows:

- Floating point values are returned on the top of the hardware FP stack.
- Integers up to 32 bits long are returned in EAX.
- 64-bit integers are passed with EAX holding the least significant 32 bits and EDX holding the most significant 32 bits.
- All other cases require the use of a return buffer, through which the value is returned.

In addition, there is a guarantee that if a return buffer is used a value is stored there only upon ordinary exit from the method. The buffer is not allowed to be used for temporary storage within the method and its contents will be unaltered if an exception occurs while executing the method.

Examples:

```
static System.Int32 f(int32 x)
```

The incoming argument (x) is placed in ECX; the return value is in EAX

```
static float64 f(int32 x, int32 y, int32 z)
```

x is passed in ECX, y in EDX, z on the top of stack; the return value is on the top of the floating point (FP) stack

```
static float64 f(int32 x, float64 y, float64 z)
```

x is passed in ECX, y on the top of the stack (not FP stack), z in EDX; the return value is on the top of the FP stack

```
virtual float64 f(int32 x, int64 y, int64 z)
```

this is passed in ECX, x in EDX, y pushed on the stack, then z pushed on the stack (hence z is top of the stack); the return value is on the top of the FP stack

```
virtual int64 f(int32 x, float64 y, float64 z)
```

this is passed in ECX, x in EDX, y pushed on the stack, then z pushed on the stack (hence z is on top of the stack); the return value is in EDX/EAX

```
virtual [mscorlib]System.Guid f(int32 x, float64 y, float64 z)
```

Since `System.Guid` is a value type the *this* pointer is passed in ECX, a pointer to the return buffer is passed in EDX, x is pushed, then y, and then z (hence z is on top the of stack); the return value is stored in the return buffer.

11.6.2.2 Varargs x86 Calling Convention

All user-specified arguments are passed on the stack, pushed in left-to-right order. Following the last argument (hence on top of the stack upon entry to the method body) a special *cookie* is passed which provides information about the types of the arguments that have been pushed.

As with the standard calling convention, the *this* pointer and a return buffer (if either is needed) are passed in ECX and/or EDX.

Values are returned in the same way as for the standard calling convention.

11.6.2.3 Fast Calls to Unmanaged Code

Transitions from managed to unmanaged code require a small amount of overhead to allow exceptions and garbage collection to correctly determine the execution context. On an x86 processor, under the best circumstances, these transitions take approximately 5 instructions per call/return from managed to unmanaged code. In addition, any method that includes calls with transitions incurs an 8 instruction overhead spread across the calling method's prolog and epilog.

This overhead can become a factor in performance of certain applications. For use in unverifiable code only, there is a mechanism to call from managed code to unmanaged code without the overhead of a transition. A "fast native call" is accomplished by the use of a **calli** instruction which indicates that the destination is managed even though the code address to which it refers is unmanaged. This can be arranged, for example, by initializing a variable of type *function pointer* in unmanaged code.

Clearly, this mechanism must be tightly constrained since the transition is essential if there is any possibility of a garbage collection or exception occurring while in the unmanaged code. The following restrictions apply to the use of this mechanism:

1. The unmanaged code must follow one of the two managed calling conventions (regular and **vararg**) that are specified below. In V1, only the regular calling convention is supported for fast native calls.

2. The unmanaged code must not execute for any extended time, since garbage collection cannot begin while executing this code. It is wise to keep this under 100 instructions under all control flow paths.
3. The unmanaged code must not throw an exception (managed or unmanaged), including access violations, etc. Page faults are not considered an exception for this purpose.
4. The unmanaged code must not call back into managed code.
5. The unmanaged code must not trigger a garbage collection (this usually follows from the restriction on calling back to managed code).
6. The unmanaged code must not block. That is, it must not call any OS-provided routine that might block the thread (synchronous I/O, explicitly acquiring locks, etc.) Again, page faults are not a problem for this purpose.
7. The managed code that calls the unmanaged method must not have a long, tight loop in which it makes the call. The total time for the loop to execute should remain under 100 instructions or the loop should include at least one call to a managed method. More technically, the method including the call must produce “fully interruptible native code.” In future versions, there may be a way to indicate this as a requirement on a method.

Note: restrictions 2 through 6 apply not only to the unmanaged code called directly, but to anything it may call.

12 Fields

Fields are typed memory locations that store the data of a program. The CLI allows the declaration of both instance and **static** fields. While **static** fields are associated with a type and shared across all instances, instance fields are associated with an instance of a type. When instantiated, the instance has its own copy of the field.

The CLI also supports global fields, which are fields not declared inside a type. Global fields must be **static**.

A field is defined by using the **.field** directive and a field declaration:

```
<field> ::= .field <fieldDecl>
```

The <fieldDecl> has the following parts:

- an optional integer specifying the offset if specific layout of a class is desired
- any number of field attributes (see section 12.2)
- a type
- a name
- and optionally either a <fieldInit> form or a data label

This is also shown by the following grammar.

<pre><fieldDecl> ::= [[<int32>]] <fieldAttr>* <type> <id> [= <fieldInit> at <dataLabel>]</pre>	<p>Comments</p> <p>[<int32>] is byte offset, for explicit layout only, ignored in global and static fields; at <label> specifies the data item label</p>
---	--

The optional field offset is ignored for static and global fields. For instances, the field will be stored at the specified offset within the portion of the instance data belonging to this class. Classes that use this feature must be declared **explicit** (see also section 7.8.1).

Global fields must have a data label associated with them. The data label specifies where the data of the field is located. Static fields of a type may, but do not need to, be assigned a data label.

Example:

The following is an instance variable declaration:

```
.field private class [.module Counter.dll]Counter counter
```

12.1 Predefined Attributes of Fields

Predefined attributes of a field specify information about accessibility, contract information, interoperation attributes, as well as information on special handling.

The following subsections contain additional information on each group of predefined attributes of a field.

<fieldAttr> ::=	Description	Section
assembly	Assembly accessibility.	12.1.1
famandassem	Family and Assembly accessibility.	12.1.1
family	Family accessibility.	12.1.1
famorassem	Family or Assembly accessibility.	12.1.1
initonly	Marks a constant field.	12.1.2
literal	Specifies metadata field. No memory is reserved for this field.	12.1.2
marshal ([<nativeType>])	Marshaling information.	12.1.3
notserialized	Field is not serialized with other fields of the class.	12.1.2
private	Private accessibility.	12.1.1
privatescope	Privatescope accessibility.	12.1.1
public	Public accessibility.	12.1.1
rtspecialname	Special treatment by runtime.	12.1.4
specialname	Special name for other tools.	12.1.4
static	Static field.	12.1.2

12.1.1 Accessibility Information

The accessibility attributes are **assembly**, **famandassem**, **family**, **famorassem**, **private**, **privatescope** and **public**. These attributes are exclusive. If you do not specify an accessibility for a field, then ILASM inserts one for you, depending upon the visibility of its owner Type, as follows:

- global, private => privatescope
- global, public => public
- non-global, top-level, public => public

- non-global, top-level, private => private
- nested, XXX => XXX (where XXX is any of the above)

Accessibility attributes are described in section 6.3.

12.1.2 Field Contract Attributes

Field contract attributes are **initonly**, **literal**, **static** and **notserialized**. These attributes may be combined. Only static fields may be **literal**. The default is an instance field that may be serialized.

static specifies that the field is associated with the type itself rather than with an instance of the type. Static fields can be accessed without having an instance of a class, e.g. by static methods. As a consequence, a **static** field is shared within an application domain (see the CTS and Remoting specifications) between all instances of a class, and any modification of this field will affect all instances. If **static** is not specified, an *instance* field is created.

initonly marks fields which are constant after they are initialized. These fields may only be mutated inside a constructor. If the field is a static field, then it may be mutated only inside the type initializer of the type in which it was declared. If it is an instance field, then it may be mutated only in one of the instance constructors of the type in which it was defined. It may not be mutated in any other method or in any other constructor, including constructors of subclasses.

Note: The CLI may not check whether **initonly** fields are mutated outside the constructors. The EE need not report any errors if a method changes the value of a constant. However, such code is not valid and an error will be reported by *PEVerify*.

notserialized specifies that this field is not serialized when an instance of this class is serialized (see section 7.4.2.5). It has no meaning on global or static fields, nor if the class doesn't have the **serializable** attribute.

literal specifies that this field represents a constant value. In contrast to **initonly** fields, **literal** fields do not exist at runtime. There is no memory allocated for them. **literal** fields become part of the metadata but cannot be accessed by the code. **literal** fields are assigned a value by using the <fieldInit> syntax (see section 12.2).

12.1.3 Interoperation Attributes

There is one attribute for interoperation with classical COM applications. The attribute is **marshal** and is used for marshalling the field to a native type whenever it is used by unmanaged code and marshalling it back to the managed form such that it can be continue to be used by managed code.

The default marshalling for each type is listed in two appendices (Unmanaged-to-Managed, and vice-versa) – see Part 5.

12.1.4 Other Attributes

The attribute **rtspecialname** indicates that the field name shall be treated in a special way by the runtime.

In contrast to **rtspecialname**, **specialname** indicates that the field name has special meaning to some tools.

12.2 Field Init Metadata

The <fieldInit> metadata can be optionally added to a field declaration. The use of this feature may not be combined with a data label.

The <fieldInit> information is available only in metadata. It does not have any affect on the actual value of the field and does not create any instructions. Thus, the <fieldInit> option does *not* initialize the field with any value but puts a value associated with this field into the metadata of the PE file. The field init metadata is typically used with **literal** fields (see section 12.1.2) or optional parameters (see section 11.5.2).

The following table lists the options for a field init. The type used has to agree with the type of the field. The description column provides additional information.

<code><fieldInit> ::=</code>	Description
<code>bytearray (<bytes>)</code>	Array of type U1 (8 bit). <bytes> specifies the actual bytes.
<code> float32 (<float64>)</code>	32 bit floating point number, with the floating point number specified in parentheses. The number needs to fit in 32 bits.
<code> float32 (<int32>)</code>	<int32> is binary representation of float
<code> float64 (<float64>)</code>	64 bit floating point number, with the floating point number specified in parentheses.
<code> float64 (<int64>)</code>	<int64> is binary representation of double
<code> int8 (<int8>)</code>	8 bit integer with the integer specified in parentheses.
<code> int16 (<int16>)</code>	16 bit integer with the integer specified in parentheses.
<code> int32 (<int32>)</code>	32 bit integer with the integer specified in parentheses.
<code> int64 (<int64>)</code>	64 bit integer with the integer specified in parentheses.
<code> <QSTRING></code>	String. <QSTRING> is stored as ASCII
<code> wchar (<QSTRING>)</code>	String. <QSTRING> is stored as Unicode.

Example:

The following example shows a typical use of this:

```
.field public static literal value class ErrorCodes no_error = int8(0)
```

The variable is a **literal** variable for which no memory is allocated. However, tools and compilers can look up the value.

12.3 Embedding Data in a PE File

CLI allows programs to store data in a PEFile.

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.

<code><decl> ::=</code>	Section
<code> .data <datadecl></code>	
<code> ...</code>	4.7

Data may also be declared as part of a class:

<code><classMember> ::=</code>	Section
<code> .data <datadecl></code>	
<code> ...</code>	7.5

Yet another alternative is to declare data inside a method:

<code><methodBodyItem> ::=</code>	Section
<code> .data <datadecl></code>	
<code> ...</code>	11.5.3

In all cases, the data should be declared with the entity in which it is used. E.g., data used only by a method should be declared in that method. Regardless of where the data is declared, it is accessible anywhere inside the assembly.

12.3.1 Data Declaration

The data declaration of a **.data** directive contains the optional attribute **tls** to specify *thread local storage* (see below). Further, it contains an optional data label and the body which defines the actual data. A data label must be used if the data shall be accessed by the code.

```
<dataDecl> ::= [tls] [<dataLabel> =] <ddBody>
```

Each PE file has a particular section whose initial contents are copied whenever a new thread is created. This section is called thread local storage. Marking data as **tls** causes the data to be put in the part of the PE File.

The body consists either of one data item or a list of data items in braces. A list of data items is similar to an array.

```
<ddBody> ::=  
    <ddItem>  
    | { <ddItemList> }
```

A list of items consists of any number of items:

```
<ddItemList> ::= <ddItem> [, <ddItemList>]
```

The list may be used to declare multiple data items associated with one label. The items will be laid out in the order declared. The first data item will be accessible directly through the label. To access the other items, pointer arithmetic needs to be used, adding the size of each data item to get to the next one in the list. The use of pointer arithmetic will make the application not verifiable.

A data item declares the type of the data and provides the data in parentheses. If a list of data item 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 simply put in brackets after the declaration. Note that the data is not accessible in a verifiable way.

```
<ddItem> ::=  
    & ( <id> )  
    | bytearray ( <bytes> )  
    | char * ( <QSTRING> )      //ASCII encoded  
    | float32 [( <float64> )] [[ <int32> ]]  
    | float64 [( <float64> )] [[ <int32> ]]  
    | int8 [( <int8> )] [[ <int32> ]]  
    | int16 [( <int16> )] [[ <int32> ]]  
    | int32 [( <int32> )] [[ <int32> ]]  
    | int64 [( <int64> )] [[ <int32> ]]  
    | wchar * ( <QSTRING> )      //Unicode encoded
```

Example:

The following declares an int32:

```
.data theInt = int32(123)
```

12.3.2 Accessing Data

The data can be accessed through a **static** variable. A **static** variable, either global or a member of a type, needs to be declared at the position of the data. The syntax for this is as follows:

```
<fieldDecl> ::= <fieldAttr>* <type> <id> at <dataLabel>
```

This is similar to a regular <fieldDecl>. After the **at** the label pointing to the location at which the data is stored is inserted. One of the field attributes must be **static**.

The data can then be accessed through the static variable. The variable may be accessed also by other modules or assemblies.

To export the data to the unmanaged world, the static variable needs to appear in a type that is exported.

Example:

The following accesses the data declared in the example of section 12.3.1. 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:

```
ldsfd int32 myInt  
// data on stack
```

12.3.3 Unmanaged Thread-local Storage

There are two mechanisms for dealing with thread-local storage (**tls**): an unmanaged mechanism and a managed mechanism. The unmanaged mechanism has a number of restrictions which are carried forward directly into the CLI. For example, the amount of thread local storage is determined when the PE file is loaded and cannot be expanded. The amount is computed based on the static dependencies of the PE file, DLLs that are loaded as a program executes cannot create their own thread local storage through this mechanism. The managed mechanism, which does not have these restrictions, is described in the Base Class Library documentation.

For unmanaged **tls** there is a particular native code sequence that can be used to locate the start of this section for the current thread. The CLI respects this mechanism. That is, when a reference is made to a static variable with a fixed RVA in the PE file and that RVA is in the thread-local section of the PE, the native code generated from the CIL will use the thread-local access sequence.

This has two important consequences:

- A static variable with a specified RVA must reside entirely in a single section of the PE file. The RVA specifies where the data begins and the type of the variable specifies how large the data area is.
- When a new thread is created it is only the data from the PE file that is used to initialize the new copy of the variable. There is no opportunity to run the class initializer. For this reason it is probably wise to restrict the use of unmanaged thread local storage to the primitive numeric types and value types with explicit layout that have a fixed initial value and no class initializer.

12.4 Initialization of Static Data

Many languages that support static data (i.e. variables that have a lifetime that is the entire program) provide for a means to initialize that data before the program begins running. There are three common mechanisms for doing this, and each is supported in the CLI.

12.4.1 Data Known at Link Time

When the correct value to be stored into the static data is known at the time the program is linked (or compiled for those languages with no linker step), the actual value can be stored directly into the PE file, typically into the **.data** area (see section 12.3). 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 class 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 the CTS specification). 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.

12.4.2 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 is possible to supply arbitrary code to run as the PE file is loaded and while the OS holds a process-wide lock.

This mechanism, while available in the CLI, is strongly discouraged. The details are provided in section 1.2.2.

12.4.3 Data Known at Run Time

When the correct value cannot be determined until class layout is computed, the user must supply code as part of a type initializer to initialize the static data. The guarantees about class initialization are covered in section 7.6.7.1. As will be explained below, global statics are modeled in the CLI as though they belonged to a class, so the same guarantees apply to both global and class statics.

