NGWS Runtime

The IL Assembly Language Programmers' Reference
(PDC Release)

Last updated: 6/15/2000

This is preliminary documentation and subject to change.

Table of Contents

1 Introduction

NGWS IL is the intermediate language emitted by all compilers that target the NGWS (Next Generation Windows Services) SDK runtime. While IL can be directly interpreted, the Runtime's "JIT compilers" can also convert IL to native machine code. These compilers normally run in a "Just-In-Time" (JIT) mode, converting methods from NGWS IL to native code only when the method is about to run the first time. They can also be used in a more traditional mode by converting an entire assembly (see section 4.1) to native code and then saving the native code for future use.

Tools that generate IL can benefit from the many services provided by the Runtime, including the IL support for early and late binding, and the fact that code compiled to IL will run on any platform supported by the Runtime. IL 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 IL so that this code may benefit from features such as cross-language inheritance, code access security, garbage-collection, and simplified COM programming.

1.1 Audience

This specification is intended for people interested in generating or analyzing programs that will be executed by the NGWS runtime. This includes those who write compilers that target the NGWS runtime (either with native code or IL), development tools or environments, or program analysis tools.

For further information about the EE, IL, and metadata, see the following specifications:

- The Virtual Object System (VOS) specification (See NGWS SDK, "Technical Overview of the NGWS Runtime")
- The IL Instruction Set specification
- The Metadata specification
- The File Format specification
- The Base Class Library specification

This document assumes that the reader is familiar with the concept of a DLL (dynamic link library). More information on dynamic link libraries can be found in MSDN.

1.2 Overview

This document focuses on writing programs directly in IL, and relies heavily on the syntax of **ilasm**, the IL assembler shipped with the NGWS SDK. In order to understand the process of creating programs in IL, it discusses

- *Execution*, i.e. the Execution Engine, a model of a machine that might directly execute IL;
- *Typing*, i.e. the underlying type system and the declarations used to define types;
- *Instructions*, i.e. the operations of the IL instruction set;

- *Deployment*, i.e. assemblies, manifests and modules. Assemblies which are the unit of deployment in NGWS, and the declarations required to package an application or component for distribution;
- *Additional Features*, such as global methods, global fields, and interoperation with existing (unmanaged) code.

1.3 Execution Engine

The NGWS execution engine (EE) is responsible for executing PE (portable executable) files. 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 detail in the Architecture specification.

However, one service of the EE is worth noting here. The EE has a garbage collector that will automatically take care unreferenced objects and memory blocks.

1.4 Validation and Verification

Validation refers to a set of tests that can be performed on a NGWS PE file , in isolation, to check that the file format, metadata, and IL are self-consistent. The **PEVerify** tool (part of the NGWS SDK) can perform these tests and report on any errors. In general, **PEVerify** is an excellent way to test the correctness of any files generated by compilers, assemblers, or file-to-file transformation tools. It is intended for use by the developers of these tools, as a means to test that they are producing acceptable output. Code that does not pass the tests in **PEVerify** can crash the Execution Engine and the JIT compilers; it is not safe to run under any circumstances.

Verification refers to a set of tests that check for consistency between separately compiled modules. In conjunction with the validation tests, verification ensures that the program cannot access memory or other resources to which it is not granted access. These tests are usually performed immediately prior to converting a method containing IL code into managed native code. The **PEVerify** tool can also perform these tests, but since the tests are sensitive to metadata from other assemblies the fact that a file passes these tests at one point in time does not guarantee that it will always pass these tests.

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

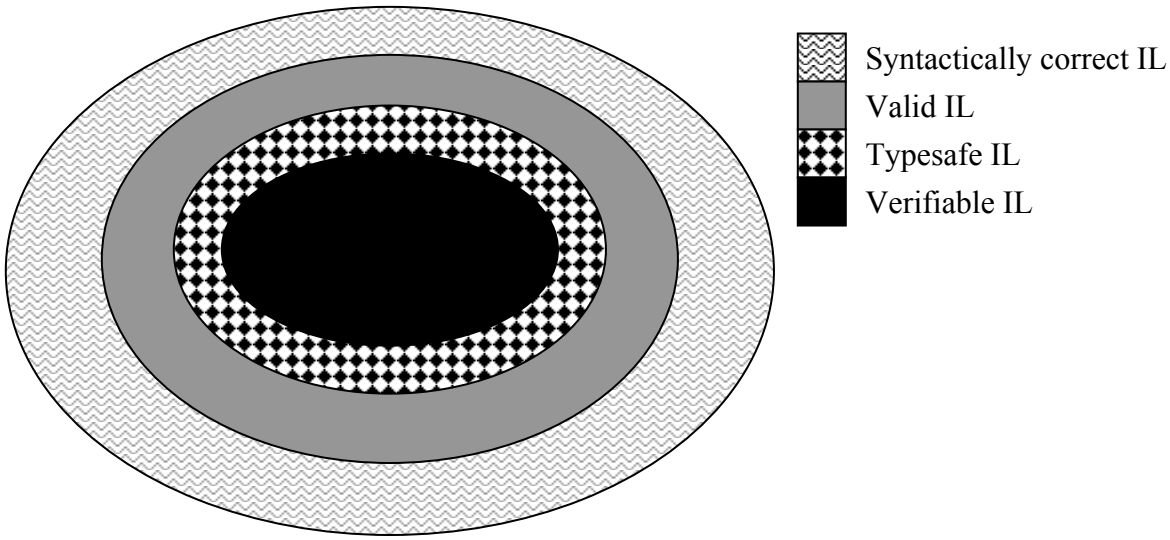

Figure 1: Relationship between sets of languages. (Figure not drawn to scale)

In the above figure, the outer circle contains all code permitted by the IL syntax. The next circle, which is solid gray, represents all code that is valid IL. Note that even if the assembler accepts an IL program, or a program follows the syntax described in this document, this code may still not be valid, because valid code must adhere to the other restrictions presented in this document. Also, the assembler may accept a somewhat more liberal syntax than presented in this document. The dotted inner circle represents all type safe code. Finally, the innermost circle contains all code that is verifiable. This document will use the term verifiable to mean code that passes **PEVerify**.

Verification is a very stringent test. There are many programs that will pass validation but will fail verification. The NGWS runtime 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 NGWS runtime allows an unsafe subset of code, that is code that does not pass verification, to be executed subject to administrative trust controls.

In general, IL is used most often with a type-safe programming language whose compilers emit IL that can be verified, but it is possible to generate IL for unsafe languages, such as C and C++. The IL emitted by the compilers for unsafe languages cannot, in general, be verified, but it will execute as a NGWS unmanaged application.

The complete set of rules for validation of the file format and metadata are covered in a separate specification. Use of the assembler guarantees that the file format is valid, although it is possible to create invalid metadata. Use of **System.Reflection.Emit** guarantees that both the file format and the metadata are valid. The rules for emitting valid as well as verifiable IL instruction sequences are included in the IL Instruction Set Specification and summarized below.

More information on verification can be found in the Verifier Specification.

1.5 The NGWS SDK IL Tools

1.5.1 The Assembler

There are five primary ways to write programs that run under the NGWS runtime:

- Write them in a programming language that generates the appropriate file format.
- Use the Reflection and ReflectionEmit classes in the NGWS SDK Base Class Library to produce an assembly on disk, in memory, or written to a stream.
- Write a compiler or other tool that directly generates the appropriate file format, using the information contained in the File Format and Metadata API specifications.
- Write a program directly in IL and convert it to an assembly using the **ilasm** program.
- Write a program in several separate parts, directly in IL, convert each to a module using **ilasm** and then combine them into an assembly using the **al** program (see section 1.5.3).

This document concentrates on the last two mechanisms, using the syntax of **ilasm** as a means of describing the overall workings of the NGWS runtime. One of the best ways to learn any new assembly language is to examine the output of other tools that produce that language. For that reason, Microsoft supplies a program, **ildasm**, that can take any file that can be executed by the NGWS runtime and produces output that can be used as input to ilasm.

ilasm takes IL as input and generates a PE file containing the IL and the required metadata. The resulting executable can be run to determine whether the IL performs as expected. The IL Assembler allows tools developers to concentrate on IL generation without being concerned with emitting IL in the PE file format. This tool is intended primarily for testing the performance of small sequences of IL and to test IL generation strategies.

An alternive to ilasm is the use of Reflection and ReflectionEmit to execute programs. This option is chosen rather by scripting languages and is described in considerably more detail in the Base Class Library documentation. This document provides cross-references to that material so that readers can understand what methods in ReflectionEmit produce the effect of various directives and attributes provided in the IL syntax of ilasm.

Although the IL Assembler is helpful in the early stages of generating IL, it has some limitations: it does not emit all possible runtime PE file features, and it cannot produce a COFF file that can be statically linked with other files (an .obj file).

For those who need complete control over the NGWS executable file format it is possible to directly generate the contents of the file. For this reason, Microsoft supplies a complete specification of the file format with the exception of the on-disk layout of the NGWS metadata. Metadata is written to a file by using the Metadata APIs, which are described elsewhere. As with ReflectionEmit, this document provides cross-references to these APIs so they may be used when ilasm is not sufficient.

There is a companion tool, **ildasm**, that takes a PE file containing IL code and creates a text file suitable as input to the IL Assembler. This ability to round trip code can be useful, for example, when compiling code in a programming language which doesn't support all of the runtime's metadata attributes. The code can be compiled, then the output run through **ildasm**, and the resulting IL text file can be had edited to add the missing attributes. This can then be run through the IL Assembler to produce a final runnable file.

In order to make this round tripping possible, the assembler does *not* perform some simple optimizations that are provided by other assemblers: it does not deduce whether to use short or long forms of instructions, but requires the input to be explicit. It does, however, check for out-of-range conditions where this is possible.

1.5.1.1 Usage of ilasm

Usage: ilasm [Options] <sourcefile> [Options]

Option	Description
/LISTING	type a formatted list file
/NOLISTING	don't type a formatted list file (default)
/QUIET	don't report assembly progress
/DLL	compile to .dll
/EXE	compile to .exe (default)

Option	Description
/DEBUG	include debug metadata
/RES=<resourcefile>	link the specified resource file (*.res) into resulting .exe or .dll
/OUT=<targetfile>	compile to file with specified name (must have extension)
/OWNER	protect the resulting file against disassembling
/OWNER=<ownersname>	protect the resulting file against unauthorised disassembling (<ownername> will be required to disassemble the file) <ownername> is arbitrary string of alphanumeric characters, without spaces in between (use underscores)

Key may be '-' or '/'.

Options are recognized by first 3 characters.

Default source file extension of IL files is ".il".

1.5.2 The Disassembler

The NGWS SDK diassembler is called **ildasm**.

Usage: ildasm [options] <file_name> [options]

Options for output redirection:	**Description**
/OUT=<file name>	Direct output to file rather than to GUI.
/TEXT	Direct output to created console window rather than to GUI.

Options for GUI or file/console output (EXE and DLL files only):	**Description**
/OWNER=<owner name>	Set owner name to disassemble a protected PE file.
/BYTES	Show actual bytes (in hex) as instruction comments.
/RAWEH	Show exception handling clauses in raw form.
/TOKENS	Show metadata tokens of classes and members.
/SOURCE	Show original source lines as comments.
/LINENUM	Include references to original source lines.
/VISIBILITY=<vis>[+<vis>...]	Only disassemble the items with specified visibility.

Options for GUI or file/console output (EXE and DLL files only):	**Description**
	(<vis> = PUB | PRI | FAM | ASM | FAA | FOA | PSC)
/PUBONLY	Only disassemble the public items (same as /VIS=PUB).
/QUOTEALLNAMES	Include all names into single quotes.
/NOBAR	Suppress disassembly progress bar window pop-up.

The following options are valid for file/console output only (if /OUT is specified):

Options for EXE and DLL files:	**Description**
/NOIL	Suppress IL assembler code output.
/HEADER	Include file header information in the output.
/ALL	Combination of /HEADER, /BYTES, /TOKENS

Options for EXE, DLL, OBJ and LIB files:	**Description**
/ITEM=<class>[::<method><sig>]	Disassembles the specified item only

Options for LIB files only:	**Description**
/OBJECTFILE=<obj_file_name>	Show MetaData of a specific object file in library
/BYREF=<class1,class2,..>	Marks the classes as marshaledbyref (non-GUI)

Option key is '-' or '/', options are recognized by first 3 characters.

Example: ildasm /tok /byt myfile.exe /out=myfile.il

1.5.2.1 The Disassembler for Power Users

For advanced users, the disassembler provides the following additional command line options:

For all following options, the option ADV has to be used as the first option:

Usage: ildasm /ADV [options] <file_name> [options]

In addition to the options above, the following additional options are supported:

Advanced options for EXE and DLL files:	**Description**
/STATS	Include statistics on the image.
/CLASSLIST	Include list of classes defined in the module.

Advanced options for EXE,DLL,OBJ and LIB files:	**Description**
/METADATA[=<specifier>]	Show MetaData, where <specifier> is:
MDHEADER	Show MetaData header information and sizes.
HEX	Show more things in hex as well as words.
CSV	Show the header sizes in Comma Separated format.
UNREX	Show unresolved externals.
VALIDATE	Validate the consistency of the metadata.

1.5.2.2 Roundtrips

ildasm and **ilasm** can be used to *round trip* code. This means that managed code compiled by any compiler can be disassembled, inspected by the user, and then again reassembled. **ildasm** has a somewhat extended syntax to support this. Those syntax elements are documented in this guide, but are marked "for round trip only".

It is not possible to ildasm to disassemble native code, such that it is not possible to round trip any program that depends on native code.

This is especially true for Microsoft VC code. The VC linker will automatically add a method called `_CRTStartup` to an exe. This method contains native code and is a problem for the disassembler. In order to make it possible to round trip, the linker has to be instructed not to add this method. This can be done by specifying an explicit entry point as in the following example:

cl /com+ <filename> /link entry:main

where main is the main method of the program. This will make it possible to round trip the code, but with the catch that since the CRT startup didn't run, no CRT method, like `printf`, can be called. Only managed methods, like `WriteLine`, must to be used.

1.5.3 The Assembly Linker

The assembly linker resolves the references of a PE file. It is called **al**.

Usage: al [options] [sources]

Options:	**Description**
/METADATA[=<specifier>]	Show MetaData, where <specifier> is:
/? or /help	display usage information

Options:	**Description**
@<filename>	read command-line options from the file
/algid:<id>	algorithm used to hash files (in hexadecimal)
/base[address]:<addr>	set the default base address of the DLL
/bugreport:<filename>	create a bug report file
/comp[any]:<text>	company name
/config[uration]:<text>	configuration string
/copy[right]:<text>	copyright message
/c[ulture]:<text>	supported culture
/delay[sign][+|-]	delay sign this assembly
/descr[iption]:<text>	description
/flags:<flags>	assembly flags (in hexadecimal)
/fullpaths	display files using fully-qualified filenames
/i[nstall][:<filename>]	install this assembly into the assembly cache
/keyf[ile]:<filename>	file containing key to sign the assembly
/keyn[ame]:<text>	key container name of key to sign assembly
/main:<method>	the fully-qualified method name of the entrypoint
/nologo	suppress the display of the startup banner
/os:<os>.<maj>.<min>	operating system, major and minor version constants
/out:<filename>	file to create for the assembly manifest
/proc[essor]:<proc>	processor constant (in hexadecimal)
/prod[uct]:<text>	product name
/productv[ersion]:<text>	product version
/title:<text>	title
/trade[mark]:<text>	trademark message
/t[ype]:lib|exe|win	create a DLL, console app, or GUI app
/v[ersion]:<version>	version (use * to auto-generate remaining numbers)
/win32icon:<filename>	use given icon for auto-generated Win32 resource
/win32res:<filename>	use given RES or OBJ file for Win32 resource

'/out' or '/install' must be specified.

Source	Description
<filename>[,<targetfile>]	add file to assembly
/embed[resource]:<filename>[,<name>[,Y|N]]	embed the file as a resource in the assembly
/link[resource]:<filename>[,<name>[,<targetfile>[,Y|N]]]	link the file as a resource to the assembly

At least one source input is required.

1.5.4 The Module Verifier

The verifier validates and verifies PE files. It is valled **PEVerify**.

Usage: PEverify <image file> [Options]

Options:	Description
/IL	Verify only the PE structure and IL
/MD	Verify only the PE structure and MetaData
/CLS	Verify only the PE structure and CLS compliance
/NOCLS	Verify only the PE structure and MetaData, w/o CLS compliance
/UNIQUE	Disregard repeating error codes
/HRESULT	Display error codes in hex format
/CLOCK	Measure and report verification times
/IGNORE=<hex.code>[,<hex.code>...]	Ignore specified error codes
/IGNORE=@<file name>	Ignore error codes specified in <file name>
/BREAK=<maxErrorCount>	Abort verification after <maxErrorCount> errors
/quiet	Display only file and Status. Do not display all errors.

PEVerify will return a list of warnings and errors for each problem it encounters. If PEVerify returns one or more errors, the code is not verifiable in the environment PEVerify was executed. The code may be verifiable in another environment.

1.5.5 The Debugger

A special debugger is included with the tools. It is called **cordbg**. More detailed documentation can be found in the Debugger Reference.

1.5.6 Compilers

There are a number of compilers that can be used to create either IL code or a PE file, which can be inspected by **ildasm**. The following is a short list of those compilers:

Language	Compiler
C#	**csc**
Visual C++	**cl /com+**
Visual Basic	**vbc**

2 Hello World Example

In this section we give a simple example to illustrate the general "feel" of NGWS IL. We are all familiar with "Hello world", which might be written in a managed language, like C#, as follows.

```
public static main() {
      System.Console.Write("Hello world\n");
}
```

The salutation is written by calling **`Write`**, a static method found in the NGWS SDK class **`System.Console`**. This example might compile into the following IL. The italicized line numbers are for explanatory purposes and are not part of the IL program.

```
1  .assembly 'hello.exe' { }
2  .method public static void main() il managed {
3     .entrypoint
4     .maxstack 1
5     ldstr "Hello world\n"
6     call void System.Console::Write(class System.String)
7     ret
8  }
```

The **.assembly** declaration on line 1 declares the *assembly name* for this program. Assemblies are the packaging unit for executable content for the NGWS runtime. The **.method** declaration on line 2 defines the global method **`main`** – the declaration gives **`main`**'s signature (**`void`**, no parameters) and its attributes (**`il`**, **`managed`**). Lines 3–8 are the body of the method, enclosed in braces. Lines 3 and 4 indicate that this method is the entry point for the assembly (**`.entrypoint`**), and that this method requires at most one stack slot (**`.maxstack`**).

The method has only three instructions. The **ldstr** instruction on line 5 pushes the string constant **`"Hello world\n"`** onto the stack and the **`call`** instruction on line 6 invokes **`System.Console::Write`**, passing the string as its only argument (note that string literals in IL are instances of the standard class **`System.String`**.) As shown, call instructions must include the full signature for the callee. Finally, line 7 returns (**`ret`**) from **`main`**.

3 General Syntax

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

3.1 General Syntax Notation

This following sections use the EBNF[1] 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. The options are sorted in alphabetical order (to be more specific: in ASCII order, ignoring "<" for syntax classes, and case-insentive).

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

3.2 Terminals

The simple syntax classes used in the grammar are:

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

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

<float> is whatever is accepted by the C function **`strtod`**which converts a string to a double precision floating point number.

<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 to insert an arbitrary byte into the string when followed by three octal digits can be used. A long string can be broken across multiple lines by using "\" as the last character in a line, in which case the line break isn't entered into the generated string.

<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 an alphabetic character, "_", "$", or "@" and is followed by any number of alphanumeric characters, "_", "@," or "?". An <ID> is used in only two ways

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

[1] See "What Can Be Done About the Unnecessary Diversity of Notation for Syntactic Definitions?", Niklaus Wirth, journal CACM, volume 20, number 11, November 1977, pages 822-823.

Thus 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

3.3 Identifiers

Identifiers are used to name entities. The IL 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. However, the single quotation marks are not necessary if the identifier conforms to the specification of an <ID>. There is no fixed maximum length for an identifier. This is summarized in the following grammar.

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

Examples of <id>:

```
A                    Test          $Test
@Foo?
'Weird Identifier'  'Odd\102Char'  'Embedded\nReturn'  ?
_X_
```

To prevent name clashes, several <id>’s may be combined to form the qualified name of an entity. E.g., one <id> may specify a certain module, another the submodule, and a third one the actual name of the entity, which is a part of the submodule. The <id>’s are separated by a dot (.). An <id> formed in this way is called a <dottedname>.

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

3.4 Labels and Lists of Labels

A simple label is just an <ID>. A label cannot contain any of the complicated characters allowed in an <id>. 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>

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

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

IL distinguishes between two kinds of labels: code labels and data labels. Code labels are always followed by a colon (":") and specify the position of an instruction to be executed. In contrast to code labels, data labels do not have the colon character and specify the location of a piece of data. The data label may not be used as a code label, and a code label may not be used as a data label.

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

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

3.5 Lists of Hex Bytes

A list of bytes is simply zero or more hex bytes. Hex bytes are pairs of characters 0 – 9, a – f, or 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 while using the dot (".") for the decimal point and "e" or "E" in front of the exponent. Both the decimal point and the exponent are optional.
2. 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.

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

```
    | <realnumber>
```

3.7 Source Line Information

To aid with debugging source line information may be included in the PDB (Program Database) file associated with an assembly file. 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.

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

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, but each of which play a different role in the NGWS runtime.

An *assembly* is the unit of deployment of software in the NGWS runtime, and is thus the largest of the groups above. An assembly can be thought as a group of one or more files that belong together, along with various properties that identify the unit of software to the mechanisms that manage the installation and removal of software within a host environment.

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 *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 of 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 prevent name clashes, and while it may appear contain types (e.g. System.Object) and other namespaces (e.g. System.XML), there is no semantics associated with this containment.

4.2 Defining an Assembly

An IL 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. An IL file that does not define an assembly must define a *module reference* and can be used in conjunction with the **al** assembly linker. Defining modules references is covered in Section 4.4.

An IL file may need to access other, external assemblies, which it does by specifying aspects of the manifests for these assemblies. This information is then used by the EE to determine which assembly is to be used.

