Developer

INSIDE MICROSOFT®

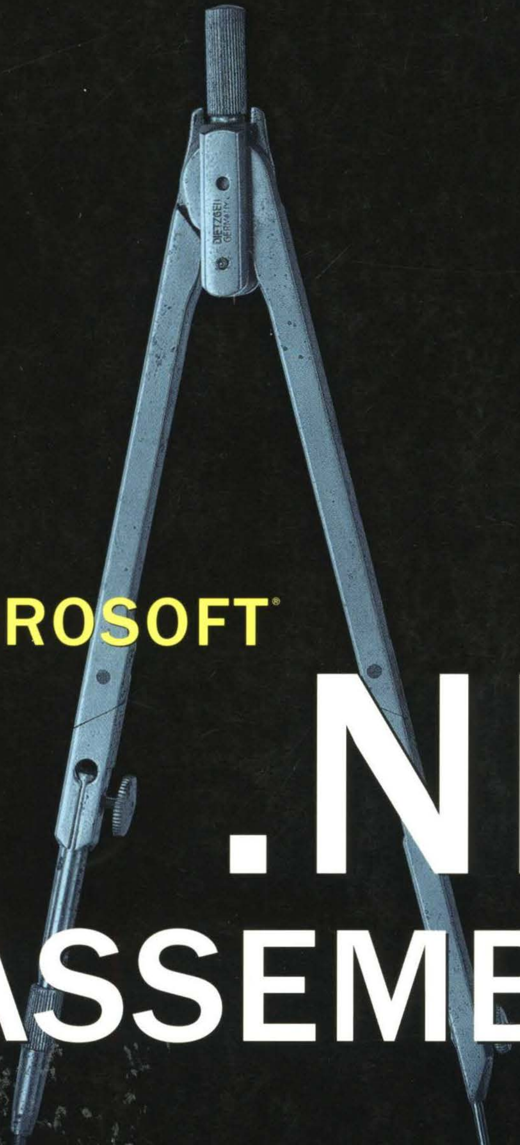
.NET IL ASSEMBLER

Lidin

Microsoft Press

Architectural Reference

Microsoft®



INSIDE MICROSOFT®

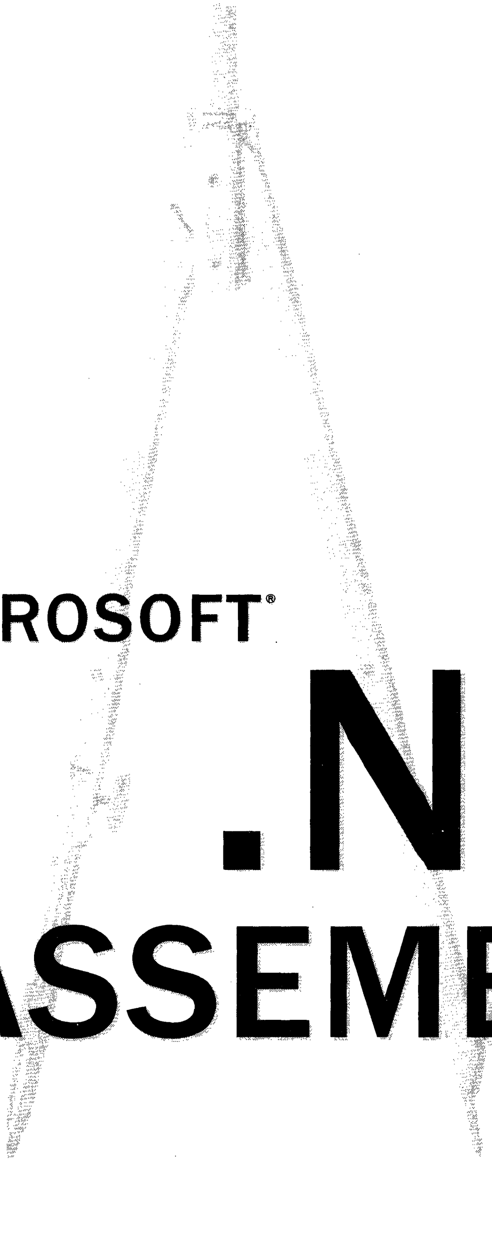
.NET IL ASSEMBLER

Microsoft® .net

Serge Lidin

Architectural Reference

Microsoft®



INSIDE MICROSOFT® .NET IL ASSEMBLER

Microsoft®
.net™

Serge Lidin

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
Lidin, Serge, 1956-
Inside Microsoft .NET IL Assembler / Serge Lidin.
p. cm.
Includes index.
ISBN 0-7356-1547-0
1. Assembling (Electronic computers) I. Title.

QA76.76.A87 L545 2002
005.2'768--dc21 2001058690

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWT 7 6 5 4 3 2

Distributed in Canada by Penguin Books Canada Limited.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to *mspinput@microsoft.com.*

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

Acquisitions Editor: Danielle Bird
Project Editor: Devon Musgrave
Technical Editor: Julie Xiao

Body Part No. X08-56864

To my family, for all their patience with me.

Contents at a Glance

Appendixes

Table of Contents

Part IV Inside the Execution Engine

Part VI Interoperation

Appendixes

Introduction

Why This Book Was Written

To tell the truth, I don't think I had much choice in this matter. Let me explain. With Microsoft .NET technology taking the world by storm, with more and more information professionals getting involved, large numbers of books covering various aspects of this technology have started to arrive—and none too soon. Alas, virtually all of these books are dedicated to .NET-based programming in high-level languages and rapid application development (RAD) environments. No doubt this is extremely important, and I am sure all these books will have to be reprinted to satisfy the demand. But what about the plumbing?

The .NET universe, like other information technology universes, resembles a great pyramid turned upside down and standing on its tip. The tip on which the .NET pyramid stands is the common language runtime. The runtime converts the intermediate language (IL) binary code into platform-specific (native) machine code and executes it. Resting on top of the runtime are the .NET Framework class library, the compilers, and environments such as Microsoft Visual Studio .NET. And above them begin the layers of application development, from instrumental to end-user-oriented. The pyramid quickly grows higher and wider.

This book is not exactly about the common language runtime—even though it's only the tip of the .NET pyramid, the runtime is too vast a topic to be described in detail in any book of reasonable (say, luggable) size. Rather, this book focuses on the next best thing: the .NET IL Assembler. IL assembly language (ILAsm) is a low-level language, specifically designed to describe every functional feature of the common language runtime. If the runtime can do it, ILAsm must be able to express it.

Unlike high-level languages, and like other assembly languages, ILAsm is platform-driven rather than concept-driven. An assembly language usually is an exact linguistic mapping of the underlying platform, which in this case is the common language runtime. It is, in fact, so exact a mapping that this language is used for describing aspects of the runtime in the ECMA standardization documents regarding the .NET common language infrastructure. (ILAsm itself, as a part of the common language infrastructure, is a subject of this standardization effort as well.) As a result of the close mapping, it is impossible to describe an assembly language without going into significant detail about the underlying platform. So, to a great extent, this book *is* about the common language runtime after all.

IL assembly language is very popular among .NET developers. No, I am not claiming that all .NET developers prefer to program in ILAsm rather than in Microsoft Managed C++, Microsoft Visual C# .NET, or Microsoft Visual Basic .NET. But all .NET developers use the IL Disassembler (ILDASM) now and then, and many use it on a regular basis. A cyan thunderbolt—the ILDASM icon (a silent praise for David Drake)—glows on the computer screens of .NET developers regardless of their language preferences and problem areas. And ILDASM text output is…? Yes, ILAsm source code.

Virtually all books on .NET-based programming that are devoted to high-level programming languages, such as Visual C# .NET or Visual Basic .NET, or to techniques such as ADO.NET at some moment mention the IL Disassembler as a tool of choice to analyze the innards of a .NET IL executable. But these volumes stop short of explaining what the disassembly text means and how to interpret it. This is an understandable choice, given the topics of these books; the detailed description of metadata structuring and IL assembly language represents a separate issue.

Now perhaps you see what I mean when I say I had no choice but to write this book. *Someone* had to, and because I had been given the responsibility of designing and developing IL Assembler and ILDASM, it was my obligation to see it through all the way.

History of ILAsm, Part I

The first versions of IL Assembler and ILDASM (under the names Asm and Dasm, respectively) were developed in early 1998 by Jonathan Forbes. The current language is very different from this original one, the only distinct common feature being the leading dots in the directive keywords. The assembler and disassembler were built as purely internal tools facilitating the ongoing development of the common language runtime and were used rather extensively inside the runtime development team.

When Jonathan went to work on Microsoft Messenger in the beginning of 1999, the assembler and disassembler fell in the lap of Larry Sullivan, head of a development group with the colorful name CROEDT (Common Runtime Odds and Ends Development Team). In April of that year, I joined the team, and Larry passed the assembler and disassembler to me. When an alpha version of the common language runtime was presented at a Technical Preview in May 1999, Asm and especially Dasm attracted significant attention, and I was told to rework the tools and bring them up to production level. So I did, with great help from Larry, Vance Morrison, and Jim Miller. Because the tools were still considered internal, we (Larry, Vance, Jim, and I) could afford to redesign the language—not to mention the implementation of the tools—radically.

A major breakthrough occurred in the second half of 1999, when IL Assembler input and ILDASM output were synchronized enough to achieve limited round-tripping. Round-tripping means that you can take a managed (IL) executable compiled from a particular language, disassemble it, add or change some ILAsm code, and reassemble it back into a modified executable. Round-tripping technique opened new avenues, and shortly thereafter it began to be used in certain production processes both inside Microsoft and by its partners.

At about the same time, third-party .NET-oriented compilers that used ILAsm as a base language started to appear. The best-known is probably Fujitsu's COBOL.NET, which made quite a splash at the Professional Developers Conference in July 2000, where the first pre-beta version of the common language runtime, along with the .NET Framework class library, compilers, and tools, was released to the developer community.

Since the release of the beta 1 version in late 2000, IL Assembler and ILDASM have been fully functional in the sense that they reflect all the features of metadata and IL, support complete round-tripping, and maintain synchronization of their changes with the changes in the runtime itself.

Who Should Read This Book

This book targets all the .NET-oriented developers who, because they work at a sufficiently advanced level, care about what their programs compile into or who are willing to analyze the end results of their programming. Here these readers will find the information necessary to interpret disassembly texts and metadata structure summaries, allowing them to develop more efficient programming techniques.

Because this analysis of disassemblies and metadata structuring is crucial in assessing the correctness and efficiency of any .NET-oriented compiler, this book should also prove especially useful for compiler developers who are targeting .NET. A narrower but growing group of readers who will find the book extremely helpful includes developers who use IL assembly language directly: for example, compiler developers targeting ILAsm as an intermediate step, developers contemplating multilanguage projects, and developers willing to exploit the capabilities of the common language runtime that are inaccessible through the high-level languages.

Finally, this book can be valuable in all phases of software development, from conceptual design to implementation and maintenance.

Organization of This Book

I begin in Part I, "Quick Start," with a quick overview of ILAsm and common language runtime features, based on a simple sample program. This overview is in no way complete; rather, it is intended to convey a general impression about the runtime and ILAsm as a language.

The following parts discuss features of the runtime and corresponding ILAsm constructs in a detailed, bottom-up manner. Part II, "Underlying Structures," describes the structure of a managed executable file and general metadata organization. Part III, "Fundamental Components," is dedicated to the components that constitute a necessary base of any application: assemblies, modules, classes, methods, fields, and related topics. Part IV, "Inside the Execution Engine," brings you, yes, inside the execution engine, describing the execution of IL instructions and managed exception handling. Part V, "Special Components," discusses metadata representation and usage of the additional components: events, properties, and custom and security attributes. And Part VI, "Interoperation," describes the interoperation between managed and unmanaged code and discusses practical applications of IL Assembler and ILDASM to multilanguage projects.

The book's five appendixes contain references concerning ILAsm grammar, metadata organization, and the IL instruction set and tool features, including IL Assembler, ILDASM, and the offline metadata validation tool.

About the Companion CD

The book contains a companion CD. If you have the Autorun feature of Microsoft Windows enabled, the CD autorun interface will start when you insert the CD in your CD-ROM drive; otherwise, you can manually run StartCD.exe from the root directory of the companion CD. The StartCD menu provides you with links to the book in eBook format, which is contained on the CD; an installation program for the book's sample files; and a link to the Microsoft Developer Network (MSDN), where you can download the Microsoft .NET Framework Software Development Kit (SDK), which you'll need in order to compile and run the samples. Notice that this link is accessible to MSDN subscribers only.

Installing the Sample Files

The sample files for the book are located in the Code folder. You can view the samples from the CD, or you can install them on your hard disk by using the installer from StartCD. Installing the sample files requires approximately 18 KB of disk space. If you have trouble running any of these files, refer to the Readme.txt file in the root directory of the companion CD.

eBook

The companion CD contains an electronic version of the book. This eBook allows you to view the book text on screen and to search the contents. For information on installing and using the eBook, see the Readme.txt file in the \eBook folder.

System Requirements

To work with the samples, you will need to install the .NET Framework with its SDK. At a bare minimum, you need the common language runtime, the .NET Framework class library, and the SDK. Visual Studio .NET and command-line compilers—except, of course, the ILAsm compiler—are not required.

Acknowledgments

First I would like to thank the editing team from Microsoft Press who worked with me on this book: Danielle Bird, Alice Turner, Robert Lyon, Mary Renaud (who didn't allow me to use propositions to end the sentences with), Julie Xiao (who learned ILAsm while looking for errors in my tables and samples and perhaps can now apply for a developer position at Microsoft), and especially Devon Musgrave. Devon, who was my editor, undoubtedly gained some gray hair working on this book: I am a professional programmer on active duty, and the things I write in human languages are usually limited to e-mail messages.

I would also like to thank my colleagues who, despite being unbelievably busy, agreed to review the draft of the book and gave me some very good advice on the contents: development leads Larry Sullivan (Runtime Platform Services), Bill Evans (Runtime Metadata), Chris Brumme (Runtime Execution Engine), Vance Morrison (Runtime JIT Compiler), program managers Erik Meijer (Runtime) and Ronald Laeremans (Visual C++), and one of our best test engineers Kevin Ransom. I greatly appreciate their help and all those "What were you thinking when you wrote..." and "No, it's exactly the other way around..." e-mails.

And of course I wish to thank all members of the common language runtime team who helped me by answering my questions, discussing the specification documents, and diving into the source code with me: Suzanne Cook, Shajan Dasan, Jim Hogg, Jim Miller, Craig Sinclair, Mei-Chin Tsai, and many others.

Microsoft Press Support Information

Every effort has been made to ensure the accuracy of the book and the contents of this companion CD. Microsoft Press provides corrections for books through the World Wide Web at:

http://www.microsoft.com/mspress/support/.

If you have comments, questions, or ideas regarding the book or this CD, or questions that are not answered by querying the Knowledge Base, please send them to Microsoft Press via e-mail to:

mspinput@microsoft.com

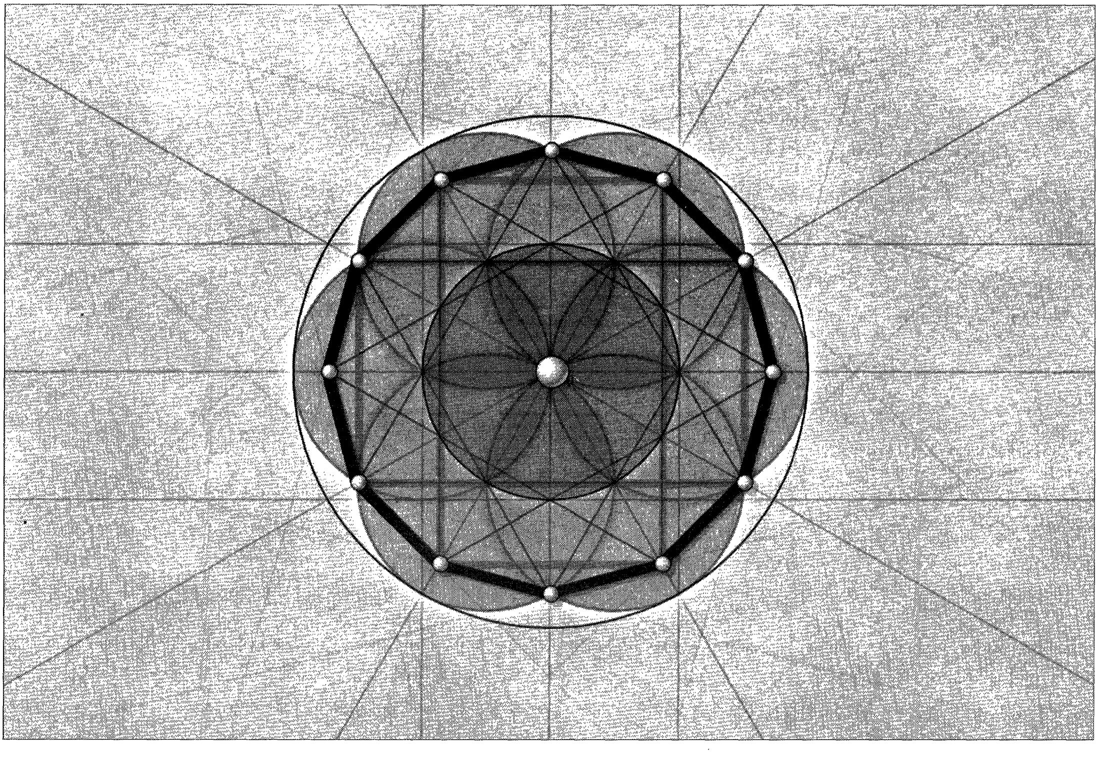
or via postal mail to:

Microsoft Press
Attn: Inside Microsoft .NET IL Assembler Editor
One Microsoft Way
Redmond, WA 98052-6399

Please note that product support is not offered through the above addresses.

Part I

Quick Start



1

Simple Sample

This chapter offers a general overview of the MSIL assembly language (ILAsm). (*MSIL* stands for Microsoft intermediate language, which will soon be discussed in this chapter.) We'll review a relatively simple program written in ILAsm, and then I'll suggest some modifications that illustrate how the concepts and elements of Microsoft .NET programming are expressed in this language.

This chapter does not teach you how to write programs in ILAsm. But it should help you to understand what the ILAsm compiler and the IL Disassembler (ILDASM) do and to use that understanding to analyze the internal structure of a .NET-based program with the help of these ubiquitous tools. You'll also learn some intriguing facts about the mysterious affairs that take place behind the scenes, within the common language runtime—intriguing enough, I hope, to prompt you to read the rest of the book.

> **Note** For your sake and mine, I'll abbreviate *IL assembly language* as *ILAsm* throughout this book. Don't confuse it with *ILASM*, which is used as the abbreviation for the ILAsm compiler in the .NET documentation.

Basics of the Common Language Runtime

The .NET common language runtime is but one of many aspects of the .NET concept, but it's the core of .NET. (Note that, for variety's sake, I'll sometimes refer to the common language runtime as *the runtime*.) Rather than focusing on an overall description of the .NET platform here, then, let's concentrate on the part of .NET where the action really happens: the common language runtime.

More Info For excellent discussions of the general structure of .NET and its components, see *Introducing Microsoft .NET* (Microsoft Press, 2001), by David S. Platt, and *Inside C#* (Microsoft Press, 2001), by Tom Archer.

Simply put, the common language runtime is a run-time environment in which .NET applications run. It provides an operating layer between the .NET applications and the underlying operating system. In principle, the common language runtime is similar to the runtimes of interpreted languages such as GBasic or Smalltalk or to the Java Virtual Machine. But this similarity is only in principle: the common language runtime is not an interpreter.

The .NET applications generated by .NET-oriented compilers (such as Microsoft Visual C# .NET, Microsoft Visual Basic .NET, ILAsm, and many others) are represented in an abstract, intermediate form, independent of the original programming language and of the target machine and its operating system. Because they are represented in this abstract form, .NET applications written in different languages can interoperate very closely, not only on the level of calling each other's functions but also on the level of class inheritance.

Of course, given the differences in programming languages, a set of rules must be established for the applications to allow them to get along with their neighbors nicely. For example, if you write an application in Visual C# .NET and name three items *MYITEM*, *MyItem*, and *myitem*, Visual Basic .NET, which is case-insensitive, will have a hard time differentiating them. Likewise, if you write an application in ILAsm and define a global method, Visual C# .NET will be unable to call the method because it has no concept of global (out-of-class) items.

The set of rules guaranteeing the interoperability of .NET applications is known as the common language specification (CLS), outlined in Partition I of the Common Language Infrastructure standardization proposal of the European Computer Manufacturers Association (ECMA). It limits the naming conventions, data types, function types, and certain other elements, forming a common denominator for different languages. It is important to remember, however, that the CLS is merely a recommendation and has no bearing whatsoever on common language runtime functionality. If your application is not CLS-compliant, it might be valid in terms of the common language runtime, but you have no guarantee that it will be able to interoperate with other applications on all levels.

The abstract intermediate representation of the .NET applications, intended for the common language runtime environment, includes two main components: metadata and managed code. *Metadata* is a system of descriptors of all structural items of the application—classes, their members and attributes, global items, and so on—and their relationships. This chapter provides some examples of metadata, and later chapters describe all the metadata structures.

The *managed code* represents the functionality of the application's methods (functions) encoded in an abstract binary form known as Microsoft intermediate language (MSIL), or common intermediate language (CIL). To simplify things, I'll refer to this encoding simply as *intermediate language (IL)*. Of course, other intermediate languages exist in the world, but as far as our endeavors are concerned, let's agree that *IL* means *CIL/MSIL*, unless specified otherwise.

The IL code is "managed" by the runtime. Common language runtime management includes, but is not limited to, three major activities: type control, structured exception handling, and garbage collection. Type control involves verification and conversion of item types during execution. Structured exception handling is functionally similar to "unmanaged" structured exception handling (C++-style), but it is performed by the runtime rather than by the operating system. Garbage collection involves automatic identification and disposal of objects no longer in use.

A .NET application, intended for the common language runtime environment, consists of one or more *managed executables*, each of which carries metadata and (optionally) managed code. Managed code is optional because it is always possible to build a managed executable containing no methods. (Obviously, such an executable can be used only as an auxiliary part of an application.) Managed .NET applications are called *assemblies*. (This statement is somewhat simplified; for more details about assemblies, application domains, and applications, see Chapter 5) The managed executables are referred to as *modules*. You can create single-module assemblies and multimodule assemblies. As illustrated in Figure 1-1, each assembly contains one prime module, which carries the assembly identity information in its metadata.

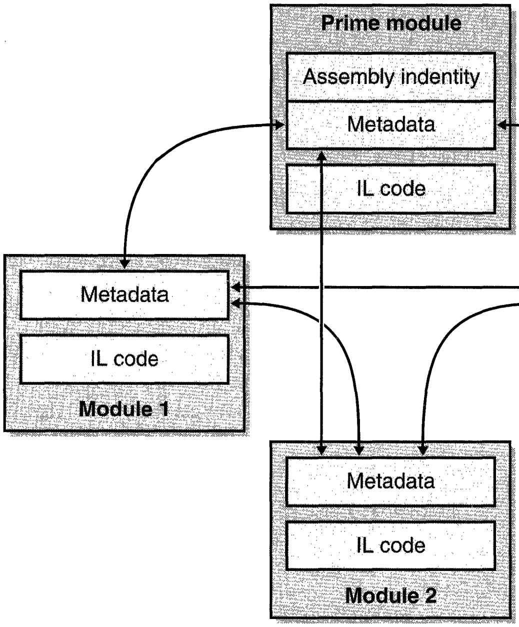


Figure 1-1 A multimodule .NET assembly.

Figure 1-1 also shows that the two principal components of a managed executable are the metadata and the IL code. The two major common language runtime subsystems dealing with each component are, respectively, the loader and the JIT (just-in-time) compiler.

In brief, the *loader* reads the metadata and creates in memory an internal representation and layout of the classes and their members. It performs this task on demand, meaning that a class is loaded and laid out only when it is referenced. Classes that are never referenced are never loaded. When loading a class, the loader runs a series of consistency checks of the related metadata.

The *JIT compiler*, relying on the results of the loader's activity, compiles the methods encoded in IL into the native code of the underlying platform. Because the runtime is not an interpreter, it does not execute the IL code. Instead, the IL code is compiled in memory into the native code, and the native code is executed. The JIT compilation is also done on demand, meaning that a method is compiled only when it is called. The compiled methods stay cached in memory. If memory is limited, however, as in the case of a small computing device such as a handheld PDA or a smart phone, the methods can be discarded if not used. If a method is called again after being discarded, it is recompiled.

The diagram shown in Figure 1-2 illustrates the sequence of creation and execution of a managed .NET application.

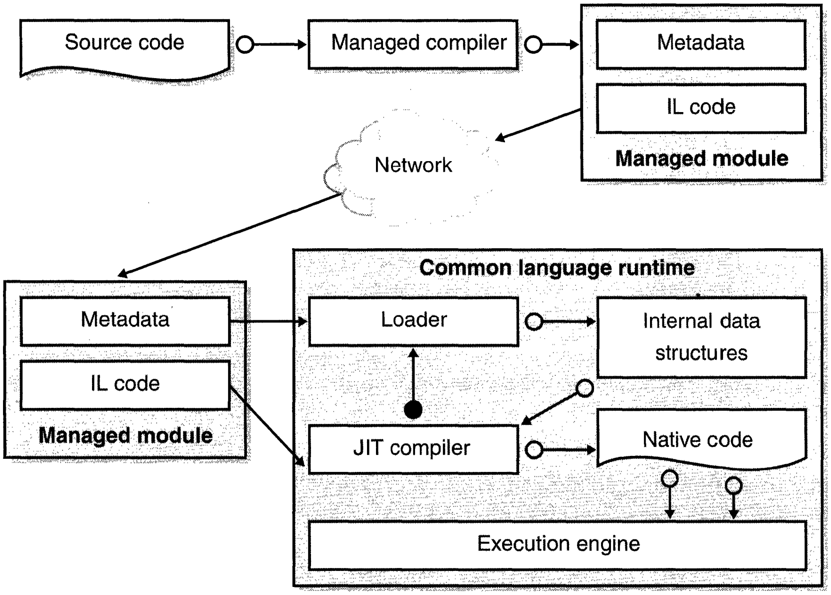


Figure 1-2 The creation and execution of a managed .NET application. Arrows with hollow circles at the base indicate data transfer; arrows with black circles represent requests and control messages.

A managed executable can be precompiled from IL to the native code, using the NGEN utility. You can do this when the executable is expected to run repeatedly from a local disk, to save time on just-in-time compilation. This is standard procedure, for example, for managed components of the .NET Framework, which are precompiled during the installation. (Tom Archer refers to this as *install-time code generation.*) In this case, the precompiled code is saved to the local disk or other storage, and every time the executable is invoked, the precompiled native-code version is used instead of the original IL version. The original file, however, must also be present because the precompiled version does not carry the metadata.

With the roles of the metadata and the IL code established, let's consider the ways you can use ILAsm to describe them.

A Simple Sample

No, the sample is not going to be "Hello, World!" This sample is a simple managed console application that prompts the user to enter an integer and then identifies the integer as odd or even. When the user enters something other than a decimal number, the application responds, "How rude!" and terminates. (See the source file Simple.il on the companion CD included with this book.)

The sample uses managed console APIs from the .NET Framework class library for console input and output, and it uses the unmanaged function *sscanf* from the C run-time library for input string conversion to an integer.

> **Note** To increase code accessibility throughout this book, all ILAsm keywords will appear in bold.

```
//----------- Program header
.assembly extern mscorlib { }
.assembly OddOrEven  { }
.module OddOrEven.exe
//----------- Class declaration
.namespace Odd.or {
    .class public auto ansi Even extends [mscorlib]System.Object {
//----------- Field declaration
        .field public static int32 val
//----------- Method declaration
        .method public static void check( ) cil managed {
            .entrypoint
            .locals init (int32 Retval)
        AskForNumber:
            ldstr "Enter a number"
            call void [mscorlib]System.Console::WriteLine(string)
            call string [mscorlib]System.Console::ReadLine()
            ldsflda valuetype CharArray8 Format
            ldsflda int32 Odd.or.Even::val
            call vararg int32 sscanf(string,int8*,...,int32*)
            stloc Retval
            ldloc Retval
            brfalse Error
            ldsfld int32 Odd.or.Even::val
            ldc.i4 1
            and
            brfalse ItsEven
            ldstr "odd!"
            br PrintAndReturn
        ItsEven:
            ldstr "even!"
            br PrintAndReturn
        Error:
            ldstr "How rude!"
        PrintAndReturn:
            call void [mscorlib]System.Console::WriteLine(string)
            ldloc Retval
            brtrue AskForNumber
            ret
        } // End of method
    } // End of class
} // End of namespace
//----------- Global items
.field public static valuetype CharArray8 Format at FormatData
//----------- Data declaration
.data FormatData = bytearray(25 64 00 00 00 00 00 00) //
% d . . . . . .
//----------- Value type as placeholder
.class public explicit CharArray8
              extends [mscorlib]System.ValueType { .size 8 }
//----------- Calling unmanaged code
.method public static pinvokeimpl("msvcrt.dll" cdecl)
    vararg int32 sscanf(string,int8*) cil managed { }
```

In the following sections, we'll walk through this source code line by line.

Program Header

```
.assembly extern mscorlib { }
.assembly OddOrEven  { }
.module OddOrEven.exe
```

`.assembly extern mscorlib { }` defines a metadata item named *Assembly Reference* (or *AssemblyRef*), identifying the external managed application (assembly) used in this program. In this case, the external application is Mscorlib.dll, the main assembly of the .NET Framework classes. (The topic of the .NET Framework class library itself is beyond the scope of this book; for further information, consult the detailed specification of the .NET Framework class library published as Partition IV of the proposed ECMA standard.)

The Mscorlib.dll assembly contains declarations of all the base classes from which all other classes are derived. Although theoretically you could write an application that never uses anything from Mscorlib.dll, I doubt that such an application would be of any use. (One obvious exception is Mscorlib.dll itself.) Thus it's a good habit to begin a program in ILAsm with a declaration of *AssemblyRef* to Mscorlib.dll, followed by declarations of other *AssemblyRefs* (if any).

The scope of an *AssemblyRef* declaration (between the curly braces) can contain additional information identifying the referenced assembly, such as version or culture (previously known as *locale*). Because this information is not mandatory for referencing Mscorlib.dll, I have omitted it from this sample. (Chapter 5 describes this additional information in detail.)

Note that although the code references the assembly Mscorlib.dll, *AssemblyRef* is declared by filename only, without the extension. Including the extension causes the loader to look for Mscorlib.dll.dll or Mscorlib.dll.exe, resulting in a run-time error.