Because the layout of managed classes need not occur until a class is first referenced, it is not possible to statically initialize managed classes by simply laying the data out in the PE file. Instead, there is a class initialization process that proceeds in the following steps:

1. All static variables are zeroed.
2. The user-supplied class initialization procedure, if any, is invoked as described in section 7.8.2.

Within a class 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 JITted, which may prove to be both a code space and an execution time problem.
- *Box value types*. When the static variable is simply a boxed version of a primitive numeric type or a value type with explicit layout, introduce an additional static variable with known RVA that holds the unboxed instance and then simply use the **box** instruction to create the boxed copy.
- *Create a managed array from a static native array of data*. This can be done by marshaling the native array to a managed array. The specific marshaler to be used depends on the native array. E.g., it may be a **safearray**.
- *Default initialize a managed array of a value type*. The .NET SDK Base Class Library will provide a method that will call the default constructor (or zero the storage if there is no default constructor) for every element of an array of unboxed value types (called `System.Runtime.CompilerServices.InitializeArray`)
- *Use Base Class Library deserialization*. The .NET SDK Base Class Library provides serialization and deserialization services. These services can be found in the `System.Runtime.Serialization` namespace. An object can be converted to a serialized form, stored in the data section and accessed using a static variable with known RVA of type **unsigned int8[]**. The corresponding deserialization mechanism can then be used in the class initializer.

13 Properties

Properties can be thought of as smart fields. Rather than just being a field that contains a value, properties are implemented by a collection of methods. Every property represents a certain value. Methods associated with this property retrieve or change this value. Typically, a property is associated with a field that stores the value. However, the value of the property may also be computed on the fly by methods.

The CLI supports the declaration of properties. From the point of view of the runtime, properties are only metadata, they associate some methods and optionally a field together. The property declaration does not have any meaning at runtime. However, the metadata can be used by compilers and other tools to inspect the property and understand what methods are associated with the property. E.g., the property declaration describes what method needs to be called in order to obtain the value of the property.

Typically, properties have a *getter* and a *setter*. The getter returns the current value of the property, while the setter updates the value of the property with a new value. Some higher level programming languages implement the getter the same way as an access to a field and the setter the same way as an assignment to a field. However, there is no such support at the runtime level. However, the metadata describes what method will implement the functionality of the getter and what method will implement the functionality of the setter.

The CLI supports **static**, **instance**, and **virtual** properties. While static properties are only associated with the type in which they are declared, **instance** and **virtual** properties are associated with an instance of the type in which they are declared.

Properties have two major advantages over fields. The first advantage is that behavior is associated with the access and updating of a value. This can be used to abstract the representation of the value and do additional computation and validation when the value is accessed or updated. Properties also give more control over restricting what values are stored when and by whom.

Virtual properties allow their getters and setters to be implemented via virtual methods. As a consequence, these methods may be overridden by subclasses and their behavior may be modified in an object oriented fashion to do more sophisticated or specialized work. Subclasses also get a chance to interrupt when a value is retrieved or updated.

In addition to the setters and getters, properties may also have other methods associated with them.

It may seem that method calls to obtain a value may be much slower than direct access to a field. However, in reality the CIL-to-native code compiler of the runtime will optimize properties, so that for trivial getters and setters there is no performance loss.

13.1 Declaring properties

A property is declared by the using the **.property** directive, followed by a property head and property members in braces. Properties may only be declared inside of types.

<code><classMember> ::=</code>	Section
<code> .property <propHead> { <propMember>* }</code>	
<code> ...</code>	7.2

Even though it is typical that the setters and getters of the property come from the type which declares the property, this is not necessary.

13.1.1 Property Head

The property head contains a calling convention (see section 11.4.1), a type and a name, and parameter declarations in parentheses.

The property head may contain the keywords **specialname** or **rtspecialname**. **specialname** marks the name of the property for other tools, while **rtspecialname** marks the name of the property as special for the runtime.

While in theory there need be no relationship between the signature of a property and the methods that implement it, in practice the declaration of the property must match the declaration of the getter

method. The calling convention, type, and parameters of a property have to be the same as they are defined for the get method.

```
<propHead> ::=  
    [specialname|rtspecialname]* <callConv> <type> <id> ( <parameters>  
        )
```

13.1.2 Property Members

A property may contain any number of property members in its body. The following table shows these and provides short descriptions for them:

<code><propMember> ::=</code>	Description	Section
<code>.backing <type> <id></code>	Backing field of the property.	
<code> .custom <customDecl></code>	Custom attribute.	17
<code> .get <callConv> <type> [<typeSpec> ::] <methodName> (<parameters>)</code>	Specifies the getter for the property.	
<code> .other <callConv> <type> [<typeSpec> ::] <methodName> (<parameters>)</code>	Specifies a method for the property other than the getter or setter.	
<code> .set <callConv> <type> [<typeSpec> ::] <methodName> (<parameters>)</code>	Specifies the setter for the property.	
<code> <externSourceDecl></code>	.line or #line	3.7

As described above, **.backing** specifies the field in the class that is associated with this property, by its type and name. A property does not need to have a backing field, but if it has such a field it should be specified in the property declaration. The backing field has to be defined in the same type as the property itself, otherwise it cannot be specified in the metadata. The field should be marked with **specialname** in its declaration to highlight it for tools and compilers.

.get specifies the getter method for this property with a MethodRef. The method needs to be defined in the type specified by `<typeSpec>` or in this class. Only one get method may be specified for a property. The `<callConv>`, `<type>` and parameters of the get method, in practice, should be the same as those specified in the property head. The get method should be marked with **specialname** in its declaration to highlight it for tools and compilers.

.set specifies the setter method for this property using a MethodRef, similar to the way the getter is specified. A property may have only one setter. The set method should be marked with **specialname** in its declaration to highlight it for tools and compilers.

.other can be used to specify other methods associated with this property. If the other methods have a special semantics, the implementer of the property needs to document these and used appropriate naming conventions.

In addition, custom attributes (see section 16) or source line declarations may be specified.

Example:

This example shows the declaration of the property used in the example in Part 5.

```
.class public auto autochar MyCount extends [mscorlib]System.Object {  
    .field private specialname int32 count           // the backing field
```

```
.method virtual hidebysig public specialname instance int32 get_Count()
{
    // body of getter
}

.method virtual hidebysig public specialname instance void
set_Count(int32 newCount) {
    // body of setter
}

.method virtual hidebysig public instance void reset_Count() {
    // body of refresh method
}

// the declaration of the property
.property int32 Count() {
    .backing int32 count
    .get instance int32 get_Count()
    .set instance void set_Count(int32)
    .other instance void reset_Count()
}
} // end of class MyCount
```

14 Events

An *event* is the change of state of an object at a particular time. Any non-trivial application reacts to events. Examples of events are hardware events, like keyboard strokes or mouse movements, or software events, like selecting a menu item.

Events become very important in a multi-tasking environment, where many processes and their threads share the same processor resources. Many applications need to show some behavior when an event occurs, but do not do any computation when no event occurred. It is inappropriate to loop and keep checking whether the state of an object changed. This technique wastes huge amounts of processor resources. In a complex system an event model that enables applications to efficiently detect events is a necessity.

This section describes the event model of the CLI and shows how applications may observe and declare events. The CLI has built-in support for events in its metadata. This allows a consistent event model across all platforms and languages that target the CLI. The CLI event model makes it possible for an event to be declared in one language and observed by an application that is written in another language.

Events are similar to properties in the sense that they are pure metadata elements. However, the metadata description of events is powerful enough to inform observers as well as event sources of how the event is used. Similar to properties, events have a number of methods associated with them. These methods have a special semantics and implement a certain behavior that is defined by the CLI event model.

An *event source* is the object where an event may occur. If an event occurs, the event source *fires* the event. An event has a special method associated with it that fires the event.

An event *observer* is an object that waits for an event and presumably shows some behavior if the event occurs. To do this, an observer needs to *listen* to an event. An observer may start listening to an event by adding itself onto the list of observers for the event. If the observer wants to discontinue listening to an event, it may remove itself from this list. An event has special methods associated with it that add an observer to the event and remove an observer from the event.

When an event is fired, the observer is notified that an event is called through a method call. A method that is dedicated to be called whenever the event is fired is called an *event handler*. When an observer starts listening to an event, it needs to register an event handler with the event source.

The CLI supports both **static** and **instance** events. While **static** events are associated with a type, **instance** events are associated with an **instance** of the type.

14.1 Implementing Events

The event source needs to be able to call the event handler of the observer. This is done using delegates (see section 10.5), which specify the signature of the event handler that must be implemented by the observer.

Delegates not only provide an ideal way of abstracting the observer's reference to event handler, they also provide additional functionality. Usually, an event will be listened to by several observers at a time and the event source needs to be prepared to be able to handle multiple observers. While the event source may implement behavior to handle multiple observers, delegates provide a standard mechanism--multicast delegates. Multicast delegates abstract the code involved in combining various delegates. When the event is fired all handlers are notified with just one call to the multicast delegate. The event source may define its own delegate type or use the delegate `System.EventHandler`. The `Invoke` method of the `System.EventHandler` delegate takes two parameters, the event source and data associated with the event.

14.2 Observing Events

Example:

This example shows how a timer event source is created and an observer added to the timer event.

Assume the timer is needed by a class called `Counter`. The following declares the needed fields in the sample class `Counter`:

```
.field private class [System.Timers]System.Timers.Timer timer
.field private class [mscorlib]System.EventHandler timerEventHandler
```

The following is the declaration of the event handler:

```
.method virtual hidebysig private void instance onTick(class
    System.Object, class [mscorlib]System.EventArgs) CIL managed {
    // body of event handler
}
```

The method below adds this class to the timer even using the handler above:

```
.method virtual hidebysig famorassem instance void SetupTimer() CIL
managed {
    .maxstack 3
    // create the timer
    ldarg.0
    ldc.r8 1000 // 1000 ms
    newobj instance void
    [System.Timers]System.Timers.Timer::.ctor(float64)
    stfld class [System.Timers]System.Timers.Timer Counter::timer
    // create the delegate
    ldarg.0 // load this pointer, needed 3 times
    dup // duplicate top of stack, 2 copies on stack
    dup // duplicate another time, 3 copies on stack
    ldvirtftn instance void Counter::onTick(class System.Object, class
    [mscorlib]System.EventArgs)
    newobj instance void [mscorlib]System.EventHandler::.ctor(class
    System.Object, int32)
    stfld class [mscorlib]System.EventHandler
    Counter::timerEventHandler
    // add this observer to the time event
    ldarg.0
    ldfld class [System.Timers]System.Timers.Timer Counter::timer
```

```

ldarg.0
ldfld class [mscorlib]System.EventHandler
Counter::timerEventHandler
call instance void
[System.Timers]System.Timers.Timer::add_Tick(class
[mscorlib]System.EventHandler)
ret
}

```

When done, the observer should be removed from the event.

14.3 Declaring Events

Events are declared inside types with the **.event** directive. Following the directive is an event head and any number of event members.

<classMember> ::=	Section
.event <eventHead> { <eventMember>* } ...	7.2

Even though it is typical that the members of the event are declared in the type which declares the event, this is not necessary.

14.3.1 Event Head

The event head contains the type and a name for the event.

```

<eventHead> ::=
    [specialname] [rtsspecialname] [<typeSpec>] <id>

```

If the event was implemented using delegates, the type refers to the type specification of the delegate. Otherwise, the type should refer to a type that specifies the signature of the handler for the event.

The event head may contain the keywords **specialname** or **rtsspecialname**. **specialname** marks the name of the property for other tools, while **rtsspecialname** marks the name of the event as special for the runtime.

14.3.2 Event Members

The grammar below shows the various members of an event:

<eventMember> ::=	Description	Section
.addon <callConv> <type> [<typeSpec> ::] <methodName> (<parameters>) .custom <customDecl>	Add method for event.	16
.fire <callConv> <type> [<typeSpec> ::] <methodName> (<parameters>) .other <callConv> <type> [<typeSpec> ::] <methodName> (<parameters>) .removeon <callConv> <type> [<typeSpec> ::] <methodName> (<parameters>)	Fire method for event. Other method.	
<externSourceDecl>	Remove method for event. .line or #line	3.7

Even though not required an event should at least provide an add and a remove method.

The **.addon** directive specifies the add method by providing a MethodRef. If the <typeSpec> is not specified, the method is assumed to be in the same type as the event. The add method may take any parameters, but typically it will take at least a reference to the event handler of the observer, e.g. in form of a delegate. The type of the parameter accepting the event handler should be same as the type used in the event head. The accessibility of the add method restricts the observers that may add themselves to the event. If this should be unrestricted, **public** needs to be used. The add method should be marked with **specialname** in its declaration to highlight it for tools and compilers.

The **.removeon** directive specifies the remove method by providing a MethodRef similar to the **.addon** directive. The remove method may also take any parameters, but typically it will take at least a reference to the event handler to be removed. This is the same reference that was used with the add method. The accessibility of the remove method should be the same as the accessibility of the add method. The remove method should be marked with **specialname** in its declaration to highlight it for tools and compilers.

The **.fire** directive is used to specify the fire method of the event. It also uses a MethodRef to refer to the implementation similar to the **.addon** and **.removeon** directives. The **.fire** directive is not required but should be provided if possible. The accessibility of the fire method determines which types may fire the event. If only the type that declares the fire method should be able to fire the event, the fire method needs to be **private**. The fire method should be marked with **specialname** in its declaration to highlight it for tools and compilers.

The implementation of the event will make use of one or more fields to store the observers. It is important that the fields that store information about the observers remain private or at most family. This will make sure that no unauthorized objects remove observers by using the fields or fire the event without the control of the event source. However, firing the event means notifying the observers, which makes access to them necessary. The fire method gives other objects the opportunity to request that an event is fired without the event having to expose its underlying data structures. In addition, the fire method abstracts the specific implementation of the event.

An event may contain any number of other method specified with the **.other** directive. From the point of view of the runtime, these methods are only associated with each other through the event. If they have special semantics, this needs to be documented by the implementer.

Events may also have custom attributes (17) associated with them and they may declare source line information.

Example:

The following example shows the declaration of an event, its corresponding delegate, and typical implementations of the add, remove, and fire method of the event. The event and the methods are declared in a class called Counter.

```
// the delegate
.class private sealed auto autochar TimeUpEventHandler extends
[mscorlib]System.MulticastDelegate {
    .method public hidebysig specialname rtsspecialname instance void
    .ctor(class System.Object object, int32 '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, class System.Object object)
    runtime managed {}
    .method public hidebysig newslot virtual instance void EndInvoke(class
    [mscorlib]System.IAsyncResult result) runtime managed {}
}

// the class that declares the event
.class public auto autochar Counter extends [mscorlib]System.Object {
```

```
// field to store the handlers, initialized to null
.field private class TimeUpEventHandler timeUpEventHandler

// the event declaration
.event TimeUpEventHandler startStopEvent {
    .addon instance void add_TimeUp(class TimeUpEvent 'handler')
    .removeon instance void remove_TimeUp(class TimeUpEvent 'handler')
    .fire instance void fire_TimeUpEvent()
}

// the add method, combines the handler with existing delegates
.method public hidebysig virtual specialname instance void
    add_TimeUp(class TimeUpEventHandler 'handler') {
    .maxstack 4
    ldarg.0
    dup
    ldfld class TimeUpEventHandler Counter::timeUpEventHandler
    ldarg 'handler'
    call class[mscorlib]System.Delegate
    [mscorlib]System.Delegate::Combine(class [mscorlib]System.Delegate,
    class [mscorlib]System.Delegate)
    castclass TimeUpEventHandler
    stfld class TimeUpEventHandler Counter::timeUpEventHandler
    ret
}

// the remove method, removes the handler from the multicast delegate
.method virtual public specialname void remove_TimeUp(class
    TimeUpEventHandler 'handler') {
    .maxstack 4
    ldarg.0
    dup
    ldfld class TimeUpEventHandler Counter::timeUpEventHandler
    ldarg 'handler'
    call class[mscorlib]System.Delegate
    [mscorlib]System.Delegate::Remove(class [mscorlib]System.Delegate, class
    [mscorlib]System.Delegate)
    castclass TimeUpEventHandler
    stfld class TimeUpEventHandler Counter::timeUpEventHandler
    ret
}

// the fire method
.method virtual family specialname void fire_TimeUpEvent() {
    .maxstack 3
    ldarg.0
    ldfld class TimeUpEventHandler Counter::timeUpEventHandler
    callvirt instance void TimeUpEventHandler::Invoke()
    ret
}
} // end of class Counter
```

15 Exception Handling

This section is a brief summary of the CLI exception handling. A more detailed description of exception handling can be found in the [Architecture specification](#).