If a manifest is given, it should appear at the beginning of an IL 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>* }                4.2.1
    | .file [nometadata] <dottedname> [.hash = ( <bytes> )]            4.2
    | .manifestres [public | private] <dottedname>                    4.2.2
        [( <QSTRING> )] { <manResDecl>* }
    | …                                                               4.5
```

All manifest declarations must be made at the top level.

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 executables. However, it is optional for modules (dll files). If such a library has an assembly declaration, it will in its own assembly. If a library does not have an assembly declaration, it will be part of the assembly it is used in.

Assembly names should not contain the file extension.

The **.manifestres** directive introduces a manifest resource declaration, described in section 4.2.2.

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

<asmAttr> ::=	Description	Section
implicitcom	COM Types are implicit, must be set in V1	4.2.4
| **noappdomain**	One instance of assembly only per application domain	4.2.4
| **nomachine**	One instance only per process	4.2.4
| **noprocess**	One instance only per machine, install time	4.2.4

4.2.1 Information about the Assembly

The following grammar shows the information you may specify about an assembly. This information is used by the NGWS SDK linker to determine if the assembly can be executed in a particular execution context. For example, if the assembly contains native code for some particular platform, then the **.processor** directive should be specified. Similarly, if you record a version number, then you can ensure that client code links to precisely the right version of the software, even if other versions are executing side-by-side.

<asmDecl> ::=	Description
.hash algorithm <int32>	Hash algorithm ID
| **.title** <QSTRING> [**(** <QSTRING> **)**]	The optional QSTRING is a description
| **.custom** <customDecl>	Custom attributes
| **.locale = (** <bytes> **)**	Information about the locale
| **.locale** <QSTRING>	Information about the locale
| **.originator = (** <bytes> **)**	Information about the originator
| **.os** <int32> **.ver** <int32> **:** <int32>	OS ID, major version and minor version
| **.processor** <int32>	Processor ID
| **.ver** <int32> **:** <int32> **:** <int32> **:** <int32>	Major version, minor version, revision, and build

The hash algorithm id is defined in the header file **wincrypt.h** which usally comes with Microsoft Visual Studio. 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.

The following table lists the constants and their corresponding value:

Constant	Value (decimal)
CALG_MD2	32769
CALG_MD4	32770
CALG_MD5	32771
CALG_SHA	32772
CALG_SHA1	32772
CALG_MAC	32773
CALG_RSA_SIGN	9216
CALG_DSS_SIGN	8704
CALG_RSA_KEYX	41984
CALG_DES	26113
CALG_3DES_112	26121
CALG_3DES	26115
CALG_RC2	26114
CALG_RC4	26625
CALG_SEAL	26626
CALG_DH_SF	43521
CALG_DH_EPHEM	43522
CALG_AGREEDKEY_ANY	43523
CALG_KEA_KEYX	43524
CALG_HUGHES_MD5	40963
CALG_SKIPJACK	26122
CALG_TEK	26123
CALG_CYLINK_MEK	26124
CALG_SSL3_SHAMD5	32776
CALG_SSL3_MASTER	19457
CALG_SCHANNEL_MASTER_HASH	19458
CALG_SCHANNEL_MAC_KEY	19459
CALG_SCHANNEL_ENC_KEY	19463
CALG_PCT1_MASTER	19460
CALG_SSL2_MASTER	19461

Constant	Value (decimal)
CALG_TLS1_MASTER	19462
CALG_RC5	26125
CALG_HMAC	32777

4.2.2 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.

```
| .manifestres [public | private] <dottedname>                    4.2.2
    [( <QSTRING> )] { <manResDecl>* }
```

If the the manifest resource is declared **public** it is exported from the assembly. If it 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
.assembly extern <dottedname>	Manifest resource is in external assembly with name <dottedname>.	4.2.2
| .custom <customDecl>	Custom attribute.	17
| .file <dottedname> at <int32>	Manifest resource is in file <dottedname> at offset <int32>.	4.2.2

4.2.3 Files in the Assembly

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

```
<decl> ::=                                                        Section
    .file [nometadata] <dottedname> [.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 NGWS EE will recompute this hashvalue when this file is referenced 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 4.2.1).

4.2.4 Operational Characteristics of Assemblies

To understand the **noappdomain**, **noprocess** and **nomachine** attributes for assemblies, you have to first understand that the execution model of the NGWS EE permits each machine to have several running instances of the EE, each as an operating system-process. In turn, each process may contain several "application domains", which are isolated areas of execution, and each applicaion domain may have many threads of execution.

Assemblies have various operational characteristics with regard to these dynamic structures. For example global, static variables are either copied once per machine, process, and/or application domain. Similarly, there is the question of how many copies of the JITted code exist for the assembly – is it one per machine, process or application domain? In NGWS these characteristics are all specified at the granularity of assemblies.

In particular, an *instance* of an assembly corresponds to one full set of generated data structures that result from loading and preparing an assembly for execution, most notable global, static variables and the underlying JITted code. The choice between "per app domain" and "per process" depends partly on whether app domains will be unloaded – if "per proceess" is chosen and the app domain is unloaded then the space taken up by the JITted code will not be freed until the entire EE process has terminated.

Per-machine assemblies only make sense in the context of install-time JIT compilation.

The above flags specify the granularity at which instances should be created. Combinations of the flags are invalid.

4.3 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 IL file. **.assembly extern** declarations are thus used for the resolution of names that refer to external entitites. The declaration includes the external assembly name, an optional alias for the assembly provided after the **as** clause, and a sequence of further declarations in braces. The option **fullorigin** specifies that the the assembly reference holds the full (unhashed) originator.

`<decl> ::=`	**Section**
`| .assembly extern [fullorigin] <dottedname> [as <QSTRING>]` `{ <asmRefDecl>* }`	4.2.1

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

`<asmRefDecl> ::=`	**Description**
`.hash = ( <bytes> )`	Hash Blob for file references
`| .custom <customDecl>`	Custom attributes
`| .locale = ( <bytes> )`	Information about the locale
`| .locale <QSTRING>`	Information about the locale

`| .originator = ( <bytes> )`	Information about the originator
`| .os <int32> .ver <int32> : <int32>`	OS ID, major version and minor version
`| .processor <int32>`	Processor ID
`| .ver <int32> : <int32> : <int32> : <int32>`	Major version, minor version, revision, and build

These declarations are very similar to those for **.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 all System classes and methods. This assembly may always be referenced implicitly, so that all System methods can be used without having to explicitly reference the assembly.

4.4 Referencing Modules

Instead of referring to a contruct 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. To do this, you must first use a **.module** declaration to define some information about the module you wish to access.

```
<decl> ::=
   | .module [[extern] <dottedname>]
```

Only the name of the module need be specified, and no other information about it.

4.5 Declarations in a Module

The remainder of an IL file is a sequence of declarations specifying the contents of a module. A single file input to the IL assembler is a sequence of declarations, defined as follows:

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

Declarations are specified by the following grammar. More information on each option can be found in the corresponding section.

`<decl> ::=`	**Section**
`| .class <classHead> { <classDecl>* }`	6
`| .custom <customDecl>`	17
`| .data <datadecl>`	13.4.1
`| .export [<exportAttr*>] <dottedname> { <exportDecl>* }`	4.6
`| .field <fieldDecl>`	13
`| .method <methodHead> { <methodDecl>* }`	12

| **.namespace** <dottedname> **{** <decl>* **}**	7.1
| **.vtfixup** <vtfixupDecl>	7.5.2.2
| <externSourceDecl>	3.7
| <securityDecl>	16

Example:

.assembly MyAssembly.dll { .ver 1 : 2 : 1 : 4 }

Nothing but a manifest for an assembly.

.class Foo { .field int32 Bar }

Declare a class named **Foo** with one (public by default) field named **Bar** which must hold a 32-bit integer.

4.6 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. 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 types exported is specified. Finally, the export declarations follow.

<decl> ::=	**Section**
| **.export** [<exportAttr*>] <dottedname> **{** <exportDecl>* **}**	4.6

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

<exportAttr> ::=	**Section**
nested assembly	6
| **nested famandassem**	6
| **nested family**	6
| **nested famorassem**	6
| **nested private**	6
| **nested public**	6
| **public**	6

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

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

4.6.1 The .comtype directive

The **.comtype** directive is for disassembling purposes only. It should not be used. Instead of **.comtype**, the **.export** directive should be used, which can be used for the same purpose (see section 4.6).

```
<decl> ::=                                                    Section
   | .comtype <comtypeHead> { <comtypeDecl>* } /* roundtrip      4.2.1
       only */
```

5 Types

Types are an abstract way of describing values and specifying what operations can be defined on the values. The NGWS type system includes both reference types (pointers and object references) and value types (primitive numeric types and user defined types which are passed by copying the value). User defined types may contain fields (data members), methods, properties and events. There are also types manufactured automatically by the NGWS runtime from a description of the type, such as pointers and arrays.

Section 6 describes reference types which are defined by classes. Section 9 describes value types while section 9.6 describes the automatically created types.

More information on types can be also found in the VOS spec.

5.1 The Type System

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

5.2 Types

The following grammar completely specifies all types that can be used in the NGWS type system:

<type> ::=	Description	Section
bool	Boolean	5.2
| **char**	Unicode character	5.2
| **class** <className>	User defined reference type. Recall that <className> includes information about nested types and a resolution scope.	5.2
| **float32**	32-bit floating point number	5.2
| **float64**	64-bit floating point number	5.2
| **int8**	Signed 8-bit integer	5.2
| **int16**	Signed 16-bit integer	5.2
| **int32**	Signed 32-bit integer	5.2
| **int64**	Signed 64-bit integer	5.2
| **method** <callConv> <type> *** (** <signature> **)**	Function type	10.5
| **native float**	Internal floating point representation	5.2
| **native int**	Signed integer whose size varies depending on platform (32- or 64-bit)	5.2
| **native unsigned int**	Unsigned integer whose size varies depending on platform (32- or 64-bit)	5.2
| <type> **&**	Managed pointer (by-ref) to <type>. Note that <type> cannot itself be a managed pointer.	10.4
| <type> *****	Unmanaged pointer to <type>	10.4
| <type> **[ ]**	Zero-based, one-dimensional array of <type>.	5.2
| <type> **[** [<bound> [,<bound>]*] **]**	Array with specified rank (number of dimensions) and element type, and bounds as shown above.	5.2
| <type> **modopt (** <className> **)**	Custom modifier that may be ignored by the caller.	5.2
| <type> **modreq (** <className> **)**	Custom modifer that the caller must understand in order to use this parameter.	5.2

| <type> **pinned**	For use in local signatures only, this local is pinned so that the value it references will not be moved by the garbage collector for the duration of this method.	5.2
| **typedref**	Typed reference, created by **mkrefany** instr.	5.2
| **value class** <className>	User defined value type.	5.2
| **unsigned int8**	Unsigned 8-bit integers	5.2
| **unsigned int16**	Unsigned 16-bit integers	5.2
| **unsigned int32**	Unsigned 32-bit integers	5.2
| **unsigned int64**	Unsigned 64-bit integers	5.2
| **void**	No type. Only allowed as a return type or as part of **void ***	5.2
| **wchar**	Unicode character	5.2

Bounds are used by arrays and are defined as follows:

<bound> ::=	**Description**
<int32>	zero lower bound, <int32> upper bound
| <int32> ...	lower bound only specified
| <int32> ... <int32>	both bounds specified

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*:

<typeSpec> ::=	**Section**
[[**.module**] <dottedname> **]**	
| <className>	5.3
| <type>	5.2

The fundamental types are specified in **CorHdr.h** as **ELEMENT_TYPE_xxx**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**.

5.3 Type References, Assemblies and Modules

Types include names, via the "<className>" grammar element. Names of nested classes (see section 7.4) are formed by using the outer class name, a slash ("/") and the name of the nested class.

Types and type names must typically be resolved by the EE. For example, types for primitive types will be self-contained, but when the EE finds a type name used to specify a class type (E.g. "**class** MyClass"), it must find the definition for this class.

Type names may define extra information specifying how resolution should happen:

```
<className> ::=                                                    Section
     [<resolutionScope>] <dottedname> [/ <dottedname>]*              3.1


<resolutionScope> ::=
     [ .module <externFileName> ]
   | [ <assemblyRefName> ]

<externFileName> ::=                                               Section
     <dottedname>                                                  3.1

<assemblyRefName> ::=                                              Section
     <dottedname>                                                  3.1
```

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. 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** directive, and an assembly reference must have been defined by a prior **.assembly extern** directive (see section 4.2.4).

The **ilasm** tool converts type names to either a *type references* or a direct references to a *type definition*. The latter is used if the assembler can find the actual definition of the type within the module being created.

Method and fields are referenced in a similar fashion (see section 12.6 for method references).

On occasion a class name can be used in isolation. In these cases the names may again be prefixed by a resolution scope, via the <className> non-terminal.

Examples of types with resolution scopes:

`class [Assembly.exe]Foo.Bar/C`
A reference to the type named `C` nested inside of the type named `Foo.Bar` in another module, named `Assembly.exe`. Notice that there must already have been a top-level **.assembly extern** directive that defines the name "Assembly.exe"

`class [.module X.mod]C.D`

A reference to the named `C.D` in the module named `X.mod` in the current assembly. There must be a top-level **.file** directive to define X.mod.

`class [mscorlib.dll]System.Console`
This is the proper way to refer to a class defined in the NGWS SDK base class library. The name of the type is `System.Console` and it is found in the assembly named `mscorlib.dll`. As you would expect, you must have a **.assembly extern** directive to define mscorlib.dll before this reference.

The following shows the grammar for a type specification:

5.4 Inheritance, Type Conformance and Subtypes

Reference types may be related to each other by inheritance and other rules, for example System.String is related to System.Object. We say that System.String is a *subtype* of System.Object, or alternatively that the former *conforms to* the latter, or that there exist *implicit conversions* between values of one type to the other type. If one type conforms to another this means that it provides the same members, contracts, and member signatures as the other type. The behavior of the types may differ.

The NGWS SDK supports *multiple type inheritance*. This is achieved by inheriting interfaces using the **implements** keyword in the class head (see section 6 for more information on classes).

The rules for conformance between reference types are as follows:

- In the trivial case, a type always conforms to itself;
- If a type B is a class type and has a superclass A, then type B conforms to type A;
- If a type B is a class or interface type and supports interfaces I1, ... I*n*, then type B conforms to each of I1 through I*n*;
- All reference types conform to System.Object;
- All array types conform to System.Array;
- An array type B[...] conforms to an array type A[...] if type C conforms to 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.

5.4.1 Conformance and Subtyping in the IL Verifier

Conformance is most crucial during verification when using the IL instructions that call methods, perform stores, and return values from methods (**call**, **callvirt**, **stfld**, **ret**, **starg**, **stloc** etc.). The exact rules applied by the verifier vary, but typically a variable may be assigned any value that has a type that conforms to the declared type of the variable. E.g., if a variable has type A, and B conforms to A, then the variable may be assigned values of type B.

5.4.2 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 will be B (or even some subtype of B). Exact types may be compared and checked by using IL instructions such as **castclass** and **isinst**, as well as the facilities available in the reflection library.

6 Visibility, Accessibility and Hiding

The VOS 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 to a method then no reference to that type can be resolved and the name of the type and its members does not participate in any way in name resolution at runtime.

Hiding controls which member names inherited from its base class are available during runtime name binding.

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.

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.

The following table summarizes this:

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

6.2 Hiding

Hiding applies to individual members of a type (nested types are *not* considered to be members for this purpose). The VOS 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).

6.3 Accessibility

There are seven accessibility modes. These can be applied to members of a type and to nested types.

Nested classes have also an accessibility associated with them in addition to their **nested** visibility.

The following table shows and describes the accessibility attributes:

Accessibility	Description
Public	Accessible by all referents.
Family or Assembly	Accessible only to referents that qualify for Family or Assembly access, but not necessarily both.
Assembly	Accessible only to referents in the same assembly that contains the implementation of the type.
Family and Assembly	Accessible only to referents that qualify for both Family and Assembly access.
Family	Accessible only by referents whose base class (immediate or indirect) defined the member or type in question.
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.

Not all accessibility attributes may be used with all members. In the following sections, for each type of member the permitted accessibility attributes are described.

The following two subsections give more information on attributes containing **family** and on **privatescope** accessibility.

6.3.1 Family Access

Most accessibility modes can be completely tested statically (i.e. at JIT time and, except for versioning problems, at source code compilation time). The modes that involve **family**, however, are different. There are two rules, one enforced by the verifier and one by the Execution Engine. The Execution Engine rule is statically testable, and is simply that access to a family member is available only to methods within that class and any of its subclasses.

The verification rule is stricter. It ensures that methods of a given class can only access family members of objects that belong to the method's class (or a subtype of the method's class). If the object is the original **this** pointer for an instance or virtual method, then the condition is automatically satisfied. Otherwise, however, the object must be known to the verifier to belong to the class of the method (or one of its subtypes). This may be true either because the statically declared type of the object satisfies the condition

or because the IL generator has inserted an explicit **castclass** instruction to force the object to be tested at runtime.

6.3.2 Privatescope Acess

privatescope effectively restricts access to the member to the same compilation unit that defines them, allowing a compiler complete control over accessibility. This access mode is most useful for implementing concepts like function-local static variables.

7 Classes

Classes define the reference types of the NGWS type system. The NGWS type system supports single code inheritance and multiple type inheritance. At runtime classes may be instantiated, allocating an *object* on the heap and creating a reference to this location. Only the reference to an object may be passed around and not an object itself. This is unlike value types, described in more detail in chapter 9.

Classes have members. NGWS supports the following groups of members:

- fields
- methods
- properties
- events
- nested classes

Fields are typed memory locations that store the data of classes and their instances. Fields are described in more detail in chapter 13.

Methods implement the behavior of classes. They are described in further detail in chapter 12.

Properties are constructs that look like fields, but are implemented via methods. These methods allow access and/or mutation of data that is internal to the object. Other methods may be associated with properties, too. More information on properties can be found in chapter 14.

Events are similar to methods but are executed when an the appropriate event is fired. More about events can be found in chapter 15.

Nested classes are classes defined inside other classes. They are defined by their own top-level **.class** declarations and are not normally considered "members", but have similar access restrictions. Nested classes are described in more detail in section 7.4.

7.1 Defining a Class

Classes may be defined at the top-level of an assembly program as the following excerpt from the grammar for an IL assembly file shows:

```
<decl> ::=                                                          Section
     .class <classHead> { <classDecl>* }                              6
   | .namespace <dottedname> { <decl>* }                              7.1
   | …                                                                4.5


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

Namespaces can be used to prevent conflicts with duplicate class names within an assembly. Namespaces are introduced by the **.namespace** directive. Each namespace has a <dottedname> and contains all its declarations in braces. Namespaces may be nested.

Namespaces are syntactic sugar. The assembler will automatically combine the <dottedname> of a namespace with the name of a class. Declaring a class "MyClass" in namespace "MySpace" would have the same effect as giving the class directly the name "'MySpace.MyClass'", quoted in single quotes to permit the use of the ".".

A class declaration consists of

- any number of class attributes (see section 7.1.1)
- a name (an <id>)
- an optional superclass (see section 5.4)
- an optional list of interfaces to be implemented (see also section 8)
- a number of declarations specifying the contents of the class (apart from nested classes)

The **extends** or **implements** clause contain specifictions of type names (see section 7.4). A <className> is a <dottedname>.

The class name may also be prefixed by a resolution scope, which specifies the context in which the class name is resolved. Resolution scopes are described in section 5.3.

The **extends** keyword defines the *superclass* of a class, a kind of inheritance called "code inheritance". In the NGWS SDK a class always inherits code from exactly one other class. If no class is specified, the class will inherit from **System.Object**.

The **implements** keyword defines the *interfaces* of a class, a kind of inheritance called "multiple type inheritance". Zero, one or more interfaces may be specified, and, given that interfaces may indeed inherit from further interfaces, the class will actually support the transitive closure of all interfaces, i.e. the immediate interfaces, the superinterfaces to those and so on.

The inheritance structure creates a hierarchy of classes and interfaces. This hierarchy must be acyclic.

7.1.1 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.	7.1.1.4
| **ansi**	Used for string marshaling across managed/ unmanaged boundary.	7.1.1.6
| **auto**	Auto layout of class.	7.1.1.2

autochar	Specifies to use platform specific char marshal across boundary.	7.1.1.6
explicit	Layout of fields are provided explicitly.	7.1.1.2
import	The class is imported from COM.	7.1.1.5
interface	The declaration is an interface declaration.	7.1.1.3
lateinit	Initialize class as late as possible.	7.1.1.7
nested assembly	Assembly accessibility for nested class.	7.1.1.1
nested famandassem	Family and Assembly accessibility for nested class.	7.1.1.1
nested family	Family accessibility for nested class.	7.1.1.1
nested famorassem	Family or Assembly accessibility for nested class.	7.1.1.1
nested private	Private accessibility for nested class.	7.1.1.1
nested public	Public accessibility for nested class.	7.1.1.1
not_in_gc_heap	Specifies that the class shall not be allocated in garbage collected heap.	7.1.1.3
private	Private accessibility.	7.1.1.1
public	Public accessibility.	7.1.1.1
rtspecialname	Special treatement by runtime.	7.1.1.7
sealed	The class cannot be subclassed anymore.	7.1.1.4
sequential	The class is layed out sequentially.	7.1.1.2
serializable	Specifiles that the fields of the class may be output to a stream.	7.1.1.5
specialname	Special treatment by tools.	7.1.1.7
unicode	Used for string marshaling across managed/unmanaged boundary.	7.1.1.6
value	Declares a value type.	7.1.1.3

7.1.1.1 Visibility and Accessibility Attributes

The visibility attributes are **nested assembly**, **nested famandassem**, **nested family**, **nested famorassem**, **nested private**, **nested public**, **private**, and **public**. Visibility attributes are described in section 6.1 and accessibility attributes are described in section 6.3. The **nested** in front of the visibility attribute specifies that the class is a nested class (see section 7.4). Top-level classes may only have **public** or assembly visibility. Nested classes have **nested** visibility, which mean that they have the same visbility as their outer class. In addition nested classes have an accessibility attribute, which specifies the range from which they can be referenced.

The visibility assembly can be specified by not using any other visibility attribute.

Visibility attributes are exclusive. The default is assembly.

7.1.1.2 Class Layout Attributes

The class layout attributes are **auto**, **explicit**, and **sequential**. These attributes are used to specify how the fields of a class are arranged. Layout attributes are 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.

Sequential specifies that the fields are layed out in sequential order by the runtime.

7.1.1.3 Class Semantics Attributes

The class semantics attributes are **interface**, **not_in_gc_heap**, and **value**. **interface** and **value** cannot be specified together. **Not_in_gc_heap** can only be used in connection with **value**.

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

If **not_in_gc_heap** is used with **value**, an instance of the value type will not be allocated on garbage collected heap. E.g., it may be allocated on the stack instead. Instance of refernce types are always allocated on garbage allocated heap.

7.1.1.4 Special Sematics Attributes

Attributes that specify special semantics are **abstract** and **sealed**. Both attributes are exclusive.

abstract specifies that this class may not be instantiated. Interfaces are implicitly abstract and may not be defined to be abstract. Typically, even though not necessary, abstract classes contain one or more abstract methods (see section 12.3.4). If a class contains abstract methods, it must be declared as an abstract class.

Sealed specifies that subclasses of the class may not override any virtual methods of this class. This effectively means that all virtual methods become instance methods.

7.1.1.5 Implementation Attributes

The implementation attributes are **import** and **serializable**. This 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 output through a data stream. E.g., the class may be sent over the network or saved to a file.

7.1.1.6 Interoperation Attributes

These attributes are for interoperation with COM and mainly focus on the treatment of strings of type LPSTR. The attributes are **ansi**, **autochar**, and **unicode**. These attributes are exclusive and the default is **ansi**.

While autochar specifies that the interpretation is done automatically, the other two interpret LPSTR as an ANSI string or Unicode string, respectively.

7.1.1.7 Special Handling Attributes

The thre attributes that are used for special handling are **lateinit**, **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.

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

7.2 Contents of a Class

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

<classDecl> ::=	Description	Section
.class <classHead> **{** <classDecl>* **}**	Defines a nested class.	7.2
| **.comtype** <comtypeHead> **{** <comtypeDecl>* **}** /* for round trip only */	Exports a COM type, use **.export** instead.	7.2
| **.custom** <customDecl>	Custom attribute.	17
| **.data** <datadecl>	Defines static data associated with the class.	7.2
| **.event** <eventHead> **{** <EventDecl>* **}**	Defines an event.	7.2
| **.export** [**public** | **private**] <dottedname> **{** <exportDecl>* **}**	Specifies entities to export.	4.6
| **.field** <fieldDecl>	Declares a field belonging to the class.	7.2
| **.method** <methodHead> **{** <methodDecl>* **}**	Declares a method of the class.	7.2
| **.override** <typeSpec> **::** <methodName> **with** <callConv> <type> <typeSpec> **::** <methodName> **(** <signature> **)**	Specifies that the first method is overriden by the definition of the new method.	7.2
| **.pack** <int32>	Used for explicit layout of fields. Fields are put at byte multiples of <int32>. E.g., .pack 8 would specify that fields should be stored at addresses that are a multiple of 8.	7.2
| **.property** <propHead> **{** <PropDecl>* **}**	Defines a property of the class.	7.2
| **.size** <int32>	Specifies that a memory block of <int32> many bytes shall be allocated for an instance of the class.	7.2

	Used for explicit layout only.	
| <externSourceDecl>	**.line** or **#line**	3.7
| <securityDecl>	**.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 nesteded type.

7.3 Special Members of Types

This section discusses some special members of a classes and other types. These members are virtual methods, instance and class constructors, and finalizers.

7.3.1 Inheritance of Virtual Methods

A virtual method specifies a contract that all of its implementations are expected to support. When a concrete class has a parent that declares a virtual method the child class must also provide an implementation for that virtual method. By default, the parent's implementation will be used for the child.

The **sealed** attribute indicates that no class may use it as a parent. Introducing a new virtual method in a sealed class is legal and is precisely the same as introducing the method as an instance method. All enums and value types are sealed.

The **final** attribute on a virtual method prohibits any child class from providing its own implementation of this virtual method. However, a new virtual method with the same name and signature may be introduced via the **newslot** attribute. This can be thought of as creating a new method that happens to have the same name.

If a child class wishes to provide its own implementation (called "overriding") it may do so only if it could have called the method it would override[2]. An override is specified by either of two methods:

- By providing a direct implementation of the virtual method using a **methodef** (and not specifying it to be **newslot**) and providing the location of the code which implements the method.
- By providing a MethodImpl whose declaration part is a **methodef** or **methodref** for the virtual method, either specifying this class or any parent class that defines the same virtual method[3].

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 **methoddef** that provides the location of the code that implements the method and is marked as **newslot**. This explicitly marks the virtual method as creating a new contract and will therefore always create a new slot in the object layout to hold the implementation of this virtual method (even if, later, a parent class defines a virtual method with the same name and signature).

[2] This requirement may be relaxed before V1 is released, since some languages appear to require the looser rule that any inherited virtual method my be overridden.

[3] This may not be permitted in V1.

It is also possible *not* to mark the method as **newslot** and a new slot will be created if no existing virtual method with that name and signature is located. This is not recommended, however, since it allows a parent class to "capture" this implementation if a version change in the parent introduces a virtual method with the same name and signature.

When computing the assignment of method implementation to slots, if a class provides a method body for a virtual method that is not marked **newslot**, then the execution engine will try to find an inherited virtual slot to reuse. It searches the chain of parents looking for a virtual method with the same name and signature. This search ignores intervening static or instance methods of the same name and signature, and it also ignores the "hide-by-name" attribute. If it is unable to locate an existing virtual method in any of its parents a new slot is created, exactly as though the definition had been marked **newslot**.

7.3.2 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**. Instance constructors may take parameters, but may not return a value. Instance constructors may be overloaded, such that a class may have several instance constructors. Each instance constrcutor must have a unique signature. At instantiation time, this signature is specified and determines which constructor is called.

7.3.3 Instance Finalizer

The finalizer gives classes a chance to execute some final code before they become garbage collected. 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();
```