.assembly OddOrEven { } defines a metadata item named *Assembly*, which, to no one's surprise, identifies the current application (assembly). Again, you could include additional information identifying the assembly in the assembly declaration—see Chapter 5 for details—but it is not necessary here. Like *AssemblyRef*, the assembly is identified by its filename, without the extension.

Why must you identify the application as an assembly? If you don't, it will not be an application at all; rather, it will be a nonprime module—part of some other application (assembly)—and as such will not be able to execute on its own. Giving the module an EXE extension changes nothing; only assemblies can be executed.

.module OddOrEven.exe defines a metadata item named *Module*, identifying the current module. Each module, prime or otherwise, carries this identification in its metadata. Note that the module is identified by its full filename, including the extension. The path, however, must not be included.

Class Declaration

```
.namespace Odd.or {
    .class public auto ansi Even extends [mscorlib]System.Object {
        ⋮
    }
    ⋮
}
```

.namespace Odd.or { … } declares a namespace. A namespace does not represent a separate metadata item. Rather, a namespace is a common prefix of the full names of all the classes declared within the scope of the namespace declaration.

.class public auto ansi Even **extends** [mscorlib]System.Object { … } defines a metadata item named *Type Definition* (*TypeDef*). Each class, structure, or enumeration defined in the current module is described by a respective *TypeDef* record in the metadata. The name of the class is *Even*. Because it is declared within the scope of the namespace *Odd.or*, its full name, by which it can be referenced elsewhere and by which the loader identifies it, is *Odd.or.Even*.

The keywords *public*, *auto*, and *ansi* define the flags of the *TypeDef* item. The keyword *public*, which defines the visibility of the class, means that the class is visible outside the current assembly. (Another keyword for class visibility is *private*, the default, which means that the class is for internal use only and cannot be referenced from outside.)

The keyword *auto* defines the class layout style (automatic, the default), directing the loader to lay out this class however it sees fit. Alternatives are *sequential* (which preserves the specified sequence of the fields) and *explicit* (which explicitly specifies the offset for each field, giving the loader exact instructions for laying out the class).

The keyword *ansi* defines the mode of string conversion within the class, when interoperating with the unmanaged code. This keyword, the default, specifies that the strings will be converted to and from "normal" C-style strings of bytes. Alternative keywords are *unicode* (strings are converted to and from Unicode) and *autochar* (the underlying platform determines the mode of string conversion).

The clause `extends [mscorlib]System.Object` defines the parent, or base class, of the class *Odd.or.Even*. The code *[mscorlib]System.Object* represents a metadata item named *Type Reference* (*TypeRef*). This particular *TypeRef* has *System* as its namespace, *Object* as its name, and *AssemblyRef mscorlib* as the resolution scope. Each class defined outside the current module is addressed by *TypeRef*. You can even address the classes defined in the current module by *TypeRefs* instead of *TypeDefs*, which is considered harmless enough but not nice.

By default, all classes are derived from the class *System.Object* defined in the assembly Mscorlib.dll. Only *System.Object* itself and the interfaces have no base class, as explained in Chapter 6

The structures—referred to as *value types* in .NET lingo—are derived from the *[mscorlib]System.ValueType* class. The enumerations are derived from the *[mscorlib]System.Enum* class. Because these two distinct kinds of *TypeDefs* are recognized solely by the classes they extend, you must use the *extends* clause every time you declare a value type or an enumeration.

Using Pseudoflags to Declare a Value Type and an Enumeration

You might want to know about a little cheat that will allow you to circumvent the necessity of repeating the *extends* clause. ILAsm has two keywords, *value* and *enum*, that can be placed among the class flags to identify, respectively, value types and enumerations if you omit the *extends* clause. (If you include the *extends* clause, these keywords are ignored.) This is, of course, not a proper way to represent the metadata, because it can give one the incorrect impression that value types and enumerations are identified by certain *TypeDef* flags. I am ashamed of the fact that ILAsm contains such lowly tricks, but I am too lazy to type *extends [mscorlib]System.ValueType* again and again. ILDASM never resorts to these cheats and always truthfully prints the *extends* clause, but ILDASM has the advantage of being a software utility.

You have probably noticed that the declaration of *TypeDef* in the sample contains three default flags: *public*, *auto*, and *ansi*. Yes, in fact, you could declare exactly the same *TypeDef* as *.class public Even { … }*, but then we would not be able to discuss the *TypeDef* flags and the *extends* clause.

Finally, I must emphasize one important fact about the class declaration in ILAsm. (Please pay attention, and don't say I haven't told you!) The languages such as Visual C# .NET and Visual Basic .NET require that all of a class's attributes and members be defined within the lexical scope of the class, defining the class as a whole in one place. ILAsm is similar *except* that the class needn't be defined all in one place.

In ILAsm, you can declare a *TypeDef* with some of its attributes and members, close the *TypeDef*'s scope, and then reopen the same *TypeDef* later in the source code to declare more of its attributes and members. This technique is referred to as *class amendment*.

When you amend a *TypeDef*, the flags, the *extends* clause, and the *implements* clause (not discussed here, in the interests of keeping the sample simple) are ignored. You should define these characteristics of a *TypeDef* the first time you declare it.

There is no limitation on the number of *TypeDef* amendments or on how many source files a *TypeDef* declaration might span. You are required, however, to completely define a *TypeDef* within one module. Thus it is impossible to amend the *TypeDef*s defined in other assemblies or other modules of the same assembly.

Chapter 6 provides extensive and detailed information about ILAsm class declarations.

Field Declaration

```
.field public static int32 val
```

`.field public static int32 val` defines a metadata item named *Field Definition* (*FieldDef*). Because the declaration occurs within the scope of class *Odd.or.Even*, the declared field belongs to this class.

The keywords *public* and *static* define the flags of the *FieldDef*. The keyword *public* identifies the accessibility of this field and means that the field can be accessed by any member for whom this class is visible. Alternative accessibility flags are as follows:

- The *assembly* flag specifies that the field can be accessed from anywhere within this assembly but not from outside.
- The *family* flag specifies that the field can be accessed from any of the classes descending from *Odd.or.Even*.
- The *famandassem* flag specifies that the field can be accessed from any of those descendants of *Odd.or.Even* that are defined in this assembly.
- The *famorassem* flag specifies that the field can be accessed from anywhere within this assembly as well as from any descendant of *Odd.or.Even*, even if the descendant is declared outside this assembly.
- The *private* flag specifies that the field can be accessed from *Odd.or.Even* only.
- The *privatescope* flag is the default. See the Caution reader aid for important information about this flag.

Caution The *privatescope* flag is a special case, and I strongly recommend that you do *not* use it. Private scope items are exempt from the requirement of having a unique parent/name/signature triad, which means that you can define two or more private scope items within the same class that have the same name and the same type. Some compilers emit private scope items for their internal purposes. It is the compiler's problem to distinguish one private scope item from another; if you decide to use private scope items, you should at least give them unique names.

Because the default accessibility is *privatescope*, which can be a problem, it's important to remember to specify the accessibility flags.

The keyword *static* means that the field is static—that is, it is shared by all instances of class *Odd.or.Even*. If you did not designate the field as static, it would be an instance field, individual to a specific instance of the class.

The keyword *int32* defines the type of the field, a 32-bit signed integer. (Types and signatures are described in Chapter 7 And, of course, *val* is the name of the field.

More Info You can find a detailed explanation of field declarations in Chapter 8

Method Declaration

```
    .method public static void check( ) cil managed {
        .entrypoint
        .locals init (int32 Retval)
:
}
```

.method public static void check( ) **cil managed** { … } defines a metadata item named *Method Definition* (*MethodDef*). Because it is declared within the scope of *Odd.or.Even*, this method is a member method of this class.

The keywords *public* and *static* define the flags of *MethodDef* and mean the same as the similarly named flags of *FieldDef* discussed in the preceding section. Not all the flags of *FieldDef*s and *MethodDef*s are identical—see Chapter 8 as well as Chapter 9 for details—but the accessibility flags are, and the keyword *static* means the same for fields and methods.

The keyword *void* defines the return type of the method. If the method had a calling convention that differed from the default, you would place the respective keyword after the flags but before the return type. Calling convention, return type, and types of method parameters define the signature of the *MethodDef*. Note that a lack of parameters is expressed as *( )*, never as *(void)*. The notation *(void)* would mean that the method has one parameter of type *void*—an illegal signature.

The keywords *cil* and *managed* define so-called implementation flags of the *MethodDef* and indicate that the method body is represented in IL. A method represented in native code rather than in IL would carry the implementation flags *native unmanaged*.

Now, let's proceed to the method body. In ILAsm, the method body (or method scope) generally contains three categories of items: instructions (compiled into IL code), labels marking the instructions, and directives (compiled into metadata, header settings, structured exception handling clauses, and so on—in short, anything but IL code). Outside the method body, only directives exist. Every declaration discussed so far has been a directive.

.entrypoint identifies the current method as the entry point of the application (the assembly). Each managed EXE file must have a single entry point. The ILAsm compiler will refuse to compile a module without a specified entry point, unless you use the /DLL command-line option.

.locals init (*int32* Retval) defines the single local variable of the current method. The type of the variable is *int32*, and its name is *Retval.* The keyword *init* means that the local variables must be initialized before the method executes. If the local variables are not designated with this keyword in even one of the assembly's methods, the assembly will fail verification (in a security check performed by the common language runtime) and will be able to run only from a local disk, when verification is disabled. For that reason, you should never forget to use the keyword *init* with the local variable declaration. If you need more than one local variable, you can list them, comma-separated, within the parentheses—for example, *.locals init (int32 Retval, string TempStr).*

```
AskForNumber:
    ldstr "Enter a number"
    call void [mscorlib]System.Console::WriteLine(string)
```

AskForNumber: is a label. It needn't occupy a separate line; the IL Disassembler marks every instruction with a label on the same line as the instruction. Labels are not compiled into metadata or IL; rather, they are used solely for the identification of certain offsets within IL code at compile time.

A label marks the first instruction that follows it. Labels don't mark directives. In other words, if you moved the *AskForNumber* label two lines up so that the directives *.entrypoint* and *.locals* separated the label and the first instruction, the label would still mark the first instruction.

An important note before we examine the instructions: IL is strictly a stack-based language. Every instruction takes something (or nothing) from the top of the stack and puts something (or nothing) onto the stack. Some instructions have parameters and some don't, but the general rule does not change: instructions take all required arguments (if any) from the stack and put the results (if any) onto the stack. No IL instruction can address a local variable or a method parameter directly, except the instructions of *load* and *store* groups, which, respectively, put the value or the address of a variable or a parameter onto the stack or take the value from the stack and put it into a variable or a parameter.

Elements of the IL stack are not bytes or words, but slots. When we talk about IL stack depth, we are talking in terms of items put onto the stack, with no regard for the size of each item. Each slot of the IL stack carries information about the type of its current "occupant." And if you put an *int32* item on the stack and then invoke an instruction, which expects, for instance, a string, the JIT compiler becomes very unhappy and very outspoken, throwing an *Unexpected Type* exception and aborting the compilation.

`ldstr "Enter a number"` is an instruction that loads the specified string constant onto the stack. The string constant in this case is stored in the metadata. We can refer to such strings as *common language runtime string constants* or *metadata string constants*. You can store and handle the string constants in another way, as explained in a few moments, but *ldstr* deals exclusively with common language runtime string constants, which are always stored in Unicode format.

`call void [mscorlib]System.Console::WriteLine(string)`is an instruction that calls a console output method from the .NET Framework class library. The string is taken from the stack as the method argument, and nothing is put back, because the method returns *void*.

The parameter of this instruction is a metadata item named *Member Reference* (*MemberRef*). It refers to the static method named *WriteLine*, which has signature *void(string)*; the method is a member of class *System.Console*, declared in the external assembly *mscorlib*. The *MemberRef*s are members of *TypeRef*s—discussed earlier in this chapter in the section "Class Declaration"—just as *FieldDef*s and *MethodDef*s are *TypeDef* members. However, there are no separate *FieldRef*s and *MethodRef*s, the *MemberRef*s cover references to both fields and methods.

You can distinguish field references from method references by their signatures. *MemberRef*s for fields and for methods have different calling conventions and different signature structures. Signatures, including those of *MemberRef*s, are discussed in detail in Chapter 7.

How does the ILAsm compiler know what type of signature should be generated for a *MemberRef*? Mostly from the context. For example, if a *MemberRef* is the parameter of a *call* instruction, it must be a *MemberRef* for a method. In certain cases in which the context is not clear, the compiler requires explicit specifications, such as *method void Odd.or.Even::check()* or *field int32 Odd.or.Even::val*.

```
call string [mscorlib]System.Console::ReadLine()
ldsflda valuetype CharArray8 Format
ldsflda int32 Odd.or.Even::val
call vararg int32 sscanf(string,int8*,...,int32*)
```

call string [mscorlib]System.Console::ReadLine() is an instruction that calls a console input method from the .NET Framework class library. Nothing is taken from the stack, and a string is put onto the stack as a result of this call.

ldsflda valuetype CharArray8 Format is an instruction that loads the address of the static field *Format* of type *valuetype CharArray8*. (Both the field and the value type are declared later in the source code and are discussed later.) IL has separate instructions for loading instance and static fields (*ldfld* and *ldsfld*) or their addresses (*ldflda* and *ldsflda*). Also note that the "address" loaded onto the stack is not exactly an address (or a C/C++ pointer), but rather a reference to the item (a field in this sample).

As you probably guessed, *valuetype CharArray8 Format* is another *MemberRef*, to the field *Format* of type *valuetype CharArray8*. Because this *MemberRef* is not attributed to any *TypeRef*, it must be a global item. (The following section discusses declaration of global items.) In addition, this *MemberRef* is not attributed to any external resolution scope, such as *[mscorlib]*. Hence, it must be a global item defined somewhere in the current module.

ldsflda int32 Odd.or.Even::val is an instruction that loads the address of the static field *val*, member of the class *Odd.or.Even*, of type *int32*. But because the method we're discussing is also a member of *Odd.or.Even*, why do we need to specify the full class name when referring to a member of the same class? Such are the rules of ILAsm: all references must be fully qualified. It might look a bit cumbersome, compared to most high-level languages, but it has its advantages. You don't need to keep track of the context, and all references to the same item look the same throughout the source code.

Because both class *Odd.or.Even* and its field *val* have been declared by the time the field is referenced, the ILAsm compiler will not generate a *MemberRef* item but instead will use a *FieldDef* item.

call vararg int32 sscanf(**string,int8***,...,**int32***) is an instruction that calls the global static method *sscanf*. This method takes three items currently on the stack (the string returned from *System.Console::ReadLine*, the reference to the global field *Format*, and the reference to the field *Odd.or.Even::val*) and puts the result of type *int32* onto the stack.

This method call has two major peculiarities. First, it is a call to an unmanaged method from the C runtime library. I'll defer explanation of this issue until we discuss the declaration of this method. (I have a formal excuse for that because, after all, at the call site managed and unmanaged methods look the same.)

The second peculiarity of this method is its calling convention, *vararg*, which means that this method has a variable argument list. *Vararg* methods have some (or no) mandatory parameters, followed by an unspecified number of optional parameters of unspecified types—unspecified, that is, at the moment of the method declaration. When the method is invoked, all the mandatory parameters (if any) plus all the optional parameters used in this invocation (if any) should be explicitly specified.

Let's take a closer look at the list of arguments in this call. The ellipsis refers to a pseudoargument of a special kind, known as a *sentinel.* A sentinel's role can be formulated as "separating the mandatory arguments from the optional ones," but I think it would be less ambiguous to say that a sentinel immediately precedes the optional arguments and that it is a prefix of the optional part of a *vararg* signature.

What is the difference? An ironclad common language runtime rule concerning the *vararg* method signatures dictates that a sentinel cannot be used when no optional arguments are specified. Thus a sentinel can never appear in *MethodDef* signatures—only mandatory parameters are specified when a method is declared—and it should not appear in call site signatures when only mandatory arguments are supplied. Signatures containing a trailing sentinel are illegal. That's why I think it is important to look at a sentinel as the beginning of optional arguments and not as a separator between mandatory and optional arguments or (heaven forbid!) as the end of mandatory arguments.

For those less familiar with C runtime functions, I should note that the function *sscanf* parses and converts the buffer string (first argument) according to the format string (second argument), puts the results in the rest of the pointer arguments, and returns the number of successfully converted items. In our sample, only one item will be converted, so *sscanf* will return 1 on success or 0 on failure.

```
stloc Retval
ldloc Retval
brfalse Error
```

stloc Retval is an instruction that takes the result of the call to *sscanf* from the stack and stores it in the local variable *Retval.* We need to save this value in a local variable because we will need it later.

ldloc Retval copies the value of *Retval* back onto the stack. We need to check this value, which was taken off the stack by the *stloc* instruction.

brfalse Error takes an item from the stack and, if it is 0, branches (switches the computation flow) to the label *Error.*

```
ldsfld int32 Odd.or.Even::val
ldc.i4 1
and
brfalse ItsEven
ldstr "odd!"
br PrintAndReturn
```

`ldsfld int32` `Odd.or.Even::val` is an instruction that loads the value of the static field *Odd.or.Even::val* onto the stack. If the code has proceeded this far, the string-to-integer conversion must have been successful, and the value that resulted from this conversion must be sitting in the field *val*. The last time we addressed this field, we used the instruction *ldsflda* to load the field address onto the stack. This time we need the value, so we use *ldsfld*.

`ldc.i4` `1` is an instruction that loads the constant *1* of type *int32* onto the stack.

`and` takes two items from the stack—the value of the field *val* and the integer constant *1*—performs a bitwise AND operation, and puts the result onto the stack. Performing the bitwise AND operation with 1 zeroes all the bits of the value of *val* except the least-significant bit.

`brfalse` `ItsEven` takes an item from the stack (the result of the bitwise AND operation) and, if it is 0, branches to the label *ItsEven*. The result of the previous instruction is 0 if the value of *val* is even, and 1 if the value is odd.

`ldstr` `"odd!"` is an instruction that loads the string *odd!* onto the stack.

`br` `PrintAndReturn` is an instruction that does not touch the stack and branches unconditionally to the label *PrintAndReturn*.

The rest of the code in the *Odd.or.Even::check* method should be clear. This section has covered all the instructions used in this method except *ret*, which is fairly obvious: it returns whatever is on the stack. If the method's return type does not match the type of the item on the stack, the JIT compiler will disapprove, throw an exception, and abort the compilation. It will do the same if the stack contains more than one item by the time *ret* is reached or if the method is supposed to return *void* (that is, not return anything) and the stack still contains an item.

Global Items

```
{
⋮
} // End of namespace
.field public static valuetype CharArray8 Format at FormatData
```

`.field public static valuetype` `CharArray8 Format` **`at`** `FormatData` declares a static field named *Format* of type *valuetype CharArray8*. As you might remember, we used a reference to this field in the method *Odd.or.Even::check*.

This field differs from, for example, the field *Odd.or.Even::val* because it is declared outside any class scope and hence does not belong to any class in particular. It is thus a global item. Global items belong to the module containing their declarations. As you've learned, a module is a managed executable file

(EXE or DLL); one or more modules constitute an assembly, which is the primary building block of a managed .NET application; and each assembly has one prime module, which carries the assembly identification information in its metadata.

Actually, a little trick is connected with the concept of global items not belonging to any class. In fact, the metadata of every module contains one special *TypeDef* named *<Module>*, which represents...any guesses? Yes, you are absolutely right.

This *TypeDef* is always present in the metadata, and it always holds the honorable first position in the *TypeDef* table. However, *<Module>* is not a proper *TypeDef*, because its attributes are very limited compared to "normal" *TypeDef*s (classes, value types, and so on). Sounds almost like real life—the more honorable the position you hold, the more limited are your options.

<Module> cannot be private. *<Module>* can have only static members, which means that all global fields and methods must be static. In addition, *<Module>* cannot have events or properties because events and properties cannot be static. (Consult Chapter 12, "Events and Properties," for details.) The reason for this limitation is obvious: given that an assembly always contains exactly one instance of every module, the concept of instantiation becomes meaningless.

The accessibility of global fields and methods differs from the accessibility of member fields and methods belonging to a "normal" class. Even public global items cannot be accessed from outside the assembly. *<Module>* does not extend anything—that is, it has no base class—and no class can inherit from *<Module>*. However, all the classes declared within a module have full access to the global items of this module, including the private ones.

This last feature is similar to class nesting and is quite different from class inheritance. (Derived classes don't have access to the private items of their base classes.) A *nested class* is a class declared within the scope of another class. That other class is usually referred to as an *enclosing class*, or an *encloser*. A nested class is not a member class or an inner class, in the sense that it has no implicit access to the encloser's instance reference (*this*). A nested class is connected to its encloser by three facts only: it is declared within the encloser's lexical scope; its visibility is "filtered" by the encloser's visibility (that is, if the encloser is private, the nested class will not be visible outside the assembly, regardless of its own visibility); and it has access to all of the encloser's members.

Because all the classes declared within a module are by definition declared within the lexical scope of the module, it is only logical that the relationship between the module and the classes declared in it is that of an encloser and nested classes.

As a result, global item accessibilities *public*, *assembly*, and *famorassem* all amount to *assembly*; *private*, *family*, and *famandassem* amount to *private*; and *privatescope* is—well, *privatescope*. The metadata validity rules explicitly state that only three accessibilities are permitted for the global fields and methods: *public* (which is actually *assembly*), *private*, and *privatescope*. The loader, however, is more serene about the accessibility flags of the global items: it allows any accessibility flags to be set, interpreting them as just described (as *assembly*, *private*, or *privatescope*).

Mapped Fields

```
.field public static valuetype CharArray8 Format at FormatData
```

The declaration of the field *Format* contains one more new item, the clause *at FormatData*. This clause indicates that the *Format* field is located in the data section of the module and that its location is identified by the data label *FormatData*. (Data declaration and labeling are discussed in the following section.)

This technique of mapping fields to data is widely used by the compilers for field initialization. It does have some limitations, however. First, mapped fields must be static. This is logical. After all, the mapping itself is static, as it is done at compile time. And even if you manage to map an instance field, all the different instances of this field will be physically mapped to the same memory, which means that you'll wind up with a static field anyway. Because the loader, encountering a mapped instance field, decides in favor of "instanceness" and completely ignores the field mapping, the mapped instance fields are laid out just like all other instance fields.

Second, the mapped fields belong in the data section and hence are unreachable for the garbage collection subsystem of the common language runtime, which provides automatic disposal of unused objects. For this reason, mapped fields cannot be of a type that is subject to garbage collection (such as *class* or *array*). Value types are permitted as types of the mapped fields, as long as these value types have no members of types that are subject to garbage collection. If this rule is violated, the loader throws a *Type Load* exception and aborts loading the module.

Third, mapping a field to a predefined memory location leaves this field wide open to access and manipulation. This is perfectly fine from the point of view of security as long as the field does not have an internal structure whose parts are not intended for public access. That's why the type of a mapped field cannot be any value type that has nonpublic member fields. The loader enforces this rule very strictly and checks for nonpublic fields all the way down. For example, if the type of a mapped field is value type A, the loader will check whether its fields are all public. If among these fields is one field of value type B, the loader will check whether value type B's fields are also all public. If among these fields are two fields of value types C and D—well, you get the picture. If the loader finds a nonpublic field at any level in the type of a mapped field, it throws a *Type Load* exception and aborts the loading.

Data Declaration

```
.field public static valuetype CharArray8 Format at FormatData
.data FormatData = bytearray(25 64 00 00 00 00 00 00)
```

.data FormatData = **bytearray**(25 64 00 00 00 00 00 00) defines a data segment labeled *FormatData*. This segment is 8 bytes long, has ASCII codes of characters *%* (0x25) and *d* (0x64) in the first 2 bytes and 0s in the remaining 6 bytes.

The segment is described as *bytearray*, which is the most ubiquitous way to describe data in ILAsm. The numbers within the parentheses represent the hexadecimal values of the bytes, without the 0x prefix. The byte values should be space-separated, and I recommend that you always use the two-digit form, even if one digit would suffice (as in the case of 0, for example).

It is fairly obvious that you can represent literally any data as a *bytearray*. For example, instead of using the quoted string in the instruction *ldstr "odd!"*, you could use a *bytearray* presentation of the string:

```
ldstr bytearray(6F 00 64 00 64 00 21 00 00 00)
```

The numbers in parentheses represent the Unicode characters *o*, *d*, *d*, *!*, and zero terminator. When you use ILDASM, you can see *bytearrays* everywhere. A *bytearray* is a universal, type-neutral form of data representation, and ILDASM uses it whenever it cannot identify the type associated with the data as one of the elementary types, such as *int32*.

On the other hand, the data *FormatData* could be defined as follows:

```
.data FormatData = int64(0x0000000000006425)
```

This would result in the same data segment size and contents. When you specify a type declaring a data segment (for instance, *int64*), no record concerning this type is entered into metadata or anywhere else. The ILAsm compiler uses the specified type for two purposes only: to identify the size of the data segment being allocated and to identify the byte layout within this segment.

Value Type as Placeholder

```
.field public static valuetype CharArray8 Format at FormatData
.data FormatData = bytearray(25 64 00 00 00 00 00 00)
.class public explicit CharArray8
             extends [mscorlib]System.ValueType { .size 8 }
```

.class public explicit CharArray8 **extends** [mscorlib]System.ValueType { **.size** 8 } declares a value type that has no members but that has an explicitly specified size, 8 bytes. Declaring such a value type is a common way to declare "just a piece of memory." In this case, we don't need to declare any members of this value type because we aren't interested in the internal structure of this piece of memory; we simply want to use it as a type of our global field *Format*, to specify the field's size. In a sense, this value type is nothing but a placeholder.

Could we use an array of 8 bytes instead and save ourselves the declaration of another value type? We could if we did not intend to map the field to the data. Because arrays are subject to garbage collection, they are not allowed as types of mapped fields.

Using value types as placeholders is popular with managed C/C++ compilers because of the need to store and address numerous ANSI string constants. The Visual C# .NET and Visual Basic .NET compilers, which deal mostly with Unicode strings, are less enthusiastic about this technique because they can directly use the common language runtime string constants, which are stored in metadata in Unicode format.

Calling Unmanaged Code

```
.method public static pinvokeimpl("msvcrt.dll" cdecl)
    vararg int32 sscanf(string,int8*) cil managed { }
```

The line **.method public static pinvokeimpl**("msvcrt.dll" *cdecl*) **vararg int32** sscanf(**string**,**int8***) **cil managed** { } declares an unmanaged method, to be called from managed code. The attribute *pinvokeimpl("msvcrt.dll" cdecl)* indicates that this is an unmanaged method, called using the mechanism known as *platform invocation*, or *P/Invoke*. This attribute also indicates that this method resides in the unmanaged DLL Msvcrt.dll and has the calling convention *cdecl*. This calling convention means that the unmanaged method handles the arguments the same way an ANSI C function does.

The method takes two mandatory parameters of types *string* and *int8** (the equivalent of C/C++ *char**) and returns *int32*. Being a *vararg* method, *sscanf* can take any number of optional parameters of any type, but, as you know already, neither the optional parameters nor a sentinel is specified when a *vararg* method is declared.

Platform invocation is the mechanism the common language runtime provides to facilitate the calls from the managed code to unmanaged functions. Behind the scenes, the runtime constructs the so-called *stub*, or *thunk*, which allows addressing of the unmanaged function and conversion of managed argument types to appropriate unmanaged types and back. This conversion is known as *parameter marshaling.*

What is being declared here is not an actual unmanaged method to be called, but a stub generated by runtime, as it is seen from the managed code. Hence the implementation flags *cil managed.* Specifying the method signature as *int32(string, int8*),* we specify the "managed side" of parameter marshaling. The unmanaged side of the parameter marshaling is defined by the actual signature of the unmanaged method being invoked.

The actual signature of the unmanaged function *sscanf* in C is *int sscanf(const char*, const char*, ...).* So the first parameter is marshaled from managed type *string* to unmanaged type *char*.* Recall that when we declared the class *Odd.or.Even,* we specified the *ansi* flag, which means that the managed strings by default are marshaled as ANSI C strings, that is, *char*.* And because the call to *sscanf* is made from a member method of class *Odd.or.Even,* we don't need to provide special information about marshaling the managed strings.

Because the second parameter of the *sscanf* declaration is *int8*,* which is a direct equivalent of *char*,* little marshaling is required. (ILAsm has type *char* as well, but it indicates a Unicode character rather than ANSI, equivalent to "unsigned short" in C, so we cannot use this type here.)

The optional parameters of the original (unmanaged) *sscanf* are supposed to be the pointers to items (variables) we want to fill while parsing the buffer string. The number and base types of these pointers are defined according to the format specification string (the second argument of *sscanf*). In this case, given the format specification string *"%d", sscanf* will expect a single optional argument of type *int*.* When we call the managed thunk of *sscanf,* we provide the optional argument of type *int32*,* which might require marshaling to a native integer pointer only if we are dealing with a platform other than a 32-bit Intel platform (for example, an Alpha or Intel 64-bit platform).

The *P/Invoke* mechanism is very useful because it gives you full access to rich and numerous native libraries and platform APIs. But don't overestimate the ubiquity of *P/Invoke.* Different platforms tend to have different APIs, so overtaxing *P/Invoke* can easily limit the portability of your applications. It's better to stick with .NET Framework class library and take some consolation in the thought that by now you can make a fair guess about what lies at the bottom of this library.

Now that we've finished analyzing the source code, find the sample file Simple.il on the companion CD, copy it into your working directory, compile it using the console command *ilasm simple* (assuming that you have installed .NET Framework and the Platform SDK), and try to run the resulting Simple.exe.

Forward Declaration of Classes

You can carry out a little experiment with the sample code. Open the source file Simple.il in any text editor and modify it by moving the declaration of the value type *CharArray8* in front of the declaration of the field *Format*:

```
{
    ⋮
} // End of namespace
.class public explicit CharArray8
              extends [mscorlib]System.ValueType { .size 8 }
.field public static valuetype CharArray8 Format at FormatData
```

Everything seems to be in order. But when you try to recompile the file, ILAsm compilation fails with the error message *Unresolved MemberRef 'Format'*.

Now modify the source file again, this time moving the declaration of value type *CharArray8* before the declaration of the namespace *Odd.or*:

```
.class public explicit CharArray8
              extends [mscorlib]System.ValueType { .size 8 }
.namespace Odd.or {
    .class public auto ansi Even extends [mscorlib]System.Object {
        .field public static int32 val
        .method public static void check( ) cil managed {
            ⋮
            ldsflda valuetype CharArray8 Format
            ⋮
        } // End of method
    } // End of class
} // End of namespace
.field public static valuetype CharArray8 Format at FormatData
```

Now when you save the source code and try to recompile it, everything is back to normal. What's going on here?