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

15.1 SEH Blocks

A Structured Exception Handling (SEH) block, called `<sehBlock>` in the grammar, may appear in a method as shown by the following grammar:

	Section
<code><methodBodyItem> ::=</code>	
<code><sehBlock></code>	
...	11.5.3

A SEH block consists of a protected block (try block), and one or more handlers, called `<sehClause>` in the grammar.

```
<sehBlock> ::=
    <tryBlock> <sehClause> [<sehClause>*]
```

15.1.1 Protected Blocks

A protected block is declared with the `.try` directive. There are two ways to define a protected block.

The first way simply uses a scope block (see section 11.5.6) after the `.try` directive that contains the instructions to be protected.

In the second way, the protected instructions are enclosed by two labels. The first label is defined at the first instruction to be protected, while the second label is defined at the first instruction that does not need to be protected, i.e. after the last instruction that needs to be protected.

There is yet another way permitted by the syntax, but only to be used by disassemblers for enabling round tripping of code. Instead of the labels, the addresses the labels represent may be used.

The three ways are summarized in the grammar below:

	Descriptions
<code><tryBlock> ::=</code>	
<code>.try <int32> to <int32></code>	For disassembler only
<code>.try <label> to <label></code>	Second label is exclusive, pointing to first instruction after the try block; both labels must be already defined in the preceding code
<code>.try <scopeBlock></code>	<code><scopeBlock></code> contains the instructions to be protected

Protected blocks may be nested inside each other. Each protected block has one and only one handler associated with it. If multiple handlers are needed to cover a certain set of instructions nested protected blocks need to be used.

There are two ways to leave a protected block, by using the **leave** instruction or with an exception. In particular, a protected block may not be left with a branch or **ret** instruction, nor are tail calls or jumps allowed inside a protected block.

The **leave** instruction takes a code label that points to the address of the next instruction to execute after the protected block is left. If protected blocks are nested, a single **leave** instruction can be used to leave all protected blocks at once.

If an exception occurs inside the protected block, the CLI will search for an appropriate handler and, if such a handler can be found, transfer control to the handler.

Note: The assembler *ilasm* requires labels used to specify a protected block to be declared before the **.try** directive.

15.1.2 Handlers

There are four kinds of handlers for protected blocks. A single protected block can have exactly one handler associated with it. If more than one handler needs to be used, nested protected blocks need to be declared. There are four different kinds of handlers:

1. A **finally** handler which is always executed whenever the protected block is left, regardless of whether by normal control flow or with an exception.
2. A **fault** handler which is always executed if an exception occurs, but not on completion of normal control flow.
3. A type-filtered handler (**catch**) that handles any exception of a specified class or any of its sub-classes.
4. A user-filtered handler (**filter**) that runs a user-specified set of CIL instructions to determine whether the exception should be handled or not.

The handlers are specified by the following grammar.

```
<sehClause> ::=
    catch <typeReference> <handlerBlock>
  | fault <handlerBlock>
  | filter <int32> <handlerBlock> /* for round tripping only */
  | filter <label> <handlerBlock>
  | finally <handlerBlock>
```

When an exception occurs, the EE will fix the state of the thread that caused the exception, so that the call and evaluation stacks will be kept unmodified. These stacks can be inspected by handlers to determine the cause of an exception. The EE will then try to find an exception handler which matches the object being thrown..

There are two classes of handlers, exception resolving handlers and exception observing handlers. Exception resolving handlers may, but do not need to, resolve an exception so that normal control flow may continue. **catch** and **filter** handlers are exception resolving handlers. **finally** and **fault** handlers only observe exceptions in the sense that they show behavior when an exception occurs but do not resolve the exception.

The handlers will be visited starting at the deepest nested one and then going to the enclosing one, step by step. If a method does not define an appropriate handler that resolves the exception, the next method on the class stack will be inspected. However, this process does not modify the call or evaluation stack fixed when the exception occurred.

What if we reach the last method on the call stack without finding a handler? The behavior in this case, at the moment, is implementation-defined. It may depend upon the kind of thread that threw the exception:

- main thread (ie corresponding to the main method in a managed image)
- user-created, managed threads
- threads which started life in unmanaged code, and transitioned into managed code paths

- a task that was enqueued to be run by a thread in the pool of managed worker threads
- the finalizer thread (where the thread that created the objects being finalized may well have ceased to exist)

Moreover, the situation is complicated by any user-registered handlers that field otherwise uncaught exceptions, and which version of the CLI image is being run (eg, a regular, retail build or a *developer* edition that supports more careful checking). Possible options for what to do on an unhandled exception include: emit a stack trace to the console, or a file; attach a debugger to the thread; prompt the user for appropriate action; write an event to a system-supplied ‘event log’, etc. These issues will be resolved during Beta-2.

The following grammar shows a handler block:

<code><handlerBlock> ::=</code>	Description
<code>handler <int32> to <int32></code>	For disassembler only
<code> handler <label> to <label></code>	Second label is exclusive, pointing to first instruction after the handler block; both labels must be already defined in the preceding code
<code> <scopeBlock></code>	<code><scopeBlock></code> contains the instructions of the handler block

In the above grammar, the labels enclose the instructions of the handler block. Alternatively, the handler block is just a scope block. The option with the integers is intended for the round tripping use only.

Each handler has a specific instruction that is used to exit the handler. It is illegal to exit a handler with an instruction other the specific instruction. Specifically, it is illegal to return from a handler or branch outside a handler. It is also illegal to execute a tail call from a handler or jump to a different method from a handler. See the Architecture specification for details.

It is also possible that the handler itself causes a new exception. Such an exception will exit the handler and continue the search for another appropriate handler.

Note: The assembler *ilasm* requires labels used to specify a protected block to be declared before the declaration of the handlers.

The following sections add more details to each type of exception handler.

15.1.2.1 Finally Handler

As noted above, the **finally** handler is executed whenever its protect block is left. A **finally** handler may be used to execute code that must always be executed and is usually code used to clean up state, like closing a file.

The **finally** handler is exited with the **endfinally** instruction. If the protected block was left with a **leave** instruction, execution will resume with the instruction indicated by the label used with the **leave** instruction after executing any enclosing **finally** clauses. Otherwise, if the protected block was left with an exception, the EE will continue to search for a resolving handler.

Note: **endfinally** and **endfault** are synonyms for each other and represent the same instruction.

Example:

The following code uses scope blocks to mark the protected block and handler.

```
.try {  
    // protected instructions  
    leave exitTry  
} finally {  
    // instructions in finally handler
```

```
        endfinally
    }
```

15.1.2.2 Fault Handler

The **fault** handler is similar to the **finally** handler, but will only be executed if an exception occurred. If the protected block was left with the **leave** instruction, the **fault** handler will be skipped.

A **fault** handler may be exited with the **endfault** instruction. The **endfault** instruction will let the EE continue to search for a resolving handler.

Note: **endfinally** and **endfault** are synonyms for each other and represent the same instruction.

Example:

The following code uses labels to mark the protected block and handler.

```
.method public static void m() {
    startTry:
        // protected instructions
        leave exitSEH
    endTry:
    startFault:
        // instructions to be executed in case of an fault
        endfault
    endFault:
        // the SEH declaration
    .try startTry to endTry fault handler startFault to endFault
    exitSEH:
        // unprotected instructions
}
```

15.1.2.3 Type-Filtered Handler

If a type-filtered handler is associated with a protected block and the protected block is left with an exception, then the EE will check whether the type of the exception objection is equal to or a subtype of the type the type-filtered handler expects.

If the type matches, the code associated with the handler will be executed. Otherwise, the EE will continue to search for the next handler.

A type-filter is declared using the **catch** keyword. Following the **catch** keyword is a reference to the type of exception object that the handler expects. After the handler block follows similar to other handlers.

When the type-filtered handler is entered, the EE will automatically push the exception object onto the stack. The handler may inspect this exception object and do operations on it. However, the handler is responsible for popping the exception object from the stack.

A type-filtered handler may be exited in one of two ways, in addition to the third way of causing a new exception. The first way is using a **leave** instruction. Similar to the case with protected blocks, the **leave** instruction takes a label as part of the instruction that points to the next instruction to be executed after the exception after executing any enclosing **finally** clauses. The next instruction must be outside the protected block from which the exception originated. It is not possible to resume execution using a type-filtered handler.

When the **leave** instruction is executed, the EE will treat the exception as resolved and restore the normal condition in the system.

The second way to exit a type-filtered handler is by not resolving the exception. This is done with the **rethrow** instruction. The **rethrow** instruction throws the exception that caused the handler to be executed again. The **rethrow** instruction expects the exception object on the stack when it is

executed. After the **rethrow** instruction is executed the EE will continue its search for a handler that resolves the exception.

Example:

```
.try {  
    // protected instructions  
    leave    exitSEH  
} catch [mscorlib]System.FormatException {  
    // pop the exception object (or inspect it)  
    pop  
    // other handler instructions  
    leave    exitSEH    // leave the catch  
}
```

15.1.2.4 User-Filtered Handler

While for type-filtered handler the EE checks whether the handler should be executed, a user-filtered handler provides its own code that checks whether the exception should be handled by a particular handler.

The user-filtered handler is declared using the **filter** syntax and consists of two parts. The first part is a label that points to code that checks whether the exception should be handled. The second part is the handler block itself.

The user defined filter is a block of code inside the method that contains the protected block, even though the filter may call another method. The EE will automatically push the exception object onto the stack for the user defined filter for inspection. However, the filter is responsible itself to **pop** the exception object from the stack. The filter may also inspect the state of the evaluation stack and local variables or other data that will help it to make the decision. When the filter has decided whether to accept the exception or not, it needs to return using the **endfilter** instruction.

The **endfilter** instruction expects an argument on the stack that defines whether this method will handle the exception or not. In general there three answers, but the first version of the CLI only honors two of those answers.

The two possible values are:

- **EXCEPTION_CONTINUE_SEARCH** (= 0): Indicates that the handler cannot process this exception and that the EE should offer the exception to the next handler on the list.
- **EXCEPTION_EXECUTE_HANDLER** (= 1): Indicates that the handler can process the exception and that the EE should halt the search for any other handlers and use the code at this handler's handler offset to process the exception.

Readers familiar with exception handling will know that the third answer is **EXCEPTION_CONTINUE_EXECUTION**, which instructs the EE to ignore the exception and continue with the next instruction after the instruction that caused the exception in the protected block. This third possibility may be supported in future versions of the CLI.

If the user-defined filter causes a new exception, which *escapes* the filter block (ie, it is not caught by a handler inside the filter), CLI will ignore that exception. The result of the filter is made to appear as it having returned **EXCEPTION_CONTINUE_SEARCH**

If the answer of the filter was **EXCEPTION_EXECUTE_HANDLER**, the EE terminates the first pass of exception handling, and arranges that the second pass will give control to the handler block associated with the user filtered handler. This handler block is very similar to the handler block of a type-filtered handler. The EE will automatically push a reference to the exception object onto the stack, which the handler must **pop** itself from the stack.

Similar to the handler block of a type-filtered exception handler, the handler is exited either with a **leave** instruction, or with a **rethrow** instruction. The handler is also exited if it causes a new exception itself.

If the **leave** instruction is used, the EE will treat the exception as resolved and restore normal conditions. The **leave** instruction takes a label that points to the address of the next instruction to execute after executing any enclosing **finally** clauses. This instruction must be outside the protected block that caused the exception.

If the handler block is exited with a **rethrow** instruction, the EE will throw the same exception again and continue its search. The **rethrow** instruction expects an object on the stack, which will be treated as the exception to throw.

Example:

```
.method public static void m () {
    br start          // jump over user defined filter
check_exception:
    // this is the user defined filter
    pop              // pop exception object or inspect
    // other filter instructions
    ldc.i4.1         // EXCEPTION_EXECUTE_HANDLER
    endfilter        // return answer to EE
    // end of user defined filter
start:
    .try {
        // protected instructions
        leave exitSEH
    } filter check_exception {
        pop          // pop exception object or inspect
        // other handler instructions
        leave exitSEH // leave the handler
    }
}
```

15.2 Throwing an Exception

An exception is thrown using the **throw** instruction. The throw instruction expects the exception object on the stack. This object can be an instance of any class, but typically is an instance of `System.Exception` or one of its subclasses.

CLS compliant languages must throw an exception object that is an instance of `System.Exception` or one of its subclasses.

The Architecture Specification describes in detail how an exception is thrown. In summary, the CLI exception system does two passes on the call stack to handle user-filtered exceptions. The first pass will search for user defined filters or type filters and if one is found execute that filter. During this pass no **finally** or **fault** handlers are executed. The second pass calls the **finally** and **fault** handlers, and then the selected filter's handler. This behavior gives the user defined filter more flexibility for its decisions.

Example:

The following code throws the `NonPositiveNumberException`, a nested class that declares the custom exception inside the class `CounterTextBox`.

```
// create the an instance of the exception
newobj instance void CounterTextBox/NonPositiveNumberException::.ctor()
throw          // throw the exception
```

16 Declarative Security

The CLI has a sophisticated security system. More information about the security system can be found [Security Guide](#). Programs may specify what permission they need in order to run correctly. However, this feature is currently not available for CIL assembly code. It is planned for Beta-2.

The following grammar is for round tripping use only. It will be used by *ildasm* to display code compiled from other languages.

```
<securityDecl> ::=  
    .capability <secAction> = ( <bytes> )  
    | .permission <secAction> <typeReference> ( <nameValPairs> )
```

In **.permission**, <typeReference> specifies the permission class and <nameValPairs> show the settings. In **.capability** the bytes show the serialized version of the security settings. The following values for <secAction> can be displayed by ILDASM:

<secAction> ::=	Description
assert	Assert permission so callers don't need it.
demand	Demand permission of all callers.
deny	Deny permission so checks will fail.
inheritcheck	Demand permission of a subclass.
linkcheck	Demand permission of caller.
permitonly	Reduce permissions so check will fail.
prejitdeny	Persisted grant set at prejit time.
prejitgrant	Persisted denied set at prejit time.
reqmin	Request minimum permissions to run.
reqopt	Request optional additional permissions.
reqrefuse	Refuse to be granted these permissions.
request	Hint that permission may be required.

```
<nameValPairs> ::= <nameValPair> [, <nameValPair>]*
```

```
<nameValPair> ::= <SQSTRING> = <SQSTRING>
```

17 Custom Attributes

Custom attributes add user-defined annotations to the metadata. Custom attributes allow an instance of a type to be stored with any element of the metadata. This mechanism can be used to store application specific information at compile time and access it either at runtime or when another tool reads the metadata. While any user-defined type can be used as an attribute it is expected that most attributes will be instances of types whose parent is `System.Attribute`. The CLI predefines some attribute types and uses them to control runtime behavior. Some languages predefine attribute types to represent language features not directly represented in the CTS. Users or other tools are welcome to define and use additional attribute types.

Custom attributes are declared using the directive **.custom**. Followed by this directive is the method declaration for a class constructor, optionally followed by a <bytes> in parentheses:

```
<customDecl> ::=  
    <ctor> [ = ( <bytes> ) ]
```

For example:

```
.custom instance void myAttribute::.ctor(bool, bool) = ( 01 00 00 01 00 00 )
```

Custom attributes can be attached to *any* item in metadata, except a custom attribute itself. Commonly, custom attributes are attached to assemblies, modules, classes, interfaces, value types, methods, fields, properties and events (the custom attribute is attached to the immediately preceding declaration)

The <bytes> item is not required if the constructor takes no arguments. In these cases, all that matters is the presence of the custom attribute.

If the constructor takes parameters, then you must specify their values in the <bytes> item. The format for this 'blob' is defined in the [Metadata API specification](#).