Finalize does nothing by default. If an object holds references to any resources, **Finalize** should be overridden in order to free these resources before the object is discarded by the GC.

This method can be overridden by a derived class, but only if necessary. Reclamation by the GC will tend to take much longer if a **Finalize** operation must be run. **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, the object can only be resurrected once; the GC will not call **Finalize** on resurrected objects.

However, **Finalize** operations are not guaranteed to be run. If the resource must be freed, the Dispose() design pattern is recommended.

7.3.4 Class constructors

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

Interfaces, classes, and value types may all have type initializers. This method must be static, take no parameters, return no value, be marked with **rtspecialname** 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.

Instance constructors are a kind of static method. Thus, they may only access static fields. Instance constructors have a special privilege in that they may write into static fields in the class that have the **initonly** attribute.

7.3.4.1 Execution Guarantees

There are three fundamental guarantees about type initialization.

1. The type initializer always starts running before
   - any instance of the type is created
   - any static member (method or field) of the type is referenced
2. A type intializer is run exactly once for any given type, unless explicitly called by user code.
3. No method other than one called directly or indirectly from the type initializer will be able to access members of a type before its initializer completes execution.

7.3.4.2 Delaying Type Initialization

Classes and interfaces can be marked as either "late initialize required" or "early initialize allowed" based on the **lateinit** attribute. 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. Early initialization is always performed on value types (it is illegal to set both **value** and **lateinit**).

7.3.4.3 Races and Deadlocks

Consider the following two class definitions:

```
class A
{ static A a; static B b;
  runtime special .cctor () { b=null; a=B.a; }
}
class B
{ static A a; static B b;
  runtime special .cctor () { a=null; b=A.b; }
}
```

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 NGWS SDK 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 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 NGWS runtime. 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 NGWS runtime deals with these problems by ensuring that all three of the guarantees are met, as well as provide two additional guarantees to code that is called out of the class initializers:

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.

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.

7.4 Nested Types

One type may be nested within another. All references to the nested type are through its **enclosing** type. The nested type has its own accessibility, and references to the nested type must therefore have access to both the enclosing type and the nested type itself.

The parent of a nested type is completely independent of the enclosing type. Methods defined within a nested type are treated as part of the enclosing type *and* as children of their parent type.

Consider:

```
public class A extends System.Object
{ family int AfamInt;
  private int AprivInt;
}
public class B extends System.Object
{ family int BfamInt;
  private int BprivInt;
```

```
 public class C extends A  // nested in B, parent is A
  { public int Foo(B myB, A myA)
    { this.BfamInt := 3;   // Illegal, not a B
      this.BprivInt := 3;  // Illegal, not a B
      myB.BfamInt := 3;    // OK, nested in B
      myB.BprivInt := 4;   // OK, nested in B
      this.AfamInt := 3;   // OK, child of A
      this.AprivInt := 4;  // Illegal, not member of A
      myA.AfamInt := 3;    // Not verifiable but OK
    }
  }
  public int Foo(B myB, A myA)
  { this.BfamInt := 3;     // OK
    this.BprivInt := 3;    // OK
    myB.BfamInt := 3;      // OK
    myB.BprivInt := 4;     // OK
    this.AfamInt := 3;     // Illegal, doesn't inherit from A
    this.AprivInt := 4;    // Illegal, doesn't inherit from A
    myA.AfamInt := 3;      // Illegal, doesn't inherit from A
  }
}
```

Objects of type "C inside of B"

Are a subtype of A. Hence, their **this** contains all of the fields, methods, properties, and events of any other instance of A.

Have access to all the members of any instances of type B.

Are *not* a subtype of B. Hence, their **this** does *not* contain the fields, methods, properties or events of a B.

7.5 Controlling Layout

In some cases, it may be useful to control the layout of fields of an instance. A set of directives make this possible.

Further it may be also desirable to control the layout of a virtual method table. Also this is possible using the NGWS IL.

7.5.1 Explicit Layout Control of Instances

Classes that require explicit layout control must be marked with the class attribute explicit (see section 7.1.1.2).

The two directives that control layout of fields are **.pack** and **.size**.

.pack specifies to put fields at specified byte multiples. E.g., **.pack** 8 would specify that fields should be stored at addresses that are a multiple of 8.

.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 unmanaged code.

Further, fields may be placed at certain indexes. This index is specified in brackets before the field attributes. A class that uses this feature must be declared **explicit**.

7.5.2 Explicit Layout of the Vtable

Virtual methods are implemented using a *vtable* (virtual method table). Rather than calling the method directly, the method call is redirected through the table, which contains the addresses of the virtual methods. These addresses may be overriden by subtypes, which has the effect of redirecting the call to the new implementation.

7.5.2.1 The override Directive

Usually, the runtime will automatically determine which method overrides wich method, by matching its name or signature depending whether hide_by_name or hide_by_signature is used.

The syntax for .override is as follows:

<classDecl> ::=	Section
.override <typeSpec> **::** <methodName> **with** <callConv> <type> <typeSpec> **::** <methodName> **(** <signature> **)**	7.5.1
| …	7.2

The first method specification will be overriden by the method specification following the **with**.

If a particular method that has a different name, but a compatible signature, shall override a method, the **.override** directive may be used. The .override directive simply specifies that the slot for the specified method in the virtual method table shall be replaced by the new definition.

7.5.2.2 The vtfixup Directive

In some cases when interoperation with unmanaged code is required, the position of the slot used for the virtual method table might need to be specified exactly. In addition, the unmanaged code might not use the correct calling convention to invoke a managed method.

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

<decl> ::=	Section
.vtfixup <vtfixupDecl>	7.5.2.2
| …	4.5

The virtual method table does not need to be a contiguous block of memory. It may be separated into sevaral chunks of memory. Each chunk is called an virtual method table entry, and has an entry number associated with it. Each entry must caputure a contiguous block of memory and is divided into slots. The slots 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, such that 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.

The syntax for **.vtfixup** takes the desired number of slots the declared 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 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 solts are 64 bits wide. If **int64** is used and the pointers on the target machine are only 32 bits wide, they higher 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.

7.6 Global (Non-class) Data 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”. They are modeled in the NGWS SDK as static fields or static methods with a parent of **mdTokenNil** rather than a **typeref** or **typedef**.

It is simplest to understand global data and methods in the NGWS SDK by imagining that they are simply members of a fictitious abstract public class, <module>, that doesn’t implement any interfaces. The parallel is very close indeed. The only noticeable difference is in how definitions of this fictitious class are treated by the metadata merge code (used by tools such as a linker), the NGWS SDK 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 (fictitious) 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 vs 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 of **mdPrivateScope**, then save them all in the output class.

- If there are duplicates and two or more have an accessibility other than **mdPrivateScope** then keep all of the **mdPrivateScope** members as well as any one of the other duplicates, silently dropping any additional members that don't have **mdPrivateScope**.

8 Interfaces

An *interface* is a class declaration where the **interface** attribute is set. This implies various constraints, for example that:

- all its methods are virtual;
- it has no fields;
- it does not inherit from a class.

An interface, IA, that promises to provide an implementation of another interface, IB, requires that any concrete class that implements IA must also implement IB.

Interfaces do not have instance methods. All virtual methods on interfaces have **public** accessibility. The interface itself must not provide an implementation of its virtual methods.

Interfaces may have static (but not instance) fields.

Interfaces may have static methods, provided they supply an implementation for them. This is used, for example, to provide a class constructor for the interface itself to initialize its static fields.

For CLS compatibility, Interfaces may not contain instance fields, and the only static method they may provide is a type initializer (named .cctor and marked with both mdSpecialName and mdRTSpecialName).

When a class contract requires an implementation for an interface the Execution Engine follows a simple set of rules to determine which method body will be used to implement, for instances of this class, each virtual method of the interface. A MethodImpl (see also section 12) can be used when the default behavior does not capture the programmer's intention.

8.1 Requirements on classes that implement interfaces

A concrete (i.e. non-abstract) class must provide an implementation for

- all methods that it introduces
- all virtual methods of interfaces that it implements
- all virtual methods it inherits from its parent

The implementation of virtual methods may be provided by:

- directly specifying an implementation
- inheritance from its parent class
- use of an explicit MethodImpl (see below)[4].

A class (concrete or abstract) may provide

[4] In V1 it may not be possible to use a MethodImpl to supply the implementation of an inherited virtual method. Their use may be restricted to providing an implementation of a virtual method on an interface, not inherited from a parent class. This remains an open issue.

- 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 parent class has specified it will implement
- alternative implementations for virtual methods inherited from its parent
- implementations for instance, static, and virtual methods inherited from an abstract parent class that did not provide an implementation

A class whose parent class defines a virtual method is permitted to override it with a more permissive accessibility, provided the original method could have been called by the overriding method[5]. Table 1 shows what is permitted.

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

OVERRIDE?	PARENT					
CHILD	**Private**	**Family**	**Assembly**	**F & A**	**F or A**	**Public**
Private	No	No	No	No	No	No
Family	No	Yes	No	No	No	No
Assembly	No	No	Same assembly	No	No	No
F & A	No	No	No	Same assembly	No	No
F or A	No	Yes	Same assembly	Yes	Same assembly	No
Public	No	Yes	Yes	Yes	Yes	Yes

Table 1. Legal Widening of Access to a Virtual Method

8.2 MethodImpls

MethodImpls are a mechanism used to explicitly specify, for a given class, what method body should be used to implement a virtual method. The virtual method may be inherited from its parent[6] or be a method on an interface that the class implements. MethodImpls can be thought of as a very inexpensive mechanism for providing a "forwarding stub" which receives calls to one method (the **declaration**) and implements them by calling another (the **body**).

A MethodImpl can be provided as part of the implementation of a class (the **implementing class**). It specifies, using a **methoddef** or **methodref**, two methods: a **declaration** and a **body**. The body provides the implementation for the declaration.

The **body** must refer to a method implemented in the implementing class or one of its parent classes. The signature of the body must match that of the associated declaration (it

[5] In V1 it may not be possible to widen the accessibility of an inherited virtual method. This remains an open issue.

[6] It is an open issue whether in V1 a MethodImpl will be allowed to specify the implementation of a virtual method inherited from a parent.

must have the same calling convention, return type, and parameter types). The method specified by the body must have an accessibility that would allow it to be called from a method in the implementing class.

Note: in V1 the body must refer to a method implemented directly in the implementing class; it cannot be an inherited method implementation.

The **declaration** must refer to a method that can be overridden by the implementing class. That is, it must be a virtual method inherited from one of its parents or an interface method from an interface that it implements. In addition, the original declaration must specify a method that it would be legal to call from the implementing class (see the earlier table).

Imagine a class **A** that declares it implements an interface **I**, and we are interested in the question "what method implements **I::Foo()**". We define a *matching method definition* as a definition for a virtual method with the same calling convention, return type, and return type as was declared for **I::Foo()**.

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

1. if **A** provides a MethodImpl for **I::Foo()**
   the MethodImpl specifies the method to use. If a MethodImpl is supplied but the body isn't a matching method definition, a class loader exception is generated.
2. *otherwise*, if **A** itself provides a matching method definition for a **public** method named **Foo**
   use that method
3. *otherwise*, if the immediate parent of **A** implements the same interface and provides an implementation for **I::Foo()**
   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**
   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 class loader exception if class **A** is concrete.

9 Value Types

A value type is introduced by a **.class** declaration where the **value** attribute is given. In contrast to reference types, when an instance of value type is created, the object is stored directly in the variable rather than a reference to the object. Thus, if a value type is passed to another variable or to an argument as part of a method call, a new copy is created.

Value types may be used like reference types. If a reference to a value type is needed, the runtime will automatically convert it into a reference type by boxing the value type and when needed unbox it again. More about boxing can be found in section 9.3.

9.1 Overview of Value Types

The following list gives additional information about value types:

1. Defining a value type also creates a corresponding boxed type.
2. In types, the unboxed type is referred to using "**value class** <className>" and the boxed type is referred to using "**class** <className>". These correspond to **ELEMENT TYPE_VALUETYPE** and **ELEMENT_TYPE_CLASS** in the C++ header file cor.h that comes with the NGWS SDK.
3. In metadata generally, the unboxed form is referred to using a **TypeDef** or **TypeRef** and the boxed form is referred to using a **TypeSpec** (derived from a signature that starts with **ELEMENT_TYPE_CLASS**).
4. A value type can have all the same kinds of members as any other type (static, instance, and virtual methods; static and instance fields; properties; and events).
5. A value type, in its boxed form, can inherit from any type that does not define instance fields. In V1, boxed value types must inherit from **System.Value**.
6. **System.Value** is the base type for value types. It provides special behavior for some of the virtual methods inherited by all types (such as **GetHashCode** and **Equals**).
7. Value types are sealed and must be marked as such.
8. Value types can have explicit layout control for fields, as can any class.

CLS: Boxed versions of value types are not allowed in the CLS. A boxed instance of a value type should be treated as System.Object.

9.2 Methods on Value Types

A value type is authored in its unboxed form. That is, the **this** pointer is considered to be a managed pointer to the type, not an object reference. The Execution Engine automatically creates unboxing stubs when needed (instance and virtual methods).

	Unboxed form	**Boxed form**	**Interface**	**Object**
Call	managed pointer to value type	object reference	illegal	object reference
Callvirt	illegal	illegal	object reference	object reference

Table 2: Type of "this" given IL instruction and class of method as specified in methodref/methoddef

To call an instance method defined directly on a value type a **call** instruction is used. The **methodref** or **methoddef** used to call the method differs based on whether the instance is boxed or unboxed. For an unboxed instance, use a **methodref/methoddef** whose parent is a **typedef/typeref/typespec** to the value type. For a boxed instance, use a **methodref/methoddef** whose parent is a **typespec** referring to the value type (with a signature starting with **ELEMENT_TYPE_CLASS**).

To call an instance method defined directly on **System.Object** or **System.Value** a **call** instruction is used. The instance must be boxed, and the **methodref/methoddef** used to specify the method must have a parent of **System.Value** or **System.Object**.

This mechanism can also be used to call, with an unboxed instance of the value type, a virtual or instance method whose implementation is known to exist for the particular value type. The **methodref/methodef** must have a parent that is a **typedef/typeref/typespec** for the value type. Notice that, even though a virtual method is being called, a **call** instruction is used. Because the call is being made with an unboxed instance, the exact type is known at compile time and a dynamic lookup is avoided. This is version brittle, however, since removing the implementation for the particular value class will cause the code to fail.

To call a virtual method with a boxed instance of the value type, a **callvirt** instruction is used. The **methodref/methoddef** can have a parent of **System.Object**, **System.Value**, or an interface that the value type explicitly implements.

9.3 Boxing and Unboxing

An instance of a (unboxed) value type can be converted to an instance of the corresponding boxed type through the built-in operation **box**. Boxing requires making a copy of the value type.

An instance of a boxed value type can be converted to an instance of the corresponding unboxed type through the built-in operation unbox. Unboxing returns a managed pointer to the unboxed type that shares state with the original object.

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.

9.4 Initializing Value Types

The Execution Engine makes only the following guarantees about the creation of instances of value types:

- Static, thread-local, and context-relative variables are initialized to **null** (for boxed instances) or all fields are zeroed/nulled (for unboxed instances)
- All elements of an array of value types are either **null** (for boxed instances) or all fields are zeroed/nulled (for unboxed instances)
- Local variables are not initialized in any way, unless the method is marked to zero-initialize its frame, in which case a boxed instance is **null** or an unboxed instance has all its fields zeroed/nulled

Value types may have any number of initializers, including a "default initializer" which has no parameters. The default initializer is optional. An initializer is recognized because it is marked with the **rtspecialname** and **specialname** attributes and named **.ctor.**

If a value type has an initializer, an instance of its unboxed type can be created as would an instance of a class: the **newobj** instruction is used along with the initializer and its parameters to allocate and initialize the instance.

Verification requires that all fields of a value type be written before they are read or the address of the value type is passed to a method other than an intializer for the value type. Every initializer for 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).

The only way to create a boxed instance of a value type is by using the **box** operation. Since this requires a managed pointer pointer to an unboxed instance it may be necessary to create a local variable to hold the unboxed value prior to boxing it.

The **initobj** instruction calls the default initializer if one exists or zeroes the object if there is no default initializer. This allows IL code generators to produce verifiable code that is resilient to the introduction or removal of a default initializer. For use within a default initializer, the **zeroobj** instruction clears the memory associated with an object.

CLS: CLS compliant producer tools must provide a way for developers to call the default initializer of a value class if it exists. Tools are urged to insert calls to the default initializer automatically, 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.

The Base Class Library provides **System.Array::Initialize()** to call the default initializer (if one exists) on or zero (if there is no default initializer) all instances in an array of unboxed value types.

9.5 Copy Constructors on Value Types

A value type can have a copy constructor, which is simply an initializer (i.e. it is named **.ctor** and has the **specialname** and **rtspecialname** attributes set) with a particular well-known signature (it is a virtual method with one argument whose type is an unmanaged pointer to the value type). The copy constructor is not called automatically, and is not supported on boxed instances (i.e. the garbage collector will not call it).

Languages that support copy construction (such as ISO C++) must insert IL code to do the copy construction as appropriate. The managed code convention is that the caller makes the copy and then passes the address of that copy to the callee.

In order to interoperate with unmanaged code that passes the copy constructed values on the stack P/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: **ELEMENT_TYPE_CMOD_REQUIRED** followed by a type reference To Be Decided. This modifier may only be place in front of a parameter whose type is a managed pointer to a value type.

CLS: Copy constructors are not part of the CLS

9.6 Using Value Types for C++ Classes

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

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++ from another language.

Similar rules can be devised for most languages.