After the first change, when the field *Format* was being referenced in the *ldsflda* instruction in the method *check*, the value type *CharArray8* had not been declared yet, so the respective *TypeRef* was emitted for it, and the signature of the field reference received the *TypeRef* as its type.

Then the value type *CharArray8* was declared, and a new *TypeDef* was created. After that, when the field *Format* was actually declared, its type was recognized as a locally declared value type, and the signature of the field definition received the *TypeDef* as its type. But, no field named *Format* with a *TypeRef* as its type was declared anywhere in this module. Hence the reference-to-definition resolution failure.

(This is an inviting moment to criticize the ILAsm compiler's lack of ability to match the signatures on a pragmatic level, with type analysis and matching the *TypeRefs* to *TypeDefs* by full name and resolution scope. Have patience, however.)

After the second change in the source code, the value type *CharArray8* was declared first so that all references to it, no matter where they happen, refer to it as *TypeDef*. A rather obvious solution.

The solution becomes not so obvious when we consider two classes, members of which use each other's class as type. Which class to declare first? Actually, both of them.

The discussion of class declaration mentioned the class amendment technique, based on the fact that ILAsm allows you to reopen a class scope to declare more class attributes and members. The general solution to the declaration/reference problem is to specify the empty-scope class definitions for all classes first. Following that, you can specify all the classes in full, with their attributes and members, as amendments. The "first wave" of class declarations should carry all class flags, *extends* clauses, and *implements* clauses and should include all nested classes (also with empty scopes). All the member declarations should be left for later.

This technique of forward declaration of classes guards against declaration/reference errors and, as a side effect, reduces the metadata size because it is unnecessary to emit redundant *TypeRefs* for locally defined classes.

(And the answer to the aforementioned criticism of the ILAsm compiler is that the compiler does signature matching in the fastest possible way, without needing more sophisticated and slower methods, as long as you use the class forward declaration. It is possible, however, that the need for the class forward declaration might be eliminated in future versions of the ILAsm compiler.)

Summary

We have touched briefly on the most important features of the common language runtime and ILAsm. You now know (in general terms) how the runtime functions, how a program in ILAsm is written, and how to define the basic components (classes, fields, and methods). You have learned that the managed code can interoperate with the unmanaged (native) code, and what the common language runtime is doing to facilitate this interoperation.

In the next chapter, we shall continue working with our simple sample to learn some more sophisticated features of the runtime and ILAsm.

2

Enhancing the Code

Let's continue tweaking our simple sample; maybe we can make it better. There are two aspects of "better" I would like to discuss in this chapter: first, reducing code size and, second, protecting our code from unpleasant surprises. Let's start with the code size.

Code Retention

The sample code presented in the previous chapter is tight. If you don't believe me, carry out a simple experiment: write a similar application in your favorite high-level Microsoft .NET language, compile it to an executable—and make sure it runs!—disassemble the executable, and compare the result to the sample offered here. Now let's try to make the code tighter yet.

First, given what you now know about field mapping and value types as placeholders, we don't need to continue employing this technique. If *sscanf* accepts *string* as the first argument, it can just as well accept *string* as the second argument too. Second, we can use (and discuss) certain "shortcuts" in the IL instruction set.

Let's have a look at our simple sample with slight modifications (source file Simple1.il). The portions of interest are marked with the comment *CHANGE!*.

```
//----------- Program header
.assembly extern mscorlib { }
.assembly OddOrEven  { }
.module OddOrEven.exe
//----------- Class declaration
.namespace Odd.or {
    .class public auto ansi Even extends [mscorlib]System.Object {
//----------- Field declaration
        .field public static int32 val
//----------- Method declaration
        .method public static void check( ) cil managed {
            .entrypoint
            .locals init (int32 Retval)
        AskForNumber:
            ldstr "Enter a number"
            call void [mscorlib]System.Console::WriteLine(string)
            call string [mscorlib]System.Console::ReadLine()
            ldstr "%d" // CHANGE!
            ldsflda int32 Odd.or.Even::val
            call vararg int32 sscanf(string,string,...,int32*) //
 CHANGE!
            stloc.0 // CHANGE!
            ldloc.0 // CHANGE!
            brfalse.s Error  // CHANGE!
            ldsfld int32 Odd.or.Even::val
            ldc.i4.1  // CHANGE!
            and
            brfalse.s ItsEven  // CHANGE!
            ldstr "odd!"
            br.s PrintAndReturn  // CHANGE!
        ItsEven:
            ldstr "even!"
            br.s PrintAndReturn  // CHANGE!
        Error:
            ldstr "How rude!"
        PrintAndReturn:
            call void [mscorlib]System.Console::WriteLine(string)
            ldloc.0  // CHANGE!
            brtrue.s AskForNumber  // CHANGE!
            ret
        } // End of method
    } // End of class
} // End of namespace
//----------- Calling unmanaged code
.method public static pinvokeimpl("msvcrt.dll" cdecl)
    vararg int32 sscanf(string,string) cil managed { }
```

The program header, class declaration, field declaration, and method header look exactly the same. The first change comes within the method body, where the loading of the address of the global field *Format* is replaced with the loading of a metadata string constant, *ldstr "%d"*. As noted earlier, we can abandon defining and using an ANSI string constant as the second argument of the call to *sscanf* in favor of using a metadata string constant (internally represented in Unicode), relying on the marshaling mechanism provided by *P/Invoke* to do the necessary conversion work.

Because we are no longer using an ANSI string constant, the declarations of the global field *Format*, the placeholder value type used as the type of this field, and the data to which the field was mapped are omitted. As you've undoubtedly noticed, there is no need to explicitly declare a metadata string constant in IL assembly language (ILAsm)—the mere mention of such a constant in the source code is enough for the ILAsm compiler to automatically emit this metadata item.

Having thus changed the nature of the second argument of our call to *sscanf*, we need to modify the signature of the *sscanf P/Invoke* thunk so that necessary marshaling can be provided. Hence the changes in the signature of *sscanf*, both in the method declaration and at the call site.

Another set of changes results from replacing the local variable loading/storing instructions *ldloc Retval* and *stloc Retval* with the instructions *ldloc.0* and *stloc.0*, respectively. IL defines special operation codes for loading/storing the first four local variables on the list, numbered 0 to 3. We gain here because while the canonic form of the instruction (*ldloc Retval*) compiles into the operation code (*ldloc*) followed by an unsigned integer indexing the local variable (in this case 0), the instructions *ldloc*.n compile into single operation codes.

You might also notice that all branching instructions (*br*, *brfalse*, *brtrue*) in the method *check* are replaced with the short forms of these instructions (*br.s*, *brfalse.s*, *brtrue.s*). A standard (long) form of an instruction compiles into an operation code followed by a 4-byte parameter (in the case of branching instructions, offset from the current position), whereas a short form compiles into an operation code followed by a 1-byte parameter. This limits the range of branching to maximums of 128 bytes backward and 127 bytes forward from the current point in the IL stream, but in this case we can safely afford to switch to short forms because our method is rather small.

Short forms that take an integer or unsigned integer parameter are defined for all types of IL instructions. So even if we declare more than four local variables, we still could save a few bytes by using the instructions *ldloc.s* and *stloc.s* instead of *ldloc* and *stloc*, as long as the index of a local variable does not exceed 255.

The high-level language compilers, emitting the IL code, automatically estimate the ranges and choose whether a long form or a short form of the instruction should be used in each particular case. The ILAsm compiler, of course, does nothing of the sort. If you specify a long or short instruction, the compiler takes it at face value—you are the boss, and you are supposed to know better. But if you specify a short branching instruction and place the target label out of range, the ILAsm compiler will diagnose an error.

Once, a colleague of mine came to me complaining that the ILAsm compiler obviously could not compile the code the IL Disassembler (ILDASM) produced. The disassembler and the compiler are supposed to work in absolute concert, so I was quite startled by this discovery. A short investigation uncovered the grim truth. In an effort to work out a special method for automatic test program generation, my colleague was compiling the initial programs written in Visual C# .NET and Microsoft Visual Basic .NET, disassembling the resulting executables, inserting test-specific ILAsm segments, and reassembling the modified code into new executables. The methods in the initial executables, produced by Visual C# .NET and Visual Basic .NET compilers, were rather small, so the compilers were emitting the short branching instructions, which, of course, were shown in the disassembly as they were. And every time my colleague's automatic utility inserted enough additional ILAsm code between a short branching instruction and its destination, the branching instruction, figuratively speaking, kissed its target label good-bye.

One more change to note in the sample: the instruction *ldc.i4 1* was replaced with *ldc.i4.1*. The logic here is the same as in the case of replacing *ldloc Retval* with *ldloc.0*: using a shortcut operation code to get rid of a 4-byte integer parameter. The shortcuts *ldc.i4.n* exist for *n* from 0 to 8, and (–1) can be loaded using the operation code *ldc.i4.m1*. The short form of the *ldc.i4* instruction—*ldc.i4.s*—works for the integers in the byte range (from –128 to 127).

Now copy the source file Simple1.il from the companion CD, compile it with the console command *ilasm simple1* into an executable (Simple1.exe), and ensure that it runs exactly as Simple.exe does. Then disassemble both executables side by side, using console commands *ildasm simple.exe /bytes* and *ildasm simple1.exe /bytes*. (The */bytes* option makes the disassembler show the actual byte values constituting the IL flow.) Find the *check* methods in the tree views of both instances of ILDASM, and double-click them to open disassembly windows, in which you can compare the two implementations of the same method to see whether the code retention worked.

Protecting the Code

Thus far, we could have been quite confident that nothing bad would happen when we called the unmanaged function *sscanf* from the managed code, so we simply called it. But who knows what terrible dangers lurk in the deep shadows of unmanaged code? *I* don't. So we'd better take steps to make sure that our application behaves in an orderly manner. For this purpose, we can employ the mechanism of structured exception handling, well known to C++ and Visual C# .NET programmers.

Examine the following light modifications of the sample (source file Simple2.il). As before, the modifications are marked with the comment *CHANGE!*.

```
//----------- Program header
.assembly extern mscorlib { }
.assembly OddOrEven  { }
.module OddOrEven.exe
//----------- Class declaration
.namespace Odd.or {
    .class public auto ansi Even extends [mscorlib]System.Object {
//----------- Field declaration
        .field public static int32 val
//----------- Method declaration
        .method public static void check( ) cil managed {
            .entrypoint
            .locals init (int32 Retval)
        AskForNumber:
            ldstr "Enter a number"
            call void [mscorlib]System.Console::WriteLine(string)
            .try {  // CHANGE!
                // Guarded block begins
            call string [mscorlib]System.Console::ReadLine()
                // pop  // CHANGE!
                // ldnull // CHANGE!
                ldstr "%d"
                ldsflda int32 Odd.or.Even::val
                call vararg int32 sscanf(string,string,...,int32*)
                stloc.0
                leave.s DidntBlowUp  // CHANGE!
                // Guarded block ends
            } // CHANGE!
            // CHANGE! --->
            catch [mscorlib]System.Exception
            { // Exception handler begins
                pop
                ldstr "KABOOM!"
                call void [mscorlib]System.Console::WriteLine(string)
                leave.s Return
            } // Exception handler ends
        DidntBlowUp:
        // <--- CHANGE!
            ldloc.0
            brfalse.s Error
```

(continued)

```
            ldsfld int32 Odd.or.Even::val
            ldc.i4.1
            and
            brfalse.s ItsEven
            ldstr "odd!"
            br.s PrintAndReturn
        ItsEven:
            ldstr "even!"
            br.s PrintAndReturn
        Error:
            ldstr "How rude!"
        PrintAndReturn:
            call void [mscorlib]System.Console::WriteLine(string)
            ldloc.0
            brtrue.s AskForNumber
        Return:  // CHANGE!
            ret
        } // End of method
    } // End of class
} // End of namespace
//---------- Calling unmanaged code
.method public static pinvokeimpl("msvcrt.dll" cdecl)
    vararg int32 sscanf(string,string) cil managed { }
```

What are these changes? One involves enclosing the "dangerous" part of the code in the scope of the so-called try block (or guarded block), which prompts the runtime to watch for exceptions thrown while executing this code segment. The exceptions are thrown if anything out of order happens—for example, a memory access violation or a reference to an undefined class or method.

```
.try {
    // Guarded block begins
    call string [mscorlib]System.Console::ReadLine()
    ldstr "%d"
    ldsflda int32 Odd.or.Even::val
    call vararg int32 sscanf(string,string,...,int32*)
    stloc.0
    leave.s DidntBlowUp
    // Guarded block ends
}
```

Note that the try block ends with the instruction *leave.s DidntBlowUp*. This instruction—*leave.s* being a short form of *leave*—switches the computation flow to the location marked with the label *DidntBlowUp*. We cannot use a branching instruction here because, according to the rules of the common language runtime exception handling mechanism, strictly enforced by the JIT compiler, the only legal way out of a try block is via a *leave* instruction.

This limitation is caused by an important function performed by the *leave* instruction: before switching the computation flow, it unwinds the stack (strips off all the items currently on the stack) and, if these items are references to object instances, disposes of them. That is why we need to store the value returned by the *sscanf* function in the local variable *Retval* before using the *leave* instruction; if we tried to do it later, the value would be lost.

catch [mscorlib]System.Exception indicates that we plan to intercept any exception thrown within the protected segment and handle this exception:

```
{
    ⋮
    leave.s DidntBlowUp
    // Guarded block ends
}
catch [mscorlib]System.Exception
{ // Exception handler begins
    pop
    ⋮
}
```

Because we are intercepting *any* exception, we specified a generic managed exception type (*[mscorlib]System.Exception*), a type from which all managed exception types are derived. Technically, we could call *[mscorlib]System.Exception* the "mother of all exceptions," but the proper term is somehow less colloquial: the "inheritance root of all exceptions."

Mentioning another, more specific, type of exception in the *catch* clause—that is, *[mscorlib]System.NullReferenceException*—would indicate that we are prepared to handle only this particular type of exception and that exceptions of other types should be handled elsewhere. This approach is convenient if you want to have different handlers for different types of exceptions, and it's the reason this mechanism is referred to as *structured* exception handling.

Immediately following the *catch* clause is the exception handler scope (the handler block):

```
catch [mscorlib]System.Exception
{ // Exception handler begins
    pop
    ldstr "KABOOM!"
    call void [mscorlib]System.Console::WriteLine(string)
    leave.s Return
} // Exception handler ends
```

When an exception is intercepted and the handler block is entered, the only thing present on the stack is always the reference to the intercepted exception—an instance of the exception type. In implementing the handler, we don't want to take pains analyzing the caught exception, so we can simply get rid of it using the instruction *pop*. In this simple application, it's enough to know that an exception has occurred, without reviewing the details.

Then we load the string constant *"KABOOM!"* onto the stack, print this string by using the console output method *[mscorlib]System.Console::WriteLine(string)*, and switch to the label *Return* by using the instruction *leave.s*. The rule "leave only by *leave*" applies to the handler blocks as well as to the try blocks. We could not simply load the string *"KABOOM!"* onto the stack and leave to *PrintAndReturn*; the *leave.s* instruction would remove this string from the stack, leaving nothing with which to call *WriteConsole*.

You might be wondering why, if we are trying to protect the call to the unmanaged function *sscanf*, we included three preceding instructions in the try block? Why not include only the call to *sscanf* in the scope of *.try*?

```
ldstr "Enter a number"
call void [mscorlib]System.Console::WriteLine(string)
.try {
    // Guarded block begins
    call string [mscorlib]System.Console::ReadLine()
    ldstr "%d"
    ldsflda int32 Odd.or.Even::val
    call vararg int32 sscanf(string,string,...,int32*)
    stloc.0
    leave.s DidntBlowUp
    // Guarded block ends
}
```

According to the exception handling rules, a guarded segment (a try block) can begin only when the method stack is empty. The closest such moment before the call to *sscanf* was immediately after the call to *[mscorlib]System.Console::WriteLine(string)*, which took the string *"Enter a number"* from the stack and put nothing back. Because the three instructions immediately preceding the call to *sscanf* are loading the call arguments onto the stack, we must open the guarded segment before any of these instructions are executed.

Perhaps you're puzzled by what seems to be a rather strict limitation. We cannot begin and end a try block anywhere we want, as we can in C++? Well, the truth is that you can do it the same way you do it in C++, but no better.

The high-level language compilers work in such a way that every completed statement in a high-level language is compiled into a sequence of instructions that begins and ends with the stack empty. In C++, our try block would look like this:

```
try {
    Retval = sscanf(System.Console::ReadLine(),
                    "%d", &val);
}
```

This feature of high-level language compilers is so universal that all high-level language decompilers use these empty-stack points within the instruction sequence to identify the beginnings and ends of completed statements.

The last task remaining is to test our protection. Copy the source file Simple2.il from the companion CD into your working directory, and compile it with the console command *ilasm simple2* into the executable Simple2.exe. Test it to ensure that it runs exactly as the previous samples do. Now let's simulate A Horrible Disaster Within Unmanaged Code. Load the source file Simple2.il into any text editor, and uncomment the instructions *pop* and *ldnull* within the try block:

```
.try {
    // Guarded block begins
    call string [mscorlib]System.Console::ReadLine()
    pop
    ldnull
    ldstr "%d"
    ldsflda int32 Odd.or.Even::val
    call vararg int32 sscanf(string,string,...,int32*)
    stloc.0
    leave.s DidntBlowUp  // CHANGE!
    // Guarded block ends
} // CHANGE!
```

The instruction *pop* removes from the stack the string returned by *ReadLine*, and *ldnull* loads a null reference instead. The null reference is marshaled to the unmanaged *sscanf* as a null pointer. *Sscanf* is not prepared to take it and will try to dereference the null pointer. The platform operating system will throw the unmanaged exception *Memory Access Violation*, which is intercepted by the common language runtime and converted to a managed exception of type *System.NullReferenceException*, which in turn is intercepted by our protection. The application will then terminate gracefully.

Recompile Simple2.il and try to run the resulting executable. You will get nothing worse than *KABOOM!* displayed on the console.

You can then modify the source code in Simple.il or Simple1.il, adding the same two instructions *pop* and *ldnull* after the call to *System.Console::ReadLine*. Recompile the source file to see how it runs without structured exception handling protection.

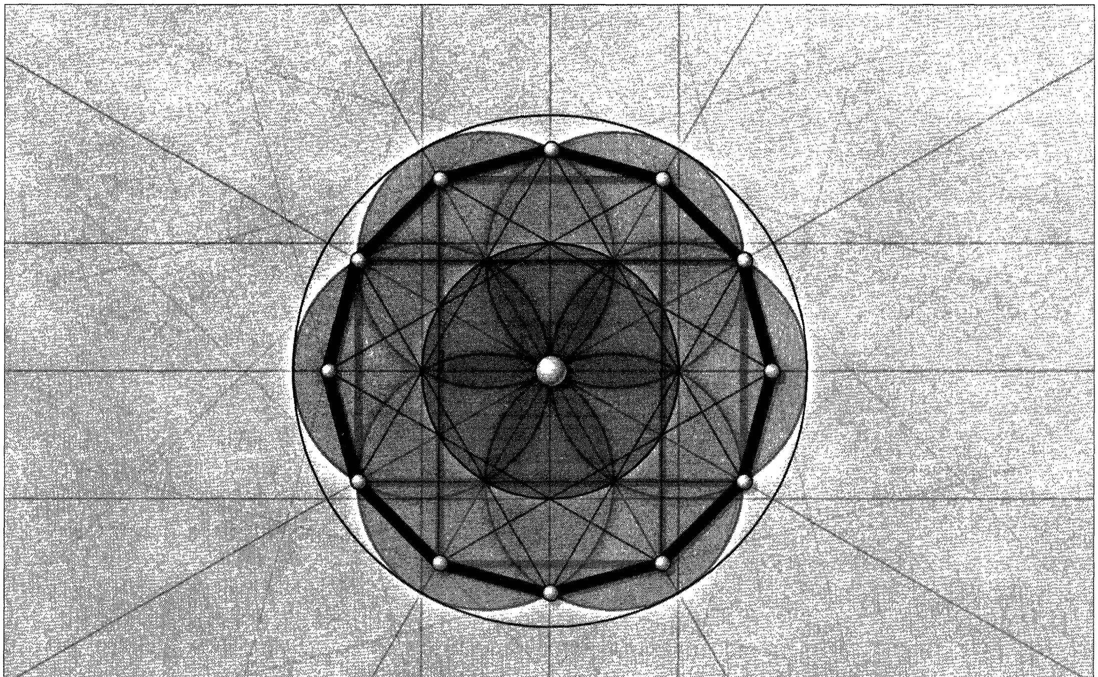
Summary

These first two chapters *did* make for a quick start, didn't they? Well, I promised you a light cavalry raid into hostile territories, and you got just that. By now you should be able to understand in general the text output the IL Disassembler produces. I hope too that you are interested in a more detailed and systematic discussion of what is going on inside the common language runtime and how ILAsm is used to describe it.

From now on, our keywords are "detailed" and "systematic." No more cavalry charges!

Part II

Underlying Structures



3

The Structure of a Managed Executable File

Chapter 1, "Simple Sample," introduced the managed executable file, known as a managed module and executed in the environment of the common language runtime. In this chapter, we'll take a detailed look at the general structure of such a file. The file format of a managed module is based on the standard Microsoft Windows Portable Executable and Common Object File Format (PE/COFF) and is an extension of this format. Thus, formally, any managed module is a proper PE/COFF file, with additional features that identify it as a managed executable file.

Because the file format of a managed module conforms to the Windows PE/COFF standard, the operating system treats the managed module as an executable. And the extended, common language runtime–specific information allows the runtime to immediately seize control over the module execution as soon as the operating system invokes the module. Figure 3-1 shows the structure of a managed PE/COFF file.

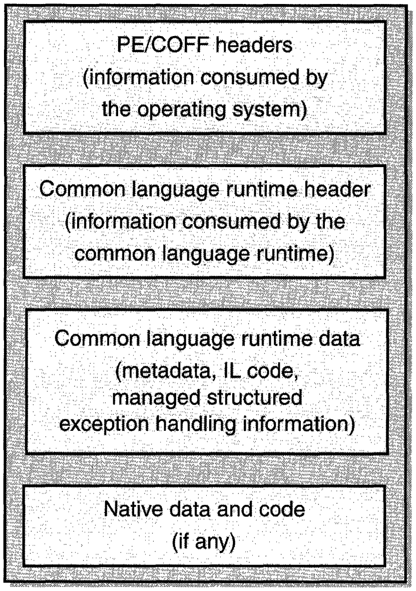


Figure 3-1 The general structure of a managed executable file.

Because IL assembly language (ILAsm) produces PE files only, this chapter concentrates on managed PE files—executables, also known as *image files* because they can be thought of as "memory images"—rather than pure COFF object files. (Actually, only one of the current managed compilers, Microsoft Managed C++ [MC++], produces object files as an intermediate step to PE files.)

This analysis of the managed PE file structure employs the following common definitions:

- **File pointer** The location of an item within the file itself, before it is processed by the loader. This location is a position (an offset) within the file as it is stored on disk.
- **Relative virtual address (RVA)** The address of an item once it has been loaded into memory, with the base address of the image file subtracted from it—in other words, the offset of an item within the image file loaded into memory. The RVA of an item almost always differs from its position within the file on disk (the file pointer).

- **Virtual address (VA)** The same as the RVA except that the base address of the image file is not subtracted. The address is referred to as *virtual* because the operating system creates a distinct virtual address space for each process, independent of physical memory. For almost all purposes, a virtual address should be considered as simply an address. A virtual address is not as predictable as an RVA because the loader might not load the image at its preferred location if a conflict exists with any image file already loaded—a so-called *base address conflict.*

- **Section** The basic unit of code or data within a PE/COFF file. In addition to code and data sections, an image file can contain a number of sections, such as *.tls* (thread local storage) or *.reloc* (relocations), that have special purposes. All the raw data in a section must be loaded contiguously.

> **Note** Throughout this chapter (and indeed throughout the book), I use the term *managed compiler* to mean a compiler that targets the common language runtime and produces managed PE files. The term does not necessarily imply that the compiler itself is a managed application.

PE/COFF Headers

Figure 3-2 illustrates the structure of operating system–specific headers of a PE file. The headers include an MS-DOS stub, the PE signature, the COFF header, the PE header, and section headers. All of these components—and the data directory table in the PE header—are discussed in the following sections.

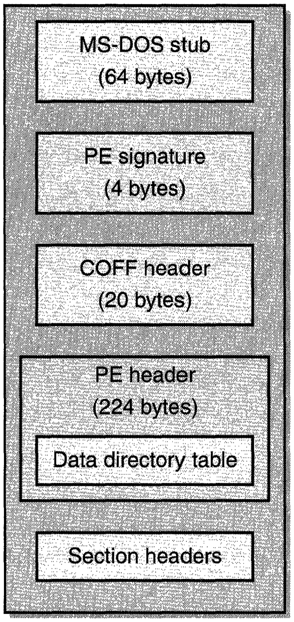


Figure 3-2 The memory layout of operating system–specific headers.

MS-DOS Stub and PE Signature

The MS-DOS stub is present in image files only. Placed at the beginning of an image file, it is a valid application that runs under MS-DOS. (Isn't that exciting!) The default stub prints the message *This program cannot be run in DOS mode* when the image file is run in MS-DOS. This is probably the least interesting part of OS-specific headers; the only relevant fact is that the MS-DOS stub, at offset 0x3C, contains the file pointer to the PE signature, which allows the operating system to properly execute the image file.

The PE signature that follows the MS-DOS stub is a 4-byte item, identifying the file as a PE format image file. The signature contains the characters *P* and *E*, followed by 2 null bytes.

COFF Header

A standard COFF header is located immediately after the PE signature of an image file. The COFF header provides the most general characteristics of a PE/COFF file, applicable to both object and executable files. The structure of the COFF header and the meaning of its fields are shown in Table 3-1.

Table 3-1 The Format of a COFF Header

Offset	Size	Field Name	Description
0	2	*Machine*	Number identifying the type of target machine. (See Table 3-2.) If the managed PE file is intended for various machine types, this field should be set to *IMAGE_FILE_MACHINE_I386* (0x014C).
2	2	*NumberOfSections*	Number of entries in the section table, which immediately follows the headers.
4	4	*TimeDateStamp*	Time and date of file creation.
8	4	*PointerToSymbolTable*	File pointer of the COFF symbol table. Because this table is never used in managed PE files, this field must be set to 0.
12	4	*NumberOfSymbols*	Number of entries in the COFF symbol table. This field must be set to 0 in managed PE files.
16	2	*SizeOfOptionalHeader*	Size of the PE header. This field is specific to PE files; it is set to 0 in COFF files.
18	2	*Characteristics*	Flags indicating the attributes of the file. (See Table 3-3.)