Example:

The following example shows a class that is marked with the `System.SerializableAttribute` and a method that is marked with the `System.Runtime.Remoting.OneWayAttribute`. The keyword **serializable** corresponds to the `System.SerializableAttribute`.

```
.class public MyClass {  
    .custom [mscorlib]System.SerializableAttribute  
    .method public static void main() {  
        .custom [mscorlib]System.Runtime.Remoting.OneWayAttribute  
        ret  
    }  
}
```

17.1 CLS Conventions: Custom Attribute Usage

In order to allow languages to provide a consistent view of custom attributes across language boundaries, a set of conventions is very helpful. The Base Class Library provides support for several different conventions defined by the CLS:

- Attributes must be instances of the class `System.Attribute`, which provides static methods to test whether attributes exist on a metadata element and retrieve their value if so.
- The use of a particular attribute class may be restricted in various ways by placing an attribute on the attribute class. The `System.AttributeUsageAttribute` is used to specify these restrictions:
 - What kinds of constructs (types, methods, assemblies, etc.) may have the attribute applied to them. By default, instances of an attribute class can be applied to any construct. This is specified by setting the value of the `ValidOn` property of `System.AttributeUsageAttribute`. Several constructs may be combined.
 - Multiple instances of the attribute class 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 parent and those explicitly applied to the child type. If multiple instance are not allowed, then an attribute of that type applied directly to the child overrides the attribute supplied by the parent. This is specified by setting the `Inherited` property of `System.AttributeUsageAttribute` to the desired value.

Notice that, 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 *allow multiple* and *inherit* rules. The EE and reflection do not supply attributes automatically.

17.2 Attributes Used by the Runtime

The metadata engine implements two sorts of Custom Attributes, called (genuine) Custom Attributes (CA), and *Pseudo Custom Attributes* (PCA). CAs and PCAs are treated differently, as follows:

- A CA is stored directly into the metadata. The “blob” which holds its defining data is not checked or parsed. That “blob” can be retrieved later.
- A PCA is recognized because its name is one of a handful of hard-wired PCAs. The engine parses its *blob* and uses this information to set bits and/or fields within the metadata tables. The engine then totally discards the blob, such that it cannot be retrieved later.

PCAs therefore serve to capture *user directives*, using the same familiar syntax the compiler provides for regular CAs, but these user directives are then stored into the more space-efficient form of metadata tables. Tables are also faster to check at runtime than (genuine) CAs. An example of a PCA is the `SerializableAttribute`. If the compiler specifies this PCA for a class, then the metadata engine simply defines the class to be **serializable**.

Many CAs are invented by higher layers of software. They are stored and returned by the EE, without knowing or caring what they mean. But all PCAs, plus a handful of regular CAs are of special interest to compilers and to the Runtime. An example of such distinguished CAs is `System.Reflection.DefaultMemberAttribute`. This is stored in metadata as a regular CA blob, but reflection uses this CA when called to invoke the default member (property) for a class.

The following subsections lists all of the PCAs and distinguished CAs, where *distinguished* means that the runtime and/or compilers pay direct attention to them.

Note that it is a Frameworks design guideline that all CAs should be named to end with “Attribute”. The runtime does not care about this convention.

For further details on these special CAs, consult the Base Class Library, or appropriate specs in the area that each covers.

17.2.1 Pseudo Custom Attributes

The Metadata engine checks for the following CAs, as part of the processing of the `DefineCustomAttribute` method. The check is solely on their name, e.g. “`DllImportAttribute`”. If a name match is found, the metadata engine parses the blob argument and sets bits and/or fields within the metadata tables. It then discards the blob. All these attributes are part of the `System` namespace, and should indicate the *ilasm* syntax, if any, they correspond to.

Attribute	Description
<code>NonSerializedAttribute</code>	
<code>SerializableAttribute</code>	

17.2.2 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
<code>AttributeUsageAttribute</code>	Used to specify how an attribute is intended to be used.
<code>ObsoleteAttribute</code>	Indicates that an element is not to be used.
<code>CLSCompliantAttribute</code>	Indicates whether or not an element is declared to be CLS compliant through an instance field on the attribute object.

17.2.3 Custom Attributes for JIT Compiler and Debugger

The CAs that control runtime behavior of the JIT-compiler and the debugger can be found in `System.Diagnostics`.

Attribute	Description
DebuggableAmbivalentAttribute	
DebuggableAttribute	
DebuggerHiddenAttribute	
DebuggerStepThroughAttribute	

17.2.4 Custom Attributes for Reflection

The following CA in `System.Reflection` is used by reflection's invoke call:

Attribute	Description
DefaultMemberAttribute	Defines the member of a type that is the default member used by <code>InvokeMember</code> .

17.2.5 Custom Attributes for Remoting

CAs that affect behavior of remoting can be found in `System.Runtime.Remoting`.

Attribute	Description
ContextAttribute	Root for all context attributes.
OneWayAttribute	
Synchronization	
ThreadAffinity	Refinement of Synchronized Context.

17.2.6 Custom Attributes for Security

The following CAs affect the security checks performed upon method invocations at runtime.

The CAs in the following table can be found in `System.Security`.

Attribute	Description
DynamicSecurityMethodAttribute	
SuppressUnmanagedCodeSecurityAttribute	
UnverifiableCodeAttribute	

The CAs in this table can be found in `System.Security.Permissions`.

Attribute	Description
CodeAccessSecurityAttribute	This is the base attribute class for declarative security using custom attributes.
EnvironmentPermissionAttribute	Custom attribute class for declarative security with <code>EnvironmentPermission</code> .
FileDialogPermissionAttribute	Custom attribute class for declarative security with <code>FileDialogPermission</code> .

FileIOPermissionAttribute	Custom attribute class for declarative security with FileIOPermission.
IsolatedStorageFilePermissionAttribute	Custom attribute class for declarative security with IsolatedStorageFilePermission.
IsolatedStoragePermissionAttribute	Custom attribute class for declarative security with IsolatedStoragePermission.
PermissionSetAttribute	Allows declarative security actions to be performed against permission sets rather than individual permissions.
PrincipalPermissionAttribute	A PrincipalPermissionAttribute can be used to declaratively demand that users running your code belong to a specified role or have been authenticated.
PublisherIdentityPermissionAttribute	Custom attribute class for declarative security with PublisherIdentityPermission.
ReflectionPermissionAttribute	Custom attribute class for declarative security with ReflectionPermission.
RegistryPermissionAttribute	
SecurityAttribute	This is the base attribute class for declarative security from which CodeAccessSecurityAttribute is derived.
SecurityPermissionAttribute	
SiteIdentityPermissionAttribute	Custom attribute class for declarative security with SiteIdentityPermission.
StrongNameIdentityPermissionAttribute	Custom attribute class for declarative security with StrongNameIdentityPermission.
UIPermissionAttribute	Custom attribute class for declarative security with UIPermission.
ZoneIdentityPermissionAttribute	Custom attribute class for declarative security with ZoneIdentityPermission.

17.2.7 Custom Attributes for TLS

A CA that denotes a TLS (thread-local storage, see section 12.3.3) field can be found in `System`.

Attribute	Description
ThreadStaticAttribute	Provides for type member fields that are relative for the thread.

17.2.8 Custom Attributes for the Assembly Linker

The following CAs are used by the *al* tool to transfer information between modules and assemblies (they are temporarily attached to a TypeRef to a class called `AssemblyAttributesGoHere`) then merged by *al* and attached to the assembly. These attributes can be found in `System.Runtime.CompilerServices`.

Attribute	Description
AssemblyCultureAttribute	
AssemblyDelaySignAttribute	
AssemblyKeyFileAttribute	

Attribute	Description
AssemblyKeyNameAttribute	
AssemblyOperatingSystemAttribute	
AssemblyProcessorAttribute	
AssemblyVersionAttribute	

17.2.9 Attributes Provided for Interoperation with COM

There are a number of custom attributes for interoperation with COM 1.x and classical COM. These attributes are located under `System.Runtime.InteropServices` in the class hierarchy. More information can also be found in the Base Class Library specification.

These attributes are shown in the following table.

Attribute	Description
ComAliasNameAttribute	Applied to a parameter or field to indicate the COM alias for the parameter or field type.
ComConversionLossAttribute	
ComEmulateAttribute	Used on a class to indicate that the class is an emulator class for another .NET Framework class.
ComImportAttribute	Used to indicate that a class or interface definition was imported from a COM type library.
ComRegisterFunctionAttribute	Used on a method to indicate that the method should be called when the assembly is registered for use from COM.
ComSourceInterfacesAttribute	Identifies the list of interfaces that are sources of events for the class.
ComUnregisterFunctionAttribute	Used on a method to indicate that the method should be called when the assembly is unregistered for use from COM.
ComVisibleAttribute	Can be applied to an individual type or to an entire assembly to control COM visibility.
DispIdAttribute	Custom attribute to specify the COM DISPID of a Method or Field.
DllImportAttribute	Used to indicate that a method is implemented as a PInvoke method in unmanaged code.
FieldOffsetAttribute	Used along with the <code>System.Runtime.InteropServices.StructLayoutAttribute.LayoutKind</code> set to explicit to indicate the physical position of each field within a class.
GuidAttribute	Used to supply the GUID of a class, interface or an entire type library.
HasDefaultInterfaceAttribute	Used to specify that a class has a COM default interface.
IdispatchImplAttribute	
ImportedFromTypeLibAttribute	Custom attribute to specify that a module is imported from a COM type library.
InAttribute	Used on a parameter or field to indicate that data should be marshaled in to the caller.
InterfaceTypeAttribute	Controls how a managed interface is exposed to COM clients (IDispatch derived or IUnknown derived).

Attribute	Description
MarshalAsAttribute	This attribute is used on fields or parameters to indicate how the data should be marshaled between managed and unmanaged code.
MethodImplAttribute	
NoComRegistrationAttribute	Used to indicate that an otherwise public, COM-creatable type should not be registered for use form COM applications.
NoIDispatchAttribute	This attribute is used to control how the class responds to queries for an IDispatch Interface.
OutAttribute	Used on a parameter or field to indicate that data should be marshaled out from callee back to caller.
PreserveSigAttribute	Used to indicate that hresult/retval signature transformation that normally takes place during Interop calls should be suppressed.
ProgIdAttribute	Custom attribute that allows the user to specify the prog ID of a .NET Framework class.
StructLayoutAttribute	Typically the runtime controls the physical layout of the data members of a class.
TypeLibFuncAttribute	Contains the FUNCFLAGS that were originally imported for this function from the COM type library.
TypeLibTypeAttribute	Contains the TYPEFLAGS that were originally imported for this type from the COM type library.
TypeLibVarAttribute	Contains the VARFLAGS that were originally imported for this variable from the COM type library.

18 CIL Instructions

18.1 Overview

This section lists and describes all CIL instructions by category. A more detailed description of the instructions can be found in the [CIL Instruction Set specification](#).

The CLI uses the model of a stack machine. Each method has an evaluation stack on which arguments to instructions are pushed. The instructions will pop these instructions and push the results onto the stack. More about the architecture of the CLI can be found in the [Architecture specification](#).

The execution of instructions can cause an exception. The possible exceptions are described in this section and the detailed list can also be found in the [CIL Instruction Set specification](#).

Any instruction can throw the general `System.ExecutionEngineException`. This exception is thrown when an error internal to the EE occurs.

The [Architecture specification](#) gives details on exceptions and rules for valid CIL sequences and verifiable code.

All instructions have an opcode that identifies them to the JIT compiler. Many instructions also accept additional data as part of the instruction. This is different from arguments to the instruction. While the arguments are on the stack, the additional data is integrated into the instruction directly following the opcode. Thus, each instruction has a certain syntax that describes how it needs to be used. A description of the integrated data for each instruction can be found in this section and a summary of the grammar in Part 5.

Some instructions have a prefix instruction associated with the them. A prefix instruction is a separate instruction with its own opcode but qualifies the following instruction in a certain way. A specific prefix instruction can only be used before selected instructions as described in the next sections. The names of all prefix instructions end with a dot (“.”).

Some instructions also have a suffix associated with them. Unlike a prefix instruction, a suffix is not a separate instruction. A suffix starts with a dot (“.”). The following sections point out which instructions have what suffixes associated with them. Often, the suffix specifies the size of the data the instruction deals with and/or includes a small constant number that is encoded efficiently using the suffix.

The general instruction syntax is as follows:

```
<instr> ::=  
    <instruction>[.ovf][.un]  
    | <prefixInstruction>
```

<instruction> specifies the desired instruction. The suffix **.ovf** specifies that the instruction checks for overflow. The suffix **.un** specifies that the instruction treats its operands on the stack as unsigned values. The optional suffixes can only be used with specific instructions as shown in the following sections.

The following grammar shows various which suffixes can be used to selected instructions.

```
<2bitNumSuffix> ::=  
    0  
    | 1  
    | 2  
    | 3
```

```
<constNumSuffix> ::=  
    4  
    | 5  
    | 6  
    | 7  
    | 8  
    | M1  
    | m1  
    | <2bitNumSuffix>  
M1 or m1 means “minus one”.
```

```
<typeSuffix> ::=  
    i  
    | i1  
    | i2  
    | i4  
    | i8  
    | r4  
    | r8  
    | ref
```

```
<exTypeSuffix> ::=  
    <typeSuffix>  
    | u  
    | u1
```

| u2
| u4
| u8

The following sections describe:

- Numeric and Logical Operations (18.2)
- Control flow instructions (18.3)
- Instructions that move data (18.4)
- Object management instructions (18.5)

18.2 Numeric and Logical Operations

Many CIL operations take numeric operands on the stack. These fall into several different categories, depending on how they deal with the types of the operands. The following operand tables summarize the legal operand types and the resulting type. 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 the CIL verifier. The types tracked by the CLI are: I4, I8, I, F, O, &, and * (see section 5.3 for the definition of & and *)

Table 1: Binary Numeric Operations

A **op** B (used for **add**, **div**, **mul**, **rem**, and **sub**, applies to all instructions unless specific instructions are specified in the table). The shaded uses are not verifiable, while items marked “-“ indicate incorrectly formed CIL sequences.

B's type A's type	I4	I8	I	F	&	O	*
I4	I4	-	I	-	& (add)	-	* (add)
I8	-	I8	-	-	-	-	-
I	I	-	I	-	& (add)	-	* (add)
F	-	-	-	F	-	-	-
&	& (add, sub)	-	& (add, sub)	-	I (sub)	-	I (sub)
O	-	-	-	-	-	-	-
*	* (add, sub)	-	* (add, sub)	-	I (sub)	-	I (sub)

Table 2: Unary Numeric Operations

Used for the **neg** instruction. All these uses of this instruction are verifiable.

Operand Type	I4	I8	I	F	&	O	*
Result Type	I4	I8	I	F	-	-	-

Table 3: Binary Comparison or Branch Operations

These return a boolean value or branch based on the top two values on the stack. Used for **beq**, **bge**, **bge.un**, **bgt**, **bgt.un**, **ble**, **ble.un**, **blt**, **blt.un**, **bne**, **bne.un**, **ceq**, **cgt**, **cgt.un**, **clt**, **clt.un**. Items marked “✓” indicate that all instructions are valid. Items marked “-” indicate invalid CIL sequences. If only a subset of instructions are permitted, the valid instructions are shown in the corresponding cell.

	I4	I8	I	F	&	O	*
I4	✓	-	✓	-	-	-	-
I8	-	✓	-	-	-	-	-
I	✓	-	✓	-	beq[.s], bne.un[.s], ceq	-	beq[.s], bne.un[.s], ceq
F	-	-	-	✓	-	-	-
&	-	-	beq[.s], bne.un[.s], ceq	-	✓ (Note)	-	✓ (Note)
O	-	-	-	-	-	beq[.s], bne.un[.s], ceq	-
*	-	-	beq[.s], bne.un[.s], ceq	-	✓ (Note)	-	✓ (Note)

Note: Except for **beq**, **bne.un** (or short versions) or **ceq** these combinations only make sense if both operands are known to be pointers to elements of the same array (CLI does not check this constraint – it’s the responsibility of the compiler to ensure it holds)

Table 4: Integer Operations

These operate only on integer types. Used for **and**, **div.un**, **not**, **or**, **rem.un**, **shl**, **shr**, **xor**. The **div.un** and **rem.un** instructions treat their arguments as unsigned integers and produce the bit pattern corresponding to the unsigned result. As described in the CLI Specification, however, the CLI makes no distinction between signed and unsigned integers on the stack. The **not** instruction is unary and returns the same type as the input. The **shl** and **shr** instructions return the same type as their first operand and their second operand must be of type U. All items marked “-” indicate incorrectly formed CIL sequences, while the others are verifiable.