9.6.1 Represent the Class as a Value Type

1. Define a custom attribute, possibly in a reserved part of the System namespace, to indicate that a value type is a VOS 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. Mark all value types that represent unmanaged classes with instances of this custom attribute.
2. Describe the object layout as 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 of necessity be a field for a pointer to the vtable and all constructors for the class must store the appropriate value (see below) into this field.
3. Instance and virtual methods should 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 "unmanaged pointer to the value type." These methods should have a custom attribute attached so that browsers (and the compiler itself) can distinguish "true" static methods from these "introduced" static methods.
4. The method body for instance and virtual methods has access to the pointer to the original object through its first parameter, and through this 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.
5. Classes that have user supplied copy constructors or destructors can't be boxed, can't be fields of managed types, and can't be passed by value directly. For this reason, there is a special metadata bit that means "unless you understand the custom attribute on this type, you shouldn't use it." Most C++ classes are usable by other languages just as any other value type would be and should *not* set this bit.
6. A value type can have a copy constructor, which is simply an initializer (i.e. it is named **.ctor** and has the **mdSpecialName** and **mdRTSpecialName** bits set) with a particular well-known signature (it is a method with one argument whose type is

an unmanaged pointer to the value type). The copy constructor is not called automatically, and is not supported on boxed instances (i.e. the garbage collector will not call it).

7. For classes that have a copy constructor, the compiler must insert code to do the copy construction as appropriate. The managed code convention is that the caller makes the copy and then passes the address of that copy to the callee. Thus methods that take a copy-constructed parameter appear in the C++ source code to be called by value, while in the IL they appear to be passed as a pointer to the copy. In addition, to allow automatic interoperation with unmanaged code via P/Invoke, copy-constructed parameters must be marked with a particular required custom attribute (via **ELEMENT_TYPE_CMOD_REQUIRED**).

9.6.2 Represent the Vtable as another Value Type

1. The vtable itself should be the value of a static variable with specified RVA. The type of this variable should be a class with unique mangled name (or a type nested within the object's type), explicit layout, and with named fields of type "function pointer to native method with correct signature."
2. The data for the vtable should be stored in an appropriate section of the PE file and the appropriate fixups must be stored in the NGWS SDK header within the PE file. If unmanaged compatibility is required the entries may be forced to 32 bits wide and the PE file header marked for 32-bit architectures only. Otherwise, the entries should be 64 bits wide.

10 Special Types

10.1 Arrays

An array is a contiguous memory block that stores a collection of values of the same type.

An **array type** is defined by specifying the element type of the array, the **rank** (number of dimensions) of the array, and the upper and lower bounds of each dimension of the array. All of these are included in any signature of an array type, although they may be marked as dynamically (rather than statically) supplied. Hence, no separate definition of the array type is needed.

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

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

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

CLS 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.

Array types form a hierarchy, with all array types inheriting from the type **System.Array**. This is an abstract class that represents all arrays regardless of the type of their elements or their rank. Arrays of one dimension with zero lower bound for their elements (sometimes called **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 bound on the array. Zero-dimensional arrays are not supported.

Consider the following examples, using the syntax of the IL Assembler (ilasm):

Static specification of type	Actual type constructed	Allowed in CLS?
`Int32[]`	vector of int32	Yes
`int32[0..5]`	vector of int32	Yes
`int32[1..5]`	array, rank 1, of int32	No
`int32[,]`	array, rank 2, of int32	Yes
`int32[0..3, 0..5]`	array, rank 2, of int32	Yes

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

10.2 Delegates

The classical COM implementation of delegates suffers from a number of problems:

- Compilers must specify the entire definition of the delegate class in metadata, bloating the metadata
- Because delegates are type distinct, each such class requires the overhead of an entire runtime class description
- Over time there are additional features that can be easily added to delegates using new methods (async, for example) but this is inhibited by the compiler providing the definition
- In a future version it would be extremely convenient if function pointers, method pointers, and delegates could be handled polymorphically by the calli instruction
- Both single-cast and multi-cast delegates are supported. There is serious support for removing multi-cast and leaving this to infrastructure services (eventing).

Due to these considerations the NGWS SDK moves delegates from their current status of type-distinct compiler-defined types toward the basic model of function pointers and arrays, which are specified structurally by the compiler and then manufactured on demand by the runtime. This does not remove any functionality, since type-distinct delegates can be recreated through the use of a value class with a single field (the underlying delegate).

10.2.1 Changes to Delegates

For the NGWS SDK, the following changes to delegates were made:

- A level of abstraction for compilers that use the unmanaged Metadata APIs was introduced.
- The methods related to combining delegates from the **System.Delegate** class to an independent class were moved.

In a future version (not during V1) we will consider extending the **calli** instruction to be polymorphic across function pointers and delegates with the equivalent signature.

With regard to the existing specification on implementing delegates, the only change is in creating the delegate class. Over time, the implementation of the verifer and the **castclass** and **isinst** instructions may be changed to allow delegates with the same signature to be considered equivalent. Compiler changes would be required to make this visible to users, but the infrastructure can be changed without impact.

10.2.2 Moved Delegate Combine Methods

The NGWS SDK has removed the existing **Combine** method off of **System.Delegate** and into a separate class. This new class, over time, will be extended to allow a variety of combination methods including, for example,

- The ability to continue on to the next combined element even if an exception occurs.
- The ability to call the combined delegates asynchronously.
- The ability to combine delegates with function pointers or explicit functions with the correct signature (not in V1).

10.2.3 Members of Delagates

Delagates need to define four Methods (no Fields), as follows:

Member	Definition
A constructor	public specialname rtspecialname instance void .ctor (class System.Object, int32) runtime managed
Invoke method	public final virtual instance <retType> Invoke (<p1Type>, <p2Type>, . . .) runtime managed
BeginInvoke method	public final virtual instance class System.IasyncResult BeginInvoke (<p1Type> , <p2Ttype>, . . . , class System.AsyncCallbackDelegate, class System.Object) runtime managed
EndInvoke method	public final virtual instance <retType> EndInvoke (<p1Type> , <p2Type>, . . . , class System.IasyncResult) runtime manage

For the BeginInvoke method, any *Out-ByRef* parameters need to be excluded.

For the EndInvoke method, any **in** parameters parameters need to be excluded.

10.3 Enumerations (Enums)

Enumerations (enums) provide type-distinct classes that are equivalent to an underlying built-in integer type. Like integers, enums can be either boxed or unboxed. In their unboxed form (only) they automatically convert to and from their underlying type. Enums also provide a name scope to provide names for values of this new type. They do *not* provide a guarantee that values of this type have one of the named values (unlike Pascal). Enumerations are marked in the metadata by setting both the **tdEnum** and the **tdValueType** bits.

CLS: Enums may only have underlying types of I1, I2, I4, or I8

Enums may have only one instance field and it must be marked with the bits **fdRTSpecialName** and **fdSpecialName**.

CLS: The field must be named **.value**.

The type of the instance field may be any built-in integer type, called the “underlying type” of the enum.

Enums, when boxed, inherit from **System.Enum** and must be explicitly marked as such.

Enums must not have any methods and are sealed (and must be marked as such). Because they cannot have methods they cannot implement interfaces.

Besides the one instance field, all other fields must be static literal fields (Init-only static fields are not supported). These must be optimized away by the compiler and no space will be allocated at runtime for these fields. The fields and their values are visible through Reflection.

CLS: All fields of an enum must be public.

Enums may not have properties or events.

When used in a signature, enums are encoded as ELEMENT_TYPE_VALUETYPE (for historical reasons). 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 from 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 **typeref/typedef/typespec** 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.

10.4 Pointer Types

A **pointer type** is defined by specifying a location signature for the location the pointer references. Any signature of a pointer type includes this location signature. Hence, no separate definition of the pointer type is needed.

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

CLS Note Pointer types are not part of the CLS. For CLS consumer there is no need to pointer types. For CLS extender there is no need to provide syntax to define or access pointer types. For CLS framework pointer types must not be externally exposed.

The syntax for declaring a pointer type is as follows:

`<type> ::=`	**Section**
`<type>` **`&`**	10.4.1.2
`| <type>` **`*`**	10.4.1.3
`| …`	5.2

The * indicates a transient pointer, while the **&** indicates a managed pointer. The type for unmanaged pointers is unsigend integer (U).

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.

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.

10.4.1.1 Unmanaged Pointers

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

- Unmanaged pointers are not reported to the garbage collector and can be used in any way that an integer can be used.
- It is best to think of unmanaged pointers as unsigned (i.e. use **conv.ovf.u** rather than **conv.ovf.i**, etc.).
- 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

10.4.1.2 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 **`ELEMENT_TYPE_BYREF`** in a signature for a return value, local variable or an argument or by using a by-ref type for a field or array element.

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

10.4.1.3 Transient Pointers

Transient pointers (*****) are intermediate between managed and unmanaged pointers. They are created within the EE by certain IL 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 IL 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 (on method call 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.
- The verifier treats transient pointers as managed pointers.

10.5 Function Pointer Types

The NGWS SDK supports function pointers. In contrast to delagates, function pointers are not object. A function pointer has a type, which is the signature of the method including its calling convention.

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

<type> ::=	**Section**
method <callConv> <type> ***** **(** <signature> **)**	10.5
| …	5.2

Variables that have the type of the function pointer may store a pointer to the function which may be used to call the method.

A pointer to a function is obtained with the **ldftn** instruction (see section 19.2.4). A function may be called with a pointer with the **calli** instruction (see section 19.2.3.4).

11 Signatures

Signatures capture the part of contracts that can be enforced by the runtime. There are two kinds of signatures: those that are used to describe the parameters to methods, properties, and events (called "method signatures" for simplicity) and those that are used to describe the local variables of a procedure.

11.1 Method Signatures

Method signatures are used to specify the types of parameters to methods, properties, and events. In addition, they are used to specify the arguments passed to a method that is called through a function pointer rather than directly by name.

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`**.

As shown by the following grammar, a signature consists of any number of paramters.

```
<signature> ::= [<param> [, <param>]*]
```

The information about an individual parameter can be seen through a `System.Reflection.ParameterInfo` and can be created using `System.Reflection.Emit.ParameterBuilder`.

```
<param> ::=
      ...
    | [<paramAttr>*] <type> [marshal ( [<nativeType>] )] [<id>]
```

An individual parameter must either be the special token "**`...`**" or have a defined type and optionally have additional information.

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 signature. For signatures on methods, properties, and events, it must be the last parameter. When describing arguments passed via a function pointer, however, it separates the arguments being past as part of the normal (fully described) parameter list and those that are being passed as part of the variable argument list.

The attributes in <paramAttr>* specify special handling of certain parameters (see **CorHdr.h** under **CorParamAttr**):

```
<paramAttr> ::=
      [in]
    | [lcid]
    | [opt]
    | [out]
    | [retval]
```

- **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.
- **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 NGWS SDK return value will be made visible to those clients.
- **opt** indicates that this and all subsequent parameters are optional.
- **lcid** indicates that this parameter provides the locale ID to unmanaged COM clients.

The default value of optional parameters can be set using the method **`SetConstant`** on type **`System.Reflection.Emit.ParameterBuilder`**. There is no syntax in the assembler for this.

Examples:

`int32`
a single parameter of type 32-bit integer, it is an input parameter

`class [mscorlib.dll]System.Reflection.MethodBase, [in][out] int32& MyInt`
a pair of parameters. The type of the first is **System.Reflect.MethodBase** from the assembly named **mscorlib.dll** and it is an input parameter. The second parameter is an in/out parameter named **MyInt** and it is a managed pointer to a 32-bit integer.

`String marshal (lpstr), ...`
a method that takes one or more arguments, with the first one being a managed string which must be marshaled to an unmanaged **LPSTR**. This would only make sense if the method implementation was marked as being unmanaged.

`String, …, int32, single`
this might be the signature of a call site to the method described just above. The "**...**" must appear in the same place in both and the type of the first argument (given here) must match the type of the first parameter (given above). In addition to this required argument we are passing two additional arguments of the specified types.

11.1.1 Marshal

The keyword **`marshal`** is used to specify how this parameter should be marshaled to or from unmanaged COM or via Pinvoke; it is used only if the implementation of the method is declared to be via COM or Pinvoke (see also section 12.7.2).

11.2 Local Variable Signatures

The local variables of a method are also 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>]*
```

Information about local variables can be created using the **`System.Reflection.Emit.LocalBuilder`** class.

```
<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.

11.3 Primitive Types in Signatures

The NGWS SDK 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**.

Name in ilasm	CLS Type	Type in Base Class Library	Description	Name from CorHdr.h (ELEMENT_TYPE _xxx)
bool	Yes	System.Boolean	Boolean (true/false)	**BOOLEAN**
char	Yes	System.Char	Unicode Character	**CHAR**
class System.Object	Yes	System.Object	Object or boxed value type	**OBJECT**
class System.String	Yes	System.String	Unicode String	**STRING**
float32	Yes	System.Single	IEEE 32-bit floating point	**R4**

Name in ilasm	CLS Type	Type in Base Class Library	Description	Name from CorHdr.h (ELEMENT_TYPE _xxx)
float64	Yes	System.Double	IEEE 64-bit floating point	**R8**
int8	No	System.Sbyte	Signed 8-bit integer	**I1**
int16	Yes	System.Int16	Signed 16-bit integer	**I2**
int32	Yes	System.Int32	Signed 32-bit integer	**I4**
int64	Yes	System.Int64	Signed 64-bit integer	**I8**
native int	No		Singed, native size integer	**I**
native unsigned int	No	System.Int[7]	Unsigned, native size integer	**U**
typedref	No	System.Typed-Reference	Pointer and runtime type	**TYPEDBYREF**
unsigned int8	Yes	System.Byte	Unsigned 8-bit integer	**U1**
unsigned int16	No	System.Uint16	Unsigned 16-bit integer	**U2**
unsigned int32	No	System.Uint32	Unsigned 32-bit integer	**U4**
unsigned int64	No	System.Uint64	Unsigned 64-bit integer	**U8**
wchar	Yes	System.Char	Unicode Character	**CHAR**

11.4 Native Data Types

This section is a brief summary of native types. More information about native types can be found in the Data Marshaling specification (see the NGWS SDK).

The NGWS SDK provides automatic marshaling to and from a variety of native (unmanaged) data types and corresponding managed data types. This information can be set using the **SetMarshal** method on the class **System.Reflection.Emit.ParameterBuilder**. The complete list of these types is in **CorHdr.h** as the enumeration **CorNativeType**.

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

[7] In the NGWS SDK, not all mathematical operations are supported for this type. Future versions may support more operations.

<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>)**	Custom marshaller. The first string is a custom marshaller type name. The second string is an optional cookie.
| **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>
| **float**	Size agnostic floating point number.
| **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.
| **lptstr**	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.
| <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. An optional multiplier may be provided to increase the size by some factor.
| **method**	A function pointer.
| **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 types. These are used for marshalling. All items that are marked `/* roundtrip only */` should not be used, however might be generated by the disassembler. The native constants for variants types for older versions of Windows can be found in MSDN.

<variantType> ::=	**Description**
blob /* roundtrip only */	Bytes prefixed with the length.
| **blob_object** /* roundtrip only */	Blob contains an object.
| **bstr**	A COM style BSTR with a prepended length and Unicode characters. (Not supported for StringBuilder)
| **bool**	Boolean. 4-byte integer value where a non-zero value represents TRUE and 0 represents FALSE.
| **carray** /* roundtrip only */	C style array.
| **cf** /* roundtrip only */	Clipboard format.
| **clsid** /* roundtrip only */	Class ID.
| **currency**	A currency structure.
| **date**	A data strucuture.
| **decimal**	16 byte fixed point number
| **error**	Return value for HRESULT methods that failed.
| **filetime** /* roundtrip only */	Structure for a file time.
| **float32**	32-bit single precision floating point number.
| **float64**	64-bit double precision floating point number.

hresult	Standard return type.
idispatch *	COM style IDispatch interface.
[**unsigned**] **int** /* roundtrip only */	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** /* roundtrip only */	Signed or unsigned 64-bit integer
iunknown *	Native pointer to COM style IUnknown interface.
lpstr /* roundtrip only */	A pointer to a null terminated array of ANSI characters.
lpwstr /* roundtrip only */	A pointer to a null terminated array of Unicode characters.
null /* roundtrip only */	SQL style null.
record	User defined type.
safearray /* roundtrip only */	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.

12 Methods

Methods specify the behavior of a program. There are several kinds of methods in the NGWS SDK:

- global methods (section 12.6)
- static methods of a class (section 12.3.2.1)
- instance methods of a class (section 12.3.2.2)
- virtual methods of a class (section 12.3.2.3)
- instance constructors, a special kind of instance method (section 7.3.2)
- class constructors, a special kind of static method (sections 7.3.3)

However, 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> **{** <methodDecl>* **}**

The following sections will give more details on these parts of a method definition.

12.1 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 12.3)
- an optional description of the kind of call to use
- a return type with optional attributes (section 11.1)
- optinal marshalling information (see also section 11.1.1)
- a method name
- a signature in brackets
- and any number of implementation attributes (section 12.3.4)

as also shown by the following syntax rule:

```
<methodHead> ::=
     <methAttr>* [<callKind>] [<paramAttr>*] <type> [marshal (
       [<nativeType>] )] <methodName> ( <signature> ) <implAttr>*
```

In the NGWS SDK, 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 <paramAttr>'s should not be used with the return type.

Method that do not have return value must use the keyword **void** as the return type. **void** is an instance of **System.Void**.

12.1.1 Method Name

Most method names are a <dottedname>. The exception are constructors. 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>
```

12.1.2 Kinds of Calls

The NGWS SDK supports various kinds of method calls. The two managed kinds of calls are **default** and **vararg**. **default** specifies the standard kind of call of the NGWS SDK. **Vararg** is specifies that the method accepts a variable number of arguments and thus needs to be called in a special way.

The unmanaged kinds of calls are primarily to support languages which are similar to C++ and cannot compile the method as a managed method.

Unmanaged cdecl ist 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.

```
<callKind> ::=
      default
    | unmanaged cdecl
    | unmanaged fastcall
    | unmanaged stdcall
    | unmanaged thiscall
    | vararg
```

12.2 Method Body

The method body contains the instruction 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 table 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.

<methodDecl> ::=	Description	Section
.custom <customDecl>	Definition of custom attributes.	17
| **.data** <datadecl>	Emits data to the data section of the method.	13.4
| **.emitbyte** <int32>	Emits an int32 to the code section of the method.	12.2

	.entrypoint	Specifies that this method is the entrypoint to the application (only one such method is allowed). 12.2
	.locals [**init**] **(** <localsSignature> **)**	Defines a set of local variables for this method. 12.2.1
	.maxstack <int32>	Definies the maximum size of the stack, specified by the int32. 12.2
	.override <typeSpec>**::**<methodName>	Sets this method as the implementation for the method specified in the instruction. 12.2
	.param [<int32> **]** [**=** <fieldInit>]	Sets method parameter number <int32> as the owner of the following custom attributes. 12.2.2
	.vtentry <int32> **:** <int32>	.vtentry <entry> : <slot> 12.2.3
	.zeroinit	Specifies that all local variables are initialized to zero in this method. 12.2
	<externSourceDecl>	**.line** or **#line** 3.7
	<instr>	An instruction 19
	<codeLabel> **:**	A label 3.4
	<scopeBlock>	See below 12.5
	<securityDecl>	**.permission** or **.capability** 16
	<sehBlock>	An exception block 18

12.2.1 .locals

.locals is used to define local variables for this method. If **init** is specified, default constructors are called for each local variable. The <localsSignature> lists the local variables. Each local variable receive a zero based index that is unique within the method. The index is assigned in increasing order based using the order of declaration beginning at the start of the method. A local variable can be either accessed by its optional name, or by its unique index. See also the desciption of scoping blocks in section 12.5.

12.2.2 .param

Sets method parameter number <int32> as the owner of the following custom attributes. **.param[0]** specifies the return value. All of the method's own custom attributes must be declared before the first .param directive. <fieldInit> if used specifies the default value.

12.2.3 .vtentry

Places the token of this method at the specified slot of the virtual method table entry. Used together with <vtfixupDecl> (section 7.5.2.2).

12.3 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.

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.	12.3.4
| **assembly**	Assembly accessibility	12.3.1
| **famandassem**	Family and Assembly accessibility	12.3.1
| **family**	Family accessibility	12.3.1
| **famorassem**	Family or Assembly accessibility	12.3.1
| **final**	Specifies that this method cannot be overriden by subclasses.	12.3.2
| **hidebysig**	Hide by signature.	12.3.2
| **newslot**	Specifies that this method shall get a new slot in the virtual method table.	12.3.3
| **pinvokeimpl (** [<QSTRING> [**as** <QSTRING>]] [<pinvAttr>*] **)**	pinvokeimpl([“DLL_name” [as “ExportName”]] [attributes])	12.3.5
| **private**	Private accessibility	12.3.1
| **privatescope**	Privatescope accessibility.	12.3.1
| **public**	Public accessibility.	12.3.1
| **rtspecialname**	The method name needs to be treated in a special way by the runtime.	12.3.6

| **specialname**	The method name needs to be treated in a special way by some tool.	12.3.4
| **static**	Specifies that this method is a static method of a type.	12.3.2
| **unmanagedexp**	Marks for exports to unmanaged world.	12.3.5
| **virtual**	Specifies that this method is a virtual method.	12.3.2

12.3.1 Accessibility Information

The accessibility attributes are **assembly**, **famandassem**, **family**, **famorassm**, **private**, **privatescope** and **public**. These attributes are exclusive. The default is privatescope. Accessibility attributes are described in section 6.3. **privatescope** is the most restrictive attribute and specifies that the method cannot be accessed unless the method definition token is provided.

12.3.2 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. Abstract methods may not be final. Only virtual methods may use **hidebysig**.

Final methods may not be overriden by subclasses of this class. This makes sure the functionality provided by the implementing class is not modified by other implementations.

Hidebysig hides implementations of the parent class using the full signature and not just the name. The defualt is hide by name, which hides all methods with the same name regardless of signature. Instead of the hidden method, the new method is executed.

Static and **virtual** specify type of method to be defined. The default is an instance method. The following subsections briefly describe each kind of method.

12.3.2.1 Static Methods

Static methods may access only be used with static fields (see section 13) of a class or call only other static methods. They may be called without creating an instance of the class. Static methods are very similar to global methods, but are a member of a type.

12.3.2.2 Instance Methods

In contrast to static methods, instance methods accept a *this* pointer and thus require that an instance of the class be created. Instance methods can access to all fields of a class and possibly their parent classes and may call any other method of a class and possibly their parent classes.

The *this* pointer is not included in the signature, but is automatically added to the signature of a method as the first parameter. However, when an instance method is called the an instance of the class must be passed to the method as the first argument explicitly, even though this argument is not apparent from the signature.

12.3.2.3 Virtual Methods

Virtual methods need to be used in Object Oriented Programming (OOP). A virtual method is like an instance method, except that its call is redirected through a virtual method table. Subclasses of this class may override the entries in this virtual method table, which means that rather than the original implementation the implementation of the subclass is called.

The *this* pointer must be passed to virtual methods similar to instance methods.

12.3.3 Virtual Method Table Information

The only attribute in this group is **newslot**. **Newslot** can only be used with virtual methods and specifies that this method shall get a new slot in the virtual method table, and not override a method from the parent class. Calls to the method by the superclasses of this method, will be redirected to the implementation of the superclass of this class. However, calls of this class and its subclasses will be redirected to the new implementation, or to any overriding version. However, the implemenation of the superclass may still be overriden by subclasses by using an explicit reference to the implementation of the superclass.

The default is that the implemenation of the subclass overrides the implementation of the superclass.

12.3.4 Implementation Attributes

The two implementaion attributes are **abstract** and **specialname**. **Abstract** can only be used with non-final virtual methods. This attributes may be combined.

Abstracts 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 7.1).

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

12.3.5 Interoperation Attributes

These attributes are for interoperation with COM+ 1.x and classical COM applications. These attributes are **pinvokeimpl** and **unmanagedexp**. This attributes may be combined.

Pinvokeimpl instructs the runtime to use the platform invoke functionalityto invoke an unmanaged method in the specified dll with the specified export name. (see also 12.7.2.2).

unmanagedexp marks this method for export to an unmanaged environment.

12.3.6 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” (class constructor).

12.4 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..

<implAttr> ::=	Description	Section
forwardref	Specifies that the body of this method is not specified with this declaration.	12.4.3
| **il**	Specifies that the method contains standard il code.	12.4.1
| **internalcall**	Used only for disassembling purposes.	12.4.3
| **managed**	Specifies that the method is a managed method.	12.4.2
| **native**	Specifies that the method contains native code.	12.4.1
| **noinlining**	Spefies that the runtime shall not attempt to inline the method.	12.4.3
| **ole**	Indicates method signature is mangled to return HRESULT, with the return value as a parameter.	12.4.4
| **oneway**	Specifies “fire and forget” convention.	12.4.3
| **optil**	Specifies that the method contains OptIL code.	12.4.1
| **runtime**	The body of the method is not defined but produced by the runtime.	12.4.1
| **synchronized**	The method will be executed in a single threaded fashion.	12.4.3
| **unmanaged**	Specifies that the method is unmanaged.	12.4.2

12.4.1 Code Implementation Attributes

The code implementation attributes are **il**, **native**, **optil**, and **runtime**. This attributes are exclusive. The default is **il**.

This attributes specify the type of code the method contains. **runtime** specifies that the implemenation of the method is automatically provided by the runtime and is primarily used for for the constructor and invoke method of delagates.

12.4.2 Managed or Unmanaged Information

The two option here are **managed** or **unmanaged**. The default is **managed**.

Managed code is code that provides enough information to allow the NGWS SDK runtime to provide a set of core services, which include

- Given an address inside the code for a method, locate the metadata describing the method
- Walk the stack
- Handle exceptions
- Store and retrieve security information

Only managed code may access managed data. To produce verifiable IL code a compiler must produce managed code.

12.4.3 Implementation Information

The attributes in this group are **forwardref**, **internalcall**, **synchronized**, **noinlining**, and **oneway**. The attributes may be combined.

Forwardref specifies that the body of the method is specified with another declaration that is part of the assembly. This is similar to a forward declaration in C.

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 methods in some cases.

Once a **oneway** method is called, the caller will not hear back from the method. The method will be executed and terminate on its own. The caller is encouraged to continue with its normal processing. The method must return void and have only **in** parameters (see section 11.1). Once the method is called, the method may have synchronous or asynchronous side-effects with respect to the caller.

12.4.4 Interoperation

The attribute **ole** used for compatibility with unmanaged COM. It instructs the runtime to convert the signature of a method when 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** paramter 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.

12.5 Scope Blocks

Scope blocks are syntactic sugar primary for readability and debugging purposes. They may be also used by compilers.

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 inner most scope block, than at the next level, and so on until the top-most level, which is the method declaration level, is reached. A declaration in an inner scope block hide declarations in the outer layers.

If duplicate declarations are used, the reference will be resolved to the first occurrence. Duplicate declarations are not recommended.

Scoping does not affect the life time of a local variable. All local variables are created and initialized when the method is entered. They stay alife 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.