The structure of the standard COFF header is defined in Winnt.h as follows:

```
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD   TimeDateStamp;
    DWORD   PointerToSymbolTable;
    DWORD   NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

The *Machine* types are also defined in Winnt.h, as listed in Table 3-2.

Table 3-2 The *Machine* Field Values

Constant	Value	Description
IMAGE_FILE_MACHINE_UNKNOWN	0	Contents assumed to be applicable to any machine type—for unmanaged PE files only.
IMAGE_FILE_MACHINE_I386	0x014c	Intel 386 or later. For managed PE files, contents are applicable to any machine type.

(continued)

Table 3-2 The *Machine* Field Values *(continued)*

Constant	Value	Description
IMAGE_FILE_MACHINE_R3000	0x0162	MIPS little endian—the least significant byte precedes the most significant byte. 0x0160 big endian—the most significant byte precedes the least significant byte.
IMAGE_FILE_MACHINE_R4000	0x0166	MIPS little endian
IMAGE_FILE_MACHINE_R10000	0x0168	MIPS little endian
IMAGE_FILE_MACHINE_WCEMIPSV2	0x0169	MIPS little endian running Microsoft Windows CE 2
IMAGE_FILE_MACHINE_ALPHA	0x0184	Alpha AXP
IMAGE_FILE_MACHINE_POWERPC	0x01F0	IBM PowerPC little endian
IMAGE_FILE_MACHINE_SH3	0x01a2	SH3 little endian
IMAGE_FILE_MACHINE_SH3E	0x01a4	SH3E little endian
IMAGE_FILE_MACHINE_SH4	0x01a6	SH4 little endian
IMAGE_FILE_MACHINE_ARM	0x01c0	ARM little endian
IMAGE_FILE_MACHINE_THUMB	0x01c2	ARM processor with Thumb decompressor
IMAGE_FILE_MACHINE_IA64	0x0200	Intel IA64
IMAGE_FILE_MACHINE_MIPS16	0x0266	MIPS
IMAGE_FILE_MACHINE_MIPSFPU	0x0366	MIPS with FPU
IMAGE_FILE_MACHINE_MIPSFPU16	0x0466	MIPS16 with FPU
IMAGE_FILE_MACHINE_ALPHA64	0x0284	ALPHA AXP64
IMAGE_FILE_MACHINE_AXP64	0x0284	ALPHA AXP64

Tip As noted in Tables 3-1 and 3-2, the best strategy for a managed PE file is to specify *IMAGE_FILE_MACHINE_I386* in the *Machine* field. Doing so ensures that the PE file will be able to execute on any machine that has the common language runtime installed.

The *Characteristics* field of a COFF header contains flags that indicate attributes of the PE/COFF file. These flags are defined in Winnt.h as shown in Table 3-3. Notice that the table refers to pure-IL managed PE files; the term *pure-IL* indicates that the image file contains no embedded native code.

Table 3-3 The *Characteristics* Field Values

Flag	Value	Description
IMAGE_FILE_RELOCS_STRIPPED	0x0001	Image file only. This flag indicates that the file contains no base relocations and must be loaded at its preferred base address. In the case of base address conflict, the operating system loader reports an error. This flag should not be set for managed PE files.
IMAGE_FILE_EXECUTABLE_IMAGE	0x0002	Flag indicates that the file is an image file (EXE or DLL). This flag should be set for managed PE files. If it is not set, this generally indicates a linker error.
IMAGE_FILE_LINE_NUMS_STRIPPED	0x0004	COFF line numbers have been removed. This flag should be set for managed PE files because they do not use the debug information embedded in the PE file itself. Instead, the debug information is saved in accompanying program database (PDB) files.
IMAGE_FILE_LOCAL_SYMS_STRIPPED	0x0008	COFF symbol table entries for local symbols have been removed. This flag should be set for managed PE files, for the reason given in the preceding entry.
IMAGE_FILE_AGGRESIVE_WS_TRIM	0x0010	Aggressively trim the working set. This flag should not be set for pure-IL managed PE files.

(continued)

Table 3-3 The *Characteristics* Field Values *(continued)*

Flag	Value	Description
IMAGE_FILE_LARGE_ADDRESS_AWARE	0x0020	Application can handle addresses beyond the 2-GB range. This flag should not be set for pure-IL managed PE files.
IMAGE_FILE_BYTES_REVERSED_LO	0x0080	Little endian. This flag should not be set for pure-IL managed PE files.
IMAGE_FILE_32BIT_MACHINE	0x0100	Machine is based on 32-bit architecture. This flag is set by the current versions of code generators producing managed PE files.
IMAGE_FILE_DEBUG_STRIPPED	0x0200	Debug information has been removed from the image file.
IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP	0x0400	If the image file is on removable media, copy and run it from the swap file. This flag should not be set for pure-IL managed PE files.
IMAGE_FILE_NET_RUN_FROM_SWAP	0x0800	If the image file is on a network, copy and run it from the swap file. This flag should not be set for pure-IL managed PE files.
IMAGE_FILE_SYSTEM	0x1000	The image file is a system file (for example, a device driver). This flag should not be set for pure-IL managed PE files.
IMAGE_FILE_DLL	0x2000	The image file is a DLL rather than an EXE. It cannot be directly run.

Table 3-3 The *Characteristics* Field Values *(continued)*

Flag	Value	Description
IMAGE_FILE_UP_SYSTEM_ONLY	0x4000	The image file should be run on a uniprocessor machine only. This flag should not be set for pure-IL managed PE files.
IMAGE_FILE_BYTES_REVERSED_HI	0x8000	Big endian. This flag should not be set for pure-IL managed PE files.

The typical *Characteristics* value produced by existing code generators—the one employed by the MC++ compiler and linker as well as the one used by all the rest of the managed compilers, including ILAsm—for an EXE image file is 0x010E (*IMAGE_FILE_EXECUTABLE_IMAGE* | *IMAGE_FILE_LINE_NUMS_STRIPPED* | *IMAGE_FILE_LOCAL_SYMS_STRIPPED* | *IMAGE_FILE_32BIT_MACHINE*). For a DLL image file, this value is 0x210E (*IMAGE_FILE_EXECUTABLE_IMAGE* | *IMAGE_FILE_LINE_NUMS_STRIPPED* | *IMAGE_FILE_LOCAL_SYMS_STRIPPED* | *IMAGE_FILE_32BIT_MACHINE* | *IMAGE_FILE_DLL*).

PE Header

The PE header, which immediately follows the COFF header, provides the information for the OS loader. Although this header is sometimes referred to as the *optional header*, it is optional only in the sense that object files usually don't contain it. For PE files, this header is mandatory.

The size of the PE header is not fixed. It depends on the number of data directories defined in the header and is specified in the *SizeOfOptionalHeader* field of the COFF header. The structure of the PE header is defined in Winnt.h as follows:

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    // Standard fields
    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;
    // NT additional fields
    DWORD   ImageBase;
```

(continued)

```
        DWORD   SectionAlignment;
        DWORD   FileAlignment;
        WORD    MajorOperatingSystemVersion;
        WORD    MinorOperatingSystemVersion;
        WORD    MajorImageVersion;
        WORD    MinorImageVersion;
        WORD    MajorSubsystemVersion;
        WORD    MinorSubsystemVersion;
        DWORD   Win32VersionValue;
        DWORD   SizeOfImage;
        DWORD   SizeOfHeaders;
        DWORD   CheckSum;
        WORD    Subsystem;
        WORD    DllCharacteristics;
        DWORD   SizeOfStackReserve;
        DWORD   SizeOfStackCommit;
        DWORD   SizeOfHeapReserve;
        DWORD   SizeOfHeapCommit;
        DWORD   LoaderFlags;
        DWORD   NumberOfRvaAndSizes;
        IMAGE_DATA_DIRECTORY
            DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
    } IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

Table 3-4 describes the fields of the PE header.

Table 3-4 PE Header Fields

Offset 32/64	Size 32/64	Field	Description
0	2	*Magic*	"Magic number" identifying the state of the image file. Acceptable values are 0x010B for a 32-bit PE file, 0x020B for a 64-bit PE file, and 0x107 for a ROM image file. Managed PE files must have this field set to 0x010B.
2	1	*MajorLinkerVersion*	Linker major version number. The MC++ compiler and linker set this field to 7; the pure-IL code generator employed by other compilers sets it to 6.
3	1	*MinorLinkerVersion*	Linker minor version number.
4	4	*SizeOfCode*	Size of the code section (*.text*), or the sum of all code sections if multiple code sections exist. The ILAsm compiler always emits the single code section.

Table 3-4 PE Header Fields *(continued)*

Offset 32/64	Size 32/64	Field	Description
8	4	*SizeOfInitializedData*	Size of the initialized data section (held in the field *SizeOfRawData* of the respective section header), or the sum of all such sections. The initialized data is defined as specific values, stored in the disk image file.
12	4	*SizeOfUninitializedData*	Size of the uninitialized data section (*.bss*), or the sum of all such sections. This data is not part of the disk file and does not have specific values, but the OS loader commits space for the data.
16	4	*AddressOfEntryPoint*	RVA of the entry point function. For unmanaged DLLs, this can be 0. For managed PE files, this value always points to the common language runtime invocation stub.
20	4	*BaseOfCode*	RVA of the beginning of the file's code section(s).
24/-	4/-	*BaseOfData*	RVA of the beginning of the file's data section(s).
28/24	4/8	*ImageBase*	Image's preferred starting virtual address. In ILAsm, this field can be specified explicitly by the directive *.imagebase <integer value>* and/or the command-line option */BASE=<integer value>*. The command-line option takes precedence over the directive.
32	4	*SectionAlignment*	Alignment of sections when loaded in memory. This setting must be greater than or equal to the value of the *FileAlignment* field. The default is the memory page size.
36	4	*FileAlignment*	Alignment of sections in the disk image file. The value should be a power of 2, from 512 to 64 K. If *SectionAlignment* is set to less than the memory page size, *FileAlignment* must match *SectionAlignment.* In ILAsm, this field can be specified explicitly by the directive *.file alignment <integer value>* and/or the command-line option */ALIGNMENT=<integer value>*. The command-line option takes precedence over the directive.
40	2	*MajorOperatingSystemVersion*	Major version number of the required OS.
42	2	*MinorOperatingSystemVersion*	Minor version number of the required OS.
44	2	*MajorImageVersion*	Major version number of the application.

(continued)

Table 3-4 PE Header Fields *(continued)*

Offset 32/64	Size 32/64	Field	Description
46	2	*MinorImageVersion*	Minor version number of the application.
48	2	*MajorSubsystemVersion*	Major version number of the subsystem.
50	2	*MinorSubsystemVersion*	Minor version number of the subsystem.
52	4	*Win32VersionValue*	Reserved.
56	4	*SizeOfImage*	Size of the image file (in bytes), including all headers. This field must be set to a multiple of the *SectionAlignment* value.
60	4	*SizeOfHeaders*	Sum of the sizes of the MS-DOS stub, the COFF header, the PE header, and the section headers, rounded up to a multiple of the *FileAlignment* value.
64	4	*CheckSum*	Checksum of the disk image file.
68	2	*Subsystem*	Subsystem required to run this image file. The values are defined in Winnt.h and are as follows: ■ *NATIVE (1)* No subsystem required (for example, a device driver) ■ *WINDOWS_GUI (2)* Runs in the Windows GUI subsystem ■ *WINDOWS_CUI (3)* Runs in Windows console mode ■ *OS2_CUI (5)* Runs in OS/2 1.x console mode ■ *POSIX_CUI (7)* Runs in POSIX console mode ■ *NATIVE_WINDOWS (8)* The image file is a native Win9x driver ■ *WINDOWS_CE_GUI (9)* Runs in the Windows CE GUI subsystem. In ILAsm, this field can be specified explicitly by the directive *.subsystem <integer value>* and/or the command-line option */SUBSYSTEM=<integer value>*. The command-line option takes precedence over the directive.
70	2	*DllCharacteristics*	Obsolete, set to 0.

Table 3-4 PE Header Fields *(continued)*

Offset 32/64	Size 32/64	Field	Description
72	4/8	*SizeOfStackReserve*	Size of virtual memory to reserve for the initial thread's stack. Only the *SizeOfStackCommit* field is committed; the rest is available in one-page increments. The default is 1 MB.
76/80	4/8	*SizeOfStackCommit*	Size of virtual memory initially committed for the initial thread's stack. The default is one page.
80/88	4/8	*SizeOfHeapReserve*	Size of virtual memory to reserve for the initial process heap. Only the *SizeOfHeapCommit* field is committed; the rest is available in one-page increments. The default is 1 MB.
84/96	4/8	*SizeOfHeapCommit*	Size of virtual memory initially committed for the process heap. The default is one page.
88/104	4	*LoaderFlags*	Obsolete, set to 0.
92/108	4	*NumberOfRvaAndSizes*	Number of entries in the *DataDirectory* array; at least 16. Although it is theoretically possible to emit more than 16 data directories, all existing managed compilers emit exactly 16 data directories, with the sixteenth (last) data directory never used (reserved).

Data Directory Table

The data directory table starts at offset 96 in a 32-bit PE header and at offset 112 in a 64-bit PE header. Each entry in the data directory table contains the relative virtual address and size of a table or a string used by the operating system. The data directory table entry is an 8-byte structure defined in Winnt.h as follows:

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

The first field, named *VirtualAddress*, is, however, not a virtual address but rather an RVA; it is the address of the table when the image file is loaded into memory, relative to the base address of the image. The RVAs given in this table do not necessarily point to the beginning of a section, and the sections containing specific tables do not necessarily have specific names. The second field is the size in bytes.

Sixteen standard data directories are defined in the data directory table:

- **Export Directory table address and size** The Export Directory table contains information about four other tables, which hold data on unmanaged exports of the PE file. Among managed compilers, only the MC++ compiler and linker and ILAsm are capable of exposing the managed methods exported by a managed PE file as unmanaged exports, to be consumed by an unmanaged caller. See Chapter 15, "Managed and Unmanaged Code Interoperation," for details.
- **Import table address and size** This table contains data on unmanaged imports consumed by the PE file. Among managed compilers, only the MC++ compiler and linker make any nontrivial use of this table, importing the unmanaged external functions used in the embedded unmanaged native code. Because other compilers, including the ILAsm compiler, do not embed the unmanaged native code in the managed PE files, Import Address tables (IATs) of the files produced by these compilers contain a single entry, that of the runtime entry function.
- **Resource table address and size** Contains unmanaged resources embedded in the PE file; managed resources aren't part of this data.
- **Exception table address and size** This table contains information on unmanaged exceptions only.
- **Certificate table address and size** The address entry points to a table of attribute certificates, which are not loaded into memory as part of the image file. As such, the first field of this entry is a file pointer rather than an RVA.
- **Base Relocation table address and size**
- **Debug data address and size** A managed PE file does not carry embedded debug data, so both entries of this data directory are set to 0.
- **Architecture data address and size**
- **Global pointer** RVA of the value to be stored in the global pointer register. The size must be set to 0.
- **TLS table address and size** Among managed compilers, only the MC++ compiler and linker and the ILAsm compiler are able to produce the code that would use the thread local storage data.
- **Load Configuration table address and size**

- **Bound Import table address and size**
- **Import Address table address and size**
- **Delay Import Descriptor address and size**
- **Common Language Runtime header address and size**
- **Reserved**

Section Headers

The table of section headers must immediately follow the PE header. Because the file header has no direct pointer to the section table, the location of this table is calculated as the total size of the file headers plus 1.

The *NumberOfSections* field of the COFF header defines the number of entries in the section header table. The section header enumeration in the table is one-based, with the order of the sections defined by the linker. The sections follow one another contiguously in the order set in the section header table, with starting RVAs aligned by the value of the *SectionAlignment* field of the PE header.

A section header is a 40-byte structure defined in Winnt.h as follows:

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[8];
    union {
            DWORD   PhysicalAddress;
            DWORD   VirtualSize;
    } Misc;
    DWORD   VirtualAddress;
    DWORD   SizeOfRawData;
    DWORD   PointerToRawData;
    DWORD   PointerToRelocations;
    DWORD   PointerToLinenumbers;
    WORD    NumberOfRelocations;
    WORD    NumberOfLinenumbers;
    DWORD   Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

The fields contained in the *IMAGE_SECTION_HEADER* structure can be described as follows:

- ***Name* (8-byte ANSI string)** Represents the name of the section. Section names start with a dot (for instance, *.reloc*). If the section name contains exactly eight characters, the null terminator is omitted. If the section name has fewer than eight characters, the array *Name* is padded with null characters. Image files cannot have section names with more than eight characters. In object files, however, section names can be longer. (Imagine a long-winded code generator emitting a section named *.myownsectionnobodyelsecouldevergrok.*) In this case, the name is placed in the string table, and the field contains the / (slash) character in the first byte, followed by an ANSI string containing a decimal representation of the respective offset in the string table.

- ***PhysicalAddress/VirtualSize* (4-byte unsigned integer)** In image files, this field holds the actual (unaligned) size in bytes of the code or data in this section.
- ***VirtualAddress* (4-byte unsigned integer)** Despite its name, this field holds the RVA of the beginning of the section.
- ***SizeOfRawData* (4-byte unsigned integer)** In an image file, this field holds the size in bytes of the initialized data on disk, rounded up to a multiple of the *FileAlignment* value specified in the PE header. If *SizeOfRawData* is less than *VirtualSize*, the rest of the section is padded with null bytes.
- ***PointerToRawData* (4-byte unsigned integer)** This field holds a file pointer to the section's first page. In image files, this value should be a multiple of the *FileAlignment* value specified in the PE header.
- ***PointerToRelocations* (4-byte unsigned integer)** This is a file pointer to the beginning of relocation entries for the section. In image files, this field is not used and should be set to 0.
- ***PointerToLinenumbers* (4-byte unsigned integer)** This field holds a file pointer to the beginning of line-number entries for the section. In managed PE files, the COFF line numbers are stripped and this field must be set to 0.
- ***NumberofRelocations* (2-byte unsigned integer)** In managed image files, this field should be set to 0.
- ***NumberOfLinenumbers* (2-byte unsigned integer)** In managed image files, this field should be set to 0.
- ***Characteristics* (4-byte unsigned integer)** This field specifies the characteristics of an image file and holds a combination of binary flags, described in Table 3-5.

The section *Characteristics* flags are defined in Winnt.h. Some of these flags are reserved, and some are relevant to object files only. Table 3-5 lists the flags that are valid for PE files.

Table 3-5 The Section *Characteristics* Flags in PE Files

Flag	Value	Description
IMAGE_SCN_SCALE_INDEX	0x00000001	TLS index is scaled (*.tls* section only).
IMAGE_SCN_CNT_CODE	0x00000020	Section contains the executable code. In ILAsm compiler–generated PE files, only the *.text* section carries this flag.
IMAGE_SCN_CNT_INITIALIZED_DATA	0x00000040	Section contains initialized data.
IMAGE_SCN_CNT_UNINITIALIZED_DATA	0x00000080	Section contains uninitialized data.
IMAGE_SCN_NO_DEFER_SPEC_EXC	0x00004000	Reset speculative exception handling bits in the type library (TLB) entries for this section.
IMAGE_SCN_LNK_NRELOC_OVFL	0x01000000	Section contains extended relocations.
IMAGE_SCN_MEM_DISCARDABLE	0x02000000	Section can be discarded as needed.
IMAGE_SCN_MEM_NOT_CACHED	0x04000000	Section cannot be cached.
IMAGE_SCN_MEM_NOT_PAGED	0x08000000	Section cannot be paged.
IMAGE_SCN_MEM_SHARED	0x10000000	Section can be shared in memory.
IMAGE_SCN_MEM_EXECUTE	0x20000000	Section can be executed as code. In ILAsm compiler–generated PE files, only the *.text* section carries this flag.
IMAGE_SCN_MEM_READ	0x40000000	Section can be read.
IMAGE_SCN_MEM_WRITE	0x80000000	Section can be written to. In ILAsm compiler–generated PE files, only the *.sdata* and *.tls* sections carry this flag.

The ILAsm compiler generates the following sections in a PE file:

- ***.text*** A read-only section containing the common language runtime header, the metadata, the IL code, managed structured exception handling information, and managed resources.
- ***.sdata*** A read/write section containing data.
- ***.reloc*** A read-only section containing relocations.
- ***.rsrc*** A read-only section containing unmanaged resources.
- ***.tls*** A read/write section containing thread local storage data.

Common Language Runtime Header

The fifteenth directory entry of the PE header contains the RVA and size of the runtime header in the image file. The runtime header, which contains all of the runtime-specific data entries and other information, should reside in a read-only, sharable section of the image file. The ILAsm compiler puts the common language runtime header in the *.text* section.

Header Structure

The common language runtime header is defined in CorHdr.h—a header file distributed as part of the Microsoft .NET Framework SDK—as follows:

```
typedef struct IMAGE_COR20_HEADER
{
    ULONG                   cb;
    USHORT                  MajorRuntimeVersion;
    USHORT                  MinorRuntimeVersion;
    // Symbol table and startup information
    IMAGE_DATA_DIRECTORY    MetaData;
    ULONG                   Flags;
    ULONG                   EntryPointToken;
    // Binding information
    IMAGE_DATA_DIRECTORY    Resources;
    IMAGE_DATA_DIRECTORY    StrongNameSignature;
    // Regular fixup and binding information
    IMAGE_DATA_DIRECTORY    CodeManagerTable;
    IMAGE_DATA_DIRECTORY    VTableFixups;
    IMAGE_DATA_DIRECTORY    ExportAddressTableJumps;

    IMAGE_DATA_DIRECTORY    ManagedNativeHeader;
} IMAGE_COR20_HEADER;
```

Table 3-6 takes a closer look at the fields of the header.

Table 3-6 Common Language Runtime Header Fields

Offset	Size	Field	Description
0	4	*Cb*	Size of the header in bytes.
4	2	*MajorRuntimeVersion*	Major portion of the minimum version of the runtime required to run the program.
6	2	*MinorRuntimeVersion*	Minor portion of the version of the runtime required to run the program.
8	8	*MetaData*	RVA and size of the metadata.
16	4	*Flags*	Binary flags, discussed in the following section. In ILAsm, this value can be specified explicitly by the directive *.corflags <integer value>* and/or the command-line option */FLAGS=<integer value>*. The command-line option takes precedence over the directive.
20	4	*EntryPointToken*	Metadata identifier (token) of the entry point for the image file; can be 0 for DLL images. This field identifies a method belonging to this module or a module containing the entry point method.
24	8	*Resources*	RVA and size of managed resources.
32	8	*StrongNameSignature*	RVA and size of the hash data for this PE file, used by the loader for binding and versioning.
40	8	*CodeManagerTable*	RVA and size of the Code Manager table. In the first release of the runtime, this field is reserved and must be set to 0.
48	8	*VTableFixups*	RVA and size in bytes of an array of virtual table (v-table) fixups. Among current managed compilers, only the MC++ compiler and linker and the ILAsm compiler can produce this array.
56	8	*ExportAddressTableJumps*	RVA and size of an array of addresses of jump thunks. Among current managed compilers, only the MC++ compiler and linker can produce this table, which allows the export of unmanaged native methods embedded in the managed PE file.
64	8	*ManagedNativeHeader*	Reserved; set to 0.

Flags Field

The *Flags* field of the common language runtime header can include one or more of the following flags:

- ***COMIMAGE_FLAGS_ILONLY* (0x00000001)** The image file contains IL code only, with no embedded native unmanaged code except the startup stub. Because common language runtime–aware operating systems (such as Windows XP) ignore the startup stub, for all practical purposes the file can be considered pure-IL. However, using this flag can cause certain ILAsm compiler–specific problems when running under Windows XP. If this flag is set, Windows XP ignores not only the startup stub but also the *.reloc* section. The *.reloc* section can contain relocations for the beginning and end of the *.tls* section as well as relocations for what is referred to as *data-on-data* (that is, data constants that are pointers to other data constants). Among existing managed compilers, only the MC++ compiler and linker and the ILAsm compiler can produce these items. The MC++ compiler and linker never set this flag because the image file they generate is never pure-IL. Currently, the ILAsm compiler is the only one capable of producing pure-IL image files that might require a *.reloc* section. To resolve this problem, the ILAsm compiler, if TLS-based data or data-on-data is emitted, clears this flag and sets the *COMIMAGE_FLAGS_32BITREQUIRED* flag instead.

- ***COMIMAGE_FLAGS_32BITREQUIRED* (0x00000002)** The image file can be loaded only into a 32-bit process. This flag is set when native unmanaged code is embedded in the PE file or when the *.reloc* section is not empty.

- ***COMIMAGE_FLAGS_IL_LIBRARY* (0x00000004)** This flag is obsolete and should not be set. Setting it—as the ILAsm compiler allows, using the *.corflags* directive—will render your module unloadable.

- ***COMIMAGE_FLAGS_STRONGNAMESIGNED* (0x00000008)** The image file is protected with a strong name signature. The *strong name signature* includes the public key and the signature hash and is a part of an assembly's identity, along with the assembly name, version number, and the culture information. This flag is set when the strong name signing procedure is applied to the image file. No compiler, including ILAsm, can set this flag explicitly.

- ***COMIMAGE_FLAGS_TRACKDEBUGDATA* (0x00010000)** The loader and the JIT (just-in-time) compiler are required to track debug information about the methods.

EntryPointToken Field

The *EntryPointToken* field of the common language runtime header contains a token (metadata identifier) of either a method definition (*MethodDef*) or a file reference (*File*). A *MethodDef* token identifies a method defined in the module (a managed PE file) as an entry point method. A *File* token is used in one case only: in the runtime header of the prime module of a multimodule assembly, when the entry point method is defined in another module (identified by the file reference) of this assembly. In this case, the module identified by the file reference must contain the respective *MethodDef* token in the *EntryPointToken* field of its runtime header.

EntryPointToken must be specified in runnable executables (EXE files). The ILAsm compiler, for example, does not even try to generate an EXE file if the source code does not define the entry point. The loader imposes limitations on the signature of the entry point method: the method must return an unsigned integer or *void*, and it must have at most one parameter of type *string* or *string[]* (vector of strings).

With nonrunnable executables (DLL files), it's a different story. Pure-IL DLLs don't need the entry point method defined, and the *EntryPointToken* field in their runtime headers should be set to 0.

Mixed-code DLLs—DLLs containing IL and embedded native code—generated by the MC++ compiler and linker must run the unmanaged native function *DllMain* immediately at the DLL invocation in order to perform the initialization necessary for the unmanaged native components of the DLL. The signature of this unmanaged function must be as follows:

```
int DllMain(HINSTANCE, DWORD, void *);
```

To be visible from the managed code and the runtime, the function *DllMain* must be declared as a platform invocation of an embedded native method (local *P/Invoke*, also known in enlightened circles as IJW—It Just Works). See Chapter 15 for details about the interoperation of managed and unmanaged code.

Note The method referred to by the *EntryPointToken* field of the common language runtime header has nothing to do with the function to which the *AddressOfEntryPoint* field of the PE header points. *AddressOfEntryPoint* always points to the runtime invocation stub, which is invisible to the runtime, is not reflected in metadata, and hence cannot have a token.

VTableFixups Field

The *VTableFixups* field of the runtime header is a data directory containing the RVA and the size of the image file's v-table fixup table. When a managed method must be called from unmanaged code, the common language runtime creates a marshaling thunk for it, and the address of this thunk is placed in the respective address table. If the managed method is called from the unmanaged native code embedded in the managed PE file, the thunk address goes to a special internal v-table. If the managed method is exported as unmanaged and is consumed somewhere outside the managed PE file, the address of the respective v-table entry must also go to the Export Address table. At loading time (and in the disk image file), the entries of this v-table contain the respective method tokens.

These v-table fixups represent the initializing information necessary for the runtime to create the thunks and lay out the respective tables. v-table fixup is defined in CorHdr.h as follows:

```
typedef struct _IMAGE_COR_VTABLEFIXUP {
    ULONG       RVA;
    USHORT      Count;
    USHORT      Type;
} IMAGE_COR_VTABLEFIXUP;
```

In this definition, *RVA* points to the location of the v-table slot containing the respective method token(s). *Count* specifies the number of entries in the slot if multiple implementations of the same method exist, overriding one another. *Type* is a combination of the following flags, providing the runtime with information about the slot and what to do with it:

- ***COR_VTABLE_32BIT* (0x01)** Each entry is 32 bits wide.
- ***COR_VTABLE_64BIT* (0x02)** Each entry is 64 bits wide.

- ***COR_VTABLE_FROM_UNMANAGED* (0x04)** The thunk created by the common language runtime must provide data marshaling between managed and unmanaged code.
- ***COR_VTABLE_CALL_MOST_DERIVED* (0x10)** This flag is not currently used.

Obviously, the first two flags are mutually exclusive. The slots of the v-table must follow each other immediately—that is, the v-table must be contiguous.

Because the v-table should be fixed up after the image has been loaded into memory, this table is located in a read/write section. (In contrast, the v-table in an unmanaged image is located in a read-only section.)

Among existing managed compilers, only the MC++ compiler and linker and the ILAsm compiler can define the v-table and its fixups.

StrongNameSignature Field

The *StrongNameSignature* field of the common language runtime header contains the RVA and size of the strong name hash, which is used by the runtime to establish the authenticity of the image file. After the image file has been created, it is hashed using the public and private encryption keys provided by the producer of the image file, and the resulting hash blob is written into the space allocated inside the image file.

If even a single byte in the image file is subsequently modified, the authenticity check fails and the image file cannot be loaded. The strong name signature does not survive a round-tripping procedure; if you disassemble a strong-named module using the IL Disassembler and then reassemble it, the module must be strong name signed again.

The ILAsm compiler puts the strong name signature in the *.text* section of the image file.

Relocation Section

The *.reloc* section of the image file contains the Fixup table, which holds entries for all fixups in the image file. The RVA and size of the *.reloc* section are defined by the Base Relocation table directory of the PE header. The Fixup table consists of blocks of fixups, each block representing the fixups for a 4-KB page. Blocks are 4-byte-aligned.

Each fixup describes the location of a specific address within the image file as well as how the OS loader should modify the address at this location when loading the image file into memory.

Each fixup block starts with two 4-byte unsigned integers: the RVA of the page containing the address to be fixed up and the size of the block. The fixup entries for this page immediately follow. Each entry is a 2-byte unsigned integer, of which 4 senior bits contain the type of relocation required. The remaining 12 bits contain the relocated address's offset within the page.

To relocate an address, the OS loader calculates the difference (delta) between the preferred base address (the *ImageBase* field of the PE header) and the actual base address where the image file is loaded. This delta is then applied to the address according to the type of relocation. If the image file is loaded at its preferred address, no fixups need be applied.

The following relocation types are defined in Winnt.h:

- ***IMAGE_REL_BASED_ABSOLUTE* (0)** This type has no meaning in an image file, and the fixup is skipped.
- ***IMAGE_REL_BASED_HIGH* (1)** The high 16 bits of the delta are added to the 16-bit field at the offset. The 16-bit field in this case is the high half of the 32-bit address being relocated.
- ***IMAGE_REL_BASED_LOW* (2)** The low 16 bits of the delta are added to the 16-bit field at the offset. The 16-bit field in this case is the low half of the 32-bit address being relocated.
- ***IMAGE_REL_BASED_HIGHLOW* (3)** The delta is added to the 32-bit address at the offset.
- ***IMAGE_REL_BASED_HIGHADJ* (4)** The high 16 bits of the delta are added to the 16-bit field at the offset. The 16-bit field in this case is the high part of the 32-bit address being relocated. The low 16 bits of the address are stored in the 16-bit word that follows this relocation. A fixup of this type occupies two slots.
- ***IMAGE_REL_BASED_MIPS_JMPADDR* (5)** The fixup applies to a MIPS *jump* instruction.
- ***IMAGE_REL_BASED_SECTION* (6)** Reserved.
- ***IMAGE_REL_BASED_REL32* (7)** Reserved.
- ***IMAGE_REL_BASED_MIPS_JMPADDR16* (9)** The fixup applies to a MIPS16 *jump* function.
- ***IMAGE_REL_BASED_IA64_IMM64* (9)** This is the same type as *IMAGE_REL_BASED_MIPS_JMPADDR16*.
- ***IMAGE_REL_BASED_DIR64* (10)** The delta is added to the 64-bit field at the offset.

- ***IMAGE_REL_BASED_HIGH3ADJ* (11)** The fixup adds the high 16 bits of the delta to the 16-bit field at the offset. The 16-bit field is the high one-third of a 48-bit address. The low 32 bits of the address are stored in the 32-bit double word that follows this relocation. A fixup of this type occupies three slots.

The only fixup type emitted by the existing managed compilers is *IMAGE_REL_BASED_HIGHLOW.*

A pure-IL PE file, as a rule, contains only one fixup in the *.reloc* section. This is for the benefit of the common language runtime startup stub, the only segment of native code in a pure-IL image file. This fixup is for the image file's IAT, containing a single entry: the runtime DLL.

Windows XP, as a common language runtime–aware operating system, needs neither the runtime startup stub nor the IAT to engage the runtime. Thus, if the common language runtime header flags indicate that the image file is IL-only (*COMIMAGE_FLAGS_ILONLY*), the operating system ignores the *.reloc* section altogether.

This optimization plays a bad joke with some image files generated by the ILAsm compiler. This compiler produces pure-IL image files but needs relocations executed if any data is located in thread local storage or if data-on-data is defined. To have these relocations executed when the image file is loaded under Windows XP, the ILAsm compiler is forced to cheat and set the common language runtime header flags as if the image file contained embedded native code (*COMIMAGE_FLAGS_32BITREQUIRED*).

Other compilers don't have these problems. Compilers generating pure-IL image files (such as Microsoft Visual C# .NET and Microsoft Visual Basic .NET) don't define TLS-based data or data-on-data.

Because the MC++ compiler and linker produce mixed-code image files, the *.reloc* sections of these image files can contain any number of relocations. But because mixed-code image files never carry IL-only common language runtime header flags, their relocations are always executed.

Text Section

The *.text* section of a PE file is a read-only section. In a managed PE file, it contains metadata tables, IL code, an Import Address table, a common language runtime header, and an unmanaged runtime startup stub. The image files generated by the ILAsm compiler additionally contain managed resources, the strong name signature hash, and unmanaged export stubs.

The ILAsm compiler emits data to the *.text* section in a particular order. When the PE file generator is initialized during the ILAsm compiler startup, space is allocated in the *.text* section for the Import Address table—which carries one lonely entry, for the startup routine of the runtime DLL—and for the runtime header.

The IL code and the managed structured exception handling tables for each method defined in the module are emitted to the *.text* section during the parsing of the source code, as soon as parsing and compilation of the next method are completed.

After all the source files representing the module have been parsed and all IL code and structured exception handling tables have been emitted, the ILAsm compiler, if so directed, allocates sufficient space in the *.text* section for the strong name signature. The signature itself is emitted later, as the last step of the file generation.

Then the ILAsm compiler analyzes and rearranges the metadata defined during the parsing of the source files and emits the metadata tables to the *.text* section. By this time, all the managed resources to be embedded in the image file are analyzed and accounted for, and their respective offsets within the managed resource directory are recorded as part of the metadata describing these resources. At this time, all the necessary fixups are made in the already emitted IL code. These fixups primarily deal with the metadata tokens, which were unknown before the metadata analysis and rearrangement or were changed during that process. Other fixups deal with references to global data constants placed in the *.sdata* section that will be discussed later in this chapter.

After all metadata has been emitted, any managed resources to be embedded in the image file are read from the respective files and emitted to the *.text* section. (For a discussion of embedding managed and unmanaged resources, see "Resources" later in this chapter.)

The next set of data to be emitted to the *.text* section consists of the export stubs for each managed method that will be exposed as an unmanaged export, to be consumed by the external unmanaged executables. (For detailed information on managed and unmanaged code interoperation, see Chapter 15.)

The last item emitted to the *.text* section is the unmanaged runtime startup stub, whose RVA is assigned to the *AddressOfEntryPoint* field of the PE header.

Figure 3-3 summarizes the general structure of the *.text* section of an image file generated by the ILAsm compiler.

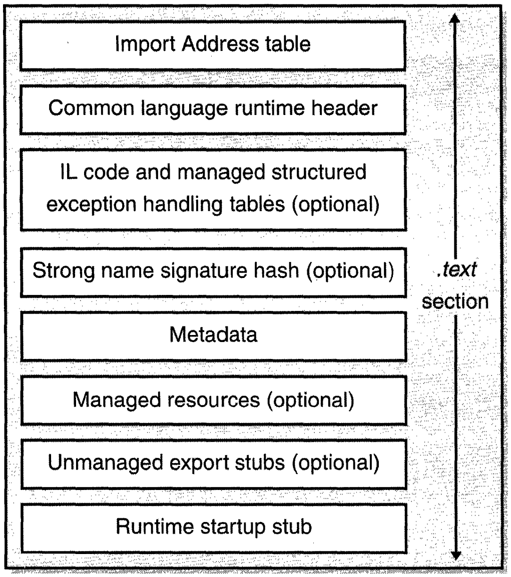

Import Address table
Common language runtime header
IL code and managed structured exception handling tables (optional)
Strong name signature hash (optional)
Metadata
Managed resources (optional)
Unmanaged export stubs (optional)
Runtime startup stub
.text section

Figure 3-3 Structure of a *.text* section emitted by the ILAsm compiler.

Data Sections

The data section (*.sdata*) of an image file generated by the ILAsm compiler is a read/write section. It contains data constants, the v-table, the unmanaged export table, and the thread local storage directory structure. The data declared as thread-specific is located in a different section, the *.tls* section.

Data Constants

The term *data constants* might be a little misleading. Located in a read/write section, data constants can certainly be overwritten, so technically they can hardly be called "constants." The term, however, refers to the usage of the data rather than to the nature of the data. Data constants represent the mappings of the static fields and usually contain data initializing the mapped fields. (Chapter 1 described the peculiarities of this field mapping; see "Mapped Fields.")

Field mapping is a convenient way to initialize the static fields with ANSI strings, blobs, or structures. An alternative way to initialize the static fields—and a more orthodox way in terms of the common language runtime—is to do it explicitly in class constructors, as discussed in the section "Constructors vs. Data Constants" in Chapter 8, "Fields and Data Constants." But this alternative is much more tedious, so no one can really blame the managed compilers for resorting to field mapping for initialization. The MC++ compiler maps all the global fields, whether they will be initialized or not.

Mapping static fields to data has its caveats. Fields mapped to the data section are, on the one hand, out of reach of runtime controlling mechanisms such as type control and garbage collection and, on the other hand, wide open to unrestricted access and modification. This causes the loader to prevent certain field types from being mapped; types of mapped fields might contain no references to objects, vectors, or arrays, nor to any nonpublic substructures. No such problems arise if a class constructor is used for static field initialization. Philosophically speaking, this is only natural: throughout the history of humanity, deviations from orthodoxy, however tempting, have always brought some unpleasant complications.

V-Table

The v-table consists of entries, and each entry consists of one or more slots. The v-table fixups we have already discussed earlier in the section "*VTableFixups* Field" specify the number and width (4 or 8 bytes) of slots in each entry. Each slot contains a metadata token of the respective method, which at execution time is replaced with the address of the method itself or the address of a marshaling thunk representing the method. Because these fixups are performed at execution time, the v-table of a managed PE file must be located in a read/write section. The ILAsm compiler puts the v-table in the *.sdata* section, together with other data.

V-tables of unmanaged image files are completely defined at link time and need base relocation fixups only, performed by the OS loader. Because no changes are made to v-tables at execution time, unmanaged image files carry their v-tables in read-only sections.

Unmanaged Export Table

The unmanaged export table in an unmanaged image file occupies a separate section named *.edata*. In image files generated by the ILAsm compiler, the unmanaged export table resides in the *.sdata* section, together with the v-table it references.

The unmanaged export table contains information about symbols that other (unmanaged) image files can access through dynamic linking. The unmanaged export table is not a single table but rather a contiguous set of five tables: the Export Directory table, the Export Address table, the Name Pointer table, the Ordinal table, and the Export Name table.

The unmanaged export information starts with the Export Directory table, which describes the rest of the export information. It is a table with only one element, containing the locations and sizes of other export tables. The structure of the sole row of the Export Directory table is defined in Winnt.h as follows:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;
    DWORD   AddressOfNames;
    DWORD   AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

Briefly, the fields of *IMAGE_EXPORT_DIRECTORY* are the following:

- ***Characteristics*** Reserved. This field should be set to 0.
- ***TimeDateStamp*** The time and date the export data was generated.
- ***MajorVersion*** The major version number. This field and the *MinorVersion* field are for information only; the ILAsm compiler does not set them.
- ***MinorVersion*** The minor version number.
- ***Name*** The RVA of the ANSI string containing the name of the exporting module.
- ***Base*** The ordinal base (usually 1). This is the starting ordinal number for exports in the image file.
- ***NumberOfFunctions*** The number of entries in the Export Address table.
- ***NumberOfNames*** Number of entries in the Export Name table.
- ***AddressOfFunctions*** The RVA of the Export Address table.
- ***AddressOfNames*** The RVA of the Export Name table.
- ***AddressOfNameOrdinals*** The RVA of the Name Pointer table.

The Export Address table contains the RVAs of exported entry points. The export ordinal of an entry point is defined as its zero-based index within the Export Address table plus the ordinal base (the value of the *Base* field of *IMAGE_EXPORT_DIRECTORY* structure).

In a managed file, the Export Address table contains the RVAs not of the exported entry points (methods) themselves but rather of unmanaged export stubs representing these entry points. (See "Text Section" earlier in this chapter.) Export stubs, in turn, contain references to respective v-table slots.

An RVA in an Export Address table can be a so-called *forwarder RVA*, identifying a re-exported entry point—that is, an entry point this module imports from another module and exports as its own. In such a case, the RVA points to an ANSI string containing the import name. The import name might be a DLL name and the name of the imported entry (*SomeDLL.someFunc*) or a DLL name and the imported entry's ordinal in this DLL (*SomeDLL.#12*).

Because the ILAsm compiler does not allow re-export, the entries in an Export Address table of an image file generated by this compiler always represent the RVAs of unmanaged export stubs.

The Name Pointer table contains RVAs of the export names from the Export Name table. These RVAs are lexically ordered to facilitate binary searches of the entry points by name.

The Ordinal table contains 2-byte indexes to the Export Address table. The Name Pointer table and the Ordinal table form two parallel arrays and operate as one intermediate lookup table, rearranging the entries so that they are lexically ordered by name.

The Export Name table contains zero-terminated ANSI strings representing the export names of the methods exported by the module. The export names might differ from the names under which the methods were declared in the module. An exported method might have no exported name at all if it is being exported by ordinal only. In this case, its ordinal is not included in the Ordinal table. The ILAsm compiler does not allow unnamed exports.

Chapter 15 examines unmanaged export information and the details of exposing managed methods as unmanaged exports.

Thread Local Storage

ILAsm and MC++ allow you to define data constants belonging to thread local storage and to map static fields to these data constants. TLS is a special storage class in which a data object is not a stack variable but is nevertheless local to each separate thread. Consequently, each thread can maintain a different value for such a variable.

The TLS data is described in the TLS directory, which the ILAsm compiler puts in the *.sdata* section. The structure of the TLS directory for 32-bit image files is defined in Winnt.h as follows:

```
typedef struct _IMAGE_TLS_DIRECTORY32 {
    DWORD   StartAddressOfRawData;
    DWORD   EndAddressOfRawData;
    PDWORD  AddressOfIndex;
    PIMAGE_TLS_CALLBACK *AddressOfCallBacks;
    DWORD   SizeOfZeroFill;
    DWORD   Characteristics;
} IMAGE_TLS_DIRECTORY32;
```

The fields of this structure can be described as follows:

- ***StartAddressOfRawData*** The starting virtual address (not an RVA) of the TLS data constants. The TLS data constants plus uninitialized TLS data together form the *TLS template*. The operating system makes a copy of the TLS template every time a thread is created, thus providing each thread with its "personal" data constants and field mapping.
- ***EndAddressOfRawData*** The ending VA of the TLS data constants. The rest of the TLS data (if any) is filled with zeros. Because the ILAsm compiler allows no uninitialized TLS data, presuming that TLS data constants represent the whole TLS template, nothing is left for the zero fill.
- ***AddressOfIndex*** The VA of the 4-byte TLS index, located in the ordinary data section. The ILAsm compiler puts the TLS index in the *.sdata* section, immediately after the TLS directory structure and the callback function pointer array terminator.
- ***AddressOfCallBacks*** The VA of an array of TLS callback function pointers. Because the array is null-terminated, this field points to 4 bytes set to 0 if no callback functions are supported. The ILAsm compiler does not support TLS callback functions, so the entire array of TLS callback function pointers consists of zero terminator. This zero terminator immediately follows the TLS directory structure in the *.sdata* section.
- ***SizeOfZeroFill*** The size of the uninitialized part of the TLS template, filled with zeros when a copy of the TLS template is being made. The ILAsm compiler sets this field to 0.
- ***Characteristics*** Reserved. This field should be set to 0.

Because the *StartAddressOfRawData*, *EndAddressOfRawData*, *AddressOfIndex*, and *AddressOfCallBacks* fields hold VAs rather than RVAs, base relocations must be defined for them in the *.reloc* section.

The RVA and size of the TLS directory structure are stored in the tenth data directory (TLS) of the PE header. TLS data constants, which form the TLS template, are stored in the *.tls* section of the image file.

Resources

Two distinct kinds of resources can be embedded in a managed PE file: unmanaged platform-specific resources and managed common language runtime–specific resources. These two kinds of resources, which have nothing in common, reside in different sections of a managed image file and are accessed by different sets of APIs.

Unmanaged Resources

Unmanaged resources reside in the *.rsrc* section of the image file. The starting RVA and size of embedded unmanaged resources are represented in the Resource data directory of the PE header.

Unmanaged resources are indexed by type, name, and language and are binary-sorted by these three characteristics in that order. A set of Resource directory tables represents this indexing as follows: each directory table is followed by an array of directory entries, which contain the ID or name of the respective level (the type, name, or language level) and the address of the next-level directory table or of a data description (a leaf node of the tree). Because three indexing characteristics are used, any data description can be reached by analyzing at most three directory tables.

By the time the data description is reached, its type, name, and language are known from the path the search algorithm traversed to arrive at the data description.

The *.rsrc* section has the following structure:

- **Resource directory tables and entries**
- **Resource directory strings** Unicode strings representing the string data addressed by the directory entries. These strings are 2-byte-aligned. Each string is preceded by a 2-byte unsigned integer representing the string's length.
- **Resource data description** A set of records addressed by directory entries, containing the size and location of actual resource data.
- **Resource data** Raw undelimited resource data, consisting of individual resource data whose address and size are defined by data description records.

A Resource directory table structure is defined in Winnt.h as follows:

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    WORD    NumberOfNamedEntries;
    WORD    NumberOfIdEntries;
} IMAGE_RESOURCE_DIRECTORY, *PIMAGE_RESOURCE_DIRECTORY;
```

The roles of these fields should be evident, in light of the preceding discussion about structuring unmanaged resources and the Resource directory tables. One exception might be the *Characteristics* field, which is reserved and should be set to 0.

Name entries, which use strings to identify type, name, or language, immediately follow the Resource directory table. After them, ID entries are stored.

A Resource directory entry (either a name entry or an ID entry) is an 8-byte structure consisting of two 4-byte unsigned integers, defined in Winnt.h as follows:

```
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
    union {
        struct {
            DWORD NameOffset:31;
            DWORD NameIsString:1;
        };
        DWORD   Name;
        WORD    Id;
    };
    union {
        DWORD   OffsetToData;
        struct {
            DWORD   OffsetToDirectory:31;
            DWORD   DataIsDirectory:1;
        };
    };
} IMAGE_RESOURCE_DIRECTORY_ENTRY, *PIMAGE_RESOURCE_DIRECTORY_ENTRY;
```

If the senior bit of the first 4-byte component is set, the entry is a name entry and the remaining 31 bits represent the name string offset; otherwise, the entry is an ID entry and the remaining bits hold the ID value.

If the senior bit of the second component is set, the item, whose offset is represented by the remaining 31 bits, is a next-level Resource directory table; otherwise, it is a Resource data description.

A Resource data description is a 16-byte structure defined in Winnt.h as follows:

```
typedef struct _IMAGE_RESOURCE_DATA_ENTRY {
    DWORD   OffsetToData;
    DWORD   Size;
    DWORD   CodePage;
    DWORD   Reserved;
} IMAGE_RESOURCE_DATA_ENTRY, *PIMAGE_RESOURCE_DATA_ENTRY;
```

The fields *OffsetToData* and *Size* characterize the respective chunks of resource data that constitute an individual resource. *CodePage* is the ID of the code page used to decode the code point values in the resource data. Usually this is the Unicode code page. Finally—no surprise here—the *Reserved* field is reserved and must be set to 0.

The ILAsm compiler creates the *.rsrc* section and embeds the unmanaged resources from the respective .RES file if this file is specified in command-line options. The compiler can embed only one unmanaged resource file per module.

When the IL Disassembler analyzes a managed PE file and finds the *.rsrc* section, it reads the data and its structure from the section and emits a .RES file containing all the unmanaged resources embedded in the PE file.

Managed Resources

The *Resources* field of the common language runtime header contains the RVA and size of the managed resources embedded in the PE file. It has nothing to do with the Resource directory of the PE header, which specifies the RVA and size of unmanaged platform-specific resources.

In PE files created by the ILAsm compiler, unmanaged resources reside in the *.rsrc* section of the image file, whereas managed resources are located in the *.text* section, along with the metadata, the IL code, and so on. Managed resources are stored in the *.text* section contiguously. Metadata carries *ManifestResource* records, one for each managed resource, containing the name of the managed resource and the offset of the beginning of the resource from the starting RVA specified in the *Resources* field. At this offset, a 4-byte unsigned integer indicates the length in bytes of the resource. The resource itself immediately follows.

When the IL Disassembler processes a managed image file and finds embedded managed resources, it writes each resource to a separate file, named according to the resource name.

When the ILAsm compiler creates a PE file, it reads all managed resources defined in the source code as embedded from the file according to the resource names and writes them to the *.text* section, each preceded by its specified length.

Summary

Let's summarize the ways the ILAsm compiler creates a managed PE file. The PE file creation is performed in four phases.

Phase One: Initialization

- Internal buffers are initialized.
- The empty template of a PE file is open in memory, including an MS-DOS stub, a PE signature, a COFF header, and a PE header.
- The Import Address table and the runtime header are allocated in the *.text* section.

Phase Two: Source Code Parsing

- The IL code is emitted to the *.text* section.
- Data constants are emitted to the *.sdata* and *.tls* sections.
- Metadata is collected in internal buffers.

Phase Three: Image Generation

- Space for the strong name signature is allocated in the *.text* section.
- Metadata is analyzed, rearranged, and emitted to the *.text* section.
- Managed resources are emitted to the *.text* section.
- Unmanaged export stubs are emitted to the *.text* section.
- Unmanaged export tables are emitted to the *.sdata* section.
- The TLS directory table is emitted to the *.sdata* section.
- The runtime startup stub is emitted to the *.text* section.
- Unmanaged resources are read from a .RES file and emitted to the *.rsrc* section.
- Necessary base relocations are emitted to the *.reloc* section.

Phase Four: Completion

- The image file is written as a disk file.
- The strong name signing procedure is applied to the file.

The ILAsm compiler allows you to explicitly set certain values in the image file headers, by means of both source code directives and the compiler's command-line options, as shown in Table 3-7. In all the cases discussed in this chapter, the command-line options take precedence over the respective source code directives.

Table 3-7 Directives and Command-Line Options for Setting Header Fields

Header	Field	Directive	Command-Line Option
PE	*ImageBase*	*.imagebase <integer value>*	*/BASE=<integer value>*
PE	*FileAlignment*	*.file alignment <integer value>*	*/ALIGNMENT=<integer value>*
PE	*Subsystem*	*.subsystem <integer value>*	*/SUBSYSTEM=<integer value>*
CLR	*Flags*	*.corflags <integer value>*	*/FLAGS=<integer value>*

4

Metadata Tables Organization

This chapter provides a general overview of metadata and how it is structured. It also describes metadata validation and the PEVerify tool, used to perform validation and verification. Later chapters will analyze individual metadata items based on the foundation presented here. I understand your possible impatience—"When will this guy quit stalling and get to the real stuff?"—but nevertheless I urge you not to skip this chapter. Far from stalling, I'm simply approaching the subject systematically. It might look the same, but the motivation is quite different.

What Is Metadata?

Metadata is, by definition, data that describes data. Like any general definition, however, this one is hardly informative. In the context of the common language runtime, *metadata* means a system of descriptors of all items that are declared or referenced in a module. Because the common language runtime programming model is inherently object-oriented, the items represented in metadata are classes and their members, with their accompanying attributes, properties, and relationships.

From a pragmatic point of view, the role played by metadata is similar to that played by type libraries in the COM world. At this general level, however, the similarities end and the differences begin. Metadata, which describes the structural aspect of a module or an assembly in minute detail, is vastly richer than the data provided by type libraries, which carry only information regarding the COM interfaces exposed by the module. The important difference, of course, is that metadata is embedded in a managed module, which allows each managed module to carry a complete formal description of its logical structure.

Structurally, metadata is a normalized relational database. This means that metadata is organized as a set of cross-referencing rectangular tables—as opposed to, for example, a hierarchical database that has a tree structure. Each column of each row of a metadata table contains either data or a reference to a row of another table. Metadata does not contain any duplicate data fields; each category of data resides in only one table of the metadata database. If another table needs to employ the same data, it references the table that holds the data.

For example, as Chapter 1, "Simple Sample," explained, a class definition carries certain binary attributes (flags). Because the behavior and features of member methods of this class are affected by the class's flags, it would be tempting to duplicate some of the class attributes, including flags, in a metadata record describing one of the methods. But data duplication leads not only to increased database size but also to the problem of keeping all the duplications synchronized.

Instead, a method descriptor contains a reference to the descriptor of the method's parent class. Such referencing does require resolving additional levels of indirection, which results in burning more processor cycles. But for massively distributed systems (and Microsoft .NET–based applications obviously target such systems), processor speed is not the problem—communication bandwidth and data integrity are.

But what do you do if, for instance, you need to find all the methods a certain class implements? Browse the entire method descriptor table to find the methods referring to this class's descriptor? No, that would be no fun at all. Instead, the class descriptor (record) carries a reference to the record of the method table that represents the first method of this class. The end of the method records belonging to this class is defined by the beginning of the next class's method records or (for the last class) by the end of the method table.

Obviously, this technique requires that the records in the method table must be ordered by their parent class. The same applies to other table-to-table relationships (class-to-field, method-to-parameter, and so on). If this requirement is met, the metadata is referred to as *optimized*, or *compressed*. Figure 4-1 shows an example of such metadata. The ILAsm compiler always emits optimized metadata.

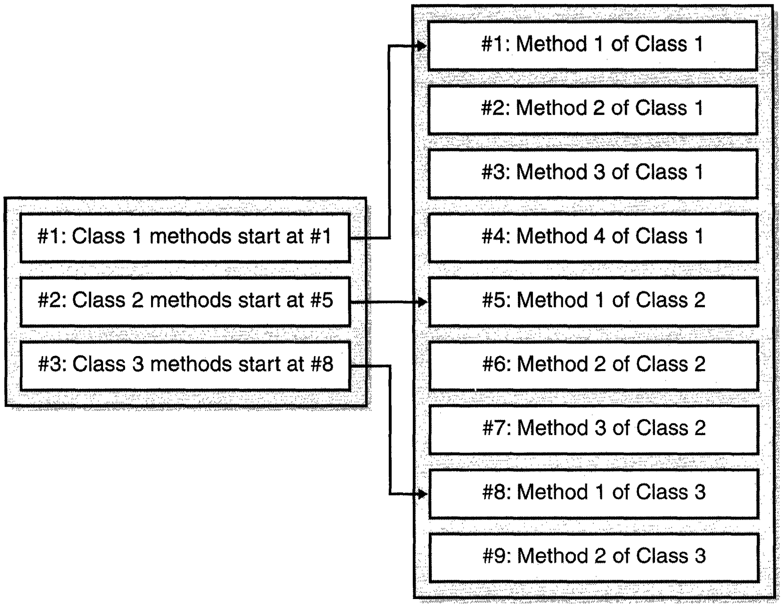


Figure 4-1 An example of optimized metadata.

It is possible, however—perhaps as a result of sloppy metadata emission—to have the child tables interleaved with regard to their parent classes. For example, class record A might be emitted first, followed by class record B, the method records of class B, and then the method records of class A; or the sequence might be class record A, then some of the method records of class A, followed by class record B, the method records of class B, and then the rest of the method records of class A.

In such a case, additional intermediate metadata tables are engaged, providing noninterleaved and ordered lookup tables. Instead of referencing the method records, class records reference the records of an intermediate table (a *pointer* table), which in turn reference the method records, as diagrammed in Figure 4-2. Metadata that uses such intermediate lookup tables is referred to as *unoptimized*, or *uncompressed*.

Note Uncompressed metadata structure is characteristic of an "edit-and-continue" scenario, in which metadata and the IL code of a module are modified while the module is loaded in memory.

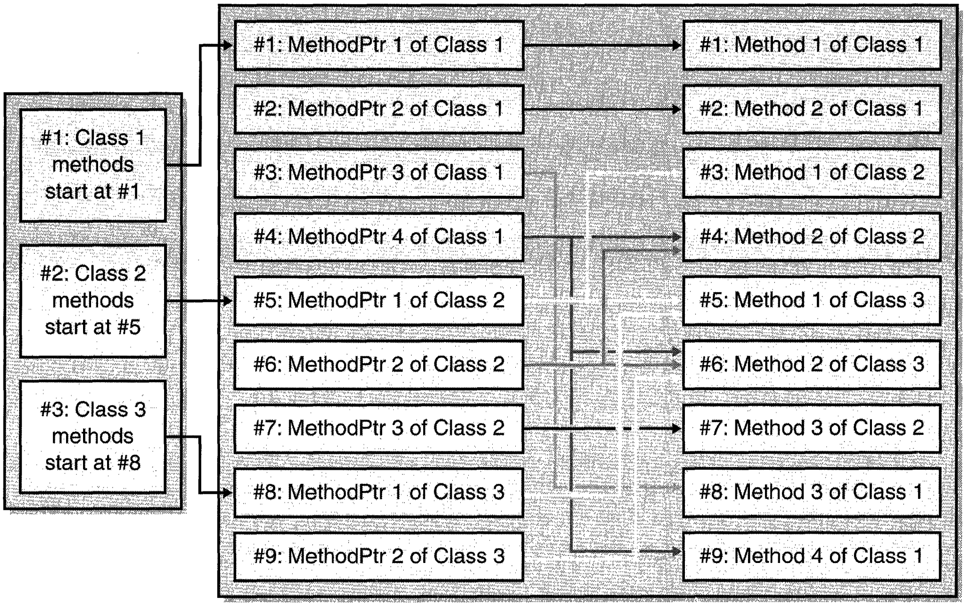

#1: Class 1 methods start at #1
#2: Class 2 methods start at #5
#3: Class 3 methods start at #8
#1: MethodPtr 1 of Class 1
#2: MethodPtr 2 of Class 1
#3: MethodPtr 3 of Class 1
#4: MethodPtr 4 of Class 1
#5: MethodPtr 1 of Class 2
#6: MethodPtr 2 of Class 2
#7: MethodPtr 3 of Class 2
#8: MethodPtr 1 of Class 3
#9: MethodPtr 2 of Class 3
#1: Method 1 of Class 1
#2: Method 2 of Class 1
#3: Method 1 of Class 2
#4: Method 2 of Class 2
#5: Method 1 of Class 3
#6: Method 2 of Class 3
#7: Method 3 of Class 2
#8: Method 3 of Class 1
#9: Method 4 of Class 1

Figure 4-2 An example of unoptimized metadata.

Heaps and Tables

Logically, metadata is represented as a set of named streams, each stream representing a category of metadata. These streams are divided into two types: metadata heaps and metadata tables.

Heaps

A metadata heap is a storage of trivial structure, holding a contiguous sequence of items. Heaps are used in metadata to store strings and binary objects. There are three kinds of metadata heaps:

- **String heap** This type of heap contains zero-terminated character strings, encoded in UTF-8. The strings follow each other immediately. Because the first byte of the heap is always 0, the first string in the heap is an empty string. The last byte of the heap must be 0 as well.
- **GUID heap** This type of heap contains 16-byte binary objects, immediately following each other. Because the size of the binary objects is fixed, length parameters or terminators are not needed.
- **Blob heap** This type of heap contains binary objects of arbitrary size. Each binary object is preceded by its length (in compressed form). Binary objects are aligned on 4-byte boundaries.

The length compression formula is fairly simple. If the length (which is an unsigned integer) is 0x7F or less, it is represented as 1 byte; if the length is greater than 0x7F but no larger than 0x3FFF, it is represented as a 2-byte unsigned integer with the senior bit set. Otherwise, it is represented as a 4-byte unsigned integer with two senior bits set. Table 4-1 summarizes this formula.

Table 4-1 The Length Compression Formula for the Blob

Value Range	Compressed Size	Compressed Value (Big-Endian)
0–0x7F	1 byte	*<value>*
0x80–0x3FFF	2 bytes	0x8000 | *<value>*
0x4000–0x1FFFFFFF	4 bytes	0xC0000000 | *<value>*

Note This compression formula is widely used in metadata. Of course, the compression works only for numbers not exceeding 0x1FFFFFFF (536,870,911), but this limitation isn't a problem because the compression is usually applied to such values as lengths and counts.

General Metadata Header

A general metadata header consists of a storage signature and a storage header. The storage signature has the following structure:

Type	Field	Description
DWORD	*lSignature*	"Magic" signature for physical metadata, currently 0x424A5342
WORD	*iMajorVersion*	Major version (1 for the first release of the common language runtime)
WORD	*iMinorVersion*	Minor version (1 for the first release of the common language runtime)
DWORD	*iExtraData*	Reserved; set to 0
DWORD	*iLength*	Length of the version string
BYTE[]	*iVersionString*	Version string

The storage header follows the storage signature, aligned on a 4-byte boundary. Its structure is simple:

Type	Field	Description
BYTE	*fFlags*	Reserved; set to 0
BYTE		[*padding*]
WORD	*iStreams*	Number of streams

The storage header is followed by an array of stream headers. The structure of a stream header looks like this:

Type	Field	Description
DWORD	iOffset	Offset in the file for this stream
DWORD	*iSize*	Size of the stream in bytes
char[16]	*rcName*	Name of the stream; a zero-terminated ANSI string no longer than seven characters

Six named streams can be present in the metadata:

- ***#Strings*** A string heap containing the names of metadata items (class names, method names, field names, and so on). The stream does *not* contain literal constants defined or referenced in the methods of the module.
- ***#Blob*** A blob heap containing internal metadata binary objects, such as default values. This stream does *not* contain binary objects defined in the methods of the module.
- ***#GUID*** A GUID heap containing all sorts of globally unique identifiers.
- ***#US*** A blob heap containing user-defined strings. This stream contains string constants defined in the user code. The strings are kept in Unicode encoding. This stream's most interesting characteristic is that the user strings can be explicitly addressed by the IL code (with the *ldstr* instruction). In addition, because it is actually a blob heap, the *#US* heap can store not only Unicode strings but any binary object, which opens some intriguing possibilities.
- ***#~*** A compressed (optimized) metadata stream. This stream contains an optimized system of metadata tables.

- *#-* An uncompressed (unoptimized) metadata stream. This stream contains an unoptimized system of metadata tables, including the intermediate lookup tables (pointer tables).

> **Note** The streams *#~* and *#-* are mutually exclusive—that is, the metadata structure of the module is either optimized or unoptimized; it cannot be both at the same time.

If no items are stored in a stream, the stream is absent (null), and the *iStreams* field of the storage header is correspondingly reduced. At least three streams are guaranteed to be present: a metadata stream (*#~* or *#-*), a string stream (*#Strings*), and a GUID stream (*#GUID*). Metadata items must be present in at least minimal configuration in even the most trivial module, and these metadata items must have names and GUIDs.

Figure 4-3 illustrates the general structure of metadata. In Figure 4-4, you can see the way streams are referenced by other streams as well as by external "consumers" such as metadata APIs and the IL code.

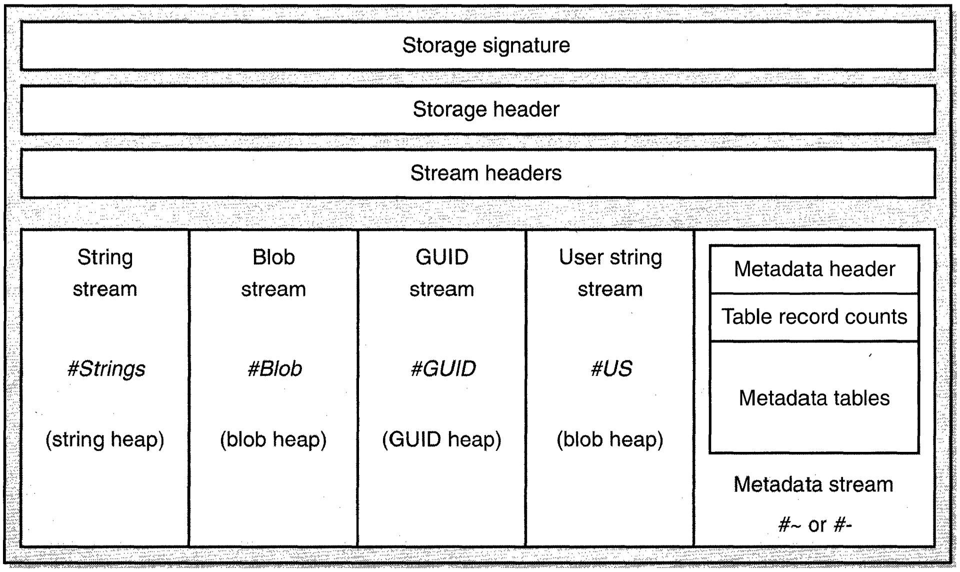


Figure 4-3 The general structure of metadata.