	I4	I8	I	F	&	O	*
I4	I4	-	I	-	-	-	-
I8	-	I8	-	-	-	-	-
I	I	-	I	-	-	-	-
F	-	-	-	-	-	-	-
&	-	-	-	-	-	-	-
O	-	-	-	-	-	-	-
*	-	-	-	-	-	-	-

Table 5: Overflow Arithmetic Operations

These operations generate an exception if the result cannot be represented in the target data type. Used for **add.ovf**, **add.ovf.un**, **mul.ovf**, **mul.ovf.un**, **sub.ovf**, **sub.ovf.un**. The shaded uses are not verifiable, while items marked “-” indicate incorrectly formed CIL sequences.

	I4	I8	I	F	&	O	*
I4	I4	-	I	-	& add.ovf.un	-	* add.ovf.un
I8	-	I8	-	-	-	-	-
I	I	-	I	-	& add.ovf.un	-	* add.ovf.un
F	-	-	-	-	-	-	-
&	& add.ovf.un, sub.ovf.un	-	& add.ovf.un, sub.ovf.un	-	I sub.ovf.un	-	I sub.ovf.un
O	-	-	-	-	-	-	-
*	* add.ovf.un, sub.ovf.un	-	* add.ovf.un, sub.ovf.un	-	I sub.ovf.un	-	I sub.ovf.un

Table 6: Conversion Operations

These operations convert from one numeric type to another. The result type is guaranteed to be representable as the data type specified as part of the operation (i.e. the **conv.u2** instruction returns a value that can be stored in a **U2**). 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 items marked “-” indicate incorrectly formed CIL sequences.

Output Operand	I1/U1 I2/U2	I4/U4	I8	U8	I
I4	Truncate ¹	No-op	Sign extend	Zero extend	Sign extend
I8	Truncate ¹	Truncate ¹	No-op	No-op	Truncate ¹
I	Truncate ¹	Truncate ¹	Sign extend	Zero extend	No-op
F	Trunc to 0 ²	Trunc to 0 ²	Trunc to 0 ²	Trunc to 0 ²	Trunc to 0 ²
&	-	-	-	Stop GC Tracking	-
O	-	-	-	-	-
*	-	-	-	Zero extend	-

Output Operand	U	All R Types
I4	Zero extend	To Float
I8	Truncate ¹	To Float
I	No-op	To Float
F	Trunc to 0 ²	Change Precision ³
&	Stop GC Tracking	-
O	-	-
*	No-op	-

Note 1: “Truncate” means that the number is truncated (i.e. the higher-order bits are set to zero) to the desired size. If the destination type is signed, the most-significant bit of the truncated value is then sign-extended to fill the full output size. Thus, converting 257 (0x101) to I1 or U1 yields 1, but truncating 129 (0x81) to U1 yields 129 (0x81) while truncating it to I1 yields -126 (0xF...F81)

Note 2: “Trunc to 0” means that the floating point number will be converted to an integer by truncation toward zero. Thus 1.1 is converted to 1 and -1.1 is converted to -1.

Note 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 IEEE 754 “round to nearest” mode to compute the low order bit of the result.

18.3 Control Flow

The operations described in the following sections alter the normal flow of control from one CIL instruction to the next. There are three main ways to alter the control flow:

1. branch instructions
2. procedure calls
3. exceptions

Branch instructions can be further subdivided into unconditional and conditional branches. There are unary, binary, and multi-way conditional branch instructions. Branch instructions can only branch to a label within the current block of code, e.g. they cannot branch to a location outside the current method or a protected block.

Branch instructions take in addition to their operands on the stack a label as an argument to which the control flow shall be redirected if the branch condition is met.

18.3.1 Unconditional Branch Instructions

The **.s** suffix indicates that the distance can be expressed in a signed 8-bit number (distances are in bytes from the end of the current instruction). See also the **leave** and **leave.s** instructions in Section 18.3.5.

br	branch within current method or protected block
br.s	branch within current method or protected block

18.3.2 Unary Compare-and-Branch and Multi-Way Branch Instructions

These instructions branch depending on the value of the topmost stack item. The **.s** suffix indicates that the distance can be expressed in a signed 8-bit number (distances are in bytes from the end of the current instruction).

brfalse	branch if false (i.e. zero)
brfalse.s	branch if false (i.e. zero)
brinst	branch if non-null object reference
brinst.s	branch if non-null object reference
brnull	branch if null object reference
brnull.s	branch if null object reference
brtrue	branch if not false (i.e. not zero)
brtrue.s	branch if not false (i.e. not zero)
brzero	branch if zero
brzero.s	branch if zero
switch	multi-way 0-based branch depending on value on top of evaluation stack

18.3.3 Binary Compare-and-Branch Instructions

These operations compare the top two elements on the evaluation stack and branch if a specific condition is true. They can be considered abbreviations for sequences of instructions using the binary comparison instructions followed by either a **brtrue** (or **brtrue.s**) or a **brfalse** (or **brfalse.s**) instruction. The **.s** suffix indicates that the distance can be expressed in a signed 8-bit number (distances are in bytes from the end of the current instruction).

beq	based on ceq and brtrue
beq.s	based on ceq and brtrue.s
bge	based on clt and brfalse
bge.s	based on clt and brfalse.s
bge.un	based on clt.un and brfalse
bge.un.s	based on clt.un and brfalse.s
bgt	based on cgt and brtrue
bgt.s	based on cgt and brtrue.s
bgt.un	based on cgt.un and brtrue
bgt.un.s	based on cgt.un and brtrue.s
ble	based on cgt and brfalse
ble.s	based on cgt and brfalse.s

ble.un	based on cgt.un and brfalse
ble.un.s	based on cgt.un and brfalse.s
blt	based on clt and brtrue
blt.s	based on clt and brtrue.s
blt.un	based on clt.un and brtrue
blt.un.s	based on clt.un and brtrue.s
bne.un	based on ceq.un and brfalse
bne.un.s	based on ceq.un and brfalse.s

18.3.4 Procedure Call and Related Instructions

These instructions move the flow of control to another procedure. See also **callvirt** and **ldvirtftn** in section 18.5.

call	call a method specified by type, name, and signature
calli	call a method specified by function pointer
jmp	branch with current arguments to another method
ldftn	create function pointer from type, name, and signature
ret	return from the current method, possibly returning a value
tail.	Convert subsequent instruction to a tail call version (drop current stack frame before call)

18.3.5 Exception Handling

The following instructions specify the control flow of exceptional code. The **.s** suffix indicates that the distance can be expressed in a signed 8-bit number (distances are in bytes from the end of the current instruction).

endfault	mark end of a fault handler
endfilter	mark end of a filter handler
endfinally	mark end of a finally handler
leave	unconditional branch that may exit a try block
leave.s	unconditional branch that may exit a try block
rethrow	throw exception again (out of a catch handler)
throw	throw an exception

18.3.6 Other Control Flow Instructions

The instructions in this section are considered to be instructions that belong to the group of control flow instructions, however do not belong to any of the above sections.

break	invoke debugger if attached
ckfinite	check that the top of stack is a finite floating point number, generating a <code>System.ArithmeticException</code> if the value is a NaN or an infinity
nop	ignored

18.4 Moving Data

The instructions presented in this sections may be used to move data from one location to another.

Method arguments and locals are numbered in increasing order starting with 0, unless explicit values are assigned (see section 11.5.6). Argument 0 is the *this* pointer for instance and virtual methods. Valid CIL requires that any argument or local is used consistently, always containing either an integer, floating point number, class, or instance of a specific value class.

- arglist** returns handle to current argument list on stack (for **vararg** methods, see section 11.5.7)
- cpblk** copy block of data from one part of memory to another (*not verifiable*).
- dup** duplicate top element of evaluation stack
- initblk** zero block of data in memory (*not verifiable*).
- ldarg, ldarg.<2bitNumSuffix>, ldarg.s**
load argument onto evaluation stack. **ldarg.0** through **ldarg.3** are short encodings for accessing the first four arguments. **ldarg.s** is used for arguments numbered 4 through 255.
- ldarga, ldarga.s**
load address of an argument. **ldarga.s** is used for arguments 0 through 255
- ldc.i4, ldc.i4.<constNumSuffix>, ldc.i4.s**
load constant as a 4-byte signed integer onto the evaluation stack. There is a short encoding for constants -1 (denoted "m1") through 8. **ldc.i4.s** is for encoding constants that fit, signed, in one byte.
- ldc.i8** load an 8-byte integer constant onto the evaluation stack
- ldc.r4** load a 32-bit floating point constant onto the evaluation stack
- ldc.r8** load a 64-bit floating point constant onto the evaluation stack
- ldind.<exTypeSuffix>**
load indirect through a pointer, type of data loaded is specified as a suffix to the instruction (*not verifiable*).
- ldloc, ldloc.<2bitNumSuffix>, ldloc.s**
load value of a local variable (numbered from 0) onto the evaluation stack. There are special small encodings for locals 0 through 3. **ldloc.s** is used for locals 4 through 255.
- ldloca, ldloca.s**
load address of local variable onto stack; **ldloca.s** is used for locals 0 through 255
- ldnull** load the null object reference
- localloc** allocate space for additional locals, dynamically. The evaluation stack must be empty when this instruction is executed.
- pop** remove the top item from the evaluation stack
- starg, starg.s**
store top of evaluation stack into an argument; **starg.s** is used for arguments 0 through 255
- stind.<typeSuffix>**
store the top of the evaluation stack into the address specified by a pointer, which is the second item on the stack. The type of data stored is specified by the suffix to the instruction. Valid CIL requires that the suffix corresponds to the basic type (integer, float, object) of the value on the top of the stack. (*Not verifiable*).
- stloc, stloc.<2bitNumSuffix>, stloc.s**
store the top of the evaluation stack into a local variable. There are short encodings for locals 0 through 3. **stloc.s** is used for locals 4 through 255.
- unaligned.** Indicates that the subsequent operation may reference data that is not aligned to the natural size of the target machine. Valid only before **ldind.<exTypeSuffix>**, **stind.<typeSuffix>**, **ldfld**, **stfld**, **ldobj**, **stobj**, **initblk**, or **cpblk**.

volatile. Indicates that the subsequent operation may reference data that is read or written asynchronously. Valid only before **ldind**, **stind**, **ldfld**, **ldsfld**, **stfld**, **stsfld**, **ldobj**, **stobj**, **initblk**, or **cpblk**.

18.5 Object Management

The CIL instruction set has direct support for creating objects, zero-based one-dimensional arrays, typed-references, and strings. It also supports casting between object types with runtime type checking, copying instances of value types, accessing fields of classes and value types, and converting between the boxed and unboxed forms of value types.

box	convert an unboxed (copy-by-value) instance of a value type into the boxed (copy-by-reference) version by allocating a <code>System.Object</code> on the heap.
callvirt	call a virtual method given an object and arguments on the evaluation stack and the types, name, and signature of the virtual method as direct arguments. If the object is null a <code>System.NullReferenceException</code> is thrown.
castclass	convert an object to any of its parent classes, specified as part of the instruction, or raise <code>System.InvalidCastException</code> .
cpobj	copy an instance of a value type from one location to another. The top of the evaluation stack points to the source object and the next-to-top points to the destination.
initobj	zero the contents of a value type. The top of the stack is the address of the instance to be zeroed.
isinst	the top of the stack must be an object reference and a type is passed as a direct argument. If the top of the stack is an instance of that type it is left on the stack, otherwise it is replaced by a null object reference. In either case, it is guaranteed that the top of the stack can be considered to be of the specified type.
ldelem	ldelem.<exTypeSuffix> load an element out of a zero-based, one-dimensional array, with range and type checking. The type of the array must match the suffix of the instruction or a <code>System.ArrayTypeMismatchException</code> is raised. An out of range subscript results in a <code>System.IndexOutOfRangeException</code> , while an attempt to access an element of the null array results in a <code>System.NullReferenceException</code> .
ldelema	load the address of an element of a zero-based, one-dimensional array, with range and type checking. The index is the top operand on the stack, the array is the second on the stack. The type is expected as a direct argument to the instruction.
ldfld	load the contents of a field of an object.
ldflda	load the address of a field of an object
ldlen	load the length of a zero-based, one-dimensional array.
ldobj	load an instance of a value type onto the evaluation stack. The top of the evaluation stack is a pointer to the instance.
ldsfd	load the contents of a static field of a class onto the evaluation stack.
ldsfla	load the address of a static field of a class onto the evaluation stack.
ldstr	load a literal instance of <code>System.String</code> onto the evaluation stack.
ldtoken	load a token representing a type, field, or method onto the evaluation stack. The instruction returns an unmanaged pointer type I (32- or 64-bits, depending on platform) and can be used for efficient type comparisons, method lookup, etc.
ldvirtftn	load a function pointer that references the implementation, in a given object, of a particular virtual method. This function pointer can then be used with the calli instruction. The method is computed at the time the ldvirtftn instruction is executed, not when the calli occurs (i.e. it returns a function pointer, not a C++ “pointer to virtual function”).
mkrefany	make a typed reference (runtime typed pointer to memory). A pointer to memory is passed on the top of the stack, and the type of data stored at that location is passed as part of the instruction itself. Verification requires that the type specified in the instruction and the type of the pointer

	match, and verification will fail if it cannot show this to be true. Thus, only stylized uses of mkrefany are verifiable.
newarr	allocate and zero-initialize a zero-based, one-dimensional array. The top of the evaluation stack specifies the total number of elements in the array, and the instruction itself specifies the data type of the elements.
newobj	allocate and initialize an object. The initializer (see Section 7.6.5) to call is specified as part of the instruction itself. The arguments, if any, to that initializer must be on the evaluation stack.
refanytype	given a typed reference on the evaluation stack, extract the type of the pointer from it. This will be the same value that would have been computed by a ldtoken instruction given the type used when the typed reference was created using mkrefany .
refanyval sizeof	given a typed reference on the evaluation stack, extract the pointer from it. See also mkrefany . returns the size in bytes of an instance of a value type. The value type is specified as part of the instruction.
	stelem.<typeSuffix> store an item into an element of an array, with type and range checking. The type is specified by the suffix of the instruction and (for stelem.ref) the object itself; any mismatch results in a <code>System.ArrayTypeMismatchException</code> . The top of the stack contains the value to be stored. The second item on the stack is the index, an unsigned integer. A <code>System.IndexOutOfRangeException</code> will be thrown if the index is larger than the size of the array. The third item on the stack is the array itself, which will result in a <code>System.NullReferenceException</code> if it is null. To store an unboxed value type into an array, use ldlema and stobj rather than stelem.<typeSuffix> .
stfld	store the top of the stack into a field of an object. The item below the top of stack must be an object reference or a pointer to an unboxed value type instance.
stobj	store an unboxed instance of a value type (on the top of the stack) at the address specified by the pointer below it on the evaluation stack.
stsfld	Store the top of the evaluation stack into a static field, specified as part of the instruction.
unbox	Return a pointer to the unboxed instance of a value type that is stored within the boxed instance on the top of the evaluation stack. The type of the boxed instance (from the object on the stack) must match the type desired (from the instruction stream) or a <code>System.InvalidCastException</code> is thrown. The result is a by-ref (managed pointer), <i>not</i> a copy of the data in the object. A copy can be made by using ldobj to copy the data onto the evaluation stack or cpobj to copy into another location that has already been computed.

18.6 Annotations

Annotations are ignored by all the CLI tools that convert CIL into managed native code. Their opcodes are reserved and their formats specified for completeness only. More information on these instruction can be found in the [CIL Instruction Set specification](#).

ann.call
ann.catch
ann.data
ann.data.s
ann.dead
ann.def
ann.hoisted
ann.hoisted_call
ann.lab
ann.live
ann.phi

ann.ref
ann.ref.s

19 Overview of File Format Extensions to COFF

This clause specifies a file format for CLI components that is based on, and is a strict extension of, the current Microsoft Windows Portable Executable (PE) and Common Object File Format (COFF). This extended PE/COFF 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.