```
<scopeBlock> ::= { <methodDecl>* }
```

12.6 Method Calls

Non-virtual methods are called with using the **call** instruction. The call instruction takes a method reference as part of the instruction and expects the arguments of the method on the stack. The first argument is pushed first onto the stack. For instance methods, the first argument is the *this* pointer to the instance.

The syntax for a **call** instruction is as follows:

call <callConv> <type> [<typeSpec> **::**] <methodName> **(** <signature> **)**

The main difference to a method definition head (see section 12.1) is that a call convention, rather than a call kind is needed. In addition, a call doesn't take predifined and implementation attributes.

Virtual methods are called using the **callvirt** instruction, which has the same syntax as the **call** instruction.

Method may be also called using a pointer to the method with the **calli** instruction.

The syntax for calli does not need to include the method name:

calli <callConv> <type> **(** <signature> **)**

A pointer to a method is loaded with the **ldftn** instruction. More about this instruction can be found in section 19.2.3.4.

12.6.1 Call Convention

A call convention includes a call kind (see section 12.1.2). In addition it has the optional keywords **instance**, or **instance explicit**.

```
<callConv> ::= [instance [explicit]] [<callKind>]
```

By default, the method call will be treated as call to a static method. If instance is specified, the method call will be treated as a method call that accepts a *this* pointer. **callvirt** instrcutions need to be always **instance**.

Explicit applies only to **instance** methods. If **explicit** is specified, the method *this* pointer is included explictly in the method signature. This is used only in the declaration of function pointers in connection with the **calli** instruction. It is not used with the **call** or **callvirt** instruction.

12.7 Global Methods

12.7.1 Managed Native Calling Conventions

There are two managed native calling conventions used on the x86. These may be changed over time. 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.

12.7.1.1 Standard x86 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.

Consider the following examples.

1. `static System.Int32 F(System.Int32 x)`
   The incoming argument (x) is placed in ECX; the return value is in EAX
2. `static double F(System.Int32 x, y, 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
3. `static double F(System.Int32 x, double y, 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
4. `virtual double F(System.Int32 x, System.Int64 y, 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 stack); the return value is on the top of the FP stack
5. `virtual System.Int64 F(System.Int32 x, double y, 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 stack); the return value is in EDX/EAX
6. `virtual System.Variant F(System.Int32 x, double y, z)`
   Recall that `System.Variant` is a value type. Hence **this** is passed in ECX, a pointer to the return buffer is passed in EDX, x is pushed then y then z (hence z is top of stack); the return value is stored in the return buffer.

12.7.1.2 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 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.

12.7.1.3 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 varargs) 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." Post-V1 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.

12.7.2 Accessing Unmanaged Methods

12.7.2.1 Via 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 NGWS SDK provides mechanisms useful to both implementers and users of existing Unmanaged 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 aren't part of a managed class), static functions, or parameterized constructors.

- Given an existing Unmanaged 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* Unmanaged 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 NGWS SDK provides automatically 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 Unmanaged COM users. This set of rules guarantees that the data types exposed to the Unmanaged COM user are unmanaged types that can be marshaled automatically by the NGWS SDK.
- **Implementers** run the **comreg** tool to register their implementation with Unmanaged COM for location and activation purposes.
- **Users** of *existing* Unmanaged COM interfaces simply import the metadata produced by **tlbimp**. They can then reference the (managed) types defined there and the NGWS SDK 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 Unmanaged COM (unmanaged) code or the NGWS SDK (managed) code: the interfaces they see use managed data types, and hence do not need explicit memory management.
- For some existing Unmanaged COM interfaces, the NGWS SDK 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.

12.7.2.2 Using Platform Invoke

Umnanged methods may be invoked using the platfrom invoke functionality of the NGWS runtime. Platfrom invoke 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 platform invoke, are marked as **pinvokeimpl**. **Pinvokeimpl** takes an optional name of the dll with an optional name of the exported method and any number of attributes.

A function declared with **pinvokeimpl** does not have a body.

If the exported name is not specified, the declared name is assumed. If the no dll name is specified the current module is assumed. Thus, both unmanaged methods in the same PE file and in another PE file may be invoked with the **pinvokeimpl** command.

<methAttr> ::=	Description	Section
pinvokeimpl ([<QSTRING> [**as** <QSTRING>]] [<pinvAttr>*] **)**	**pinvokeimpl(** [**"**DLL_name**"** [**as "**ExportName**"**]] [attributes]**)**	12.7.2.2
| …		12.3

The following grammar shows the attributes of a **pinvokeimpl** instruction.

<pinvAttr> ::=	**Description**
Ansi	ANSI character set.
| **autochar**	Determine character set automatically.
| **cdecl**	Standard C style call.
| **fastcall**	C style fastcall.
| **lasterr**	Indicates that method supports C style last error querying.
| **nomangle**	Pinvoke is to use the member name as specified.
| **ole**	Indicates method signature is mangled to return HRESULT, with the return value as a parameter.
| **stdcall**	Standard C++ style call.
| **thiscall**	The method accepts an implicit this pointer.
| **unicode**	Unicode character set.
| **winapi**	Pinvoke will use native callconv appropriate to target windows platform.

12.7.2.3 Via Function Pointers

Unmanaged pointers can also be called via function pointers. There is no difference between calling managed or unmanaged fuctions with pointers. However, the unmanaged function needs to be declared with **pinvokeimpl** as decribed in section 12.7.2.2. Calling managed methods with function pointers is described in section 12.6.

12.7.2.4 Unmanaged Mechanisms: "It Just Works" and "Platform Invoke"

It Just Works (IJW) scenarios are designed for programmers who wish to use existing unmanaged data types for interoperation with unmanaged code. The NGWS SDK 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 NGWS SDK. 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 NGWS SDK 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 It Just Works 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.

12.7.2.5 Calling from Managed to Unmanaged

From the point of view of an IL code generator, both IJW and Pinvoke are handled in the same way for calls to statically named methods. There is a call to a method using the ordinary IL 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 to indicated it is implemented by 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 NGWS SDK execution engine and any IL-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 the verifier) that takes a function pointer and converts it from any given calling convention to any other, by producing a transition stub as needed.

12.7.2.6 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 NGWS SDK 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.
- 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 *un*managed entry point. The exporting module will have exported this entry point, and made it point to a table entry (also fixed up by the NGWS SDK 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.

12.7.3 Exporting Managed Methods to the Unmanaged World

The **.vtfixup** directive can be used to export managed method to the unmanaged world. More information about this command can be found in section 7.5.2.2.

13 Fields

Fields store the data of a program. The NGWS SDK allows the declaration of

- global fields (section 13.3)
- instance fields of a type (section 13.2)
- static fields of a type (section 13.2)

This section specifies the syntax common to all fields. 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 13.1)
- a type
- a name
- and either a <fieldInit> form or a data label

This is also shown by the following grammar.

<fieldDecl> ::=	**Comments**
[**[** <int32> **]**] <fieldAttr>* <type> <id> [**=** <fieldInit> | **at** <dataLabel>]	[<int32>] is field offset, for explicit layout only, ignored in global fields; **at** <label> specifies the data item label

The optional field offset is ignored for global fields. For classes, the field will be stored at the specified offset for the class. Classes that use this feature must be declared **explicit** (see also section 7.1.1.2).

Global fields must have a data label associated with them. The data label specifies where the data of the field is located.

13.1 Field Attributes

Field attributes can be optionally added to a field declaration. The use of field attributes may not be combined with a data label.

Field attributes are for metadata only. They do not have any affect on the actual value of the field and do 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. Field attributes are typically used with **literal** fields (see section 13.2.2).

The following table lists the options for a field init. The used type has to agree with the type of the field. The description column provides additional information.

<fieldInit> ::=	**Description**
bytearray (<bytes> **)**	Array of type U1 (8 bit). <bytes> specifies the actual bytes.
| **float32 (** <float64> **)**	32 bit floating point number, with the floating point number

	specified in parentheses. The number needs to fit in 32 bits.
| **float32 (** <int32> **)**	<int32> is binary representation of float
| **float64 (** <float64> **)**	64 bit floating point number, with the floating point number specified in parentheses.
| **float64 (** <int64> **)**	<int64> is binary representation of double
| **int8 (** <int8> **)**	8 bit integer with the integer specified in parentheses.
| **int16 (** <int16> **)**	16 bit integer with the integer specified in parentheses.
| **int32 (** <int32> **)**	32 bit integer with the integer specified in parentheses.
| **int64 (** <int64> **)**	64 bit integer with the integer specified in parentheses.
| <QSTRING>	String. <QSTRING> is stored as ASCII
| **wchar * (** <QSTRING> **)**	Pointer to a string. <QSTRING> is converted to Unicode.

13.2 Predefined Attributes on 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> ::=	**Derscription**	**Section**
assembly	Assembly accessibility.	13.2.1
| **famandassem**	Family and Assembly accessibility.	13.2.1
| **family**	Family accessibility.	13.2.1
| **famorassem**	Family or Assembly accessibility.	13.2.1
| **initonly**	Field can only be mutated inside a constructor.	13.2.2
| **literal**	Field represents a constant literal.	13.2.2
| **marshal (** [<nativeType>] **)**	Marshaling information.	13.2.3
| **notserialized**	Field is not serialized with other fields of the class.	13.2.2
| **pinvokeimpl (** [<QSTRING> [**as** <QSTRING>]] [<pinvAttr>*] **)**	pinvokeimpl([“DLL_ name” [as “ExportName”]] [attributes])	13.2.3
| **private**	Private accessibility.	13.2.1

privatescope	Privatescope accessibility.	13.2.1
public	Public accessibility.	13.2.1
rtspecialname	Special treatement by runtime.	13.2.4
specialname	Special name for other tools.	13.2.4
static	Field is a static field.	13.2.2

13.2.1 Accessibility Information

The accessibility attributes are **assembly**, **famandassem**, **family**, **famorassm**, **private**, **privatescope** and **public**. These attributes are exclusive. The default is privatescope. Accessibility attributes are described in section 6.3. **privatescope** is the most restrictive attribute and specifies that the field cannot be accessed unless the field definition token is provided.

13.2.2 Field Contract Attributes

Method contract attributes are **initonly**, **literal**, **static** and **notserialized**. These attributes may be combined. Only static fields may be **literal**.

Static specifies that the field is associated with the type itself rather than with an instance of the type, which is true for instance variables. Thus, the field can be accessed without having an instance of a class, e.g. by static methods. As an effect of this static, this field is shared between all instances of a class and any modification of this field will affect all instances.

Initonly fields may only be mutated inside of a (class or instance) constructor. Once the constructor exits, **initonly** fields behave like constants. This feature may be used to initalize constants, which have values that cannot be computed at compile time.

Notserialized specifies that this field is not serialized when an instance of this class is serialized (see section 7.1.1.5).

literal specifies that this field represents a constant value. **literal** fields do not have any other purpose than for debugging. **Literal** fields become part of the metadata but cannot are not added as variables to the name space of a class. Thus, they cannot be accessed by the code. **literal** fields may be assigned a value by using the <initField> syntax (see section 13).

13.2.3 Interoperation Attributes

The attribute **pinvokeimpl** is used for interoperation with COM+ 1.x and classical COM applications. **Pinvokeimpl** specifies that the field is defined and shall be imported from an unmanaged environment in the specified dll with the specified export name. (see also 13.4.2).

The attribute **marshal** 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. This feature is also used for interoperation with COM+ 1.x and classical COM.

13.2.4 Other Attributes

The attribute **rtspecialname** indicates that the field name shall be treated in a special way by the runtime.

In contrast to **respecialname**, **specialname** indicates that the field name special meaning to some tools.

13.3 Global Fields

The use of metadata allows the static data to be significantly more self-describing, since the metadata associates name and type information with it. Some forms of layout (overlapping static areas, as in FORTRAN COMMON and EQUIVALENCE) will be hard to describe, but the metadata is capable of describing arbitrary formats provided they do not contain embedded references to managed objects.

In the NGWS model, there are at least three kinds of static fields:

1. The existing NGWS SDK static fields. Space for these is allocated and initialized (zeroed/nulled) by the NGWS SDK class loader.
2. Static fields that represent data laid out within the same PE file. The metadata for these fields contains an additional RVA field that specifies where in the image the static variable is located. The NGWS class loader neither allocates nor initializes the space.

CLS: There is no requirement to consume or emit these

3. Static fields that represent unmanaged data laid out in and exported by another PE file via unmanaged DLL export mechanism. The metadata for these fields contains both the name of the PE file in which the data resides as well as the name by which it was exported from that file. The NGWS class loader neither allocates nor initializes the space.

CLS: There is no requirement to consume or emit these

In all cases, a type initializer method can be used to update the values of static fields prior to the first attempt to access them. The definition of a static field can be marked with Custom Attributes (to be defined post-V1) to indicate section placement, padding, etc.

13.3.1 Initializing 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 NGWS SDK.

13.3.1.1 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 13.4). 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 NGWS SDK, 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) using an appropriate call into the (managed "reflection emit" or unmanaged) metadata APIs.

This mechanism, however, does not interact well with the NGWS SDK notion of an application domain. 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.

13.3.1.2 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 NGWS SDK, is strongly discouraged. The details are provided in the File Format specification.

13.3.1.3 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 under Topic 6. As will be explained below, global statics are modeled in the NGWS SDK 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 three steps:

1. All static variables are zeroed.
2. Any static that has been specially marked in the metadata to be initialized with a constant managed string or primitive numeric type is initialized by the EE.
3. The user-supplied class initialization procedure, if any, is invoked as described in Topic 6.

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 IL **box** instruction to create the boxed copy.

- **Create a managed array from a static native array of data.** The NGWS SDK base class library will supply a method that can be used to convert a native array of primitive numeric types or value types with explicit layout and no embedded object references into a managed array. The values are stored as static data at a known RVA.
- **Default initialize a managed array of a value type.** The NGWS 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.
- **Use Base Class Library deserialization.** The NGWS SDK base class library provides serialization and deserialization services. 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 "native array of 8-bit unsigned integers". The corresponding deserialization mechanism can then be called in the class initializer.

13.3.2 Unmanaged Thread-local Storage

There are two mechanisms for dealing with thread-local storage: an unmanaged mechanism and a managed mechanism. The unmanaged mechanism has a number of restrictions which are carried forward directly into the NGWS SDK. 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 elsewhere.

Each PE file has a particular section whose initial contents are copied whenever a new thread is created. There is a particular native code sequence that can be used to locate the start of this section for the current thread. NGWS SDK 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 IL 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.

13.4 Embedding Data in a PE File

There are several ways to declare a data field that is stored in a PE file. In all cases, the **.data** directive is used.

Data can be embedded in a PE file by using the **.data** directive at the top-level.

```
<decl> ::=                                                    Section
```

```
      .data <datadecl>                                          13.4
    | …                                                         4.5
```

Data may also be declared as part of a class:

```
<classDecl> ::=                                                 Section
      .data <datadecl>                                          13.4
    | …                                                         7.2
```

Yet another alternative is to declare data inside a method:

```
<methodDecl> ::=                                                Section
      .data <datadecl>                                          13.4
    | …                                                         12.2
```

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 delared in that method. Uneffected by where the data is declared, it is accessible anywhere inside the assembly.

13.4.1 Data Decleration

The data declaration of a .data directive contains the optional attribute **tls** to specify thread local storage. Further, it contains an optional data label and the body which defines the actual data. The label will be almost always used.

```
<dataDecl> ::= [tls] [<dataLabel> =] <ddBody>
```

The body consists either of one data item or a list of data items in braces. A list of data items is similar to an array.

```
<ddBody> ::=
      <ddItem>
    | { <ddItemList> }
```

A list of items consists of any number of items:

```
<ddItemList> ::= <ddItem> [, <ddItemList>]*
```

A data item declares the type of the data and provides the data in parentheses. If an array needs to be specified, the size of the array is provided in brackets, after the initialization value for each item of the array.

```
<ddItem> ::=
      & ( <id> )
    | bytearray ( <bytes> )
    | char * ( <QSTRING> )
```

```
| float32 [( <float64> )] [[ <int32> ]]
| float64 [( <float64> )] [[ <int32> ]]
| int8 [( <int8> )] [[ <int32> ]]
| int16 [( <int16> )] [[ <int32> ]]
| int32 [( <int32> )] [[ <int32> ]]
| int64 [( <int64> )] [[ <int32> ]]
| wchar * ( <QSTRING> )
```

13.4.2 Accessing Data

The data can be accessed through a static variable. A static variable, either at the top level or inside a class 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.

14 Properties

Portperties are similar to fields. However, instead of being just a field, properties have a list of methods associated with them to retrieve and mutate fields. It is important to understand that a property itself is not a field.

Porperties are always associated with a field that contains the data. In addition they have a get and set method, which retrieve or mutate the property, respectively. In higher level languages, assignment to a property might invoke the set method of the property. On the IL level, the invocation needs to be specified explicitly, similar to a method call. Properties may also have other methods assoiated with them.

When a property is created, a field for it is created automatically.

A property is very similar to having a separte class for a field, that defines its setters and getters and using a reference to an instance of the class, instead of storing the value of the field. However, the use of properties makes this object oriented approach of managing data easier and allows the definition of the needed methods in the same place as the field.

A property is declared by the using the **.property** directive, followed by a property head and property declarations in braces. Proprties are defined only inside of classes.

<classDecl> ::=	Section
.property <propHead> **{** <PropDecl>* **}**	14
| …	7.1

14.1 Property Head

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.

Similar to methods, a call kind is specified (see section 12.1.2), a type and a name, and finally its prarameters in parentheses.

```
<propHead> ::=
    [specialname|rtspecialname]* <callKind> <type> <id> ( <signature> )
```

The <callKind>, <type>, and parameters of a property have to be the same as for the get method of the property.

14.2 Property Declarations

A property may contain any number of property declarations in its body. The following table shows these and provides short descriptions for them:

<propDecl> ::=	Description	Section
.backing <type> <id>	Backing field of the property. For documentation purposes.	14.2

`| .custom <customDecl>`	Custom attribute.	14.2
`| .get <callConv> <type> [<typeSpec> ::] <methodName> ( <signature> )`	Specifies the getter for the property.	14.2
`| .other <callConv> <type> [<typeSpec> ::] <methodName> ( <signature> )`	Specifies the another method for the property.	14.2
`| .set <callConv> <type> [<typeSpec> ::] <methodName> ( <signature> )`	Specifies a setter for the property.	14.2
`| <externSourceDecl>`	**.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. This is for documentation purposes only, and may be omitted.

.get specifies the get method for this property, providing a method specification. The method needs to be defined in the type specified by <typeSpec> or in this class. Only one get method may exist. The <callKind>, <type> and parameters for a the get method have to be the same as in the property declaration. The get method should take parameters only if the field is an indexed field. The get method should return the value of the property.

.set specifies the set method for this property, similar to **.get**. Only one set method may exist. The return type of set should be **void**. The set method should have the same <callKind> as the property. It will take the new parameter as a value and may take additional arguments for indexed fields.

.other can be used to specify several other methods associated with this property.

In addition, custom attributes or source line declarations may be specified.

15 Events

The NGWS SDK supports the event model of programming. Special event handlers may be provided that handle Windows and custom events.

Events are declared inside classes with the **.event** directive. Following the directive is an event head and any number of event declarations.