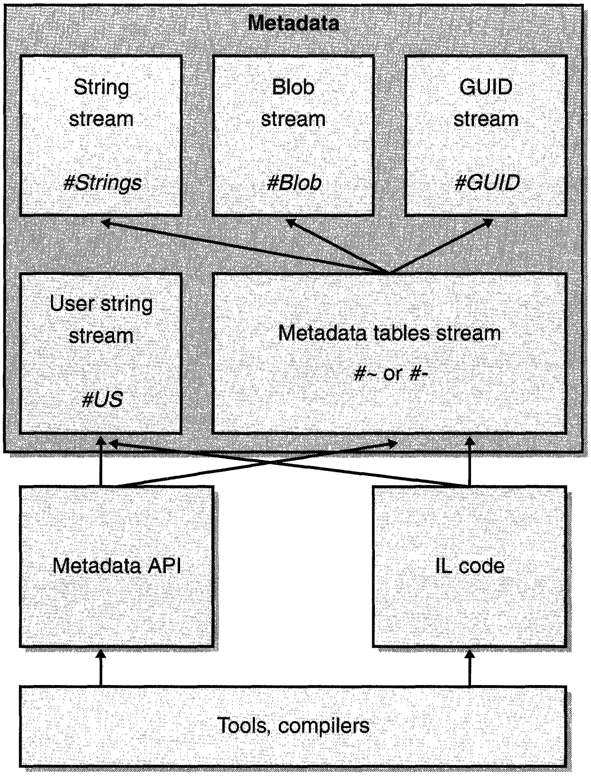


Figure 4-4 Stream referencing.

Metadata Table Streams

The metadata streams #~ and #- begin with the following header:

Size	Field	Description
4 bytes	*Reserved*	Reserved; set to 0.
1 byte	*Major*	Major version of the table schema (1 for the first release of the common language runtime).
1 byte	*Minor*	Minor version of the table schema (0 for the first release of the common language runtime).
1 byte	*Heaps*	Binary flags indicate the offset sizes to be used within the heaps. A 4-byte unsigned integer offset is indicated by 0x01 for a string heap, 0x02 for a GUID heap, and 0x04 for a blob heap. If a flag is not set, the respective heap offset is presumed to be a 2-byte unsigned integer.

Size	Field	Description
		A # stream can also have special flags set: flag 0x20, indicating that the stream contains only changes made during an edit-and-continue session, and flag 0x80, indicating that the metadata might contain items marked as deleted.
1 byte	*Rid*	Bit count of the maximal record index to all tables of the metadata; calculated at run time (during the metadata stream initialization).
8 bytes	*MaskValid*	Bit vector of present tables, each bit representing one table (1 if present).
8 bytes	*Sorted*	Bit vector of sorted tables, each bit representing a respective table (1 if sorted).

This header is followed by a sequence of 4-byte unsigned integers indicating the number of records in each table marked 1 in the *MaskValid* bit vector.

Like any database, metadata has a schema. The *schema* is a system of descriptors of metadata tables and columns—in this sense, it is "meta-metadata." A schema is not a part of metadata, nor is it an attribute of a managed PE file. Rather, a metadata schema is an attribute of the common language runtime and is hard-coded. It should not change in the future unless there's a major overhaul of the runtime.

Each metadata table has the following descriptors:

Type	Field	Description
pointer	*pColDefs*	Pointer to an array of column descriptors
BYTE	*cCols*	Number of columns in the table
BYTE	*iKey*	Index of the key column
WORD	*cbRec*	Size of a record in the table

Column descriptors, to which the *pColDefs* fields of table descriptors point, have the following structure:

Type	Field	Description
BYTE	*Type*	Code of the column's type
BYTE	*oColumn*	Offset of the column
BYTE	*cbColumn*	Size of the column in bytes

Type, the first field of a column descriptor, is especially interesting. The metadata schema of the first release of the common language runtime identifies the following codes for column types:

0–63 Column holds the record index (RID) to another table; the specific value indicates which table. The width of the column is defined by the *Rid* field of the metadata stream header.

64–95 Column holds a coded token referencing another table; the specific value indicates the type of coded token. Tokens are references carrying the indexes of both the table and the record being referenced. The table being addressed and the index of the record are defined by the coded token value.

96 Column holds a 2-byte signed integer.

97 Column holds a 2-byte unsigned integer.

98 Column holds a 4-byte signed integer.

99 Column holds a 4-byte unsigned integer.

100 Column holds a 1-byte unsigned integer.

101 Column holds an offset in the string heap (the *#Strings* stream).

102 Column holds an offset in the GUID heap (the *#GUID* stream).

103 Column holds an offset in the blob heap (the *#Blob* stream).

The metadata schema defines 44 tables. Given the range of RID type codes, the common language runtime definitely has room for growth. At the moment, the following tables are defined:

- **Module** The current module descriptor.
- **TypeRef** Class reference descriptors.
- **TypeDef** Class or interface definition descriptors.
- **FieldPtr** A class-to-fields lookup table, which does not exist in optimized metadata (#~ stream).
- **Field** Field definition descriptors.
- **MethodPtr** A class-to-methods lookup table, which does not exist in optimized metadata (#~ stream).
- **Method** Method definition descriptors.
- **ParamPtr** A method-to-parameters lookup table, which does not exist in optimized metadata (#~ stream).
- **Param** Parameter definition descriptors.
- **InterfaceImpl** Interface implementation descriptors.

- **MemberRef** Member (field or method) reference descriptors.
- **Constant** Constant value descriptors that map the default values stored in the *#Blob* stream to respective fields, parameters, and properties.
- **CustomAttribute** Custom attribute descriptors.
- **FieldMarshal** Field or parameter marshaling descriptors for managed/unmanaged interoperations.
- **DeclSecurity** Security descriptors.
- **ClassLayout** Class layout descriptors that hold information about how the loader should lay out respective classes.
- **FieldLayout** Field layout descriptors that specify the offset or sequencing of individual fields.
- **StandAloneSig** Stand-alone signature descriptors. Signatures per se are used in two capacities: as composite signatures of local variables of methods, and as parameters of the call indirect (*calli*) IL instruction.
- **EventMap** A class-to-events mapping table. This is not an intermediate lookup table, and it does exist in optimized metadata.
- **EventPtr** An event-map-to-events lookup table, which does not exist in optimized metadata (#~ stream).
- **Event** Event descriptors.
- **PropertyMap** A class-to-properties mapping table. This is not an intermediate lookup table, and it does exist in optimized metadata.
- **PropertyPtr** A property-map-to-properties lookup table, which does not exist in optimized metadata (#~ stream).
- **Property** Property descriptors.
- **MethodSemantics** Method semantics descriptors that hold information about which method is associated with a specific property or event and in what capacity.
- **MethodImpl** Method implementation descriptors.

- **ModuleRef** Module reference descriptors.
- **TypeSpec** Type specification descriptors.
- **ImplMap** Implementation map descriptors used for the platform invocation (*P/Invoke*) type of managed/unmanaged code interoperation.
- **FieldRVA** Field-to-data mapping descriptors.
- **ENCLog** Edit-and-continue log descriptors that hold information about what changes have been made to specific metadata items during in-memory editing. This table does not exist in optimized metadata (#~ stream).
- **ENCMap** Edit-and-continue mapping descriptors. This table does not exist in optimized metadata (#~ stream).
- **Assembly** The current assembly descriptor, which should appear only in prime module metadata.
- **AssemblyProcessor** This table is unused in the first release of the runtime.
- **AssemblyOS** This table is unused in the first release of the runtime.
- **AssemblyRef** Assembly reference descriptors.
- **AssemblyRefProcessor** This table is unused in the first release of the runtime.
- **AssemblyRefOS** This table is unused in the first release of the runtime.
- **File** File descriptors that contain information about other files in the current assembly.
- **ExportedType** Exported type descriptors that contain information about public classes exported by the current assembly, which are declared in other modules of the assembly. Only the prime module of the assembly should carry this table.
- **ManifestResource** Managed resource descriptors.
- **NestedClass** Nested class descriptors that provide mapping of nested classes to their respective enclosing classes.
- **TypeTyPar** Reserved for future use.
- **MethodTyPar** Reserved for future use.

The structural aspects of the various tables and their validity rules are discussed in later chapters, along with the corresponding ILAsm constructs.

RIDs and Tokens

Record indexes and tokens are the unsigned integer values used for indexing the records in metadata tables. RIDs are simple indexes, applicable only to an explicitly specified table, and tokens carry the information identifying metadata tables they reference.

RIDs

A RID is a record identifier, which is simply a one-based row number in the table containing the record. The range of valid RIDs stretches from 1 to the record count of the addressed table, inclusive. RIDs are used in metadata internally only; metadata emission and retrieval APIs do not use RIDs as parameters.

The RID column type codes (0–63) serve as zero-based table indexes. Thus the type of the column identifies the referenced table, while the value of the table cell identifies the referenced record. This works fine as long as we know that a particular column always references one particular table and no other. Now if we only could combine RID with table identification.

Tokens

Actually, we can. The combined identification entity, referred to as a *token*, is used in all metadata APIs and in all IL instructions. A token is a 4-byte unsigned integer whose senior byte carries a zero-based table index (the same as the internal metadata RID type). The remaining 3 bytes are left for the RID.

There is a significant difference between token types and internal metadata RID types, however: whereas internal RID types cover all metadata tables, the token types are defined for only a limited subset of the tables, as noted in Table 4-2.

Table 4-2 Token Types and Their Referenced Tables

Token Type	Value (RID Type << 24)	Referenced Table
mdtModule	0x00000000	Module
mdtTypeRef	0x01000000	TypeRef
mdtTypeDef	0x02000000	TypeDef
mdtFieldDef	0x04000000	Field

(continued)

Table 4-2 Token Types and Their Referenced Tables *(continued)*

Token Type	Value (RID Type << 24)	Referenced Table
mdtMethodDef	0x06000000	Method
mdtParamDef	0x08000000	Param
mdtInterfaceImpl	0x09000000	InterfaceImpl
mdtMemberRef	0x0A000000	MemberRef
mdtCustomAttribute	0x0C000000	CustomAttribute
mdtPermission	0x0E000000	DeclSecurity
mdtSignature	0x11000000	StandAloneSig
mdtEvent	0x14000000	Event
mdtProperty	0x17000000	Property
mdtModuleRef	0x1A000000	ModuleRef
mdtTypeSpec	0x1B000000	TypeSpec
mdtAssembly	0x20000000	Assembly
mdtAssemblyRef	0x23000000	AssemblyRef
mdtFile	0x26000000	File
mdtExportedType	0x27000000	ExportedType
mdtManifestResource	0x28000000	ManifestResource

The 24 tables that do not have associated token types are not intended to be accessed from "outside," through metadata APIs or from IL code. These tables (excluding the TypeTyPar and MethodTyPar tables, which are reserved for future use) are of an auxiliary or intermediate nature and should be accessed indirectly only, through the references contained in the "exposed" tables, which have associated token types.

The validity of these tokens can be defined simply: a valid token has a type from Table 4-2, and it has a valid RID—that is, a RID in the range 1 to the record count of the table of a specified type.

An additional token type, quite different from the types listed in Table 4-2, is *mdtString* (0x70000000). Tokens of this type are used to refer to the user-defined Unicode strings stored in the *#US* stream.

Both the type component and the RID component of user-defined string tokens differ from those of metadata table tokens. The type component of a user-defined string token (0x70) has nothing to do with column types (the maximal column type is 103 = 0x67), which is not surprising, considering that no column type corresponds to an offset in the *#US* stream. Because metadata tables never reference the user-defined strings, it's not necessary to define a column type for the strings. In addition, the RID component of a user-defined string token does not represent a RID because no table is being referenced. Instead, the 3 lower bytes of a user-defined string token hold an offset in the *#US* stream.

The definition of the validity of a user-defined string token is more complex. The RID component is valid if it is greater than 0 and if the string it defines starts at a 4-byte boundary and is fully contained within the *#US* stream. The last condition is checked in the following way: The bytes at the offset specified by the RID component of the token are interpreted as the compressed length of the string. (Don't forget that the *#US* stream is a blob heap.) If the sum of the offset and the size of compressed length brings us to a 4-byte boundary, and if this sum plus the calculated length are within the *#US* stream size, everything is fine and the token is valid.

Coded Tokens

The discussion thus far has focused on the "external" form of tokens. You have every right to suspect that the "internal" form of tokens, used inside the metadata, is different—and it is.

Why can't the external form also be used as internal? Because the external tokens are huge. Imagine, 4 bytes for each token, when we fight for each measly byte, trying to squeeze the metadata into as small a footprint as possible. (Bandwidth! Don't forget about the bandwidth!) Compression? Alas, because of the type component occupying the senior byte, external tokens represent very large unsigned integers and thus cannot be efficiently compressed, even though their middle bytes are full of zeros. We need a fresh approach.

The internal encoding of tokens is based on a simple idea: A column must be given a token type only if it might reference several tables. (Columns referencing only one table have a respective RID type.) But any such column certainly does not need to reference *all* the tables.

So our first task is to identify which group of tables each such column might reference and form a set of such groups. Let's assign each group a number, which will be a coded token type of the column. Because coded token types occupy a range from 64 to 95, we can define up to 32 groups.

Now, every group contains two or more table types. Let's enumerate them within the group and see how many bits we will need for this enumeration. This bit count will be a characteristic of the group and hence of the respective coded token type. The number assigned to a table within the group is called a *tag*.

This tag plays a role roughly equivalent to that of the type component of an external token. But, unwilling to once again create large tokens full of zeros, we will this time put the tag not in the most significant bits of the token but rather in the least significant bits. Then let's left-shift the RID *n* bits and add the left-shifted RID to the tag, where *n* is the bit width of the tag. Now we've got a coded token.

What about the coded token size? We know which metadata tables form each group, and we know the record count of each table, so we know the maximal possible RID within the group. Say, for example, that we would need *m* bits to encode the maximal RID. If we can fit the maximal RID (*m* bits) and the tag (*n* bits) into a 2-byte unsigned integer (16 bits), we win, and the coded token size for this group will be 2 bytes. If we can't, we are out of luck and will have to use 4-byte coded tokens for this group. No, we won't even consider 3 bytes—it's unbecoming.

To summarize, a coded token type has the following attributes:

- Number of referenced tables (part of the schema)
- Array of referenced table IDs (part of the schema)
- Tag bit width (part of the schema, derived from the number of referenced tables)
- Coded token size, either 2 or 4 bytes (computed at the metadata opening time from the tag width and the maximal record count among the referenced tables)

Table 4-3 lists the twelve coded token types defined in the metadata schema of the first release of the common language runtime.

Table 4-3 Coded Token Types

Coded Token Type	Tag
TypeDefOrRef (64): 3 referenced tables, tag size 2	
TypeDef	0
TypeRef	1
TypeSpec	2
HasConstant (65): 3 referenced tables, tag size 2	
Field	0
Param	1
Property	2
HasCustomAttribute (66): 19 referenced tables, tag size 5	
Method	0
Field	1
TypeRef	2
TypeDef	3

Table 4-3 Coded Token Types *(continued)*

Coded Token Type	Tag
Param	4
InterfaceImpl	5
MemberRef	6
Module	7
DeclSecurity	8
Property	9
Event	10
StandAloneSig	11
ModuleRef	12
TypeSpec	13
Assembly	14
AssemblyRef	15
File	16
ExportedType	17
ManifestResource	18
HasFieldMarshal (67): 2 referenced tables, tag size 1	
Field	0
Param	1
HasDeclSecurity (68): 3 referenced tables, tag size 2	
TypeDef	0
Method	1
Assembly	2
MemberRefParent (69): 5 referenced tables, tag size 3	
TypeDef	0
TypeRef	1
ModuleRef	2
Method	3
TypeSpec	4
HasSemantics (70): 2 referenced tables, tag size 1	
Event	0
Property	1
MethodDefOrRef (71): 2 referenced tables, tag size 1	
Method	0

(continued)

Table 4-3 Coded Token Types *(continued)*

Coded Token Type	Tag
MemberRef	1
MemberForwarded (72): 2 referenced tables, tag size 1	
Field	0
Method	1
Implementation (73): 3 referenced tables, tag size 2	
File	0
AssemblyRef	1
ExportedType	2
CustomAttributeType (74): 5 referenced tables, tag size 3	
TypeRef	0
TypeDef	1
Method	2
MemberRef	3
String	4
ResolutionScope (75): 4 referenced tables, tag size 2	
Module	0
ModuleRef	1
AssemblyRef	2
TypeRef	3

Note The coded token type range (64–95) provides room to add another twenty types in the future, should it ever become necessary.

Coded tokens are part of metadata's internal affairs. The ILAsm compiler, like all other compilers, never deals with coded tokens. Compilers and other tools read and emit metadata through the metadata import and emission APIs, either directly or through managed wrappers provided in the .NET Framework class library—*System.Reflection* for metadata import and *System.Reflection.Emit* for metadata emission. The metadata APIs automatically convert standard 4-byte tokens to and from coded tokens. IL code also uses only standard 4-byte tokens.

Nonetheless, the preceding definitions are useful to us for two reasons. First, we will need them when we discuss individual metadata tables in later chapters. Second, these definitions provide a good hint about the nature of relationships between the metadata tables.

Metadata Validation

This "good hint," however, is merely a hint. The definitions in the preceding section provide information about which tables we *can* reference from a column of a certain type. It does not mean that we *should* reference all the tables we can. Some of the groups of token types listed in Table 4-3 are wider than is actually acceptable in the first release of the common language runtime. For example, the *MemberRefParent* group, which describes the tables that can contain the parents of a *MemberRef* record, includes the TypeDef table. But the metadata emission APIs will not accept a *TypeDef* token as the parent token of a *MemberRef*; and even if such metadata was somehow emitted, the loader would reject it.

Even APIs provide very few safeguards (most of them fairly trivial) as far as metadata validity is concerned. Metadata is an extremely complex system, and literally hundreds of validity rules need to be enforced.

High-level language compilers, such as Microsoft Visual Basic .NET or Microsoft Visual C# .NET compilers, provide a significant level of protection against invalid metadata emission because they shield the actual metadata specification and emission from programmers. Because high-level languages are concept-driven and concept-based, and it is the compiler's duty to relate the language concepts to the metadata structures and IL code constructs, a compiler can be built to emit valid structures and constructs. (Well, more or less.) On the other hand, ILAsm, like other assemblers, is a platform-oriented language and allows a programmer to generate an enormously wide range of metadata structures and IL constructs, only a fraction of which represent a valid subset.

In view of this bleak situation, we need to rely on external validation and verification tools. (Speaking of "validation and verification" is not an exercise in tautology—the term *validation* is usually applied to metadata, and *verification* to IL code.) One such tool is the common language runtime itself. The loader tests metadata against many of the validity rules, especially those whose violation could break the system. The runtime subsystem responsible for JIT compilation performs IL code verification. These processes are referred to as *run-time validation and verification*.

PEVerify, a stand-alone tool included in the .NET Framework SDK, offers more exhaustive validation and verification. PEVerify employs two independent subsystems, MDValidator and ILVerifier. MDValidator can also be invoked through the IL Disassembler.

You can find information about PEVerify and the IL Disassembler in the appendixes. Later chapters discuss various validity rules along with the related metadata structures and IL constructs.

Summary

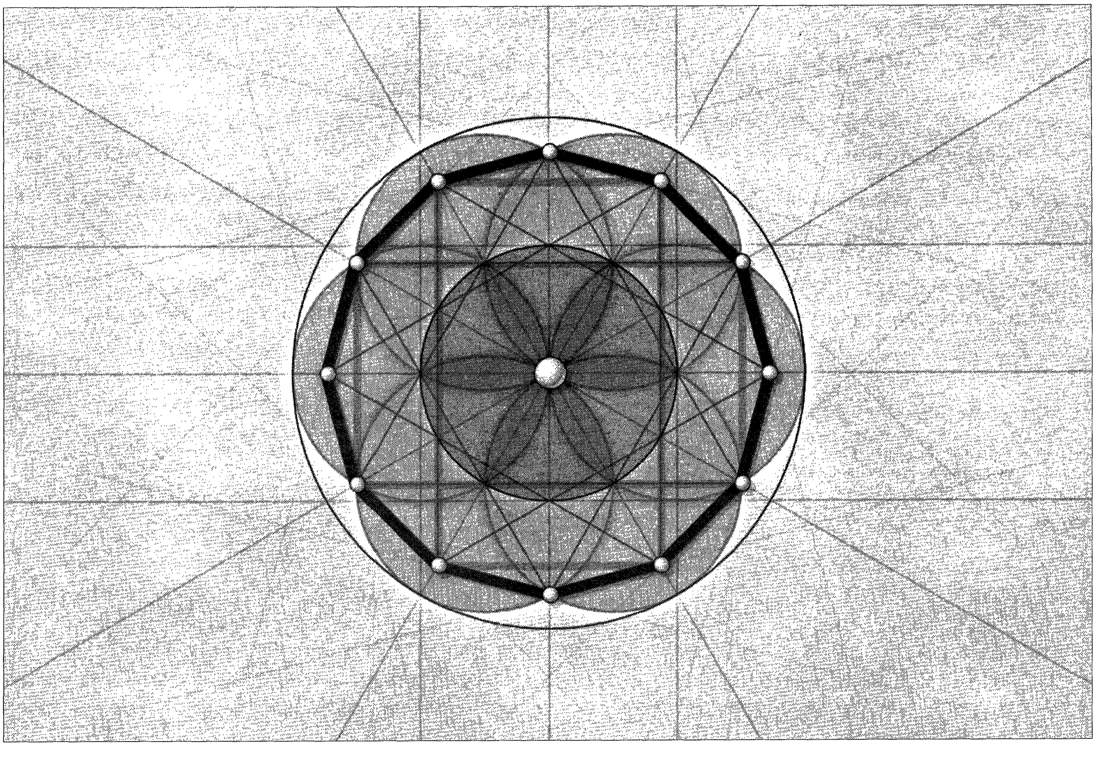
Now that we know how the metadata is organized in principle, we are ready to examine the particular metadata items and the tables representing them. All further considerations shall concentrate on four metadata streams—*#Strings*, *#Blob*, *#US*, and *#~*—because the *#GUID* stream is referenced in one metadata table only (the Module table) and the *#-* stream (unoptimized metadata) is never emitted by the ILAsm compiler.

Here's some advice for those of you who wonder if it would be a good idea to spoof the metadata header to get access to the data beyond the metadata under the pretense of manipulating the metadata: forget it. The runtime loader has safeguards analyzing the consistency of the metadata headers and the metadata itself. If an inconsistency is detected, the loader refuses to open the metadata streams. Tinkering with the metadata headers does not lead to erroneous or unpredictable behavior of the module; instead, it renders the module unloadable, period.

And on this cheerful note, let's proceed to discussion of the "real" metadata items.

Part III

Fundamental Components



5

Modules and Assemblies

This chapter discusses the organization, deployment, and execution of assemblies and modules. It also provides a detailed examination of the metadata segment responsible for assembly and module identity and interaction: the manifest. As you might recall from Chapter 1, "Simple Sample," an assembly can include several modules. Any module of a multimodule assembly can—and does, as a rule—carry its own manifest, but only one module per assembly carries the manifest that contains the assembly's identity. This module is referred to as the *prime* module. Thus each assembly, whether multimodule or single-module, contains only one prime module.

What Is an Assembly?

An assembly is basically a deployment unit, a building block of a managed application. Assemblies are reusable, allowing different applications to use the same assembly. Assemblies carry a full self-description in their metadata, including version information that allows the common language runtime to use a specific version of an assembly for a particular application.

This arrangement eliminates what's known as "DLL Hell," the situation created when upgrading one application renders another application inoperative because both happen to use the same DLL(s). A typical example of DLL Hell occurred with the release of the game Microsoft Age of Empires II, a sequel to Age of Empires. Because the sequel used a more advanced version of the Microsoft DirectX DLL, which was incompatible with Age of Empires, the original game ceased to work when the sequel was installed. To deal with the situation, Microsoft had to issue a new version of the DirectX DLL that was consumable by both games.

Private and Shared Assemblies

Assemblies are classified as either *private* or *shared*. Structurally and functionally, these two kinds of assemblies are the same, but they differ in how they are named and deployed and in the level of version checks performed by the loader.

A *private assembly* is considered part of a particular application, not intended for use by other applications. A private assembly is deployed in the same directory as the application or in a subdirectory of this directory. This kind of deployment shields the private assembly from other applications, which should not have access to it.

Being part of a particular application, a private assembly is usually created by the same author (person, group, or organization) as other components specific to this application and is thus considered to be primarily the author's responsibility. Consequently, naming and versioning requirements are relaxed for private assemblies, and the common language runtime does not enforce these requirements. The name of a private assembly must be unique within the application.

A *shared assembly* is not part of a particular application and is designed to be used widely by various applications. Shared assemblies are usually authored by groups or organizations other than those responsible for the applications that use these assemblies. A prominent example of shared assemblies is the set of assemblies constituting the Microsoft .NET Framework class library.

As a result of such positioning, the naming and versioning requirements for shared assemblies are much stricter than those for private assemblies. Names of shared assemblies must be globally unique. Additional assembly identification is provided by *strong names*, which use cryptographic public/private key pairs to ensure the name's uniqueness and to prevent name spoofing. A strong name also provides the consumer of the shared assembly with information about the identity of the assembly publisher. If the common language runtime cryptographic checks pass, the consumer can be sure that the assembly comes from the expected publisher, assuming that the publisher's private encryption key was not compromised.

Shared assemblies are deployed into the global assembly cache (GAC). The GAC stores multiple versions of shared assemblies side by side. The loader typically looks for the shared assemblies in the GAC.

Note Under some circumstances, an application might need to deploy a shared assembly in its directory to ensure that the appropriate version is loaded. In such a case, the shared assembly is being used as a private assembly, so it is not in fact shared, whether it is strong-named or not.

Application Domains as Logical Units of Execution

Operating systems and run times typically provide some form of isolation between applications running on the system. This isolation is necessary to ensure that code running in one application cannot adversely affect other, unrelated applications. In modern operating systems, this isolation is achieved by using process boundaries, where a process, occupying a unique virtual address space, runs exactly one application and scopes the resources that are available for that process to use.

Managed code execution has similar needs for isolation. Such isolation can be provided at lower cost in a managed application, however, considering that managed applications run under the control of the common language runtime and are verified to be type-safe.

The runtime allows multiple applications to be run in a single operating system process, using a construct called an *application domain* to isolate the applications from one another. In many respects, application domains are the common language runtime equivalent of an operating system process.

Specifically, isolation in managed applications means the following:

- Different security levels can be assigned to each application domain, giving the host a chance to run the applications with varying security requirements in one process.
- Applications can be independently stopped and debugged.
- Code running in one application cannot directly access code or resources from another application. (Doing so could introduce a security hole.)

- Faults in one application cannot affect other applications by bringing down the entire process.
- Each application has control over where the code loaded on its behalf comes from and what version the code being loaded is. In addition, configuration information is scoped by the application.

The following examples describe scenarios in which it is useful to run multiple applications in the same process:

- ASP.NET runs multiple Web applications in the same process. In ASP/IIS (Internet Information Services), application isolation was achieved by process boundaries, which proved too expensive to scale appropriately.
- Microsoft Internet Explorer runs code from multiple sites in the same process as the browser code itself. Obviously, code from one site should not be able to affect code from another site.
- Database engines need to run code from multiple user applications in the same process.
- Application server products might need to run code from multiple applications in a single process.

Hosting environments such as ASP.NET or Internet Explorer need to run managed code on behalf of the user and take advantage of the application isolation features provided by application domains. In fact, it is the host that determines where the application domain boundaries lie and in what domain user code is run, as these examples show:

- ASP.NET creates application domains to run user code. Domains are created per application as defined by the Web server.
- Internet Explorer by default creates one application domain per site (although developers can customize this behavior).
- In Shell EXE, each application launched from the command line runs in a separate application domain occupying one process.
- Microsoft Visual Basic for Applications (VBA) uses the default application domain of the process to run the script code contained in a Microsoft Office document.
- Windows Foundation Classes (WFC) Forms Designer creates a separate application domain for each form being built. When a form is edited and rebuilt, the old application domain is shut down, the code is recompiled, and a new application domain is created.

Because isolation demands that the code or resources of one application must not be directly accessible from code running in another application, no direct calls are allowed between objects in different application domains. Cross-domain communications are limited to passing the objects, which are either copied or accessed via proxy and which fall into one of the following three categories:

- **Unbound objects** are marshaled by value across domains. This means that the receiving domain gets a copy of the object to play with instead of the original object.
- **AppDomain-bound objects** are marshaled by reference across domains, which means that cross-domain access is always accomplished through proxies.
- **Context-bound objects** are also marshaled by reference across domains as well as between contexts within the same domain.

The common language runtime relies on the verifiable type safety of the code to provide fault isolation between domains at a much lower cost than that incurred by the process isolation used in operating systems. Because isolation is based on static type verification, hardware ring transitions or process switches are not necessary.

Manifest

The metadata that describes an assembly and its modules is referred to as a *manifest*. The manifest carries the following information:

- **Identity**, including a simple textual name, an assembly version number, an optional culture if the assembly contains localized managed resources, and an optional public key if the assembly is strong-named. This information is defined in two metadata tables: Module and Assembly (in the prime module only).
- **Contents**, including types and managed resources exposed by this assembly for external use and the location of these types and resources. The metadata tables that contain this information are ExportedType (in the prime module only) and ManifestResource.
- **Dependencies**, including other (external) assemblies this assembly references and, in the case of a multimodule assembly, other modules of the same assembly. You can find the dependency information in these metadata tables: AssemblyRef, ModuleRef, and File.

- **Requested permissions**, specific to the assembly as a whole. More specific requested permissions might also be defined for certain types (classes) and methods. This information is defined in the DeclSecurity metadata table. (Chapter 14, "Security Attributes," describes requested permissions and their declaration.)
- **Custom attributes**, specific to the manifest components. Custom attributes provide additional information used by compilers and other tools. The common language runtime recognizes a limited number of custom attributes. Custom attributes are defined in the CustomAttribute metadata table. (Refer to Chapter 13, "Custom Attributes," for more information on this topic.)