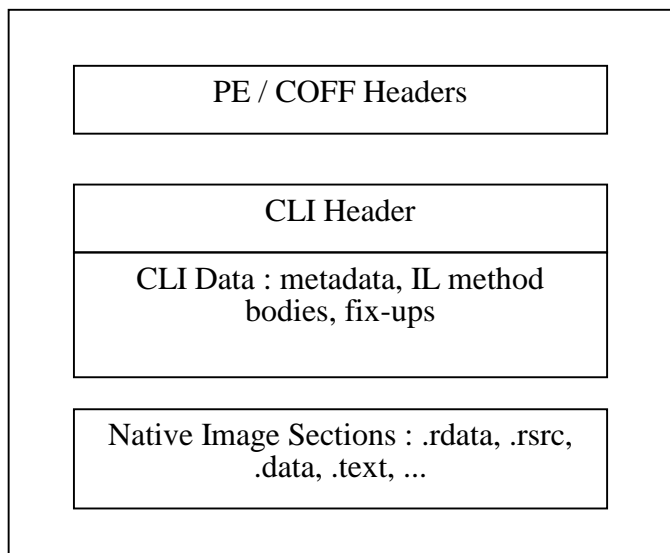
This section provides a brief overview of and motivation for the design approach, including a summary of requirements and constraints. Subsequent sections present the technical specifications as a delta from the current Windows PE/COFF file format, in sufficient detail that a tool or compiler can use the specifications to emit valid runtime images.

The entire document assumes familiarity with the current PE/COFF structure and terminology. For more information, refer to the “Microsoft Portable Executable and Common Object File Format Specification”

19.1 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:

- Standard PE/COFF headers, with specific guidelines on how field values should be set in a runtime file
- A runtime header that contains all of the runtime specific data entries. Currently, the runtime header is read-only and may be placed in any read-only section
- Any of the data one currently finds in a valid PE/COFF image, including imports/exports, data, and code. This spec calls out specific areas where we use this data in the runtime.



The image is a full PE/COFF file image. The normal PE/COFF headers apply. The runtime header is found using directory entry `IMAGE_DIRECTORY_COR_DESCRIPTOR` in the PE header. The runtime header in turn contains the address and locations of the runtime data 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.).

While the bulk of the file format is generated by tools directly, the metadata portion is emitted through an API that abstracts the tools from the underlying data structures. This is in part because the data structures are many and complex, having been tuned for performance and size, and because we want to be able to do

additional tuning of the structures without impacting the tools that are emitting them. And, it is in part because the runtime metadata engine even today supports a number of different formats exposed in a uniform way through the same API. For example, for COM Interop, a consumer of runtime metadata can import a typelib as though it were a perfectly valid runtime metadata file. Refer to the [Metadata Interfaces](#) spec for details on emitting and consuming the metadata portion of a runtime image.

19.2 Producers and Consumers of the Runtime File Format

Development tools and compilers will emit runtime images that can be packaged and deployed across a range of runtime-enabled platforms. Development tools will range from RAD tools (including scripting languages) to high-level language compilers. The first category of tools will compile and emit files in a single pass from the development environment. Scripting tools may not even have a need to persist the resulting file, but simply regenerate the code every time it's executed. The second category of tools has an incremental approach, first emitting intermediate compilation units and then linking them together with resources into a loadable runtime image.

The file format needs to accommodate not only what the runtime will require in order to load and execute these files, but it needs to make it reasonably straightforward for this range of different tools with different internal data structures and compilation models to emit metadata and code efficiently (along with imports/exports, fix-ups, debugging information, etc.).

Consumers of the runtime file format include the runtime itself as well as development tools and administrative tools. The runtime consumes metadata and CIL in order to JIT-compile CIL to native code. The loader consumes metadata to load classes and track managed data structures. Development tools will import metadata to enable references to runtime types and members. Administrative tools will consume metadata to browse classes and configure services.

19.3 Requirements Addressed by the Runtime File Format Design

Initial exploration of alternative design approaches ranged from introducing an entirely new file format for the runtime that would co-exist side-by-side with today's PE file format, to ensuring that the runtime format was a natural extension of today's Windows PE file. In having chosen the latter approach, it may be instructive to review the requirements that drove the design and spec work.

An Option of CIL or Native Code

A developer who wants to target a range of runtime platforms may want to build a component or assembly of components once and compile to native when needed for a particular platform. Options for "when needed" range from deployment time to install time to execution time. In this scenario, the code is emitted as CIL, plus the metadata that the runtime JIT compiler(s) use to compile the CIL to native.

A developer building a runtime component or application in his or her favorite language may have reason to compile code directly to native. For example, if the code is known to target only a specific platform, there may be no perceived benefit from going through an intermediate language. This does not mean that the developer need forego the benefits of the runtime managed services. In the design presented in this document, the target file format is today's PE file, either .exe or .dll.

To be more specific, the runtime recognizes managed native code and unmanaged native code. Both are compiled in any language to the native instruction set of a CPU. Unlike unmanaged native code, managed native code has additional metadata and coding conventions used by the runtime to enable garbage collection, exceptions and other runtime features. The current file format specification does not describe these metadata and file format extensions. Unmanaged native code is fully supported, emitted using all of the structures of today's PE/COFF.

A Combination of CIL and Native Code

The runtime will accept a file containing a mixture of CIL and native code. The runtime file format accommodates either one or both naturally in a single format, without requiring compilers to emit, and OS loaders to recognize, a range of different formats for specialized purposes.

Self-Contained Environments

Although based on today's Windows PE/COFF, the structure of the sections is intended to be subset-able for self-contained environments that are directly integrated with the runtime. In particular, these environments may be willing to trade off full OS services, like page sharing between processes, for image

size. Observe that the structure of the format headers and sections pictured earlier lends itself to a structure that consists solely of the runtime header and the data sections that make up the CIL portion of the image.

32- and 64-Bit Support

Support for both 32-bit and 64-bit requires a number of accommodations in the file format design, including:

- Support for agnostic-sized integers
- Data fix-ups

Although 64-bit is not fully supported in this version of the CLI, the underpinnings are reflected in this specification since moving toward 64-bit is integral to the design of the file format and the runtime.

20 Emitting A Valid CLI Image

This section covers the structure of the file headers, section headers, and extensions to the native PE data that may be used by the runtime.

20.1 File Headers

The image starts with an MS-DOS header, followed by the COFF header, and PE header.

20.1.1 Signature

The PE format calls for an MS-DOS stub to be placed at the front of the module. This stub is then used to tell DOS users that the module cannot be run in DOS mode.

At offset 0x3c is the offset to the PE signature. The signature will remain “PE\0\0” as it is today.

20.1.2 COFF Header

Immediately after the signature is the COFF header consisting of the following:

Offset	Size	Field	Description
0	2	Machine	Number identifying type of target machine. See below
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
8	4	Pointer to Symbol Table	The COFF symbol table is not used. Set this value to 0
12	4	Number of Symbols	Always 0
16	2	Optional Header Size	Size of the optional header, the format is described below
18	2	Characteristics	Flags indicating attributes of the file

20.1.2.1 Machine Type

If an image is intended to run on a single processor type, then the machine type should be set accordingly. If the image is intended to run on more than one processor type, runtime images will use a machine type of IMAGE_FILE_MACHINE_I386

20.1.2.2 Characteristics

An image that contains native code may have any of the standard flags from the PE file format specification as appropriate.

A CIL-only dll has the following characteristics:

Flag	Value	Description
------	-------	-------------

IMAGE_FILE_EXECUTABLE_IMAGE	0x0002	Image only. Indicates that the image file is valid and can be run. If this flag is not set, it generally indicates a linker error
IMAGE_FILE_LINE_NUMS_STRIPPED	0x0004	COFF line numbers have been removed
IMAGE_FILE_LOCAL_SYMS_STRIPPED	0x0008	COFF symbol table entries for local symbols have been removed
IMAGE_FILE_DEBUG_STRIPPED	0x0200	Debugging information removed from image file
IMAGE_FILE_DLL	0x2000	The image file is a dynamic-link library (DLL). Such files are considered executable files for almost all purposes, although they cannot be directly run

Currently we do not anticipate support for IMAGE_FILE_SYSTEM (to produce device drivers and systems level code written in CIL)

20.1.3 Optional Header

The PE/COFF Optional Header is required for a runtime image¹. It is located immediately after the COFF Header and is sometimes referred to as the PE Header. This header contains the following information:

Offset	Size	Header part	Description
0	28	Standard fields	These are defined for all implementations of COFF, including UNIX®.
28	68	NT-specific fields	These include additional fields to support specific features of Windows NT (for example, subsystem)
96	128	Data directories	These fields are address/size pairs for special tables, found in the image file and used by the operating system (for example, Import Table and Export Table)

20.1.3.1 Optional Header Standard Fields

These fields are required for all COFF files. They contain loader information as follows:

Offset	Size	Field	Description
0	2	Magic	Unsigned integer identifying the state of the image file. Set this value to 0x10B, meaning an executable file
2	1	LMajor	Linker major version number, tool specific
3	1	LMinor	Linker minor version number, tool specific
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 (BSS), or the sum of all such sections if there are multiple BSS sections
16	4	Entry Point RVA	Address of entry point, relative to image base, when executable file is loaded into memory. See the section below on entry points

¹ It is called optional because when the COFF format is used for an object file it is not required

20	4	Base Of Code	Address, relative to image base, of beginning of code section, when loaded into memory
24	4	Base Of Data	Address, relative to image base, of beginning of data section, when loaded into memory

20.1.3.2 Optional Header Windows NT-Specific Fields

These fields are Windows NT specific:

Offset	Size	Field	Description
28	4	Image Base	Preferred address of first byte of image when loaded into memory; must be a multiple of 64K
32	4	Section Alignment	Alignment (in bytes) of sections when loaded into memory. Must be greater or equal to File Alignment. Default is the page size for the architecture
36	4	File Alignment	Alignment factor (in bytes) used to align pages in image file. Valid values are a power of 2 between 512 and 64K. Unless otherwise necessary, use 512
40	2	OS Major	Major version number of required OS
42	2	OS Minor	Minor version number of required OS
44	2	User Major	Major version number of image
46	2	User Minor	Minor version number of image
48	2	SubSys Major	Major version number of subsystem
50	2	SubSys Minor	Minor version number of subsystem
52	4	Reserved	
56	4	Image Size	Size, in bytes, of image, including all headers; must be a multiple of Section Alignment
60	4	Header Size	Combined size of MS-DOS Header, PE Header, and Object Table
64	4	File Checksum	Image file checksum. The algorithm for computing is incorporated into IMAGHELP.DLL. The following are checked for validation at load time: all drivers, any DLL loaded at boot time, and any DLL that ends up in the server
68	2	SubSystem	Subsystem required to run this image. See note below
70	2	DLL Flags	Obsolete
72	4	Stack Reserve Size	Size of stack to reserve. Only the Stack Commit Size is committed; the rest is made available one page at a time, until reserve size is reached. Stacks for CIL will be handled by the runtime. This value should be set using the same switches as used today
76	4	Stack Commit Size	Size of stack to commit
80	4	Heap Reserve Size	Size of local heap space to reserve. Only the Heap Commit Size is committed; the rest is made available one page at a time, until reserve size is reached
84	4	Heap Commit Size	Size of local heap space to commit

88	4	Loader Flags	Obsolete
92	4	Number of Data Directories	Number of data-dictionary entries in the remainder of the Optional Header. Each describes a location and size

20.1.3.2.1 SubSystem Settings

The runtime Loader itself does not do anything with the subsystem setting of the PE. The value chosen, however, can impact on what Windows platforms the image may be run. For example, setting this value to IMAGE_SUBSYSTEM_WINDOWS_CE_GUI means the image can't be run on any non-CE device. In addition, IMAGE_SUBSYSTEM_NATIVE is not supported because the runtime cannot run in kernel mode for this release. It is recommend that either IMAGE_SUBSYSTEM_WINDOWS_GUI or IMAGE_SUBSYSTEM_WINDOWS_CUI be used for this setting.

20.1.3.2.2 Stack Reserve Size

For now, the default stack size should be used

Recommended size information will be supplied in a later release.

20.1.3.3 Optional Header Data Directories

Data directories give the address and size of tables used by Windows. Each data directory entry is as follows:

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD    RVA;
    DWORD    Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

The first field, RVA, is the relative virtual address of the table. The RVA is the address of the table, when loaded, relative to the base address of the image. The second field gives the size in bytes. The data directories, which form the last part of the Optional Header, are listed below.

Offset	Size	Field	Description
96	8	Export Table	Export Table address and size
104	8	Import Table	Import Table address and size
112	8	Resource Table	Resource Table address and size
120	8	Exception Table	Exception Table address and size. For Managed Native Code, this table will contain the MIH mapping information for the Code Manager
128	8	Certificate Table	Attribute Certificate Table address and size
136	8	Base Relocation Table	Base Relocation Table address and size
144	8	Debug	Debug data starting address and size
152	8	Copyright	Copyright string address and length
160	8	Global Ptr	Relative virtual address of the global pointer register. Size member of this structure is set to 0
168	8	TLS Table	Thread Local Storage (TLS) Table address and size

176	8	Load Config Table	Load Configuration Table address and size
184	8	Bound Import	Bound Import Table address and size
192	8	IAT	Import Address Table address and size
200	8	Delay Import Descriptor	Address and size of the Delay Import Descriptor
208	8	Runtime Header	Runtime Header with directories for runtime data
216	8	Reserved	

20.1.4 Storing Runtime Data in Sections

The runtime defines several types of data formats that are used by the engine to execute code. This includes things like the metadata, CIL method bodies, garbage-collection encodings, and others. Each type of data may be placed in any part of the PE image so long as the section the data is placed in has the same attributes as required. Each data format described in the following section identifies what type of page attributes the data must have in the final image.

1.2.2 Runtime Header

Directory entry IMAGE_DIRECTORY_COR_DESCRIPTOR contains the location of the runtime Header in the image. This 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. This value matches the COR_VERSION_MAJOR macro the runtime is compiled with
6	2	MinorRuntimeVersion	The minor portion of the version. Use the COR_VERSION_MINOR macro
8	8	MetaData	Location and size of the meta data in this image
16	4	Flags	Flags describing this runtime image. See “Runtime Flags” below
20	4	EntryPointToken	Token for the MethodDef of the entry point for the image
24	8	Resources	Location of CLI resources. These resources are not the same as the WIN32 resource section of the PE file
32	8	StrongNameSignature	Location of the hash data for this PE file used by the CLI loader for binding and versioning
40	8	CodeManagerTable	Location and size of the Code Manager Table. See below
48	8	VTableFixups	Location and size of an array of locations in the file that contain an array of function pointers (e.g., vtable slots). See discussion below

56	8	ExportAddressTableJumps	Location and size of an array of RVA's to where jump thunks are written
64	8	EEInfoTable	Reserved for IOJ ²
72	8	HelperTable	Reserved for IOJ
80	8	DynamicInfo	Reserved for IOJ
88	8	DelayLoadInfo	Reserved for IOJ
96	8	ModuleImage	Reserved for IOJ
104	8	ExternalFixups	Reserved for IOJ
112	8	RidMap	Reserved for IOJ
120	8	DebugMap	Reserved for IOJ
128	8	IPMap	OBSOLETE: this directory is being replaced by the P-Data map already defined in the PE specification

More details on parts of this header are included below. Those directories that are specific to Managed Native Code can be found in the chapter that covers MNC.