```
<classDecl> ::=                                                    Section
     .event <eventHead> { <EventDecl>* }                             15
   | …                                                               7.1
```

15.1 Event Head

The event head may contain the keywords **specialname** or **rtspecialname**. **specialname** marks the name of the property for other tools, while **rtspecialname** marks the name of the event as special for the runtime.

Further, the event head contains the type and a name for the event.

```
<eventHead> ::=
     [specialname | rtspecialname]* [<typeSpec>] <id>
```

15.2 Event Declaration

The event declaration has the following syntax:

<eventDecl> ::=	Description	Section
.addon <callConv> <type> [<typeSpec> ::] <methodName> (<signature>)	AddOn method for event.	15.2
| .custom <customDecl>	Custom attribute.	15.2
| .fire <callConv> <type> [<typeSpec> ::] <methodName> (<signature>)	Fire method for event.	15.2
| .other <callConv> <type> [<typeSpec> ::] <methodName> (<signature>)	Other method.	15.2
| .removeon <callConv> <type> [<typeSpec> ::] <methodName> (<signature>)	RemoveOn method for method.	15.2
| <externSourceDecl>	**.line** or **#line**	3.7

The AddOn method is called by an object that wants to be notified when this event occurs. The RemoveOn method is called by an object that does not want to be notified anymore by when this event occurs.

The Fire method is called to fire this event and notifiy all registerd objects.

16 Declarative Security

The NGWS SDK has a sophisticated security system. Programs may specify what permission they need in order to run correctly. However, this feature is currently not available for IL assembly code. It might or might not be available in the final product or a future version.

The following grammar is for round tripping use only. It will be used by ildasm to display code compiled from other languages. However, it is not recommended using the following directives with the current version of ilasm.

```
<securityDecl> ::=
      .capability <secAction> = ( <bytes> )
    | .capability <secAction> <SQSTRING>
    | .permission <secAction> <classname> ( <nameValPairs> )
```

In **.permission**, <className> specifies the permission class and <nameValPairs> show the settings. In **.capability** the bytes show the serialized version of the security settings.

<secAction> ::=	**Description**
assert	Assert permission so callers don't need.
| **demand**	Demand permission of all caller.
| **deny**	Deny permissions 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 NGWS SDK 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 VOS. Users or other tools are welcome to define and use additional attribute types.

Custom attributes are declared using the directive **.custom**. Follwed by this directive is a custom attribute type, and either an optional <bytes> in parentheses or a string as shown by the following grammar:

```
<customDecl> ::=
     <customAttrType> [= ( <bytes> ) | = <QSTRING> ]
```

A custom attribute type is either a type specification or a calling convention, followed by a type, followed by an optional type specification and by a method name and parameters in parentheses.

```
<customAttrType> ::=
     <callConv> <type> [<typeSpec> ::] <methodName> ( <signature> )
   | <typeSpec>
```

There are fundamentally three kinds of attributes that can be created:

- Those that are specified with only a type for the attribute. The type is used simply to identify the attribute, and there is no value specified for the attribute itself. These can serve as simple flags, where the existence of the attribute is sufficient to convey all the necessary information. These are specified by providing a `typeSpec` for the `customAttrType` and providing no additional data. These attributes are not CLS compliant.
- Similar to the above, but providing data along with the type. These are specified using a `typeSpec` for the `customAttrType` and providing arbitrary additional. These, too, are not CLS compliant.
- Providing a type initializer and an encoded value that specifies the arguments to be passed to that initializer and the names and values of specific fields that should be initialized. These are specified using the more complex form of `customAttrType` to specify the type initializer (named **.ctor**) and the syntax beginning with **=** to specify the encoding of the value according to the separate specification. These are the only form that is CLS compliant.

While it is possible to use the assembler to create a custom attribute that has a method reference other than a constructor, the meaning of this is unspecified and it should be avoided.

17.1 Custom Attribute Usage: CLS Conventions

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 metadata (types, methods, assemblies, etc.) may have the attribute applied to them. By default, instances of an attribute class can be applied to any metadata item.
- 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.
- 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.

Notice that, since these are CLS rules and not part of the VOS 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 Execution Engine and Reflection will not supply attributes automatically.

17.2 Attributes Used by the Runtime

The Metadata engine implements two sorts of Custom Attribute, called (genuine) Custom Attributes, and pseudo Custom Attributes. In the remainder of this document, we’ll abbreviate these terms to CA and PCA. Both CAs and PCAs are “handed over” to Metadata via the DefineCustomAttribute method. But they 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 on Metadata’s hard-wired list of 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.” So you cannot retrieve that ‘blob’ later – it doesn’t exist

PCAs therefore serve to capture user “directives,” using the same familiar syntax the compiler provides for regular CAs – but these ‘directives’ are then stored into the more space-efficient form of metadata tables. Tables are also faster to check at runtime than full-bloodied (genuine) CAs. An example of a PCA is the SerializableAttribute – if the compiler calls DefineCustomAttribute with this PCA as an argument, the Metadata engine simply sets the tdSerializable bit on the target class definition.

Many CAs are invented by higher layers of software – Metadata stores them, and returns them, 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 "distinguised" 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 in "Attribute" (Neither Metadata or Runtime check, or care, about this convention).

For further details on this long list of 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 for the DefineCustomAttribute method. The check is solely on their name – for example "DllImportAttribute" – their namespace is ignored. 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 throws the "blob" on the floor (this is the definition of a PCA – see above). All these are attributes can be found in `System`.

Attribute	Description
NonSerializedAttribute	
SerializableAttribute	

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
AttributeUsageAttribute	Used to specify how an attribute is intended to be used.
ObsoleteAttribute	Indicates that an element is not to be used.
CLSCompliantAttribute	Indicates whether or not an element is declared to be CLS compliant through an instance field on the attribute object.

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 on `System.Diagnostics`.

Attribute	Description
DebuggableAmbivalentAttribute	
DebuggableAttribute	
DebuggerHiddenAttribute	
DebuggerStepThroughAttribute	

17.2.4 Custom Attributes for Reflection

CA in `System.Reflection` that is used by Reflection's Invoke call – it invokes the property for the Type defined in this CA:

Attribute	Description
DefaultMemberAttribute	Defines the member of a type that is the default member used by InvokeMember.

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

Attribute	Description
	custom attributes.
EnvironmentPermissionAttribute	Custom attribute class for declarative security with EnvironmentPermission.
FileDialogPermissionAttribute	Custom attribute class for declarative security with FileDialogPermission.
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) 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 'hung off' a TypeRef to a class called AssemblyAttributesGoHere) then merged by **al** and 'hung off' the assembly. These attributes can be found in `System.Runtime.CompilerServices`.

Attribute	Description
AssemblyCultureAttribute	
AssemblyDelaySignAttribute	
AssemblyKeyFileAttribute	
AssemblyKeyNameAttribute	
AssemblyOperatingSystemAttribute	
AssemblyProcessorAttribute	
AssemblyVersionAttribute	

17.2.9 Attributes Provided for Language Interop

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 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 COM+ 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 P/Invoke method in unmanaged code.
FieldOffsetAttribute	Used along with the System.Runtime.InteropServices.

Attribute	Description
	StructLayoutAttribute.LayoutKind set to **Explicit** to indicate he 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).
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 shuld 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 COM+ 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 Exception Handling

This section is a brief summary of the NGWS SDK exception handling. More information can be found in the Architecture specification.

The EE 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 boxed instances of some subclass of **System.Exception**. Users can create their own exception classes by subclassing **System.Exception**.

A protected block, called <sehBlock> in the grammar may appear in a method as shown by the following excerpt:

<methodDecl> ::=	**Section**
<sehBlock>	18
| …	12.2

A protected block consists of a try block, and one or more handlers, called <sehClause> in the grammar.

```
<sehBlock> ::=
     <tryBlock> <sehClause> [<sehClause>*]
```

18.1 Try Block

The try block contains the instructions that may throw an exception which needs to be handled. Try block are declared by the **.try** directive. There are two ways to define a try block.

The first way simply uses a scope block (see section 12.5) after **.try** directive that contains the instructions to be protected.

In the second way, the instructions that shall be protected need to be 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 the code. Instead of the labels, the addresses the labels represents may be used.

The three ways are summarized in the grammar below:

<tryBlock> ::=	**Descriptions**
.try <int32> **to** <int32>	For disassembler only
| **.try** <label> **to** <label>	Second label is exclusive, pointing to first instruction after the try block; both labels must be already defined in the preceding code
| **.try** <scopeBlock>	<scopeBlock> contains the instructions to be protected

18.2 Handlers

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

1. A **finally** handler which must be executed whenever the block exits, regardless of whether that occurs by normal control flow or by an unhandled exception.
2. A **fault** handler which must be 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 IL instructions to determine whether the exception should be ignored (i.e. execution should resume), handled by the associated handler, or passed on to the next protected block.

The handlers are specified by the following grammar.

```
<sehClause> ::=
     catch <className> <handlerBlock>
   | fault <handlerBlock>
   | filter <int32> <handlerBlock>
   | filter <label> <handlerBlock>
   | finally <handlerBlock>
```

The <className> after **catch** specifies the class of exception to be caught.

The <label> in the **filter** clause specifies the code that checks the exception. The integer is used only for round tripping purposes.

When an exception is thrown, the execution engine will first check for a filter clause. The filter code specifies whether the method wishes to handle the exception. If yes, the execution engine will proceede with the corresponding type-filtered handler. Otherwise, the next method in the call stack will be investigated, and so on. If no method that wishes to handle the exception can be found, the exception is dump to the default ouput.

Within the exception filter, code executes within the same context of the procedure that caused the exception. In other words, all local variables contain the same values they contained when the exception occurred.

The following grammar shows a handler block:

<handlerBlock> ::=	**Description**
handler <int32> **to** <int32>	For dissassembler only
| **handler** <label> **to** <label>	Second label is exclusive, pointing to first instruction after the handler block;both labels must be already defined in the preceding code
| <scopeBlock>	<scopeBlock> 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 inteded for the round tripping use only.

18.3 Throwing an Exception

To throw an exception the instruction **throw** is used. To throw the same exception from a catch block the **rethrow** instruction is used.

A protected block may be left using the **leave** instruction. A branch may not be used to leave a protected block. When a protected block is left, the corresponding **finally** clause will be automatically executed. The **leave** instruction may appear in either a **try** block or a **catch** block.

A **filter** block may be exited with the **endfilter** instruction only. The **endfilter** instruction expects an argument on the stack that defines whether this method will handle the exception or not. The two possible values are:

- **EXCEPTION_CONTINUE_SEARCH (= 0)**: Indicates that the handler cannot process this exception and that the execution engine 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 execution engine should halt the search for any other handlers and use the code at this handler's handler offset to process the exception.

A **finally** block may be exited only with a **endfinally** instruction.

19 IL Instructions

19.1 Overview

See the separate IL Instruction Set specification for details.

19.2 Opcodes by Category

See the Architecture specification for details on exceptions, rules for valid IL sequences and verifiable code. Note that all opcodes can throw the general **`System.ExecutionEngineException`**.

19.2.1 General Instruction Syntax

The general instruction syntax is as follows:

<instr> ::= <instruction>[**.ovf**][**.un**]

<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.

Arguments may be passed to instructions in two ways (possibly combined). One way is using the stack. Another way is as a direct argument, which is assembled as a part of the instruction.

19.2.2 Numeric and Logical Operations

These operations deal with the top one or two items on the evaluation stack, typically popping their arguments and pushing a result. See the Architecture specification for details on exceptions, rules for valid IL sequences and verifiable code. In general, valid IL requires that the types of the operand(s) are numeric, and for binary operations they must be of the same basic numeric type (I4, I8, or float). Many operations in this category may throw **`System.ArithmeticException`**.

Pointers (type **&** and *****) are considered to be unsigned integer values representing byte addresses. Thus, for example, subtraction of two pointers returns an answer that has an integer type.

19.2.2.1 Unary Operations

The top of the evaluation stack must be a numeric type (integer type for **`not`**) and the result is of the same type.

`neg` arithmetic negation (i.e. subtract from zero).

`Not` bitwise logical complement.

19.2.2.2 Binary Numeric Operations, No Overflow Check

The following table summarizes the result type based on the type of the top two items on the evaluation stack. A **op** B (applies to all instructions unless specific instructions are

specified in the table). The shaded uses are not verifiable, while items marked "-" indicate incorrectly formed IL sequences, as are any uses where either operation is an object reference. If only a subset of instructions are permitted, the valid instructions are shown in the corresponding cell.

B's type A's type	I4	I8	I	F	&(by-ref)	* (transient pointer)
I4	I4	-	I	-	& (add)	* (add)
I8	-	I8	-	-	-	-
I	I	-	I	-	& (add)	* (add)
F	-	-	-	F	-	-
&	&(add, sub)	-	&(add, sub)	-	I (sub)	I (sub)
*	* (add, sub)	-	* (add, sub)	-	I (sub)	I (sub)

`add` signed addition, ignore overflow
`div` signed division, may throw **`System.DivideByZeroException`**
`mul` signed multiplication, ignore overflow
`rem` signed remainder, may throw **`System.DivideByZeroException`**
`sub` signed subtraction, ignore overflow

19.2.2.3 Binary Integer Operations, No Overflow Check

These operate only on integer types. The **div.un** and **rem.un** instructions treat their arguments as unsigned integers and produce the bit pattern corresponding to the unsigned result. The EE makes no distinction between signed and unsigned integers on the stack. The **shl**, **shr,** and **shr.un** instructions return the same type as their first operand and their second operand must be of type U. All items marked "-" indicate invalid IL sequences, while the others are verifiable. If only a subset of instructions are permitted, the valid instructions are shown in the corresponding cell. Use of any other data types for either operand is invalid IL.

	I4	I8	I
I4	I4	-	I
I8	-	I8	-
I	I	-	I

`and` bitwise logical AND
`div.un` unsigned division, may throw **`System.DivideByZeroException`**
`or` bitwise logical OR
`rem.un` unsigned remainder, may throw **`System.DivideByZeroException`**
`shl` left shift
`shr` right shift
`shr.un` unsigned right shift
`xor` bitwise logical XOR

19.2.2.4 Operations with Overflow Checks

These operations generate a **System.OverflowException** exception if the result cannot be represented in the target data type. This table shows the returned type given the two operands on the top of the evaluation stack. The shaded uses are not verifiable, while items marked "-" indicate invalid IL sequences. If only a subset of instructions are permitted, the valid instructions are shown in the corresponding cell. These instructions are also invalid if either operand is a floating point number or an object reference.

	I4	I8	I	& (by-ref)	* (transient pointer)
I4	I4	-	I	& add.ovf.un	* add.ovf.un
I8	-	I8	-	-	-
I	I	-	I	& add.ovf.un	* add.ovf.un
&	& add.ovf.un, sub.ovf.un	-	& add.ovf.un, sub.ovf.un	I sub.ovf.un	I sub.ovf.un
*	* add.ovf.un, sub.ovf.un	-	* add.ovf.un, sub.ovf.un	I sub.ovf.un	I sub.ovf.un

In the case of an overflow, the instructions below throw a `System.OverflowException`.

`add.ovf`	signed addition
`add.ovf.un`	unsigned addition
`mul.ovf`	signed multiplication
`mul.ovf.un`	unsigned multiplication
`sub.ovf`	signed subtraction
`sub.ovf.un`	unsigned subtraction

19.2.2.5 Comparison Operations

These operations compare the top two elements on the evaluation stack and push either a 4-byte zero for false or a 4-byte one for true. The following table summarizes legal combinations of operands for these instructions. Items marked "✓" indicate that all instructions are valid. Items marked "-" indicate invalid IL 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	✓	-	✓	-	ceq	-	ceq
F	-	-	-	✓	-	-	-
&	-	-	ceq	-	✓ (Note)	-	✓ (Note)

	I4	I8	I	F	&	O	*
O	-	-	-	-	-	ceq, cgt, cle.un	-
*	-	-	ceq	-	✓ (Note)	-	✓ (Note)

Note: Except for ceq these combinations only make sense if both operands are known to be pointers to elements of the same array.

`Ceq`	compare for equality
`cgt`	compare for greater than, signed
`cgt.un`	compare for greater than, unsigned or unordered (floating point)
`clt`	compare for less than, signed
`clt.un`	compare for less than, unsigned

19.2.2.6 Conversions

These operations convert the top of the evaluation stack into the specified numeric type. 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.

A value on the stack that has type I is automatically converted by the runtime into the destination type when it is stored to a location that is of type I4, I2 or I1. However, this feature needs to be enjoyed with special care. Such a conversions might throw a **System.OverflowException**. In order not to be confronted with surprises, compiler writers and other developpers are strongly encouraged to use explicit type conversions in every case. This will gurantee that the code does not cause unexpected behavior due to implicit type conversions when executed on a 64 bit platform.

In the following table, the shaded uses are not verifiable, while items marked "-" indicate invalid IL sequences. If only a subset of instructions are permitted, the valid instructions are shown in the corresponding cell. It is invalid IL for either operand to be an object reference.

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]
& (by-ref)	-	-	-	Stop GC Tracking	-
* (transient pointer)	-	-	-	Zero extend	-

Output Operand	U	All R Types	& (by-ref)
I4	Zero extend	To Float	-

Output Operand	U	All R Types	& (by-ref)
I8	**Truncate**[1]	**To Float**	**-**
I	**No-op**	**To Float**	**-**
F	**Trunc to 0**[2]	**Change Precision**[3]	**-**
&	**Stop GC Tracking**	**-**	**No-op**
*****	**No-op**	**-**	**Start GC Tracking**

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.

The general syntax for conv instructions is as follows:

<conv instruction> ::= **conv.**<to type> | **conv.ovf.**<to type>[**.un**]

<to type> specifies the type to which type the operand on top of the stack shall be converted.

The optional suffix **.ovf** specifies that overflow checking is done before converting. A `System.OverflowException` is thrown if the operand is not in the range permitted by the destination type. The .**ovf** suffix cannot be used for floating point conversions. Floating point conversions change the precision of the operand, or result in *infinity* if the operand is too large (or *negative infinity* if the operand is too small) for the destination type (i.e., in conversions from float64 to float32).

The optional suffix **.un** specifies that the operand shall be treated as an unsigned value. As a consequence of this, using the **.un** suffix has a more restricive effect than just using the **.ovf** suffix. A `System.OverflowException` will be thrown if the operand is negative or outside the range permitted by the destination type. The **.un** suffix does not make sense, and thus cannot be used, without the **.ovf** suffix. The exception is **conv.r.un**, which converts only unsigned integers to a floating point numbers (of type F). **conv.r.un** cannot be used to convert from one floating point type to another. The **.un** suffix cannot be used with other floating type conversions.

The following is a complete list of all conv instructions:

`conv.i`	convert to signed integer (4- or 8-byte depending on platform)
`conv.i1`	convert to signed 1 byte integer
`conv.i2`	convert to signed 2 byte integer
`conv.i4`	convert to signed 4 byte integer
`conv.i8`	convert to signed 8 byte integer
`conv.ovf.i`	like **conv.i**, but may throw `System.OverflowException`
`conv.ovf.i.un`	like **conv.ovf.i**, but does throw exception for negative values
`conv.ovf.i1`	like **conv.i1**, but may throw `System.OverflowException`

conv.ovf.i1.un	like **conv.ovf.i1**, but does throw exception for negative values
conv.ovf.i2	like **conv.i2**, but may throw **System.OverflowException**
conv.ovf.i2.un	like **conv.ovf.i2**, but does throw exception for negative values
conv.ovf.i4	like **conv.i4**, but may throw **System.OverflowException**
conv.ovf.i4.un	like **conv.ovf.i4**, but does throw exception for negative values
conv.ovf.i8	like **conv.i8**, but may throw **System.OverflowException**
conv.ovf.i8.un	like **conv.ovf.i8**, but does throw exception for negative values
conv.ovf.u	like **conv.u**, but may throw **System.OverflowException**
conv.ovf.u.un	like **conv.ovf.u**, but does throw exception for negative values
conv.ovf.u1	like **conv.u1**, but may throw **System.OverflowException**
conv.ovf.u1.un	like **conv.ovf.u1**, but does throw exception for negative values
conv.ovf.u2	like **conv.u2**, but may throw **System.OverflowException**
conv.ovf.u2.un	like **conv.ovf.u2**, but does throw exception for negative values
conv.ovf.u4	like **conv.u4**, but may throw **System.OverflowException**
conv.ovf.u4.un	like **conv.ovf.u4**, but does throw exception for negative values
conv.ovf.u8	like **conv.u8**, but may throw **System.OverflowException**
conv.ovf.u8.un	like **conv.ovf.u8**, but does throw exception for negative values
conv.r.un	convert to system-specified floating point from unsigned integer
conv.r4	convert to IEEE 32-bit floating point
conv.r8	convert to IEEE 64-bit floating point
conv.u	convert to unsigned integer (4- or 8-byte depending on platform)
conv.u1	convert to unsigned 1 byte integer
conv.u2	convert to unsigned 2 byte integer
conv.u4	convert to unsigned 4 byte integer
conv.u8	convert to unsigned 8 byte integer

19.2.3 Control Flow

The operations described in the following sections alter the normal flow of control from one IL 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.

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.

19.2.3.1 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 19.2.3.5.

br branch within current method
br.s branch within current method

19.2.3.2 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 zero
brfalse.s branch if 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 zero
brtrue.s branch if not zero
brzero branch if zero
brzero.s branch if not zero
ckfinite check that the top of stack is a finite floating point number, generating a **System.ArithmeticException** if the value is a NaN or an infinity
switch multi-way 0-based branch depending on value on top of evaluation stack

19.2.3.3 Binary Compare-and-Branch Instructons

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. See Section 19.2.2.5 for a table of valid and verifiable operand types. 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).

The following table summarizes legal combinations of operands for these instructions. Items marked "✓" indicate that all instructions are valid. Items marked "-" indicate invalid IL 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]**	-	**beq[.s], bne.un[.s]**
F	-	-	-	✓	-	-	-
&	-	-	**beq[.s], bne.un[.s]**	-	✓ **(Note)**	-	✓ **(Note)**
O	-	-	-	-	-	**beq[.s],**	-

	I4	I8	I	F	&	O	*
						bne.un[.s]	
*	-	-	beq[.s], bne.un[.s]	-	✓ (Note)	-	✓ (Note)

Note: Except for beq, bne.un (or short versions) these combinations only make sense if both operands are known to be pointers to elements of the same array.

`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`

19.2.3.4 Procedure Call and Related Instructions

These instructions move the flow of control to another procedure. See also **callvirt** and **ldvirtftn** in Section 19.2.5

`arglist`	returns handle to current argument list on stack (for varargs methods)
`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
`jmpi`	branch with current arguments to another method using function pointer
`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 tailcall version (drop current stack frame before call)

19.2.3.5 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).

Endfilter mark end of a filter handler

endfinally mark end of a fault or finally handler

leave unconditional branch that may exit a try block

leave.s unconditional branch that may exit a try block

rethrow throw the current exception again (out of a catch handler)

throw throw an exception

19.2.3.6 Other Control Flow Instructions

The instructions in this section are considered to be instructions that belong to the group of control flow instructions, however don't belong to any of the above sections.

Nop ignored

break invoke debugger if attached

19.2.4 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 from 0. Argument 0 is the **this** pointer for instance and virtual methods. Valid IL 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.

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.0, ldarg.1, ldarg.2,
ldarg.3, 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 load address of an argument

ldarga.s load address of an argument, short form, for arguments 0 through 255

ldc.i4, ldc.i4.0, ldc.i4.1, ldc.i4.2,
ldc.i4.3, ldc.i4.4, ldc.i4.5, ldc.i4.6,
ldc.i4.7, ldc.i4.8, ldc.i4.M1, ldc.i4.m1,
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.i, ldind.i1, ldind.i2, ldind.i4,
ldind.i8, ldind.r4, ldind.r8, ldind.ref,
ldind.u1, ldind.u2, ldind.u4, ldind.u8
load indirect through a pointer, type of data loaded is specified as a suffix to the instruction (*not verifiable*).

Ldloc, ldloc.0, ldloc.1, ldloc.2,
ldloc.3, 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 load address of local variable onto stack, long form (locals 256 and over)

ldloca.s load address of local variable onto stack, short form (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 store top of evaluation stack into an argument, for arguments 256 and over

starg.s store top of evaluation stack into an argument, for arguments 0 through 255

stind.i, stind.i1, stind.i2, stind.i4,
stind.i8, stind.r4, stind.r8, stind.ref
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 IL 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.0, stloc.1, stloc.2,
stloc.3, 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.*, stind.*, ldfld, stfld, ldobj, stobj, initblk,** or **cpblk.**

Volatile. Indicates that the subsequent operation my reference data that is read or written asynchronously. Valid only before **ldind, stind, ldfld, ldsfld, stfld, stsfld, ldobj, stobj, initblk,** or **cpblk.**

19.2.5 Object Management

The IL instruction set has direct support for creating objects, zero-based one-dimensional arrays, typed-references, and strings. It also support 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 System.Object 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 **System.NullReferenceException** is thrown.

Castclass convert an object to any of its parent classes, specified as part of the instruction, or raise **System.InvalidCastException**.

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 other argument 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.i, ldelem.i1, ldelem.i2, ldelem.i4,
ldelem.i8, ldelem.r4, ldelem.r8, ldelem.ref,
ldelem.u1, ldelem.u2, ldelem.u4, ldelem.u8
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 **System.ArrayTypeMistmatchException** is raised. An out of range subscript results in a **System.IndexOutOfRangeException**, while an attempt to access an element of the null array results in a **System.NullReferenceException**.

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. The object must not be derived from **System.MarshalByRefObject** or **System.ContextBoundObject**.

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.

Ldsfld load the contents of a static field of a class onto the evaluation stack.

Ldsflda load the address of a static field of a class onto the evaluation stack.

Ldstr load a literal instance of **System.String** onto the evaluation stack.

Ldtoken load a token representing a type, field, or method onto the evaluation stack. The token is of type **U** (unsigned 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 the verifier 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 7.3.2) 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 given a typed reference on the evaluation stack, extract the pointer from it. See also **mkrefany**.

Sizeof returns the size in bytes of an instance of a value type. The value type is specified as part of the instruction.

Stelem.i, stelem.i1, stelem.i2, stelem.i4, stelem.i8, stelem.r4, stelem.r8, stelem.ref

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 **System.ArrayTypeMismatchException**. The top of the stack contains the value to be stored. The second item on the stack is the index, an unsigned integer. A **System.IndexOutOfRangeException** will be thrown if the index is larger than the size of the array. Below the index is the array itself, which will result in a **System.NullReferenceException** if it is null. To store an unboxed value type into an array, use **ldelema** and **stobj** rather than **stelem.***.

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 set) or a **System.InvalidCastException** is thrown. The result is a by-ref (managed pointer), *not* 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.

19.2.6 Annotations

Annotations are ignored by all the NGWS SDK tools that convert IL 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 IL 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

20 Sample IL Programs

20.1 Mutually Recursive Program (with tail calls)

Consider this managed C++ program:

```
#import <mscorlib.dll>

[managed] class EvenOdd
{ static bool IsEven(int N)
  { return (N==0) ? true : IsOdd(N-1);
  }
  static bool IsOdd(int N)
  { return (N==0) ? false : IsEven(N-1);
  }
public:
  static void Test(int N)
  { Console::Write(N);
    Console::Write(L" is ");
    if (IsEven(N)) Console::WriteLine(S"even");
    else Console::WriteLine(S"odd");
    return;
  }
};
void start()
{ EvenOdd::Test(5);   EvenOdd::Test(2);
  EvenOdd::Test(100); EvenOdd::Test(1000001);
  return;
}
```

This can be hand-translated into the following IL assembly program. Notice that references to types must use the fully qualified name (like "System.String"). This is independent of the "/NOAUTOINHERIT" flag to the assembler, which governs whether classes defined by the IL code implicitly inherit from System.Object.

```
.assembly test.exe { }
.class EvenOdd
{ .method private static bool IsEven(int32 N) il managed
  { .maxstack   2
    ldarg.0             // N
    ldc.i4.0
    bne.un      NonZero
    ldc.i4.1
    ret
NonZero:
    ldarg.0
    ldc.i4.1
    sub
    tail. Call bool EvenOdd::IsOdd(int32)
    ret
  } // end of method 'EvenOdd::IsEven'
```

```
  .method private static bool IsOdd(int32 N) il managed
  { .maxstack   2
    // Demonstrates use of argument names and labels
    // Notice that the assembler does not covert these
    // automatically to their short versions
    ldarg       N
    ldc.i4.0
    bne.un      NonZero
    ldc.i4.0
    ret
NonZero:
    ldarg       N
    ldc.i4.1
    sub
    tail. Call bool EvenOdd::IsEven(int32)
    ret
  } // end of method 'EvenOdd::IsOdd'

  .method public static void Test(int32 N) il managed
  { .maxstack   1
    ldarg       N
    call        void System.Console::Write(int32)
    ldstr       " is "
    call        void System.Console::Write(class System.String)
    ldarg       N
    call        bool EvenOdd::IsEven(int32)
    brfalse     LoadOdd
    ldstr       "even"
Print:
    call        void System.Console::WriteLine(class System.String)
    ret
LoadOdd:
    ldstr       "odd"
    br          Print
  } // end of method 'EvenOdd::Test'
} // end of class 'EvenOdd'

//Global method

.method public static void main() il managed
{ .entrypoint
  .maxstack     1
  ldc.i4.5
  call          void EvenOdd::Test(int32)
  ldc.i4.2
  call          void EvenOdd::Test(int32)
  ldc.i4        100
  call          void EvenOdd::Test(int32)
  ldc.i4        1000001
  call          void EvenOdd::Test(int32)
  ret
} // end of global method 'main'
```

20.2 Using Value Types

Consider the following program in pseudo code:

```
value class Rational extends Object implements Icomparable
{ int Numerator, Denominator;

  virtual bool Icomparable::CompareTo(Object o)
  { return (this.Numerator==((boxed Rational) o)->Numerator) &&
           (this.Denominator==((boxed Rational) o)->Denominator);
  }

  virtual String Object::ToString()
  { StringBuilder SB = new StringBuilder();
    SB.AppendFormat("The value is: {0}/{1}",
                    (Object) this.Numerator,
                    (Object) this.Denominator);
    return SB.ToString();
  }

  Rational Mul(Rational By)
  { Rational Result;
    Result.Numerator = this.Numerator * By.Numerator;
    Result.Demominator = this.Denominator * By.Denominator;
    return Result;
  }
}

Main()
{ Rational Half = { 1, 2 }, Third = { 1, 3 }, Temporary;
  Object H = (Object) Half, T = (Object) Third;
  WriteLine(H.Icomparable::CompareTo(H)); // Interface call
  WriteLine(Half.CompareTo(T));           // Virtual call
  WriteLine(Half.ToString());             // Virtual call
  WriteLine(T.ToString());          // Virtual call on Object
  WriteLine((Half.Mul(Third)).ToString());
}
```

Which translates into verifiable IL as follows:

```
.assembly rational.exe { }
.class value sealed Rational extends System.ValueType
                            implements System.Icomparable
{ .field int32 Numerator
  .field int32 Denominator

  .method virtual int32 CompareTo(class System.Object o)
  // Implements Icomparable::CompareTo(Object)
  { ldarg.0     // this as managed pointer
    ldfld int32 value class Rational::Numerator
    ldarg.1     // o as object
    unbox value class Rational
    ldfld int32 value class Rational::Numerator
    beq.s TryDenom
    ldc.i4.0
    ret
```

```
TryDenom:
    ldarg.0     // this as managed pointer
    ldfld int32 value class Rational::Denominator
    ldarg.1     // o as object
    unbox value class Rational
    ldfld int32 class Rational::Denominator
    ceq
    ret
  }
  .method virtual class System.String ToString()
  // Implements Object::ToString
  { .locals init (class System.Text.StringBuilder SB,
                  class System.String S,
                  class System.Object N,
                  class System.Object D)
    newobj void System.Text.StringBuilder::.ctor()
    stloc.s SB
    ldstr "The value is: {0}/{1}"
    stloc.s S
    ldarg.0     // Managed pointer to self
    dup
    ldflda int32 value class Rational::Numerator
    box System.Int32
    stloc.s N
    ldflda int32 value class Rational::Denominator
    box System.Int32
    stloc.s D
    ldloc.s SB
    ldloc.s S
    ldloc.s N
    ldloc.s D
    call instance class System.Text.StringBuilder
      System.Text.StringBuilder::AppendFormat(class System.String,
                                              class System.Object,
                                              class System.Object)
    callvirt instance class System.String System.Object::ToString()
    ret
  }
  .method value class Rational Mul(value class Rational)
  { .locals init (value class Rational Result)

    ldloca.s Result
    dup
    ldarg.0     // this
    ldfld int32 value class Rational::Numerator
    ldarga.s    1     // arg
    ldfld int32 value class Rational::Numerator
    mul
    stfld int32 value class Rational::Numerator
    ldarg.0     // this
    ldfld int32 value class Rational::Denominator
    ldarga.s    1     // arg
    ldfld int32 value class Rational::Denominator
    mul
    stfld int32 value class Rational::Denominator
    ldloc.s Result
    ret
```

```
  }
}
.method void main()
{ .entrypoint
  .locals init (value class Rational Half,
                value class Rational Third,
                value class Rational Temporary,
                class System.Object H,
                class System.Object T)
  // Initialize Half, Third, H, and T
  ldloca.s Half
  dup
  ldc.i4.1
  stfld int32 value class Rational::Numerator
  ldc.i4.2
  stfld  int32 value class Rational::Denominator
  ldloca.s Third
  dup
  ldc.i4.1
  stfld int32 value class Rational::Numerator
  ldc.i4.3
  stfld int32 value class Rational::Denominator
  ldloca.s Half
  box value class Rational
  stloc.s H
  ldloca.s Third
  box value class Rational
  stloc.s T
  // WriteLine(H.Icomparable::CompareTo(H))
  // Call CompareTo via interface using boxed instance
  ldloc H
  dup
  callvirt int32 System.Icomparable::CompareTo(class System.Object)
  call void System.Console::WriteLine(bool)
  // WriteLine(Half.CompareTo(T))
  // Call CompareTo via value type directly
  ldloca.s Half
  ldloc T
  call instance int32
  value class Rational::CompareTo(class System.Object)
  call void System.Console::WriteLine(bool)
  // WriteLine(Half.ToString())
  // Call virtual method via value type directly
  ldloca.s Half
  call instance class System.String class Rational::ToString()
  call void System.Console::WriteLine(class System.String)
  // WriteLine(T.ToString)
  // Call virtual method inherited from Object, via boxed instance
  ldloc T
  callvirt class System.String System.Object::ToString()
  call void System.Console::WriteLine(class System.String)
  // WriteLine((Half.Mul(T)).ToString())
  // Mul is called on two value types, returning a value type
  // ToString is then called directly on that value type
  // Note that we are required to introduce a temporary variable
  //   since the call to ToString requires a managed pointer (address)
  ldloca.s Half
```

```
  ldloc.s Third
  call instance value class Rational
         Rational::Mul(value class Rational)
  stloc.s Temporary
  ldloca.s Temporary
  call instance class System.String Rational::ToString()
  call void System.Console::WriteLine(class System.String)
  ret
}
```

21 Appendix A: ILASM Complete Grammar

21.1 Assembler Grammar

The input file to the assembler must be considered legal according to the grammar for <IL File> given here. Items in **bold face** are lexical tokens to be typed exactly as specified here.

IL assembler is a case-sensitive language, like C/C++ or C#. It means that all keywords, instructions, and other lexical tokens are to be used exactly as specified here (e.g., **.class**, not **.Class**). The names of classes, methods, fields, etc. are also case-sensitive.

The syntactic classes below are sorted in alphabetical order (case-insensitive).

```
<IL file> ::= <decl>*
```

<asmAttr> ::=	Section
implicitcom	4.2
| **noappdomain**	4.2
| **nomachine**	4.2
| **noprocess**	4.2

<asmDecl> ::=	Section
.hash algorithm <int32>	4.2.1
| **.title** <QSTRING> [**(** <QSTRING> **)**]	4.2.1
| **.custom** <customDecl>	17
| **.locale = (** <bytes> **)**	4.2.1
| **.locale** <QSTRING>	4.2.1
| **.originator = (** <bytes> **)**	4.2.1
| **.os** <int32> **.ver** <int32> **:** <int32>	4.2.1
| **.processor** <int32>	4.2.1
| **.ver** <int32> **:** <int32> **:** <int32> **:** <int32>	4.2.1

<asmRefDecl> ::=	Section
.hash = (<bytes> **)**	4.2.1
| **.custom** <customDecl>	17
| **.locale = (** <bytes> **)**	4.2.1
| **.locale** <QSTRING>	4.2.1
| **.originator = (** <bytes> **)**	4.2.1
| **.os** <int32> **.ver** <int32> **:** <int32>	4.2.1
| **.processor** <int32>	4.2.1

	Section
	.ver <int32> : <int32> : <int32> : <int32>	4.2.1

<assemblyRefName> ::=	**Section**
<dottedname>	3.2

<bound> ::=	**Section**
<int32>	5.2
| <int32> …	5.2
| <int32> … <int32>	5.2

<bytes> ::=	**Section**
<hexbyte> [<hexbyte>*]	3.5

<callConv> ::=	**Section**
[**instance** [**explicit**]] [<callKind>]	12.6.1

<callKind> ::=	**Section**
default	12.1.2
| **unmanaged cdecl**	12.1.2
| **unmanaged fastcall**	12.1.2
| **unmanaged stdcall**	12.1.2
| **unmanaged thiscall**	12.1.2
| **vararg**	12.1.2

<classAttr> ::=	**Section**
abstract	7.1.1.4
| **ansi**	7.1.1.6
| **auto**	7.1.1.2
| **autochar**	7.1.1.6
| **explicit**	7.1.1.2
| **import**	7.1.1.5
| **interface**	7.1.1.3
| **lateinit**	7.1.1.7
| **nested assembly**	7.1.1.1
| **nested famandassem**	7.1.1.1
| **nested family**	7.1.1.1
| **nested famorassem**	7.1.1.1

	nested private 7.1.1.1
	nested public 7.1.1.1
	not_in_gc_heap 7.1.1.3
	private 7.1.1.1
	public 7.1.1.1
	rtspecialname 7.1.1.7
	sealed 7.1.1.4
	sequential 7.1.1.2
	serializable 7.1.1.5
	specialname 7.1.1.7
	unicode 7.1.1.6
	value 7.1.1.3

<classDecl> ::=	Section
.class <classHead> **{** <classDecl>* **}**	7.2
.comtype <comtypeHead> **{** <comtypeDecl>* **}** /* for round trip only */	7.2
.custom <customDecl>	17
.data <datadecl>	7.2
.event <eventHead> **{** <EventDecl>* **}**	7.2
.export [**public** | **private**] <dottedname> **{** <exportDecl>* **}**	4.6
.field <fieldDecl>	7.2
.method <methodHead> **{** <methodDecl>* **}**	7.2
.override <typeSpec> **::** <methodName> **with** <callConv> <type> <typeSpec> **::** <methodName> **(** <signature> **)**	7.2
.pack <int32>	7.2
.property <propHead> **{** <PropDecl>* **}**	7.2
.size <int32>	7.2
<externSourceDecl>	3.7
<securityDecl>	16

<classHead> ::=	Section
<classAttr>* <id> [**extends** <className>] [**implements** <className> [, <className>]*]	7.1

<className> ::=	Section
[<resolutionScope>] <dottedname> [**/** <dottedname>]*	5.3

	Section
<codeLabel> ::=	
<label> :	3.4

	Section
<comtAttr> ::=	
nested assembly	6
| **nested famandassem**	6
| **nested family**	6
| **nested famorassem**	6
| **nested private**	6
| **nested public**	6
| **private**	6
| **public**	6

	Section
<comtypeDecl> ::=	
.assembly extern <dottedname>	4.2.1
| **.class int32**	-
| **.comtype** <dottedname>	-
| **.custom** <customDecl>	17
| **.exeloc** <dottedname>	-
| **.file** <dottedname>	-

	Section
<comtypeHead> ::=	
<comtAttr>* <dottedname> **(** <QSTRING> **)**	-

	Section
<customAttrType> ::=	
<callConv> <type> [<typeSpec> **::**] <methodName> **(** <signature> **)**	17
| <typeSpec>	17

	Section
<customDecl> ::=	
<customAttrType> [**= (** <bytes> **)** | **=** <QSTRING>]	17

	Section
<dataDecl> ::=	
[**tls**] [<dataLabel> =] <ddBody>	13.4.1

	Section
<ddBody> ::=	

```
     <ddItem>                                                          13.4.1
   | { <ddItemList> }                                                  13.4.1
```

```
<ddItem> ::=                                                          Section
     & ( <id> )                                                       13.4.1
   | bytearray ( <bytes> )                                            13.4.1
   | char * ( <QSTRING> )                                             13.4.1
   | float32 [( <float64> )] [[ <int32> ]]                            13.4.1
   | float64 [( <float64> )] [[ <int32> ]]                            13.4.1
   | int8 [( <int8> )] [[ <int32> ]]                                  13.4.1
   | int16 [( <int16> )] [[ <int32> ]]                                13.4.1
   | int32 [( <int32> )] [[ <int32> ]]                                13.4.1
   | int64 [( <int64> )] [[ <int32> ]]                                13.4.1
   | wchar * ( <QSTRING> )                                            13.4.1
```

```
<ddItemList> ::=                                                      Section
     <ddItem> [, <ddItemList>]*                                       13.4.1
```

```
<dataLabel> ::=                                                       Section
     <label>                                                          3.4
```