The diagram in Figure 5-1 shows the mutual references that take place between the metadata tables constituting the manifest.

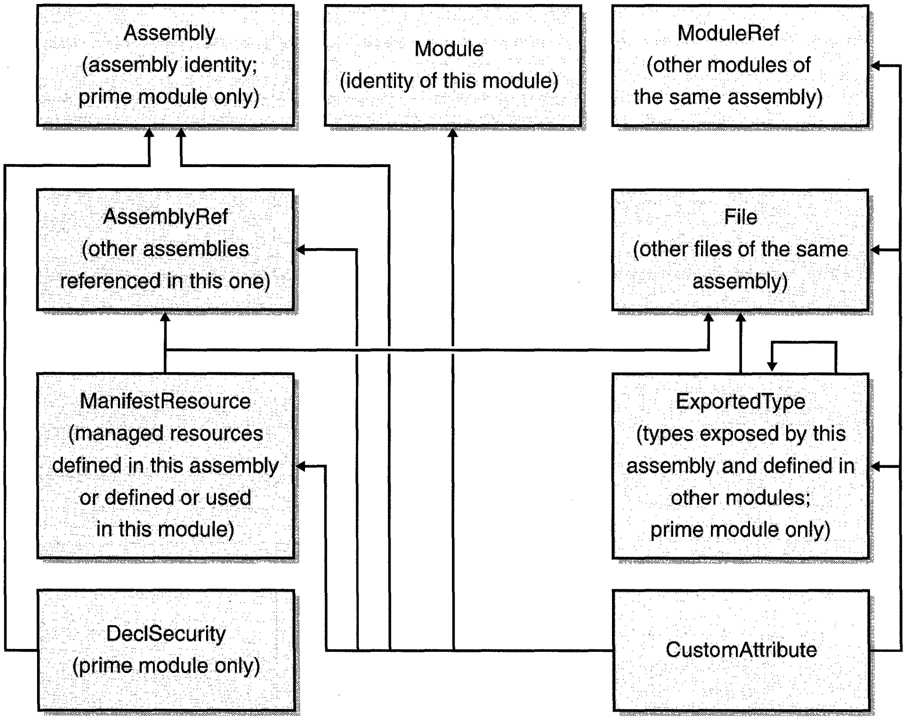


Figure 5-1 Mutual references between the manifest's metadata tables.

Assembly Metadata Table and Declaration

The Assembly metadata table contains at most one record, which appears in the prime module's metadata. The table has the following column structure:

- ***HashAlgId* (4-byte unsigned integer)** The ID of the hash algorithm used in this assembly to hash the files. The value must be one of the *CALG_** values defined in the header file Wincrypt.h. The default hash algorithm is *CALG_SHA* (a.k.a. *CALG_SHA1*) (0x8004). ECMA specifications consider this algorithm to be standard, offering the best widely available technology for file hashing.
- ***MajorVersion* (2-byte unsigned integer)** The major version of the assembly.
- ***MinorVersion* (2-byte unsigned integer)** The minor version of the assembly.
- ***BuildNumber* (2-byte unsigned integer)** The build number of the assembly.
- ***RevisionNumber* (2-byte unsigned integer)** The revision number of the assembly.
- ***Flags* (4-byte unsigned integer)** Assembly flags indicating limitations on running different versions of this assembly side by side.
- ***PublicKey* (offset in the *#Blob* stream)** A binary object representing a public encryption key for a strong-named assembly.
- ***Name* (offset in the *#Strings* stream)** The assembly name, which must be nonempty and must not contain a path or a filename extension.
- ***Locale* (offset in the *#Strings* stream)** The culture (formerly known as locale) name, such as *en-US* (American English) or *fr-CA* (Canadian French). The culture name must match one of hundreds of culture names "known" to the runtime through the .NET Framework class library, but this validity rule is rather meaningless: to use a culture, the specific language support must be installed on the target machine. If the language support is not installed, it doesn't matter whether the culture is "known" to the runtime.

In ILAsm, the *Assembly* is declared in the following way:

```
.assembly <flags> <name> { <assemblyDecl>* }
```

where *<flags>* ::=

```
<none> // No limitations on side-by-side running of the assembly
| noappdomain  // No side-by-side running within one AppDomain
| noprocess  // No side-by-side running within one process
| nomachine // No side-by-side running on the same machine
```

and *<assemblyDecl>* ::=

```
.hash algorithm <int32>  // Set hash algorithm ID
| .ver <int32>:<int32>:<int32>:<int32> // Set version numbers
| .publickey = ( <bytes> ) // Set public encryption key
| .locale <quotedString> // Set assembly culture
| <securityDecl> // Set requested permissions
| <customAttrDecl> // Define custom attribute(s)
```

In this declaration, *<int32>* denotes an integer number, at most 4 bytes in size. The notation *<bytes>* represents a sequence of two-digit hexadecimal numbers, each representing 1 byte; this form, *bytearray*, is often used in ILAsm to represent binary objects of arbitrary size. Finally, *<quotedString>* denotes, in general, a composite quoted string—that is, a construct such as *"ABC"+"DEF"+"GHI"*. The concatenation with the plus sign is useful for defining very long strings, although in this case we don't need concatenation for strings such as *en-US* or *nl-BE*.

Note In addition to the three flags related to side-by-side execution, three more, which are not relevant to the discussion at hand, are available. One indicates whether the assembly holds a full public key. This flag is never set explicitly; rather, it is set when a *PublicKey* entry is defined. The other two flags, *EnableJITcompilerTracking* and *DisableJITcompileroptimizer*, are related to the debug mode of the JIT (just-in-time) compiler and are set at the module load time.

AssemblyRef Metadata Table and Declaration

The AssemblyRef (assembly reference) metadata table defines the external dependencies of an assembly or a module. Both prime and nonprime modules can—and do, as a rule—contain this table. The only assembly that does not depend on any other assembly, and hence has an empty AssemblyRef table, is Mscorlib.dll, the root assembly of the .NET Framework class library.

The column structure of the AssemblyRef table is as follows:

- ***MajorVersion* (2-byte unsigned integer)** The major version of the assembly.
- ***MinorVersion* (2-byte unsigned integer)** The minor version of the assembly.
- ***BuildNumber* (2-byte unsigned integer)** The build number of the assembly.
- ***RevisionNumber* (2-byte unsigned integer)** The revision number of the assembly.
- ***Flags* (4-byte unsigned integer)** Assembly reference flags, which indicate whether the assembly reference holds a full unhashed public key or a "surrogate" (public key token).
- ***PublicKeyOrToken* (offset in the *#Blob* stream)** A binary object representing a public encryption key for a strong-named assembly or a token of this key. A key token is an 8-byte representation of a hashed public key.
- ***Name* (offset in the *#Strings* stream)** A referenced assembly name, which must be nonempty and must not contain a path or a filename extension.
- ***Locale* (offset in the *#Strings* stream)** The culture name.
- ***HashValue* (offset in the *#Blob* stream)** A binary object representing a hash of the metadata of the referenced assembly's prime module. Because this value is ignored by the loader in the first release of the common language runtime, it can safely be omitted.

In ILAsm, an *AssemblyRef* is declared in the following way:

```
.assembly extern <name> { <assemblyRefDecl>* }
```

where *<assemblyRefDecl>* ::=

```
| .ver <int32>:<int32>:<int32>:<int32> // Set version numbers
| .publickey = ( <bytes> ) // Set public encryption key
| .publickeytoken = ( <bytes> ) // Set public encryption key token
| .locale <quotedString> // Set assembly locale
| .hash = ( <bytes> ) // Set hash value
| <customAttrDecl> // Define custom attribute(s)
```

As you might have noticed, ILAsm does not provide a way to set the flags in the *AssemblyRef* declaration. The explanation is simple: the only flag relevant to an *AssemblyRef* is the flag indicating whether the *AssemblyRef* carries a full unhashed public encryption key, and this flag is set only when the *.publickey* directive is used.

When referencing a strong-named assembly, you are required to specify *.publickeytoken* (or *.publickey*, which is rarely used in *AssemblyRefs*) and *.ver*. The only exception to this rule among the strong-named assemblies is Mscorlib.dll.

If *.locale* is not specified, the referenced assembly is presumed to be "culture-neutral."

An interesting situation arises when we need to use two or more versions of the same assembly side by side. An assembly is identified by its name, version, public key (or its token), and culture. It would be extremely cumbersome to list all these identifications every time we reference an assembly: "I want to call method *Bar* of class *Foo* from assembly *SomeOtherAssembly*, and I want the version number such-and-such, the culture *nl-BE*, and …" Of course, if we didn't need to use different versions side by side, we could simply refer to an assembly by name.

ILAsm provides an *AssemblyRef* aliasing mechanism to deal with such situations. The *AssemblyRef* declaration can be extended as shown here:

```
.assembly extern <name> as <alias> { <assemblyRefDecl>* }
```

And whenever we need to reference this assembly, we can use its *<alias>*, as seen in this example:

```
.assembly extern SomeOtherAssembly as OldSomeOther
{ .ver 1:1:1:1 }
.assembly extern SomeOtherAssembly as NewSomeOther
{ .ver 1:3:2:1 }
⋮
call int32 [OldSomeOther]Foo::Bar(string)
⋮
call int32 [NewSomeOther]Foo::Bar(string)
⋮
```

The alias is not a part of metadata. Rather, it is simply a language tool, needed to identify a particular *AssemblyRef* among several same-name *AssemblyRefs*. IL Disassembler generates aliases for *AssemblyRefs* whenever it finds same-name *AssemblyRefs* in the module metadata.

The Loader in Search of Assemblies

When we define an *AssemblyRef* in the metadata, we expect the loader to find exactly this assembly and load it into the application domain. Let's have a look at the process of finding an external assembly and binding it to the referencing application.

Given an *AssemblyRef*, the process of binding to that assembly is influenced by these factors:

- The application base (*AppBase*), which is a URL to the referencing application location (that is, to the directory in which your application is located). For executables, this is the directory containing the EXE file. For Web applications, the *AppBase* is the root directory of the application as defined by the Web server.
- Version policies specified by the application, by the publisher of the shared assembly being referenced, or by the administrator.
- Any additional search path information given in the application configuration file.
- Any code base (*CodeBase*) locations provided in the configuration files by the application, the publisher, or the administrator. The *CodeBase* is a URL to the location of the referenced external assembly.
- Whether the reference is to a shared assembly with a strong name or to a private assembly.

As illustrated in Figure 5-2, the loader performs the following steps to locate a referenced assembly:

1. Initiate the binding. Basically, this means taking the relevant *AssemblyRef* record from the metadata and seeing what it holds—its external assembly name, whether it is strong-named, whether culture is specified, and so on.
2. Apply the version policies, which are statements made by the application, by the publisher of the shared assembly being referenced, or by the administrator. These statements are contained in XML configuration files and simply redirect references to a particular version (or set of versions) of an assembly to a different version.

The .NET Framework retrieves its configuration from a set of configuration files. Each file represents settings that have different scopes. For example, the configuration file supplied with the installation of the common language runtime has settings that can affect all applications that use that version of the runtime. The configuration file supplied with an application has settings that affect only that one application.

3. Check the *CodeBase*. Now that the common language runtime knows which version of the assembly it is looking for, it begins the process of locating it. If the *CodeBase* has been supplied (in the same XML configuration file), it points the runtime directly at the executable to load; otherwise, the runtime needs to look in the *AppBase* and the GAC, as described in step 4. If the executable specified by the *CodeBase* matches the assembly reference, the process of finding the assembly is complete, and the external assembly can be loaded. In fact, even if the executable specified by the *CodeBase* does *not* match the reference, the common language runtime stops searching. In this case, of course, the search is considered a failure, and no assembly load follows.

4. Check the GAC or the *AppBase* or both. If the *CodeBase* hasn't been supplied, the remainder of the process depends on whether the referenced assembly is private or strong-named.

   If the reference is to a private assembly, the process probes the *AppBase*. The probing involves consecutive searching in the directories defined by the *AppBase*, the private binary path (binpath) from the same XML configuration file, the culture of the referenced assembly, and its name. The *AppBase* plus directories specified in the binpath form a set of root directories *{<root$_k$>, k=1...N}*. If the *AssemblyRef* specifies the culture, the search is performed in directories *<root$_k$>/<culture>* and then in *<root$_k$>/<culture>/<name>*; otherwise, the directories *<root$_k$>* and then *<root$_k$>/<name>* are searched. When searching for a private assembly, the process ignores the version numbers. If the assembly is not found by probing, the binding fails.

   If the assembly is strong-named, the process first looks in the global assembly cache. If the strong-named assembly is not found in the GAC, the process probes the *AppBase* as just described, and in this case it also checks the version numbers.

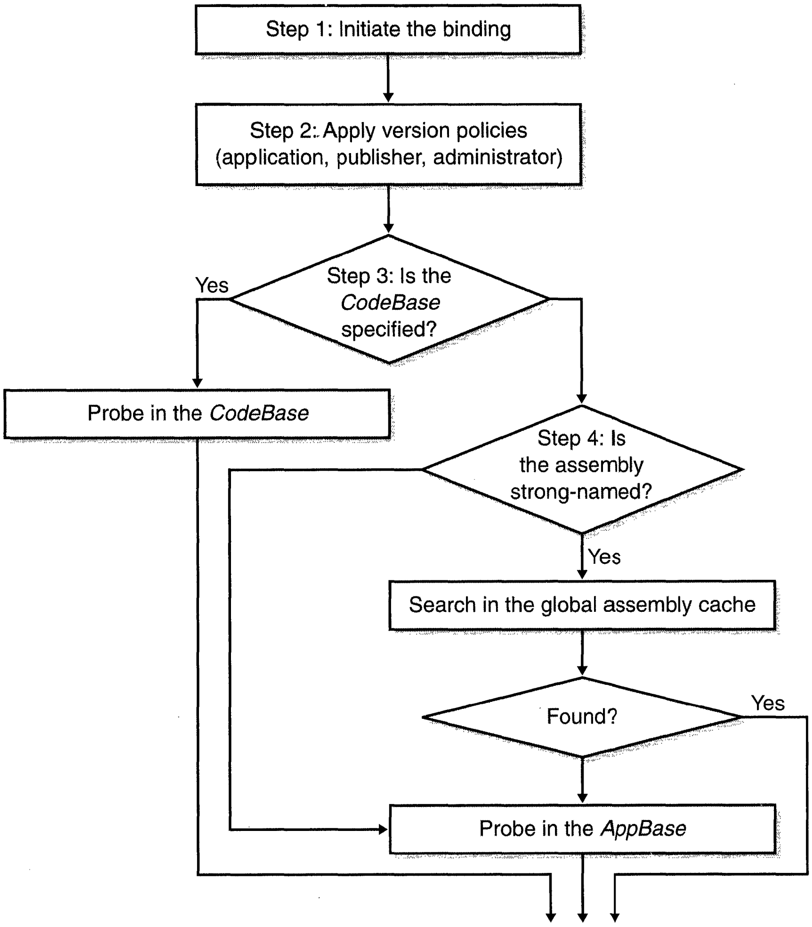

Step 1: Initiate the binding
Step 2:. Apply version policies (application, publisher, administrator)
Step 3: Is the *CodeBase* specified?
Yes
Probe in the *CodeBase*
Step 4: Is the assembly strong-named?
Yes
Search in the global assembly cache
Found?
Yes
Probe in the *AppBase*

Figure 5-2 Searching for a referenced assembly.

Module Metadata Table and Declaration

The Module metadata table contains a single record that provides the identification of the current module. The column structure of the table is as follows:

- ***Generation* (2-byte unsigned integer)** Used only at run time, in edit-and-continue mode.
- ***Name* (offset in the *#Strings* stream)** The module name, which is the same as the name of the executable file with its extension but without a path. The length should not exceed 512 characters, counting the zero terminator.

- ***Mvid* (offset in the *#GUID* stream)** A globally unique identifier, assigned to the module as it is generated.
- ***EncId* (offset in the *#GUID* stream)** Used only at run time, in edit-and-continue mode.
- ***EncBaseId* (offset in the *#GUID* stream)** Used only at run time, in edit-and-continue mode.

Because only one entry of the *Module* record can be set explicitly (the *Name* entry), the module declaration in ILAsm is quite simple:

```
.module <name>
```

ModuleRef Metadata Table and Declaration

The ModuleRef metadata table contains descriptors of other modules referenced in the current module. The set of "other modules" includes subsets of both managed and unmanaged modules.

The relevant managed modules are the other modules of the current assembly. In ILAsm, they should be declared explicitly, and their declarations should be paired with *File* declarations (discussed in the following section).

The unmanaged modules described in the ModuleRef table are simply unmanaged DLLs containing methods called from the current module using the platform invocation mechanism—*P/Invoke*, discussed in Chapter 15, "Managed and Unmanaged Code Interoperation." These *ModuleRef* records should not be paired with *File* records. They need not be explicitly declared in ILAsm because in ILAsm the DLL name is part of the *P/Invoke* specification.

A *ModuleRef* record contains only one entry, the *Name* entry, which is an offset in the *#Strings* stream. The *ModuleRef* declaration in ILAsm is not much more sophisticated than the declaration of *Module*:

```
.module extern <name>
```

As in the case of *Module*, <*name*> in *ModuleRef* is the name of the executable file with its extension but without a path, not exceeding 512 characters.

File Metadata Table and Declaration

The File metadata table describes other files of the same assembly that are referenced in the current module. In single-module assemblies, this table is empty. The table has the following column structure:

- ***Flags* (4-byte unsigned integer)** Binary flags characterizing the file. In this version, this entry is mostly reserved for future use; the only flag currently defined is *File contains no metadata* (0x00000001). This flag indicates that the file in question is not a managed PE file but rather a pure resource file.
- ***Name* (offset in the *#Strings* stream)** The filename, subject to the same rules as the names in *Module* and *ModuleRef*. This is the only occurrence of data duplication in the metadata model: the *File* name matches the name used in the *ModuleRef* with which this *File* record is paired. However, because the names in both records are not physical strings but rather offsets in the string heap, the data might not actually be duplicated; instead, both records might reference the same string in the heap.
- ***HashValue* (offset in the *#Blob* stream)** The blob representing the hash of the file, used to authenticate the files in a multifile assembly. Even in a strong-named assembly, the strong name signature resides only in the prime module and covers only the prime module. Nonprime modules in an assembly are authenticated by their hash values.

The *File* declaration in ILAsm looks like the following:

```
.file <flag> <name>  .hash = ( <bytes> )
```

where <*flag*> ::=

```
<none>          // The file is a managed PE file
| nometadata    // The file is a pure resource file
```

If the hash value is not explicitly specified, the ILAsm compiler finds the named file and computes the hash value using the hash algorithm specified in the *Assembly* declaration.

The *File* declaration can also have a *.entrypoint* clause, as shown in this example:

```
.file MainClass.dll
.hash = (01 02 03 04 05 06  … )
.entrypoint
```

This sort of *File* declaration can occur only in the prime module and only when the entry point method is defined in a nonprime module of the assembly. This clause of the *File* declaration does not affect the metadata, but it puts the appropriate file token in the *EntryPointToken* entry of the common language runtime header. See Chapter 3, "The Structure of a Managed Executable File," for details about *EntryPointToken* and the runtime header.

The prime module of an assembly, especially a runnable application (EXE), must have a valid token in the *EntryPointToken* field of the common language runtime header; and this token must be either a *Method* token, if the entry point method is defined in the prime module, or a *File* token. In the latter case, the loader loads the relevant module and inspects its common language runtime header, which must contain a valid *Method* token in the *EntryPointToken* field.

Managed Resource Metadata and Declaration

A *resource* is any nonexecutable data that is logically deployed as a part of an application. The data can take any number of forms such as strings, images, persisted objects, and so on. As Chapter 3 described, resources can be either managed or unmanaged (platform-specific). These two kinds of resources have different formats and are accessed using managed and unmanaged APIs, respectively.

An application often must be customized for different cultures. A *culture* is a set of preferences based on a user's language, sublanguage, and cultural conventions. In the .NET Framework, the culture is described by the *CultureInfo* class from the .NET Framework class library. A culture is used to customize operations such as formatting dates and numbers, sorting strings, and so on.

You might also need to customize an application for different countries or regions. A *region* defines a set of standards for a particular country or region of the world. In the .NET Framework, the class library describes a region using the *RegionInfo* class. A region is used to customize operations such as formatting currency symbols.

Localization of an application is the process of sharing the application's executable code with the application's resources that have been customized for specific cultures. Although a culture and a region together constitute a *locale*, localization is not concerned with customizing an application to a specific region. The .NET Framework and the common language runtime do not support localization of component metadata, instead relying solely on the managed resources for this task.

The .NET Framework uses a hub-and-spoke model for packaging and deploying resources. The hub is the *main* assembly, which contains the nonlocalizable executable code and the resources for a single culture (referred to as the *neutral culture*). The neutral culture is the fallback culture for the application. Each spoke connects to a *satellite* assembly that contains the resources for a single culture. Satellite assemblies do not contain code.

The advantages of this model are obvious. First, resources for new cultures can be added incrementally after an application is deployed. Second, an application needs to load only those satellite assemblies that contain the resources needed for a particular run.

The resources used in or exposed by an assembly can reside in one of the following locations:

- In separate resource file(s) in the same assembly. Each resource file can contain one or more resources. The metadata descriptors of such files carry the *nometadata* flag.
- Embedded in managed modules of the same assembly.
- In another (external) assembly.

Because the resource data is not directly used or validated by the deployment subsystem or the loader, it can be of any kind.

All resource data embedded in a managed PE file resides in a contiguous block inside the *.text* section. The *Resources* data directory in the common language runtime header provides the relative virtual address (RVA) and size of embedded managed resources. Each individual resource is preceded by a 4-byte unsigned integer holding the resource's length in bytes. Figure 5-3 shows the layout of embedded managed resources.

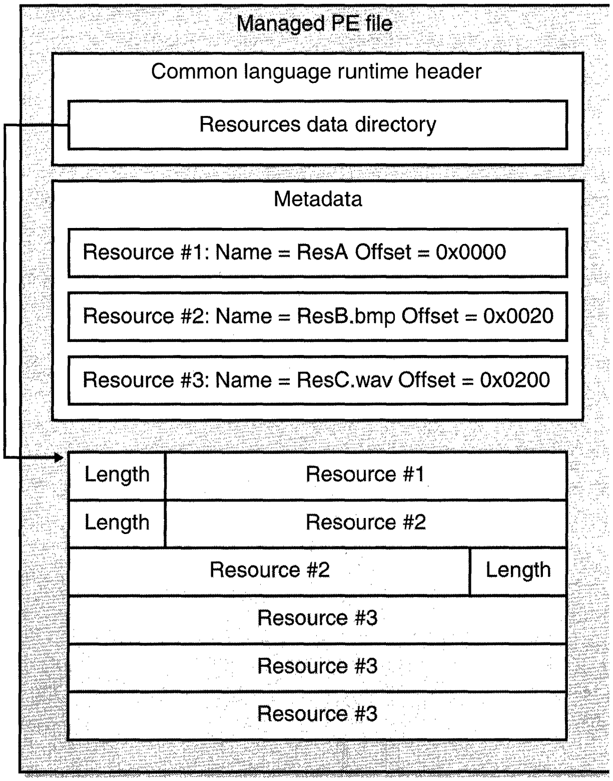

Managed PE file
Common language runtime header
Resources data directory
Metadata
Resource #1: Name = ResA Offset = 0x0000
Resource #2: Name = ResB.bmp Offset = 0x0020
Resource #3: Name = ResC.wav Offset = 0x0200
Length
Resource #1
Length
Resource #2
Resource #2
Length
Resource #3
Resource #3
Resource #3

Figure 5-3 The layout of embedded managed resources.

The ManifestResource metadata table, describing the managed resources, has the following column structure:

- ***Offset* (4-byte unsigned integer)** Location of the resource within the managed resource segment to which the Resources data directory of the common language runtime header points. This is *not* an RVA; rather, it is an offset within the managed resource segment.
- ***Flags* (4-byte unsigned integer)** Binary flags indicating whether the managed resource is public (accessible from outside the assembly) or private (accessible from within the current assembly only).
- ***Name* (offset in the *#Strings* stream)** Nonempty name of the resource, unique within the assembly.
- ***Implementation* (coded token of type Implementation)** Token of the respective *AssemblyRef* record if the resource resides in another assembly or of the respective *File* record if the resource resides in another file of the current assembly. If the resource is embedded in the current module, this entry is set to 0. If the resource is imported from another assembly, the offset need not be specified; the loader will ignore it.

ILAsm syntax for the declaration of a managed resource is as follows:

```
.mresource <flag> <name> { <mResourceDecl>* }
```

where *<flag>* ::= *public* | *private* and *<mResourceDecl>* ::=

```
.assembly extern <alias>      // Resource is imported from another
                              // assembly
| .file <name> at <int32>     // Resource resides in another
                              // file of this assembly;
                              // <int32> is the offset
| <customAttrDecl> // Define custom attribute for this resource
```

The default flag value is *private*.

The directives *.assembly extern* and *.file* in the context of a managed resource declaration refer to the resource's *Implementation* entry and are mutually exclusive. If *Implementation* references the *AssemblyRef* or *File* before it has been declared, the ILAsm compiler will diagnose an error.

If the *Implementation* entry is empty, the resource is presumed embedded in the current module. In this case, the ILAsm compiler creates the PE file, loads the resource from the file according to the resource's name, and writes it into the *.text* section of the PE file, automatically setting the *Offset* entry of the *ManifestResource* record. When the IL Disassembler disassembles a PE file into a text file, the embedded managed resources are saved into binary files named after these resources, which allows the ILAsm compiler to easily pick them up if the PE file needs to be reassembled.

ILAsm does not offer any language constructs to address the managed resources because IL lacks the means to do so. Managed APIs provided by the .NET Framework class library—specifically, the *System.Resources.ResourceManager* class—are used to load and manipulate managed resources.

ExportedType Metadata Table and Declaration

The ExportedType metadata table contains information about the public classes (visible outside the assembly) that are declared in nonprime modules of the assembly. Only the prime module's manifest can carry this table.

This table is needed because the loader expects the prime module of an assembly to hold information about *all* classes exported by the assembly. The union of the classes defined in the prime module and those in the ExportedType table gives the loader the full picture.

On the other hand, the intersection of the classes defined in the prime module and those in the ExportedType table must be nil. As a result, the ExportedType table can be nonempty only in the prime module of a multimodule assembly.

The ExportedType table has the following column structure:

- ***Flags* (4-byte unsigned integer)** Binary flags indicating accessibility of the exported type. The flags we are interested in are *public* and *nested public*; other accessibility flags—identical to the class accessibility flags discussed in Chapter 6, "Namespaces and Classes,"—are syntactically admissible but are not used to define true exported types. Other flags can be present in pseudo-*ExportedTypes* only, which the loader can use to resolve unscoped type references in multimodule assemblies.

  Some explanation is in order. Any time a type (class) is referenced in a module, the resolution scope should be provided to indicate where the referenced class is defined (in the current module, in another module of this assembly, or in another assembly). If the resolution scope is not provided, the referenced type should be declared in the current module. However, if this type cannot be found in the module referencing it, *and* if the manifest of the prime module carries a same-name pseudo-*ExportedType* record indicating where the type is actually defined, the loader is nevertheless able to resolve the type reference. None of the current Microsoft managed compilers, including the ILAsm compiler, uses this rather bizarre technique.

- ***TypeDefId* (4-byte unsigned integer)** An uncoded token referring to a record of the TypeDef table of the module where the exported class is defined. This is the *only* occasion in the entire metadata model in which a module's metadata contains an explicit value of a metadata token from another module. This token is used as something of a hint for the loader and can be omitted without any ill effects. If the token is supplied, the loader retrieves the specific *TypeDef* record from the respective module's metadata and checks the full name of *ExportedType* against the full name of *TypeDef.* If the names match, the loader has found the class it was looking for; if the names do not match, or if the token was not supplied in the first place, the loader finds the needed *TypeDef* by its full name. My advice: never specify a *TypeDefId* token explicitly when programming in ILAsm. This shortcut works only for automatic tools such as the Assembly Linker (AL) and only under certain circumstances.
- ***TypeName* (offset in the *#Strings* stream)** Exported type's name; must be nonempty.
- ***TypeNamespace* (offset in the *#Strings* stream)** Exported type's namespace; can be empty. Class names and namespaces are discussed in Chapter 6.
- ***Implementation* (coded token of type Implementation)** Token of the *File* record indicating the file of the assembly where the exported class is defined or the token of another *ExportedType*, if the current one is nested in another one.

The exported types are declared in ILAsm as follows:

```
.class extern <flag> <namespace>.<name> { <expTypeDecl>* }
```

where <*flag*> ::= *public* | *nested public* and <*expTypeDecl*> ::=

```
.file <name>     // File where exported class is defined
| .class extern <namespace>.<name> // Enclosing exported type
| .class <int32> // Set TypeDefId explicitly
| <customAttrDecl> // Define custom attribute for this ExportedType
```

The directives *.file* and *.class extern* define the *Implementation* entry and are mutually exclusive. As in the case of the *.mresource* declaration, the *File* or *ExportedType* must be declared before being referenced by the *Implementation* entry.

It is fairly obvious that if *Implementation* is specified as *.class extern*, we are dealing with a nested exported type, and *Flags* must be set to *nested public*. Inversely, if *Implementation* is specified as *.file*, we are dealing with a top-level unnested class, and *Flags* must be set to *public*.

Order of Manifest Declarations in ILAsm

The general rule in ILAsm (and not only in ILAsm) is "declare, then reference." In other words, it's always safer, and in some cases outright required, to declare a metadata item before referencing it. There are times when you can reference a yet-undeclared item—for example, calling a method that is defined later in the source code. But you cannot do this in the manifest declarations.

If we reexamine the diagram shown in Figure 5-1, which illustrates the mutual references between the manifest metadata tables, we can discern the following list of dependencies:

- Exported types reference files and enclosing exported types.
- Manifest resources reference files and external assemblies.
- Every manifest item can have associated custom attributes, and custom attributes reference external assemblies and (rarely) external modules. (See Chapter 13 for details.)

To comply with the "declare, then reference" rule, the following sequence of declarations is recommended for ILAsm programs, with the manifest declarations preceding all other declarations in the source code:

1. *AssemblyRef* declarations (*.assembly extern*), because of the custom attributes. The reference to the assembly Mscorlib should lead the pack because most custom attributes reference this assembly.
2. *ModuleRef* declarations (*.module extern*), again because of the custom attributes.
3. *Assembly* declaration (*.assembly*). Because the ILAsm compiler takes different paths in compiling Mscorlib.dll and compiling other assemblies, it is better to let it know which path to take as soon as possible. However, this is less important if you are *not* compiling Mscorlib.dll; by default the compiler assumes that it is compiling a "conventional" module.
4. *File* declarations (*.file*) because *ExportedType* and *ManifestResource* declarations might reference them.
5. *ExportedType* declarations (*.class extern*), with enclosing *ExportedType* declarations preceding the nested *ExportedType* declarations.
6. *ManifestResource* declarations (*.mresource*).