1.2.2.1 Runtime Header Definition

Following is the structure definition for the header:

```
// CLI header structure
typedef struct IMAGE_COR20_HEADER
{
    // Header versioning
    DWORD          cb;
    WORD           MajorRuntimeVersion;
    WORD           MinorRuntimeVersion;

    // Symbol table and startup information
    IMAGE_DATA_DIRECTORY MetaData;
    DWORD               Flags;
    DWORD               EntryPointToken;

    // Binding information
    IMAGE_DATA_DIRECTORY Resources;
    IMAGE_DATA_DIRECTORY StrongNameSignature;

    // Regular fixup and binding information
    IMAGE_DATA_DIRECTORY CodeManagerTable;
    IMAGE_DATA_DIRECTORY VTableFixups;
    IMAGE_DATA_DIRECTORY ExportAddressTableJumps;

    // Managed Native Code
    IMAGE_DATA_DIRECTORY EEInfoTable;
    IMAGE_DATA_DIRECTORY HelperTable;
    IMAGE_DATA_DIRECTORY DynamicInfo;
}
```

² “Install-O-Jit” – an informal term that describes how you can have the runtime engine compile the IL in PE modules and save the compiled, native code. This would typically be done when you install an application on a computer – thus the name.

```

IMAGE_DATA_DIRECTORY DelayLoadInfo;
IMAGE_DATA_DIRECTORY ModuleImage;
IMAGE_DATA_DIRECTORY ExternalFixups;
IMAGE_DATA_DIRECTORY RidMap;
IMAGE_DATA_DIRECTORY DebugMap;
IMAGE_DATA_DIRECTORY IAPMap;
} IMAGE_COR20_HEADER;

```

20.1.5.2 Runtime Flags

The following flags describe this runtime image and are used by the loader.

Flag	Value	Description
COMIMAGE_FLAGS_ILONLY	0x00000001	Image contains only CIL code so that it is not required to run on a specific CPU. Any native code in the module is the x86 startup stub which may safely be ignored if the OS is runtime aware
COMIMAGE_FLAGS_32BITREQUIRED	0x00000002	Image may only be loaded into a 32-bit process
COMIMAGE_FLAGS_TRACKDEBUGDATA	0x00010000	When set, the runtime and JIT are required to track debugging information about methods. See the Debugging Architecture Specification for more details.

20.1.5.3 Entry Point Meta Data Token

The entry point token is always a MethodDef token (refer to the [Metadata Interfaces](#) spec for details on metadata tokens). The signature and implementation flags in metadata for the method indicate how the entry is run. The entry point, if given, is always a managed method using the DEFAULT calling convention.

For a DLL image, the entry point is one of:

- int DllMain(HINSTANCE, DWORD, void *)

For an EXE image, the entry point is one of:

- unsigned main(void)
- void DEFAULT main(System.String[])

Although this approach means that the runtime has parsing smarts built into it, it offers the ability for any tool (like the assembler) to have a default implementation. A tool is always free to insert their own entry point wrapper using “unsigned main(void)” which in turn parses the command line and delegates to the user’s entry point code of the same signature.

The entry point RVA in the PE header must always be either the x86 entry point stub (which loads and calls the Runtime), or be 0 (for a runtime aware version of Windows only). If Managed and Unmanaged code are mixed in the same image, this is still the case. To make an unmanaged method the entry point for an image, define a Pinvoke MethodDef for the method and use that token. For example, here is a dump of the metadata for a C++ console application with an unmanaged entry point supplied by a static copy of the CRT:

Global functions

Method #1

```

-----
MethodName: _mainCRTStartup (06000008)
Flags      : [Public] [Static] [ReuseSlot] [PinvokeImpl] (00002011)
RVA       : 0x00001116
ImplFlags  : [Native] [Unmanaged] [Implemented] (00000014)

```

```
CallConvntn: [DEFAULT]
ReturnType: UI4
No arguments.
Pinvoke Map Data:
Entry point:
Module ref:      1a000001
Mapping flags:    [NoMangle] [CharSetNotSpec] [CallConvStdcall]
(00000301)
Ordinal:         00000000

ModuleRef #1
-----
ModuleRef: (1a000001) :
GUID : {00000000-0000-0000-0000-000000000000}
MVID : {00000000-0000-0000-0000-000000000000}
```

20.1.5.4 VTable Fixup

Certain unmanaged C++ classes, which choose not to follow the common type system runtime model, may have virtual functions which need to be represented in a v-table. These v-tables are laid out by the C++ 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. V-tables are emitted into a *read-write* section of the PE file. (This is different from unmanaged native code, where a v-table can be placed in a read-only section and shared between processes.) It is recommended that a tool try to emit v-tables adjacent to each other in the image in order to minimize the number of entries in this table (ie: you can do all v-tables with 1 entry if they are all adjacent). The runtime header contains the location and size of an array that looks like:

```
#define COR_VTABLE_32BIT      0x01          // V-table slots are 32-bits in
size.
#define COR_VTABLE_64BIT      0x02          // V-table slots are 64-bits in
size.
#define COR_VTABLE_FROM_UNMANAGED 0x04      // If set, transition from
unmanaged.
#define COR_VTABLE_CALL_MOST_DERIVED 0x10   // Call most derived method
described by
// token, only valid for
virtuals.

typedef struct _IMAGE_COR_VTABLEFIXUP
{
    ULONG      RVA;          // Offset of v-table array in image.
    USHORT     Count;        // How many entries at location.
    USHORT     Type;         // COR_VTABLE_xxx type of entries.
} IMAGE_COR_VTABLEFIXUP;
```

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.

This structure is also used for managed native code transitions. Please see the chapter on Managed Native Code for more details.

20.1.5.5 Resources

This header entry points to the CLI managed resources for an image, not to be confused with the traditional Win32 resource format. Please see the resource specification for more details on the format of this data.

20.1.5.6 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. The routines in StrongName.h in the public SDK can be used work with this data. Please see the [Design Specification for Assembly Metadata](#) for more details.

20.2 Section Headers

The section headers come immediately after the COFF optional header (PE Header). Refer to the “Microsoft® Portable Executable and Common Object File Format Specification” for settings on standard sections.

The runtime data for a module is always located through the runtime Header. The only requirement on a tool that emits this data is to place the data in a section which has the required characteristics. Currently all of the runtime data for a module (meta data, method bodies, etc.) are read only and may be placed in any read only section. The data may be shared between processes safely.

20.3 Modifications to Existing PE Data

Runtime modules may contain any data one currently finds in a valid PE image, including resources, imports/exports, data, and code. See the PE specification for more details. This section calls out specific areas where we use existing data in the runtime.

20.3.1 Import Address Table (IAT)

The IAT is used for one of the following:

1. To import mscoree.dll (ie: the runtime engine) for the x86 loader thunk
2. In a mixed managed/unmanaged image, for legacy imports

The IAT burns into the image slots for methods of a fixed size and is therefore not portable between 32 and 64-bit systems. For this reason, it is ideal to use the metadata to import legacy methods and data through the PInvoke feature. For an image using the IAT only to import the runtime, the IAT is completely ignored and therefore not a problem (presumes an OS loader which is CLI-aware).

20.3.2 Export Section (.edata)

Exporting as unmanaged native is interesting when one is implementing an existing interface (such as Win32, print drivers, or ODBC), but you want to write your code using the runtime. The EAT will always have pointers to unmanaged code as down-level clients expect this. There are no plans to use the EAT/IAT to directly export/import managed methods. Rather the metadata should be used to perform this duty.

Please see the Export Address Table Jumps directory entry for more details on how to export methods as unmanaged from an image.

20.3.3 Thread Local Storage Table

The programmer may mark an instance of a static field as TLS. This means that the memory for the instance is associated with the logical thread being executed. Each logical thread has its own copy of the instance of the variable. For an CIL image, a compiler generates instructions to access the field as they would any other piece of static data, by specifying the RVA of the data. The runtime does a range check of the RVA on the instruction to see if it falls in the TLS table, and if so, it abstracts the location of the memory relative to the thread automatically.

Per the current Windows standard, only PEs in the originating EXE or explicitly imported DLL's (through the IAT) are included in the calculation for TLS data at process startup. DLLs that are dynamically loaded later may not successfully use the TLS attribute. Runtime does not change this restriction.

TLS storage for native code is unchanged from the current PE specification. The same restrictions apply.

20.3.4 Relocations

Relocations in a pure CIL image are required only for the x86 startup stub which access the IAT to load the runtime engine on down level loaders. On an OS which is runtime-aware and marked for CIL-only execution, the relocation section, IAT, and any native code is ignored.

When building a mixed CIL/native image, relocations may be needed as they are in a traditional image.

Finally, if the image contains embedded RVAs in user data, then relocations must be emitted for these references. Relocations are not required nor recommended for arguments to CIL instructions (such as `ldptr`). These extra relocations are used by tools that need to rewrite or modify the image in some manner, such as a linker which combines images together.

21 Common Intermediate Language

This section describes how to format and emit CIL methods in the image. A method consists of a `COR_ILMETHOD` method structure and the instructions for the method. A call descriptor contains an RVA to this method structure. There is no required ordering for methods in this section, which allows a tuning tool to reorder the methods if better locality of reference and working set size can be obtained by so doing.

The `COR_ILMETHOD` describes all information about a method that is not needed by callers of the method. This includes:

1. Amount of resources needed for the Operand stack
2. Amount of resources for local variables as well as which local variables contain pointers into the garbage collected heap
3. Amount of resources for the locspace instruction
4. Exception handling information

Unfortunately, the description of the operand stack and the local variable frame is complicated by the fact that stack slots can hold items of unknown size. The size of `I` and `REF` types is unknown by the code generator, as well as by-value objects, don't even have an upper bound on their size.

21.1 Local Variable Layout

One function of the `COR_ILMETHOD` structure is to indicate the layout of the local variables. This is needed for two reasons.

1. To be able to find all pointers into the garbage-collected heap at GC time
2. To indicate the sizes of the local variables so the local frame can be calculated

When local variables are present, use the `COR_ILMETHOD_FAT` structure definition and provide a value for the `LocalVarSigTok` field. This is a metadata token for a signature that describes the precise layout of each local on the stack. Please see the [MetadataStructures](#) spec for more details on signature tokens.

21.2 File Format Structure Definitions

This section contains the layout of the data structures used to describe an CIL method and its exceptions. C language definitions can be found in the `CORHDR.H` file in the SDK. In addition, some unsupported C++ style declarations can be found in `CORHLPR.*`, also in the SDK. These helpers greatly simplify emitting and decoding the file format structures and are used in the runtime engine itself.

21.2.1 Method Body

The body of a method is defined in the general form.

Exception handling data is emitted after the method body. If exception-handling data is present, then `CorILMethod_MoreSects` must be specified in the method header and for each chained item after that.

21.2.1.1 Method Header Type Values

The three least significant bits of the first byte of the method header indicate what type of header is present. These 3 bits will be one and only one of the following:

Value	Value	Description
CorILMethod_TinyFormat	2	The method header is defined by the IMAGE_COR_ILMETHOD_TINY structure and the size of the code is even
CorILMethod_TinyFormat1	6	The method header is defined by the IMAGE_COR_ILMETHOD_TINY structure and the size of the code is odd. Using two values for tiny allows for an extra bit of size data
CorILMethod_FatFormat	3	The method header is defined by the IMAGE_COR_ILMETHOD_FAT structure

21.2.1.2 Tiny Format

The IMAGE_COR_ILMETHOD_TINY structure comes in two flavors with a 5 or 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 need be no bigger than 8 bytes

The first encoding has the following format:

Start Bit	Count of Bits	Description
0	2	CorILMethod_TinyFormat = 0x2
2	6	Size of the method body immediately following this header. Used only when the size of the method is even and under 2^6

The second encoding has the following format:

Start Bit	Count of Bits	Description
0	3	CorILMethod_TinyFormat1 = 0x6
3	5	Size of the method body immediately following this header. Used only when the size of the method is odd and under 2^5

21.2.1.3 Fat Format

The fat format is used whenever the Tiny format won't work. This may be true for one or more of the following reasons:

- The method is too large to encode the size
- There are exceptions
- There are extra data sections
- There are local variables
- The operand stack needs more than 8 bytes

The encoding of the first byte of the header in this case is as follows:

Start Bit	Count of Bits	Description
0	2	CorILMethod_Fat = 0x3
2	1	Reserved
3	1	CorILMethod_MoreSects = 0x8 or 0 to indicate no more sections
4	1	CorILMethod_InitLocals = 0x10 or 0 to indicate no local variable init

The rest of the Fat format is described below

21.2.1.4 IMAGE_COR_ILMETHOD

The metadata points to an instance of this structure. Method headers must be DWORD aligned. There are two possible formats to emit, Tiny or Fat. This structure is a union of these two types. The first byte indicates which type is present (see the “Method Header Type Values” for more details)

```
typedef union
{
    COR_ILMETHOD_TINY    Tiny;
    COR_ILMETHOD_FAT     Fat;
} COR_ILMETHOD;
```

21.2.1.5 IMAGE_COR_ILMETHOD_TINY

This structure must always be DWORD aligned. The layout of this structure is as follows:

```
typedef struct IMAGE_COR_ILMETHOD_TINY
{
    BYTE Flags_CodeSize;
} IMAGE_COR_ILMETHOD_TINY;
```

See the description of the header above.

21.2.1.6 IMAGE_COR_ILMETHOD_FAT

This structure must always be DWORD aligned. The layout of this structure is as follows:

```
typedef struct IMAGE_COR_ILMETHOD_FAT
{
    unsigned Flags      : 12;    // Flags
    unsigned Size       :  4;    // size in DWords of this structure
    // (currently 3)
    unsigned MaxStack   : 16;    // maximum number of items (I4, I, I8,
    // obj ...),
    // on the operand stack
    DWORD   CodeSize;        // size of the code
    DWORD   LocalVarSigTok;  // token that indicates the signature of
    the local
    // vars (0 means none)
} IMAGE_COR_ILMETHOD_FAT;
```

Offset	Size	Field	Description
0	12 bits	Flags	Flags (described below)

12 bits	4 bits	Size	Size of this header expressed as the count of DWORDs occupied
16 bits	16 bits	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

21.2.1.6.1 Flags for Method Headers

The first byte of a method header may also contain the following flags, valid only for the Fat format, that indicate how the method is to be executed:

Flag	Value	Description
CorILMethod_InitLocals	0x0010	Call default constructor on all local variables
CorILMethod_MoreSects	0x0008	Set to indicate a section attribute follows the current attribute. Used to chain in exception information among other things

21.2.2 Section Data

Certain types of data may be found after a method header. Currently only exceptions are described in such a way, but other usages are envisioned. These sections are only valid when the method is encoded with an IMAGE_COR_ILMETHOD_FAT header with the CorILMethod_MoreSects bit set. Section data must be DWORD aligned after the end of the method body as described by the method header.

Every section is at least two bytes. The first byte of the section data contains the flags describing the section as follows:

Flag	Value	Description
CorILMethod_Sect_Reserved	0	Reserved for future use
CorILMethod_Sect_EHTable	1	Exception handling data. See the IMAGE_COR_ILMETHOD_SECT_EH structure for more encoding details
CorILMethod_Sect_OptILTable	2	Reserved for future use
CorILMethod_Sect_FatFormat	0x40	Data format is of the fat variety, meaning there is a 3 byte length. If not set, the header is small with a 1 byte length
CorILMethod_Sect_MoreSects	0x80	Another data section occurs after this current section

21.2.3 IMAGE_COR_ILMETHOD_SECT_EH

Exceptions are declared as an extra section of data that comes after a method header. This structure defines an element of the Exception Handling information for a method. The structure must be emitted on a DWORD boundary. This structure is a union of the IMAGE_COR_ILMETHOD_SECT_EH_SMALL and IMAGE_COR_ILMETHOD_SECT_EH_FAT structure types, depending on how much data is present.