```
<decl> ::=                                                            Section
     .assembly <asmAttr>* <dottedname> { <asmDecl>* }                  4.2.1
   | .assembly extern [fullorigin] <dottedname> [as <QSTRING>]         4.2.1
       { <asmRefDecl>* }
   | .class <classHead> { <classDecl>* }                               6
   | .comtype <comtypeHead> { <comtypeDecl>* }                         7.2
   | .custom <customDecl>                                              17
   | .data <datadecl>                                                  13.4.1
   | .export [<exportAttr*>] <dottedname> { <exportDecl>* }            4.6
   | .field <fieldDecl>                                                13
   | .file [nometadata] <dottedname> [.hash = ( <bytes> )]             4.2
   | .manifestres [public | private] <dottedname>                      4.2.2
       [( <QSTRING> )] { <manResDecl>* }
   | .method <methodHead>  { <methodDecl>* }                           12
   | .module [[extern] <dottedname>]                                   4.4
```

	.namespace <dottedname> **{** <decl>* **}**	7.1
|	**.vtfixup** <vtfixupDecl>	7.5.2.2
|	<externSourceDecl>	3.7
|	<securityDecl>	16

<dottedname> ::=	**Section**
<id> [. <id>]*	3.1

<eventDecl> ::=	**Section**
.addon <callConv> <type> [<typeSpec> **::**] <methodName> **(** <signature> **)**	15.2
| **.custom** <customDecl>	15.2
| **.fire** <callConv> <type> [<typeSpec> **::**] <methodName> **(** <signature> **)**	15.2
| **.other** <callConv> <type> [<typeSpec> **::**] <methodName> **(** <signature> **)**	15.2
| **.removeon** <callConv> <type> [<typeSpec> **::**] <methodName> **(** <signature> **)**	15.2
| <externSourceDecl>	3.7

<eventHead> ::=	**Section**
[**specialname** | **rtspecialname**]* [<typeSpec>] <id>	15.1

<exportAttr> ::=	**Section**
nested assembly	6
| **nested famandassem**	6
| **nested family**	6
| **nested famorassem**	6
| **nested private**	6
| **nested public**	6
| **public**	6

<exportDecl> ::=	**Section**
.class <int32>	4.6
| **.custom** <customDecl>	4.6
| **.file** <dottedname>	4.6
| **.nestedtype** <dottedname>	4.6

<externFileName> ::=	Section
<dottedname>	3.1

<externSourceDecl> ::=	Section
.line <int32> [<SQSTRING>]	3.7
| **#line** <int32> <QSTRING>	3.7

<fieldAttr> ::=	Section
assembly	13.2.1
| **famandassem**	13.2.1
| **family**	13.2.1
| **famorassem**	13.2.1
| **initonly**	13.2.2
| **literal**	13.2.2
| **marshal (** [<nativeType>] **)**	13.2.3
| **notserialized**	13.2.2
| **pinvokeimpl (** [<QSTRING> [**as** <QSTRING>]] [<pinvAttr>*] **)**	13.4.2
| **private**	13.2.1
| **privatescope**	13.2.1
| **public**	13.2.1
| **rtspecialname**	13.2.4
| **specialname**	13.2.4
| **static**	13.2.2

<fieldDecl> ::=	Section
[**[** <int32> **]**] <fieldAttr>* <type> <id> [**=** <fieldInit> | **at** <dataLabel>]	13

<fieldInit> ::=	Section
bytearray (<bytes> **)**	13.1
| **float32 (** <float64> **)**	13.1
| **float32 (** <int32> **)**	13.1
| **float64 (** <float64> **)**	13.1
| **float64 (** <int64> **)**	13.1
| **int8 (** <int8> **)**	13.1
| **int16 (** <int16> **)**	13.1
| **int32 (** <int32> **)**	13.1

	int64 (<int64> **)** 13.1
|	<QSTRING> 13.1
|	**wchar * (** <QSTRING> **)** 13.1

<float64> ::=	**Section**
float32 (<int32> **)**	3.6
| **float64 (** <int64> **)**	3.6
| <realnumber>	3.6

<handlerBlock> ::=	**Section**
handler <int32> **to** <int32> /* For round trip use only */	18.2
| **handler** <label> **to** <label>	18.2
| <scopeBlock>	18.2

<id> ::=	**Section**
<ID>	3.1
| <SQSTRING>	3.1

<implAttr> ::=	**Section**
forwardref	12.4.3
| **il**	12.4.1
| **internalcall**	12.4.3
| **managed**	12.4.2
| **native**	12.4.1
| **noinlining**	12.4.3
| **ole**	12.4.4
| **oneway**	12.4.3
| **optil**	12.4.1
| **runtime**	12.4.1
| **synchronized**	12.4.3
| **unmanaged**	12.4.2

<label> ::=	**Section**
<id>	3.4

<labels> ::=	**Section**
<labeloroffset> [, <labeloroffset>]*	3.4

<labeloroffset> ::=	Section
<int32> /* For round trip use only */	3.4
| <label>	3.4

<local> ::=	Section
[**[**<int32>**]**] <type> [<id>]	11.2

<localsSignature> ::=	Section
<local> [, <local>]*	11.2

<manResDecl> ::=	Section
.assembly extern <dottedname>	4.2.2
| **.custom** <customDecl>	17
| **.file** <dottedname> **at** <int32>	4.2.2

<methAttr> ::=	Section
abstract	12.3.4
| **assembly**	12.3.1
| **famandassem**	12.3.1
| **family**	12.3.1
| **famorassem**	12.3.1
| **final**	12.3.2
| **hidebysig**	12.3.2
| **newslot**	12.3.3
| **pinvokeimpl (** [<QSTRING> [**as** <QSTRING>]] [<pinvAttr>*] **)**	12.7.2.2
| **private**	12.3.1
| **privatescope**	12.3.1
| **public**	12.3.1
| **rtspecialname**	12.3.6
| **specialname**	12.3.4
| **static**	12.3.2
| **unmanagedexp**	12.3.5
| **virtual**	12.3.2

<methodDecl> ::=	Section
.custom <customDecl>	17

	.data <datadecl> 13.4
|	**.emitbyte** <int32> 12.2
|	**.entrypoint** 12.2
|	**.locals** [**init**] **(** <localsSignature> **)** 12.2.1
|	**.maxstack** <int32> 12.2
|	**.override** <typeSpec>::<methodName> 12.2
|	**.param [** <int32> **]** [**=** <fieldInit>] 12.2.2
|	**.vtentry** <int32> **:** <int32> 12.2.3
|	**.zeroinit** 12.2
|	<externSourceDecl> 3.7
|	<instr> 19
|	<codeLabel> 3.4
|	<scopeBlock> 12.5
|	<securityDecl> 16
|	<sehBlock> 18

<methodHead> ::=	**Section**
<methAttr>* [<callKind>] [<paramAttr>*] <type> [**marshal (** [<nativeType>]**)**] <methodName> **(** <signature> **)** <implAttr>*	12.1

<methodName> ::=	**Section**
.cctor	12.1.1
| **.ctor**	12.1.1
| <dottedname>	12.1.1

<nameValPair> ::=	**Section**
<SQSTRING> **=** <SQSTRING>	16

<nameValPairs> ::=	**Section**
<nameValPair> [, <nameValPair>]*	16

<nativeType> ::=	**Section**
[ ]	11.4
| **as any**	11.4
| **bool**	11.4
| [**ansi**] **bstr**	11.4

	byvalstr 11.4
	custom (<QSTRING> , <QSTRING>) 11.4
	error 11.4
	fixed array [int32] 11.4
	fixed sysstring [int32] 11.4
	float 11.4
	float32 11.4
	float64 11.4
	[**unsigned**] **int** 11.4
	[**unsigned**] **int8** 11.4
	[**unsigned**] **int16** 11.4
	[**unsigned**] **int32** 11.4
	[**unsigned**] **int64** 11.4
	interface 11.4
	lpstr 11.4
	lpstruct 11.4
	lptstr 11.4
	lpvoid 11.4
	lpwstr 11.4
	<nativeType> ***** 11.4
	<nativeType> **[ ]** 11.4
	<nativeType> **[** int32 **]** 11.4
	<nativeType> **[.size .param = int32** [*** int32**] **]** 11.4
	method 11.4
	safearray <variantType> 11.4
	struct 11.4
	tbstr 11.4
	variant bool 11.4

<param> ::=	**Section**
...	11.1
| [<paramAttr>*] <type> [**marshal (** [<nativeType>] **)**] [<id>]	11.1

<paramAttr> ::=	**Section**
[in]	11.1
| **[lcid]**	11.1

	[opt] 11.1
|	**[out]** 11.1
|	**[retval]** 11.1

<pinvAttr> ::=	**Section**
ansi	12.7.2.2
| **autochar**	12.7.2.2
| **cdecl**	12.7.2.2
| **fastcall**	12.7.2.2
| **lasterr**	12.7.2.2
| **nomangle**	12.7.2.2
| **ole**	12.7.2.2
| **stdcall**	12.7.2.2
| **thiscall**	12.7.2.2
| **unicode**	12.7.2.2
| **winapi**	12.7.2.2

<propDecl> ::=	**Section**
.backing <type> <id>	14.2
| **.custom** <customDecl>	14.2
| **.get** <callConv> <type> [<typeSpec> **::**] <methodName> **(** <signature> **)**	14.2
| **.other** <callConv> <type> [<typeSpec> **::**] <methodName> **(** <signature> **)**	14.2
| **.set** <callConv> <type> [<typeSpec> **::**] <methodName> **(** <signature> **)**	14.2
| <externSourceDecl>	3.7

<propHead> ::=	**Section**
[**specialname**|**rtspecialname**]* <callKind> <type> <id> **(** <signature> **)**	14.1

<resolutionScope> ::=	**Section**
[.module <externFileName> **]**	4.4
| **[** <assemblyRefName> **]**	4.3

<scopeBlock> ::=	**Section**
{ <methodDecl>* **}**	12.5

<secAction> ::=	Section
assert	16
| **demand**	16
| **deny**	16
| **inheritcheck**	16
| **linkcheck**	16
| **permitonly**	16
| **prejitdeny**	16
| **prejitgrant**	16
| **reqmin**	16
| **reqopt**	16
| **reqrefuse**	16
| **request**	16

<securityDecl> ::=	Section
.capability <secAction> **= (** <bytes> **)**	16
| **.capability** <secAction> <SQSTRING>	16
| **.permission** <secAction> <className> **(** <nameValPairs> **)**	16

<sehBlock> ::=	Section
<tryBlock> <sehClause> [<sehClause>*]	18

<sehClause> ::=	Section
catch <className> <handlerBlock>	18.2
| **fault** <handlerBlock>	18.2
| **filter** <int32> <handlerBlock>	18.2
| **filter** <label> <handlerBlock>	18.2
| **finally** <handlerBlock>	18.2

<signature> ::=	Section
[<param> [, <param>]*]	11.1

<tryBlock> ::=	Section
.try <int32> **to** <int32> /* For round trip use only */	18.1
| **.try** <label> **to** <label>	18.1
| **.try** <scopeBlock>	18.1

<type> ::=	Section
bool	5.2
| **char**	5.2
| **class** <className>	5.2
| **float32**	5.2
| **float64**	5.2
| **int8**	5.2
| **int16**	5.2
| **int32**	5.2
| **int64**	5.2
| **method** <callConv> <type> ***** **(** <signature> **)**	10.5
| **native float**	5.2
| **native int**	5.2
| **native unsigned int**	5.2
| <type> **&**	10.4
| <type> *****	10.4
| <type> **[ ]**	5.2
| <type> **[** [<bound> [,<bound>]*] **]**	5.2
| <type> **modopt (** <className> **)**	5.2
| <type> **modreq (** <className> **)**	5.2
| <type> **pinned**	5.2
| **typedref**	5.2
| **value class** <className>	5.2
| **unsigned int8**	5.2
| **unsigned int16**	5.2
| **unsigned int32**	5.2
| **unsigned int64**	5.2
| **void**	5.2
| **wchar**	5.2

<typeSpec> ::=	Section
[[**.module**] <dottedname> **]**	
| <className>	5.3
| <type>	5.2

<variantType> ::=	**Section**
blob /* for roundtrip only */	11.4
| **blob_object** /* for roundtrip only */	11.4
| **bstr**	11.4
| **bool**	11.4
| **carray** /* for roundtrip only */	11.4
| **cf** /* for roundtrip only */	11.4
| **clsid** /* for roundtrip only */	11.4
| **currency**	11.4
| **date**	11.4
| **decimal**	11.4
| **error**	11.4
| **filetime** /* for roundtrip only */	11.4
| **float32**	11.4
| **float64**	11.4
| **hresult** /* for roundtrip only */	11.4
| **idispatch ***	11.4
| [**unsigned**] **int** /* for roundtrip only */	11.4
| [**unsigned**] **int8**	11.4
| [**unsigned**] **int16**	11.4
| [**unsigned**] **int32**	11.4
| [**unsigned**] **int64** /* for roundtrip only */	11.4
| **iunknown ***	11.4
| **lpstr** /* for roundtrip only */	11.4
| **lpwstr** /* for roundtrip only */	11.4
| **null** /* for roundtrip only */	11.4
| **record**	11.4
| **safearray** /* for roundtrip only */	11.4
| **storage** /* for roundtrip only */	11.4
| **stored_object** /* for roundtrip only */	11.4
| **stream** /* for roundtrip only */	11.4
| **streamed_object** /* for roundtrip only */	11.4
| **userdefined** /* for roundtrip only */	11.4
| **variant ***	11.4
| <variantType> **&**	11.4
| <variantType> **[ ]**	11.4

```
   | <variantType> vector                                                    11.4

<vtfixupAttr> ::=                                                          Section
     fromunmanaged                                                         7.5.2.2
   | int32                                                                 7.5.2.2
   | int64                                                                 7.5.2.2

<vtfixupDecl> ::=                                                          Section
     [ <int32> ] <vtfixupAttr>* at <dataLabel>                             7.5.2.2
```

21.2 Instruction syntax

21.2.1 Comments

The assembler supports both single line comments (started with "//") and multi-line comments (started with "/*" and ending with "*/").

21.2.2 Labels

A label is specified by a string of characters (technically, an <ID>) followed by a colon. For example:

```
        ldc.i4  1
mylabel:
        ldc.i4  2
        br.1    mylabel
```

Forward branches are allowed.

21.2.3 Full Grammar for Instructions

The remainder of this section describes all of the instruction formats supported by the assembler. The full syntax is here, and each individual instruction format is described in subsequent sections. Instruction names are case-sensitive.

While each section specifies the exact list of instructions that are included in a grammar class, this information is subject to change over time. The precise format of an instruction can be determined can be found by combining the information in the file **opcode.def** (in the SDK) with the information in the following table:

Grammar Class	Format(s) Specified in opcode.def
<instr_brtarget>	`InlineBrTarget, ShortInlineBrTarget`

<instr_field>	InlineField
<instr_i>	InlineI, ShortInlineI
<instr_i8>	InlineI8
<instr_method>	InlineMethod
<instr_none>	InlineNone
<instr_phi>	InlinePhi
<instr_r>	InlineR, ShortInlineR
<instr_rva>	InlineRVA
<instr_sig>	InlineSig
<instr_string>	InlineString
<instr_switch>	InlineSwitch
<instr_tok>	InlineTok
<instr_type>	InlineType
<instr_var>	InlineVar, ShortInlineVar

```
<instr> ::=
    <instr_brtarget> <int32>
  | <instr_brtarget> <label>
  | <instr_field> <type> [ <typeSpec> :: ] <id>
  | <instr_i> <int32>
  | <instr_i8> <int64>
  | <instr_method>
      <callConv> <type> [ <typeSpec> :: ] <methodName> ( <signature> )
  | <instr_none>
  | <instr_phi> <int16>*
  | <instr_r> ( <bytes> )  // <bytes> represent the binary image of
                           // float or double (4 or 8 bytes, respectively)
  | <instr_r> <float64>
  | <instr_r> <int64>   // integer is converted to float with possible
                        // loss of precision
  | <instr_sig> <callConv> <type> ( <signature> )
  | <instr_string> bytearray ( <bytes> )
  | <instr_string> <QSTRING>
  | <instr_switch> ( <labels> )
  | <instr_tok> field <type> [ <typeSpec> :: ] <id>
  | <instr_tok> method
      <callConv> <type> [ <typeSpec> :: ] <methodName> ( <signature> )
  | <instr_tok> <typeSpec>
  | <instr_type> <typeSpec>
```

```
  | <instr_var> <int32>
  | <instr_var> <localname>
```

21.2.4 Instructions with no operand

These instructions require no operands, so they simply appear by themselves.

```
<instr> ::= <instr_none>
<instr_none> ::= // Derived from opcode.def
   add             | add.ovf      | add.ovf.un      | and          |
   ann.catch       | ann.def      | ann.hoisted     | ann.lab      |
   arglist         | break        | ceq             | cgt          |
   cgt.un          | ckfinite     | clt             | clt.un       |
   conv.i          | conv.i1      | conv.i2         | conv.i4      |
   conv.i8         | conv.ovf.i   | conv.ovf.i.un   | conv.ovf.i1  |
   conv.ovf.i1.un  | conv.ovf.i2  | conv.ovf.i2.un  | conv.ovf.i4  |
   conv.ovf.i4.un  | conv.ovf.i8  | conv.ovf.i8.un  | conv.ovf.u   |
   conv.ovf.u.un   | conv.ovf.u1  | conv.ovf.u1.un  | conv.ovf.u2  |
   conv.ovf.u2.un  | conv.ovf.u4  | conv.ovf.u4.un  | conv.ovf.u8  |
   conv.ovf.u8.un  | conv.r.un    | conv.r4         | conv.r8      |
   conv.u          | conv.u1      | conv.u2         | conv.u4      |
   conv.u8         | cpblk        | div             | div.un       |
   dup             | endfilter    | endfinally      | initblk      |
   jmpi            | ldarg.0      | ldarg.1         | ldarg.2      |
   ldarg.3         | ldc.i4.0     | ldc.i4.1        | ldc.i4.2     |
   ldc.i4.3        | ldc.i4.4     | ldc.i4.5        | ldc.i4.6     |
   ldc.i4.7        | ldc.i4.8     | ldc.i4.M1       | ldelem.i     |
   ldelem.i1       | ldelem.i2    | ldelem.i4       | ldelem.i8    |
   ldelem.r4       | ldelem.r8    | ldelem.ref      | ldelem.u1    |
   ldelem.u2       | ldelem.u4    | ldind.i         | ldind.i1     |
   ldind.i2        | ldind.i4     | ldind.i8        | ldind.r4     |
   ldind.r8        | ldind.ref    | ldind.u1        | ldind.u2     |
   ldind.u4        | ldlen        | ldloc.0         | ldloc.1      |
   ldloc.2         | ldloc.3      | ldnull          | localloc     |
   mul             | mul.ovf      | mul.ovf.un      | neg          |
   nop             | not          | or              | pop          |
   refanytype      | rem          | rem.un          | ret          |
   rethrow         | shl          | shr             | shr.un       |
   stelem.i        | stelem.i1    | stelem.i2       | stelem.i4    |
   stelem.i8       | stelem.r4    | stelem.r8       | stelem.ref   |
   stind.i         | stind.i1     | stind.i2        | stind.i4     |
   stind.i8        | stind.r4     | stind.r8        | stind.ref    |
   stloc.0         | stloc.1      | stloc.2         | stloc.3      |
```

```
   sub            | sub.ovf     | sub.ovf.un     | tail.       |
   throw          | volatile.   | xor
```

Examples:

```
   ldlen
   not
```

21.2.5 Instructions that Refer to Parameters or Local Variables

These instructions take one operand, which references a parameter or local variable of the current method. The variable can be referenced by its number (starting with variable 0) or by name (if the names are supplied as part of a signature using the form that supplies both a type and a name).

```
<instr> ::= <instr_var> <int32> |
            <instr_var> <localname>
<instr_var> ::= // Derived from opcode.def
   ann.dead  | ann.live | ann.ref
   ann.ref.s | ldarg    | ldarg.s  | ldarga
   ldarga.s  | ldloc    | ldloc.s  | ldloca
   ldloca.s  | starg    | starg.s  | stloc
   stloc.s
```

Examples:

```
   stloc 0          // store into 0th local
   ldarg X3      // load from argument named X3
```

21.2.6 Instructions that Take a Single 32-bit Integer Argument

These instructions take one operand, which must be a 32-bit integer.

```
<instr> ::= <instr_i> <int32>
<instr_i> ::= // Derived from opcode.def
   ldc.i4 | ldc.i4.s | unaligned.
```

Examples:

```
   ldc.i4 123456  // Load the number 123456
   ldc.i4.s 10    // Load the number 10
```

21.2.7 Instructions that Take a Single 64-bit Integer Argument

These instructions take one operand, which must be a 64-bit integer.

```
<instr> ::= <instr_i8> <int64>
<instr_i8> ::= // Derived from opcode.def
   ldc.i8
```

Examples:

```
   ldc.i8 0x123456789AB
   ldc.i8 12
```

21.2.8 Instructions that Take a Single Floating Point Argument

These instructions take one operand, which must be a floating point number.

```
   <instr> ::= <instr_r> <float64> |
               <instr_r> <int64>   |
               <instr_r> ( <bytes> )  // <bytes> is binary image
<instr_r> ::= // Derived from opcode.def
   ldc.r4 | ldc.r8
```

Examples:

```
   ldc.r4 10.2
   ldc.r4 10
   ldc.r4 0x123456789ABCDEF
   ldc.r8 (00 00 00 00 00 00 F8 FF)
```

21.2.9 Branch instructions

The assembler does not optimize branches. The branch must be specified explicitly as using either the short or long form of the instruction. If the displacement is too large for the short form, then the assembler will display an error.

```
<instr> ::=
   <instr_brtarget> <int32> |
   <instr_brtarget> <label>
<instr_brtarget> ::= // Derived from opcode.def
   ann.data  | ann.data.s | beq    | beq.s    | bge    | bge.s    |
   bge.un    | bge.un.s   | bgt    | bgt.s    | bgt.un | bgt.un.s |
   ble       | ble.s      | ble.un | ble.un.s | blt    | blt.s    |
```

```
blt.un    | blt.un.s   | bne.un | bne.un.s | br     | br.s     |
brfalse   | brfalse.s  | brtrue | brtrue.s | leave  | leave.s
```

Example:

```
br.s 22
br foo
```

21.2.10 Instructions that Take a Method as an Argument

These instructions reference a method, either in another class (first instruction format) or in the current class (second instruction format).

```
<instr> ::=
   <instr_method>
     <callConv> <type> [ <typeSpec> :: ] <methodName> ( <signature> )
<instr_method> ::= // Derived from opcode.def
   ann.call | ann.hoisted_call | call  | callvirt | jmp |
   ldftn    | ldvirtftn        | newobj
```

Examples:

```
call instance int32 C.D.E::X(class W, native int)
ldftn vararg char F(...)   // Global Function F
```

21.2.11 Instructions that Take a Field of a Class as an Argument

These instructions reference a field of a class.

```
<instr> ::=
   <instr_field> <type> <typeSpec> :: <id>
<instr_field> ::= // Derived from opcode.def
   ldfld | ldflda | ldsfld | ldsflda | stfld | stsfld
```

Examples:

```
ldfld native int X::IntField
stsfld int32 Y::AnotherField
```

21.2.12 Instructions that Take a Type as an Argument

These instructions reference a type.

```
<instr> ::= <instr_type> <typeSpec>
<instr_type> ::= // Derived from opcode.def
   box     | castclass | cpobj    | initobj | isinst    |
   ldelema | ldobj     | mkrefany | newarr  | refanyval |
   sizeof  | stobj     | unbox
```

Examples:

```
   initobj System.Console
   sizeof class X
```

21.2.13 Instructions that Take a String as an Argument

These instructions take a string as an argument.

```
<instr> ::= <instr_string> <QSTRING>
<instr_string> ::= // Derived from opcode.def
   ldstr
```

Examples:

```
   ldstr "This is a string"
   ldstr "This has a\nnewline in it"
```

21.2.14 Instructions that Take a Signature as an Argument

These instructions take a stand-alone signature as an argument.

```
<instr> ::= <instr_sig> <callConv> <type> ( <signature> )
<instr_sig> ::= // Derived from opcode.def
   calli
```

Examples:

```
   calli class A.B(wchar *)
   calli vararg bool(int32[,] X, ...)
      // Returns a boolean, takes at least one argument.  The first
      // argument, named X, must be a two-dimensional array of
```

```
      // 32-bit ints
```

21.2.15 Instructions that Take a Metadata Token as an Argument

This instruction takes a metadata token as an argument. The token can reference a type, a method, or a field of a class.

```
<instr> ::= <instr_tok> <typeSpec> |
            <instr_tok> method
               <callConv> <type> <typeSpec> :: <methodName>
                          ( <signature> ) |
            <instr_tok> method
               <callConv> <type> <methodName>
                          ( <signature> ) |
            <instr_tok> field <type> <typeSpec> :: <id>
<instr_tok> ::= // Derived from opcode.def
   ldtoken
```

Examples:

```
ldtoken class System.Console
ldtoken method int32 X::Fn()
ldtoken method bool GlobalFn(int32 &)
ldtoken field class X.Y Class::Field
```

21.2.16 The SSA Φ-Node Instruction

This instruction embeds a static single assignment (SSA) Φ-Node into the instruction stream as an annotation.

```
<instr> ::= <instr_phi> <int16>*
<instr_phi> ::= // Derived from opcode.def
   ann.phi
```

Examples:

```
ann.phi 10 3 15
ann.phi 3 -2 0x3
```

21.2.17 Switch instruction

The switch instruction takes a set of labels or decimal relative values.

```
<instr> ::= <instr_switch> ( <labels> )
```

```
<instr_switch> ::= // Derived from opcode.def
   switch
```

Examples:

```
   switch (0x3, -14, Label1)
   switch (5, Label2)
```