21.2.3.1 IMAGE_COR_ILMETHOD_SECT_EH_SMALL

The layout of this structure as is a follows:

```
typedef struct IMAGE_COR_ILMETHOD_SECT_EH_SMALL
{
    IMAGE_COR_ILMETHOD_SECT_SMALL SectSmall;
    WORD Reserved;
```

```

    IMAGE_COR_ILMETHOD_SECT_EH_CLAUSE_SMALL  Clauses[1];      // actually
variable size
} IMAGE_COR_ILMETHOD_SECT_EH_SMALL;

```

Offset	Size	Field	Description
0	1	Kind	CorExceptionFlag type for this block
1	1	DataSize	Size of the data for the block not including the header
2	2	Reserved	DWORD padding
4	<i>n</i>	Clauses	One or more clauses as defined by DataSize

21.2.3.2 CorExceptionFlag Values

The following flag values are used for each exception-handling clause:

Flag	Value	Description
COR_IEXCEPTION_CLAUSE_NONE	0x0000	
COR_IEXCEPTION_CLAUSE_FILTER	0x0001	Entry is for an exception filter
COR_IEXCEPTION_CLAUSE_FINALLY	0x0002	A finally clause
COR_IEXCEPTION_CLAUSE_FAULT	0x0004	Fault clause (finally that is called on exception only)

21.2.3.3 IMAGE_COR_ILMETHOD_SECT_EH_CLAUSE_SMALL

The small form of the exception clause should be used whenever the code size for the try block and handler code is smaller than or equal to 256 bytes. The format for a small exception clause is as follows:

```

typedef struct IMAGE_COR_ILMETHOD_SECT_EH_CLAUSE_SMALL
{
    CorExceptionFlag    Flags          : 16;
    unsigned            TryOffset      : 16;
    unsigned            TryLength      : 8;  // relative to start of try
block
    unsigned            HandlerOffset  : 16;
    unsigned            HandlerLength  : 8;  // relative to start of
handler
    union {
        DWORD          ClassToken;
        DWORD          FilterOffset;
    };
} IMAGE_COR_ILMETHOD_SECT_EH_CLAUSE_SMALL;

```

Offset	Size	Field	Description
0	2	Flags	CorExceptionFlag type for this block
2	2	TryOffset	Offset of a try block
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

21.2.3.4 IMAGE_COR_ILMETHOD_SECT_EH_FAT

Used to describe a large exception clause that will not fit in the COR_ILMETHOD_SECT_EH_SMALL structure.

```
typedef struct IMAGE_COR_ILMETHOD_SECT_EH_FAT
{
    IMAGE_COR_ILMETHOD_SECT_FAT    SectFat;
    IMAGE_COR_ILMETHOD_SECT_EH_CLAUSE_FAT  Clauses[1];    // actually
variable size
} IMAGE_COR_ILMETHOD_SECT_EH_FAT;
```

Offset	Size	Field	Description
0	1	Kind	Which type of exception block is being used
1	3	DataSize	How big is the data excluding the header size
4	<i>n</i>	Clauses	One or more clauses describing exception handling

21.2.4 IMAGE_COR_ILMETHOD_SECT_EH_CLAUSE_FAT

Use this structure when the smaller clause cannot handle the size of the data. The layout of this structure is as follows:

```
typedef struct IMAGE_COR_ILMETHOD_SECT_EH_CLAUSE_FAT
{
    CorExceptionFlag    Flags;
    DWORD               TryOffset;
    DWORD               TryLength;    // relative to start of try block
    DWORD               HandlerOffset;
    DWORD               HandlerLength; // relative to start of handler
    union {
        DWORD           ClassToken;    // use for type-based exception
handlers
        DWORD           FilterOffset;  // use for filter-based exception
handlers
    }
    // (COR_ILEXCEPTION_FILTER is
set)
};
} IMAGE_COR_ILMETHOD_SECT_EH_CLAUSE_FAT;
```

Offset	Size	Field	Description
0	4	Flags	CorExceptionFlag type for this block
4	4	TryOffset	Offset of a try block
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
----	---	---------------------	--

22 Code Transitions

For first release of the runtime product, one is allowed to mix unmanaged native code and managed CIL code in the same image. Not supported but planned for a future release is “Managed Native Code”, which is managed code emitted using the native instructions for a given CPU instead of CIL. This section of the specification documents the layout of these mixed images.

Following is a definition of terms:

Managed	Code that requires the runtime for execution. A managed program can use the common type system and take advantage of other runtime features that an unmanaged program cannot. Additional metadata is required in the image to be managed. This includes information about garbage collection, exception handling, and the locations of methods in the file. A managed program cannot run without the assistance, at runtime, of the CLI
Unmanaged	Code that does not require the runtime for execution. This code may not use the common type system or other features of the runtime. Traditional native code (before the CLI) is considered unmanaged
Code Manager	A code manager contains the code required to walk the state of a running program (such as tracing the stack and track GC references). CLI supports pluggable code managers, and provides default versions for JIT’ed code, CIL, and F-Jit
Thunk	A (typically) small piece of code used to provide a transition between two pieces of code where special handling is required

22.1 Call Transitions

22.1.1 Transition Types

A Transition is any boundary between types of code that requires special handling. When the caller and the callee code have the same properties, then no special transition is required to make the call. There are three main categories discussed where transitions may be required:

Runtime Type	A method may be managed or unmanaged . Transitions between the two types in either direction require a thunk
Code Type	A method body may be stored in CIL , Opt-CIL³ , or native byte codes. Transition between CIL and Opt-CIL is handled by the Runtime. Transitions between any form of CIL and Native require a thunk
Location Type	The implementation for a method may appear either in the local image, or may be imported from another image. When calling an imported method, a thunk may be required.

Any permutation of types above can result in the requirement for a thunk. This thunk is generated by the runtime during execution and will set up the correct state for the call. For example, when making a transition from managed to unmanaged, the GC state for the thread is marked as pre-emptive (meaning it will not read or write GC reference) and an exception filter is put in place to protect the managed code from errors that occur. It is beyond the scope of this document to describe the different types of thunks and their contents.

³ Opt-IL is not supported for first release of the product. Some file format elements allow this format for future use.

22.1.1.1 Effects on Like Pieces of Code

If the caller and callee have the same properties (say managed-native calling managed-native in the same module), then there is no special handling required. Only the minimal amount of metadata should be emitted in this case, and normal code generation rules apply (ie: just make a PC relative call to the target).

22.1.1.2 Effects on CIL Code

When the caller is any form of CIL, the runtime has complete control over the execution of the code. It will almost always be JIT-compiled, in which case the jitter may generate any transition thunks inline with the code.

22.1.1.3 Effects on Native Code

When the caller is native code, there is no opportunity to change code once the compiler has emitted it⁴. Because of this, a mechanism is required which allows the runtime to provide the thunk capable of setting up the right state. The rest of this section describes this mechanism in detail.

22.2 Runtime Header Support for Transitions

This section contains descriptions of directory entries in the header to support managed native code.

22.2.1 VTableFixups

The VTableFixups directory entry is used to declare transitions. In order to have a transition, for example from managed to unmanaged, one does the following:

1. Allocate a slot in a read/write portion of the file
2. Point a IMAGE_COR_VTABLEFIXUP entry at the slot
3. Set the Type to indicate managed to unmanaged
4. Place the token of the target in the slot

The following transition types are interesting:

Combination	Description
managed to managed Token	For non-common type system v-table fix-ups. For example, a C++ compiler creating its own v-tables
managed to unmanaged Token	For calling an unmanaged call site within a PE image where full PInvoke metadata is not required
COR_VTABLE_FROM_UNMANAGED to managed Token	For calling a managed call site from unmanaged code, such as in a classic v-table

Unmanaged to unmanaged through a transition is not interesting and not supported.

22.2.2 Export Address Table Fix-ups

Exporting a managed method using an unmanaged entry point is supported through the EAT. Only unmanaged entry points may be found in the EAT to provide compatibility with down level loaders. One uses the metadata to import/export managed methods in managed code.

In a pure CIL image, the EAT itself would be sufficient to find all of these fix-up entries. But in a mixed managed/unmanaged image, only those EAT entries that need a thunk are to be fixed up and touching other entries would cause corruption. Therefore, the ExportAddressTableJumps table is used to identify which entries in the EAT are for managed methods.

Building an export table amounts to the following steps:

⁴ Note that we could consider modifying the code in the .text section, but this causes the pages to be marked writable in the process using up extra working set size. This solution was discarded.

1. Use the VTableFixup support described above to allocate an unmanaged to managed slot
2. Put the token of the managed method to be exported into this slot
3. Allocate a 32-byte piece of memory, called the “EAT Jump Thunk”, in the image that will not move. Put the address of this memory in the EAT of the PE format for the method.
4. Initialize the first 4 bytes of the EAT Jump Thunk with the RVA of the slot allocated in step #1. Initialize the next 4 bytes after this RVA to zero (reserved for future use)
5. Create the ExportAddressTableJumps table and put the RVA of the EAT Jump Thunk into this table.
6. At runtime, the following sequence of events takes place:
7. The OS loader will do the transitive closure of all IAT’s of all images being loaded. This will take the address from each EAT and place it into the IAT before any user code runs.
8. When the runtime loader runs, it will process all VTableFixup table entries and will replace the token from step #2 above with a pointer to an unmanaged thunk for the managed method.
9. Next the runtime loader walks the ExportAddressTableJumps entries to find EAT Jump Thunks that need to get replaced. It uses the RVA stored in the jump thunk to find the address of the unmanaged thunk from the previous step. Finally, it replaces the EAT Jump Thunk contents with a native jump instruction which jumps to the unmanaged thunk.

It is worth pointing out a couple of caveats that drove this design:

- On down-level OS loaders, the runtime loader runs only when the OS turns control over to the PE entry point. Therefore, the EAT cannot be initialized until that point. By requiring the EAT Jump Thunk, this requirement is met. On future OS’s that cannot encode the jump thunk in the predetermined sized limit, an OS change is required.
- Calling an EAT entry for a managed function in your DLL’s initialization routine is forbidden.
- The runtime loader does not track the location of the thunks it builds. Because of this, using the VTableFixup table allows the same thunk to be shared many times. The second 4 bytes of the EAT Jump Thunk are set to zero in case this restriction is removed in the future and an alternate way of building the data is required.

23 Entry Points

This section contains an overview of entry point handling in loaded images.

23.1 Runtime API’s

The following API’s are exported from mscoree.dll and are used to bootstrap the runtime.

23.1.1 _CorExeMain

This method is called to start execution of an executable program. The API will retrieve the handle of the image using GetModuleHandle(NULL). The module is then added to the loaded module list.

```
__int32 __stdcall _CorExeMain();
```

23.1.2 _CorDllMain

This method is called for the entry point of a DLL. This routine uses the handle to the module passed in to add the module to the loaded list.

```
__int32 __stdcall _CorDllMain(HINSTANCE, DWORD, void *);
```

23.1.3 Entry Points for Windows CE

Windows CE throws out the PE header to reduce overall working set. Because of this, all required data from the PE header is passed to the runtime entry points at runtime. The prototypes on this platform are therefore different.

23.2 Shut Down Requirements

The runtime must be given a chance to shut itself down in the process in order to do things like GC finalization, and cleanup for COM interop. This is done as transparently as possible from a developer's point of view. However, some problems can occur with ill-behaved applications. One such example is creating an unmanaged executable image (ie: a .exe) that supplies its own entry point but does not call ExitProcess. Such an application may work if it were single threaded, but would fail to work with the runtime as the runtime system threads do not get a notification to shut down.

23.3 Entry Point Stubs

23.3.1 Runtime Aware OS Loader

A runtime-aware operating system can recognize a runtime image by checking directory entry 14 in the PE header. The general loading algorithm is as follows:

1. If directory entry 14 is not set, the image is not runtime-aware and should be loaded using normal load sequences.
2. If the CPU type is not x86, and does not match the current CPU, the OS should reject the image with an error.
3. If the CPU type is x86, check the flags in the heading for COMIMAGE_FLAGS_ILONLY. When this bit is set, it means the x86 code in the image is just a loader stub, and should be ignored. If this flag is not set, the image requires an x86 capable CPU to run.

If the image is a 32-bit PE format, and the flags in the runtime header have the COMIMAGE_FLAGS_32BITREQUIRED bit set, the image cannot be loaded in a 64-bit process. In this case, the OS should reject the load with an error.

If the image is a PE 32+ PE format (64-bit), then the image must be loaded into a 64-bit process.

23.3.2 Non Runtime Aware OS Loader

Existing supported platforms include Win '9x and x86 NT 4.0 and above⁵. For this reason, runtime images should emit an x86 startup stub that these OS platforms will understand and load. The load algorithm is therefore:

1. The OS will validate the image is an x86 image (Intel only), and page the image into the process.
2. The OS will invoke the native entry point, which is an x86 load stub. This stub will call the entry point API of mscoree (_CorExeMain or _CorDllMain) through the IAT section of the image.
3. The mscoree entry point code will use the module handle to load the runtime metadata out of the image, including the user's entry point specified in the runtime header, if one was given.
4. The runtime will then invoke the user's entry point code, jitting it if required.

23.3.3 Sample x86 Stubs

The following stubs are used by runtime tools when emitting applications:

```
// *****  
// Stubs.h  
//
```

⁵ Windows CE version 3 is a runtime-aware OS loader. The runtime will not work on a Windows CE platform before this version.

```
// This file contains a template for the default entry point stubs of a runtime
// CIL only program. One can emit these stubs (with some fix-ups) and make
// the code supplied the entry point value for the image. The fix-ups will
// in turn cause mscoree.dll to be loaded and the correct entry point to be
// called.
//
//*****
#pragma once

//*****
// This stub is designed for a Windows application. It will call the
// _CorExeMain function in mscoree.dll. This entry point will in turn load
// and run the CIL program.
//
// void ExeMain(void)
// {
//     _CorExeMain();
// }
//
// The code calls the imported functions through the iat, which must be
// emitted to the PE file. The two addresses in the template must be replaced
// with the address of the corresponding iat entry which is fixed up by the
// loader when the image is paged in.
//*****
static const BYTE ExeMainTemplate[] =
{
    // Jump through IAT to _CorExeMain
    0xFF, 0x25,                // jmp iat[_CorDllMain entry]
    0x00, 0x00, 0x00, 0x00,    // address to replace
};

#define ExeMainTemplateSize      sizeof(ExeMainTemplate)
#define CorExeMainIATOffset     2

//*****
// This stub is designed for a Windows application. It will call the
// _CorDllMain function in mscoree.dll with with the base entry point
// for the loaded DLL.
// This entry point will in turn load and run the CIL program.
//
// BOOL APIENTRY DllMain( HANDLE hModule,
//                         DWORD ul_reason_for_call,
//                         LPVOID lpReserved )
// {
//     return _CorDllMain(hModule, ul_reason_for_call, lpReserved);
// }
//
// The code calls the imported function through the iat, which must be
// emitted to the PE file. The address in the template must be replaced
// with the address of the corresponding iat entry which is fixed up by the
// loader when the image is paged in.
//*****

static const BYTE DllMainTemplate[] =
{
```

```
// Call through IAT to CorDllMain
0xFF, 0x25,          // jmp iat[_CorDllMain entry]
0x00, 0x00, 0x00, 0x00, // address to replace
};

#define DllMainTemplateSize    sizeof(DllMainTemplate)
#define CorDllMainIATOffset    2

#ifdef _ALPHA_

const BYTE ExeMainTemplate[] =
{
    // load the high half of the address of the IAT
    0x00, 0x00, 0x7F, 0x27, // ldah t12,_imp__CorExeMain(zero)
    // load the contents of the IAT entry into t12
    0x00, 0x00, 0x7B, 0xA3, // ldl t12,_imp__CorExeMain(t12)
    // jump to the target address and don't save a return address
    0x00, 0x00, 0xFB, 0x6B, // jmp zero,(t12),0
};

#define ExeMainTemplateSize    sizeof(ExeMainTemplate)
#define CorExeMainIATOffset    0

const BYTE DllMainTemplate[] =
{
    // load the high half of the address of the IAT
    0x42, 0x00, 0x7F, 0x27, // ldah t12,_imp__CorDLLMain(zero)
    // load the contents of the IAT entry into t12
    0x04, 0x82, 0x7B, 0xA3, // ldl t12,_imp__CorDLLMain(t12)
    // jump to the target address and don't save a return address
    0x00, 0x00, 0xFB, 0x6B, // jmp zero,(t12),0
};

#define DllMainTemplateSize    sizeof(DllMainTemplate)
#define CorDllMainIATOffset    0
```

24 Metadata Format

To be specified.

Free printed copies can be ordered from:

ECMA

114 Rue du Rhône

CH-1204 Geneva

Switzerland

Fax: +41 22 849.60.01

Email: documents@ecma.ch

Files of this Standard can be freely downloaded from the ECMA web site (www.ecma.ch). This site gives full information on ECMA, ECMA activities, ECMA Standards and Technical Reports.

ECMA
114 Rue du Rhône
CH-1204 Geneva
Switzerland

See inside cover page for obtaining further soft or hard copies.