Note Remember that only the manifests of prime modules carry *Assembly* and *ExportedType* declarations.

Single-Module and Multimodule Assemblies

A single-module assembly consists of a sole prime module. Manifests of single-module assemblies carry neither File nor ExportedType tables: there are no other files to declare, and all types are defined in the prime module.

The advantages of single-module assemblies include lower overhead, easier deployment, and slightly greater security. Overhead is lower because only one set of headers and metadata tables must be read, transmitted, and analyzed. Assembly deployment is simpler because only one PE file must be deployed. And the level of security can be slightly higher because the prime module of the assembly can be protected with a strong name signature, which is extremely difficult to counterfeit and virtually guarantees the authenticity of the prime module. Nonprime modules are authenticated only by their hash values (referenced in *File* records of the prime module) and are theoretically easier to spoof.

Note As a rule, shared assemblies are single-module, probably because of their instrumental nature.

Manifests of the modules of a multimodule assembly carry File tables, and the manifest of the prime module of such an assembly might or might not carry ExportedType tables, depending on whether any public types are defined in nonprime modules.

The advantages of multimodule assemblies include easier development and … lower overhead. (No, I am not pulling your leg.) Both advantages stem from the obvious modularity of the multimodule assemblies.

Multimodule assemblies are easier to develop because if you distribute the functionality among the modules well, the modules can be developed independently and then incrementally added to the assembly. (I didn't say that a multimodule assembly was easier to *design*.)

Lower overhead at run time results from the way the loader operates: it loads the modules only when they are referenced. So if only a part of your assembly's functionality is engaged in a certain execution session, only part of the modules constituting your assembly might be loaded. Of course, you cannot count on any such effect if the functionality is spread all over the modules and if classes defined in different modules cross-reference each other.

A well-known technique for building a multimodule assembly from a set of modules is based on a "spokesperson" approach: the functional modules are analyzed, and an additional prime module is created, carrying nothing but the manifest and (maybe) a strong name signature. Such a prime module carries no functionality or positive definitions of its own whatsoever—it is only a front for functional modules, a "spokesperson" dealing with the loader on behalf of the functional modules. The Assembly Linker tool, distributed with the .NET Framework, uses this technique to build multimodule assemblies.

Metadata Validity Rules

In this section, I'll summarize the validity rules for metadata contained in a manifest. Because some of these rules have a direct bearing on how the loader functions, the respective checks are performed at run time. Other rules describe "well-formed" metadata; violating one of these rules might result in rather peculiar effects during the program execution, but it does not represent a crash or security breach hazard, so the loader does not perform these checks. You can find the complete set of metadata validity rules in Partition II of the ECMA Standard Proposal; the sections that follow here review the most important of them.

> **Warning** ILAsm *does* allow you to generate invalid metadata. Thus, it's extremely important to carefully check your modules after compilation.

To find out whether any of the metadata in a module is invalid, you can run the PEVerify utility, included in the .NET Framework SDK, using the option */MD* (metadata validation). Alternatively, you can invoke the IL Disassembler by using the option */ADV* (advanced). Choose View, MetaInfo, Validate, and then press Ctrl+M. Both utilities use the Metadata Validator (MDValidator), which is built into the common language runtime.

Assembly Table Validity Rules

- The record count of the table must be no more than 1. This is not checked at run time because the loader ignores all *Assembly* records except the first one.
- The *Flags* entry must have bits set only as defined in the *CorAssemblyFlags* enumeration in CorHdr.h. For the first release of the common language runtime, the valid mask is 0xC031.
- The *Locale* entry must be set to 0 or must refer to a nonempty string in the string heap that matches a known culture name. You can obtain a list of known culture names by using a call to the *CultureInfo.GetCultures* method, from the .NET Framework class library.
- [run time] If *Locale* is not set to 0, the referenced string must be no longer than 1023 characters plus the zero terminator.
- [run time] The *Name* entry must refer to a nonempty string in the string heap. The name must be the module filename excluding the extension, the path, and the drive letter.
- [run time] The *PublicKey* entry must be set to 0 or must contain a valid offset in the *#Blob* stream.

AssemblyRef Table Validity Rules

- The *Flags* entry can have only the least significant bit set (corresponding to the *afPublicKey* value; see the *CorAssemblyFlags* enumeration in CorHdr.h).
- [run time] The *PublicKeyOrToken* entry must be set to 0 or must contain a valid offset in the *#Blob* stream.
- The *Locale* entry must comply with the same rules as the *Locale* entry of the Assembly table (discussed in the preceding section).
- The table must not have duplicate records with simultaneously matching *Name*, *Locale*, *PublicKeyOrToken,* and all *Version* entries.
- [run time] The *Name* entry must refer to a nonempty string in the string heap. The name must be the prime module filename excluding the extension, the path, and the drive letter.

Module Table Validity Rules

- [run time] The record count of the table must be at least 1.
- The record count of the table must be exactly 1. This is not checked at run time because the loader uses the first *Module* record and ignores the others.
- [run time] The *Name* entry must refer to a nonempty string in the string heap, no longer than 511 characters plus the zero terminator. The name must be the module filename including the extension and excluding the path and the drive letter.
- The *Mvid* entry must refer to a nonzero GUID in the GUID heap. The value of the *Mvid* entry is generated automatically and cannot be specified explicitly in ILAsm.

ModuleRef Table Validity Rules

- [run time] The *Name* entry must refer to a nonempty string in the string heap, no longer than 511 characters plus the zero terminator. The name must be a filename including the extension and excluding the path and the drive letter.

File Table Validity Rules

- The *Flags* entry can have only the least significant bit set (corresponding to the *ffContainsNoMetaData* value; see the *CorFileFlags* enumeration in CorHdr.h).
- [run time] The *Name* entry must refer to a nonempty string in the string heap, no longer than 511 characters plus the zero terminator. The name must be a filename including the extension and excluding the path and the drive letter.
- [run time] The string referenced by the *Name* entry must not match _S[N][[C]*]_, where
    - *S* ::= con | aux | lpt | prn | nul | com
    - *N* ::= 0..9
    - *C* ::= $ | :
- [run time] The *HashValue* entry must hold a valid offset in the *#Blob* stream.

- The table must not contain duplicate records whose *Name* entries refer to matching strings.
- The table must not contain duplicate records whose *Name* entries refer to strings matching this module's name.

ManifestResource Table Validity Rules

- [run time] The *Implementation* entry must be set to 0 or must hold a valid *AssemblyRef* or *File* token.
- [run time] If the *Implementation* entry does not hold an *AssemblyRef* token, the *Offset* entry must hold a valid offset within limits specified by the *Resources* data directory of the common language runtime header of the target file.
- [run time] The *Flags* entry must hold either 1 or 2—*mrPublic* or *mrPrivate*, respectively.
- [run time] The *Name* entry must refer to a nonempty string in the string heap.
- The table must not contain duplicate records whose *Name* entries refer to matching strings.

ExportedType Table Validity Rules

- The record count of the table must be 0 if the Assembly table is empty.
- The record count of the table must be 0 if the File table is empty.
- There must be no rows with *TypeName* and *TypeNamespace* matching *Name* and *Namespace*, respectively, of any row of the TypeDef table.
- The *Flags* entry must hold one of the visibility flags of the enumeration *CorTypeAttr* (see CorHdr.h). Valid flags are 0 through 7.
- [run time] The *Implementation* entry must hold a valid *ExportedType* or *File* token.
- [run time] The *Implementation* entry must not hold an *ExportedType* token pointing to this record.
- If the *Implementation* entry holds an *ExportedType* token, the *Flags* entry must hold a nested visibility value in the range 2–7.

- If the *Implementation* entry holds a *File* token, the *Flags* entry must hold the *tdNonPublic* or *tdPublic* visibility value (0 or 1).
- [run time] The *TypeName* entry must refer to a nonempty string in the string heap.
- [run time] The *TypeNamespace* entry must be set to 0 or must refer to a nonempty string in the string heap.
- [run time] The combined length of the strings referenced by *TypeName* and *TypeNamespace* must not exceed 1023 characters.
- The table must not contain duplicate records whose *Implementation* entry holds a *File* token, and whose *TypeName* and *TypeNamespace* entries refer to matching strings.
- The table must not contain duplicate records whose *Implementation* entries hold the same *ExportedType* token and whose *TypeName* entries refer to matching strings.

6

Namespaces and Classes

As earlier chapters have discussed, the common language runtime computational model is inherently object-oriented. The concept of class—or, to use more precise runtime terminology, the concept of a *type*—is the central principle around which the entire computational model is organized. The type of an item—a variable, a constant, a parameter, and so on—defines both data representation and the behavioral features of the item. Hence one type can be substituted for another only if both these aspects are equivalent for both types—for instance, a derived type can be interpreted as the type from which it is derived.

The ECMA standard specification of the common language infrastructure divides types into *value types* and *reference types*, depending on whether an item type represents a data item itself or a reference (an address or a location indicator) to a data item.

Reference types include object types, interface types, and pointer types. Object types—classes—are types of self-describing values, either complete or partial. Types with partial self-describing values are called *abstract classes*. Interface types are always types of partial self-describing values. Interfaces usually represent subsets of behavioral features exposed by classes; a class is said to *implement* the respective interface. Pointer types are simply references to items, indicating item locations.

The common language runtime object model supports only single type inheritance, and multiple inheritance is simulated through implementation of multiple interfaces. Because of that, the runtime object model is absolutely hierarchical, with the *System.Object* class at the root of the tree. (See Figure 6-1.) Interface types, however, are not part of the type hierarchy because they are inherently incomplete and have no implementation of their own.

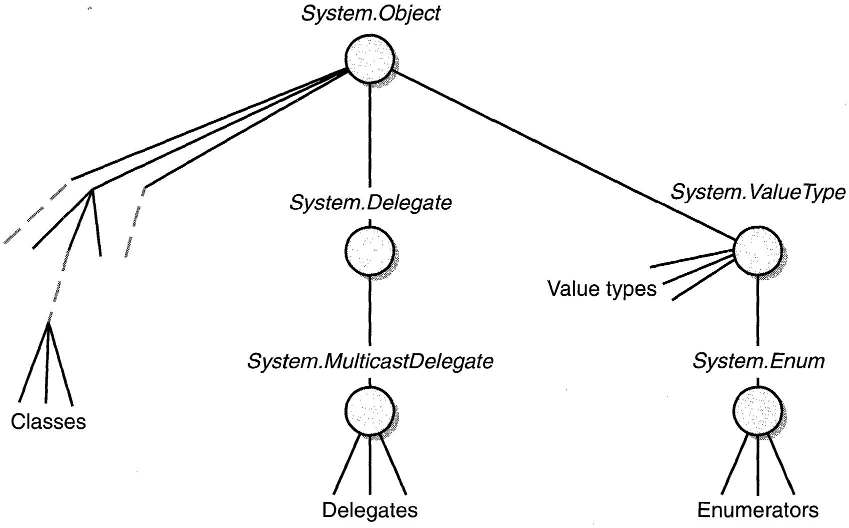


Figure 6-1 The common language runtime type hierarchy.

All types (except interfaces) are derived eventually from *System.Object*. This chapter examines types and their declarations, dividing the types into five categories: classes, interfaces, value types, enumerators, and delegates. These categories are not mutually exclusive—for example, delegates are classes and enumerators are value types—but the types of each category have distinct features.

Class Metadata

From a structural point of view, all five categories of types have identical metadata representations. Thus we can talk about class metadata, or type metadata, in a general sense.

Class metadata is grouped around two distinct concepts: *type definition* (*TypeDef*) and *type reference* (*TypeRef*). *TypeDef*s and related metadata describe the types declared in the current module, whereas *TypeRef*s describe references to types that are declared somewhere else. Because it obviously takes more information to adequately define a type than to refer to one already defined, *TypeDef*s and related metadata are far more complex than *TypeRef*s.

When defining a type, you should supply the following information:

- The name of the type being defined
- Flags indicating special features the type should have
- The type from which this type is derived
- The interfaces this type implements
- How the loader should lay out this class
- Whether this type is nested in another type—and if so, in which one
- Where fields and methods of this type (if any) can be found

When referencing a type, only its name and resolution scope need be specified. The resolution scope indicates where the definition of the referenced type can be found: in this module, in another module of this assembly, or in another assembly. In the case of referencing the nested types, the resolution scope is another *TypeRef*.

Figure 6-2 shows the metadata tables that engage in type definition and referencing but not the tables related to identification of type members—fields and methods, for example, and their attributes. The arrows denote cross-table referencing by means of metadata tokens. In the following sections, we'll have a look at all the metadata tables involved.

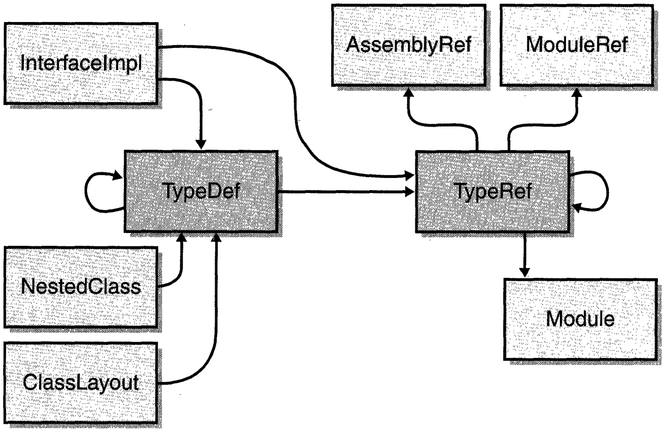


Figure 6-2 Metadata tables that engage in type definition and referencing.

TypeDef Metadata Table

The TypeDef table is the main table containing type definition information. Each record in this table has six entries:

- ***Flags* (4-byte unsigned integer)** Binary flags indicating special features of the type. Because the *TypeDef* flags are numerous and important, this chapter discusses them separately; see "Class Attributes."
- ***Name* (offset in the *#Strings* stream)** The name of the type. This entry must not be empty.
- ***Namespace* (offset in the *#Strings* stream)** The namespace of the type. This entry can be empty. The namespace plus the name constitute the full name of the type.
- ***Extends* (coded token of type *TypeDefOrRef*)** A token of the type's parent—that is, of the type from which this type is derived. This entry must be set to 0 for all interfaces and for one class, the type hierarchy root class *System.Object*. For all other types, this entry should carry a valid reference to the TypeDef or TypeRef table.
- ***FieldList* (record index [RID] to the Field table)** An index to the Field table, marking the start of the field records belonging to this type.
- ***MethodList* (RID to the Method table)** An index to the Method table, marking the start of the method records belonging to this type.

TypeRef Metadata Table

The TypeRef metadata table has a much simpler structure than the TypeDef table. Each record in this table has three entries:

- ***ResolutionScope* (coded token of type *ResolutionScope*)** An indicator of the location of the type definition. This entry is set to 0 if the referenced type is defined in the current assembly—which IL assembly language (ILAsm) does not allow—or to 1 (the *Module* token) if the referenced type is defined in the same module. *ResolutionScope* can be a token referencing the ModuleRef table if the type is defined in another module of the same assembly, a token referencing the AssemblyRef table if the type is defined in another assembly, or a token referencing the TypeRef table if the type is nested in another type. Having *TypeRef*s for the types defined in the same module does not constitute a metadata error, but it is redundant and should be avoided if possible.
- ***Name* (offset in the *#Strings* stream)** The name of the referenced type. This entry must not be empty.
- ***Namespace* (offset in the *#Strings* stream)** The namespace of the referenced type. This entry can be empty. The namespace plus the name constitute the full name of the type.

InterfaceImpl Metadata Table

If the defined type implements one or several interfaces, the corresponding *TypeDef* record is referenced in one or several records of the InterfaceImpl metadata table. This table serves as a lookup table, providing information about "what is implementing what," and it is ordered by implementing type. The InterfaceImpl table has only two entries in each record:

- ***Class* (RID to the TypeDef table)** An index to the TypeDef table, indicating the implementing type.
- ***Interface* (coded token of type *TypeDefOrRef*)** An indicator of the implemented type, which can reside in either the TypeDef table or the TypeRef table. The implemented type must be marked as an interface.

NestedClass Metadata Table

If the defined type is nested in another type, its *TypeDef* record is referenced in another lookup table: the NestedClass metadata table. (For more information about nesting, see "Nested Types" later in this chapter.) Like the InterfaceImpl table, the NestedClass table has only two entries per record:

- ***NestedClass* (RID to the TypeDef table)** An indicator of the nested type (the *nestee*).
- ***EnclosingClass* (RID to the TypeDef table)** An indicator of the type in which the current type is nested (the *encloser*, or *nester*).

Because types of both entries are RIDs in the TypeDef table, the nestee and the encloser cannot be defined in different modules or assemblies.

ClassLayout Metadata Table

Usually, the loader has its own ideas about how to lay out the type being loaded. Certain types, however, must be laid out in a specific manner, and they carry metadata information regarding these specifics.

The ClassLayout metadata table provides additional information about the packing order and total size of the type. In the section "Value Type as Placeholder" in Chapter 1, for example, when we declared a "placeholder" type without any internal structure, we used such additional information—the total size of the type.

A record in the ClassLayout metadata table has three entries:

- ***PackingSize* (2-byte unsigned integer)** The alignment factor in bytes. This entry must be set to 0 or to a power of 2, from 1 to 128.
- ***ClassSize* (4-byte unsigned integer)** The total requested layout size of the type. If the type has instance fields and the summary size of these fields, aligned by *PackingSize*, is different from *ClassSize*, the loader allocates the larger of the two sizes for the type.
- ***Parent* (RID to the TypeDef table)** An index of the type definition record to which this layout belongs. The ClassLayout table should not contain any duplicate records with the same *Parent* entry value.

Namespace and Full Class Name

It is time to talk seriously about names in the common language runtime and ILAsm. So far, in Chapter 5, "Modules and Assemblies," you've encountered only names that were in fact filenames and hence had to conform to well-known filename conventions. From now on, however, you'll need to deal with names in general, so it will be important to know the rules.

ILAsm Naming Conventions

Names in ILAsm are either simple or composite. Composite names are composed of simple names and special connection symbols such as a dot. For example, *System* and *Object* are simple names, and *System.Object* is a composite name. The length of either kind of name in ILAsm is not limited syntactically, but metadata rules impose certain limitations on the name length.

The simplest form of a simple name is an *identifier*, which in ILAsm must begin with an alphabetic character or one of the following characters:

#, $, @, _

and continue with alphanumeric characters or one of the following:

?, $, @, _

These are examples of valid ILAsm identifiers:

- *Object*
- *_Never_Say_Never_Again_*
- *men@work*

Important One obvious limitation on ILAsm identifiers: an ILAsm identifier must not match any of the (rather numerous) ILAsm keywords.

The common language runtime accepts a wide variety of names with very few limitations. Certain names—for example, *.ctor* (an instance constructor), *.cctor* (a class constructor), and *_Deleted** (a metadata item marked for deletion during an edit-and-continue session)—are reserved for internal use by the runtime. Generally, however, the runtime is liberal about names. As long as a name serves its purpose—identifying a metadata item unambiguously—and cannot be misinterpreted, it is perfectly fine.

To cover this variety, ILAsm offers an alternative way to present a simple name: as a single-quoted literal. For example, these are valid ILAsm simple names:

- *'123'*
- *'Space Between'*
- *'&%!'*

One of the most frequently encountered kinds of composite names is the *dotted name*, a name composed of simple names separated by a dot:

```
<dotted_name> ::= <simple_name>[.<simple_name>*]
```

Examples of dotted names include the following:

- *System.Object*
- *'123'.'456'.'789'*
- *Foo.Bar.'&%!'*

Namespaces

Simply put, *namespaces* are the common prefixes of the full names of classes. The full name of a class is a dotted name; the last simple name it contains is the class name, and the rest is the namespace of the class.

It takes longer, perhaps, to explain what namespaces are *not*. Namespaces are not metadata items—they do not have an associated metadata table, and they cannot be referenced by tokens. Namespaces also have no direct bearing on assemblies. The name of an assembly might or might not match in full or in part the namespace(s) used in the assembly. One assembly might use several namespaces, and the same namespace can be used in different assemblies.

So why does the metadata model even bother with namespaces and class names instead of simply using the full class names? The answer is simple: economy of space. Let's suppose that we define two classes with the full names *Foo.Bar* and *Foo.Baz*. Since the names are different, in the full-name model we would have to store two full names in the string heap: *Foo.Bar\0Foo.Baz\0*. But if we split the full names into namespaces and names, we need to store only *Foo\0Bar\0Baz\0*. This is quite a difference when you consider the number of possible classes.

Namespaces in ILAsm are declared in the following way:

```
.namespace MyNamespace
{
   : // Classes declared here
   // Have full name "MyNamespace.<simple_name>"
}
```

Namespaces can be nested, as shown here:

```
.namespace MyNamespace
{
   : // Classes declared here
   // Have full name "MyNamespace.<simple_name>"
   .namespace X
   {
     : // Classes declared here
     // Have full name "MyNamespace.X.<simple_name>"
   }
}
```

Or they can be unnested. This is how the IL Disassembler represents namespaces in the disassembly text:

```
.namespace MyNamespace
{
   : // Classes declared here
   // Have full name "MyNamespace.<simple_name>"
}
.namespace MyNamespace.X
{
   : // Classes declared here
   // Have full name "MyNamespace.X.<simple_name>"
}
```

Full Class Names

As the preceding section explained, a full class name is a dotted name, composed of the class's namespace and the name of the class. The loader resolves class references by their full names and resolution scopes, so the general rule is that no classes with identical full names should be defined in the same module. For multimodule assemblies, an additional (less strict) rule prohibits defining *public* classes—classes visible outside the assembly—with identical full names in the same assembly.

In ILAsm, a class is always referenced by its full name, even if it is referenced from within the same namespace. This makes class referencing context-independent.

The name of a class should be simple. Theoretically, a class name could contain a dot without violating metadata rules, but I recommend avoiding dotted class names, because they bring at best mild confusion.

ILAsm does not allow dotted names as class names, but you can bypass this restriction by quoting the dotted name, thus turning it into a simple name and avoiding a syntax error:

```
.namespace X
{
   .class public 'Y.Z'
   {
      ⋮
   }
}
```

And because a class is always referenced by its full name, a class with a dotted name will not pose any resolution problems (it will be referenced as *X.Y.Z* anyway), and the module will compile and work. But if you disassemble the module, you'll find that the left part of the dotted name of the class has migrated to the namespace, courtesy of the metadata emission API:

```
.namespace X.Y
{
   .class public Z
   {
      ⋮
   }
}
```

Although this is not what you intended, it has no dire consequences—just a case of mild confusion. If you know and expect this effect, and don't get confused that easily, you can even forgo the namespace declarations altogether and declare classes by their full names, to match the way they are referenced:

```
.class public 'X.Y.Z'
{
   ⋮
}
```

The first release of the common language runtime imposes a limitation on the full class name length, specifying that it should not exceed 1023 bytes in UTF-8 encoding. The ILAsm compiler, however, does *not* enforce this limitation. Single quotes, should they be used for simple names in ILAsm, are a purely lexical tool and don't make it to the metadata; thus they don't contribute to the total length of the full class name.

Class Attributes

An earlier section ("Class Metadata") listed the various pieces of information included in a type definition. In the simplest case, when only the TypeDef metadata table is involved, the ILAsm syntax for a type definition is as follows:

```
.namespace <dotted_name> {
   ⋮
   .class <flags> <simple_name> extends <class_ref> {
      ⋮
   }
}
```

The *<dotted_name>* value specified in the *.namespace* directive defines the *TypeDef*'s *Namespace* entry, *<simple_name>* specified in the *.class* directive defines the *TypeDef*'s *Name* entry, *<class_ref>* specified in the *extends* clause defines the *Extends* entry, and *<flags>* defines the *Flags* entry.

Flags

The numerous *TypeDef* flags can be divided into several groups, as described here.

- **Visibility flags (binary mask 0x00000007)**:
  - ❑ ***private* (0x00000000)** The type is not visible outside the assembly. This is the default.
  - ❑ ***public* (0x00000001)** The type is visible outside the assembly.
  - ❑ ***nested public* (0x00000002)** The nested type has public visibility.
  - ❑ ***nested private* (0x00000003)** The nested type has private visibility.
  - ❑ ***nested family* (0x00000004)** The nested type has family visibility—that is, it is visible to descendants of the enclosing class only.
  - ❑ ***nested assembly* (0x00000005)** The nested type is visible within the assembly only.
  - ❑ ***nested famandassem* (0x00000006)** The nested type is visible to the descendants of the enclosing class residing in the same assembly.

    - ❑ ***nested famorassem* (0x00000007)** The nested type is visible to the descendants of the enclosing class either within or outside the assembly and to every type within the assembly with no regard to "lineage."

- **Layout flags (binary mask 0x00000018)**:
    - ❑ ***auto* (0x00000000)** The type fields are laid out automatically, at the loader's discretion. This is the default.
    - ❑ ***sequential* (0x00000008)** The loader should preserve the order of the fields.
    - ❑ ***explicit* (0x00000010)** The type layout is specified explicitly, and the loader should follow it. (See Chapter 8, "Fields and Data Constants," for information on field declaration.)
- **Type semantics flags (binary mask 0x000005A0)**:
    - ❑ ***interface* (0x00000020)** The type is an interface. If this flag is not specified, the type is presumed to be a class or a value type; if this flag is specified, the default parent is set to nil.
    - ❑ ***abstract* (0x00000080)** The class is abstract—that is, it has abstract member methods. As such, this class cannot be instantiated and can be used only as a parent of another type or types. This flag is invalid for value types.
    - ❑ ***sealed* (0x00000100)** No types can be derived from this type. All value types and enumerators must carry this flag.
    - ❑ ***specialname* (0x00000400)** The type has a special name. How special depends on the name itself. This flag indicates to the metadata API and the loader that the name has a meaning in which they might be interested—for instance, *_Deleted**.
- **Type implementation flags (binary mask 0x00103000)**:
    - ❑ ***import* (0x00001000)** The type (a class or an interface) is imported from a COM type library.
    - ❑ ***serializable* (0x00002000)** The type can be serialized into sequential code by the serializer provided in the Microsoft .NET Framework class library.
    - ❑ ***beforefieldinit* (0x00100000)** The type can be initialized any time before the first access to a static field. If this flag is not set, the type is initialized before the first access to one of its static fields or methods.

- **String formatting flags (binary mask 0x00030000)**:
    - ***ansi*** **(0x00000000)** When interoperating with native methods, the managed strings are by default marshaled to and from ANSI strings. Managed strings are instances of the *System.String* class defined in the .NET Framework class library. *Marshaling* is a general term for data conversion on the managed and unmanaged code boundaries. (See Chapter 15, "Managed and Unmanaged Code Interoperation," for detailed information.) String formatting flags specify only default marshaling and are irrelevant when marshaling is explicitly specified. This flag, *ansi*, is the default flag for a class and hence represents a "default default" string marshaling.
    - ***unicode*** **(0x00010000)** By default, managed strings are marshaled to and from Unicode.
    - ***autochar*** **(0x00020000)** The default string marshaling is defined by the underlying platform.
- **Reserved flags (binary mask 0x0004080)**:
    - ***rtspecialname*** **(0x00000800)** The name is reserved by the common language runtime and has a special meaning. This flag is legal only in combination with the *specialname* flag. The keyword *rtspecialname* has no effect in ILAsm and is provided for informational purposes only. The IL Disassembler uses this keyword to show the presence of this reserved flag. Reserved flags cannot be set at will—this flag, for example, is set automatically by the metadata emission API when it emits an item with the *specialname* flag set and the name recognized as specific to the common language runtime.
    - **<no keyword> (0x00040000)** The type has declarative security metadata associated with it. This flag is set by the metadata emission API when respective declarative security metadata is emitted.
- **Semantics pseudoflags (no binary mask)** These are not true binary flags that define the *Flags* entry of a *TypeDef* record but rather are lexical pseudoflags modifying the default parent of the class:
    - ***value*** The type is a value type. The default parent is *System.ValueType.*
    - ***enum*** The type is an enumerator. The default parent is *System.Enum.*

Class References

The nonterminal symbol *<class_ref>* in the *extends* clause represents a reference to a type and translates into a *TypeDef* or a *TypeRef*. The general syntax of a class reference is as follows:

```
<class_ref> ::= [<resolution_scope>]<full_type_name>
```

where

```
<resolution_scope> ::= [<assembly_ref_alias>]
        | [.module <module_ref_name>]
```

Note that the square brackets in the definition of *<resolution_scope>* are syntactic elements; they do not indicate that any portion of the definition is optional.

Here are a few examples of class references:

```
[mscorlib]System.ValueType // Type is defined in another assembly
[.module Second.dll]Foo.Bar // Type is defined in another module
Foo.Baz  // Type is defined in this module
```

If the resolution scope of a class reference points to an external assembly or module, the class reference is translated into a *TypeRef* metadata record, with the full type name providing values for the *Name* and *Namespace* entries and the resolution scope providing an *AssemblyRef* or a *ModuleRef* token for the *ResolutionScope* entry.

If the resolution scope is not defined—that is, if the referenced type is defined somewhere in the current module—the class reference is translated into the respective *TypeDef* record.

Parent of the Type

Having resolved the class reference to a *TypeRef* or *TypeDef* token, we thus provided the value for the *Extends* entry of the *TypeDef* record under construction. This token references the type's parent—that is, the type from which the current type is derived.

The type referenced in the *extends* clause must not be sealed and must not be an interface; otherwise, the loader will fail to load the type. When a type is *sealed*, no types can be derived from it.

If the *extends* clause is omitted, the ILAsm compiler assigns a default parent depending on the flags specified for the type:

- ***Interface*** No parent. The interfaces are not derived from other types.
- ***value*** The parent is *[mscorlib]System.ValueType.*

- ***enum*** The parent is *[mscorlib]System.Enum*.
- **None of the above** The parent is *[mscorlib]System.Object*.

If the *extends* clause is present, the *value* and *enum* flags are ignored, and the *interface* flag causes a compilation error.

If the type layout is specified as *sequential* or *explicit*, the type's parent must also have the corresponding layout, unless the parent is *[mscorlib]System.Object*, *[mscorlib]System.ValueType*, or *[mscorlib]System.Enum*. The rationale is that the type might inherit fields from its parent, and the type cannot have a mixed layout—that is, it cannot have some fields laid out automatically and some laid out explicitly or sequentially. However, an auto-layout type can be derived from a type having any layout; in this case, information about the parent's field layout plays no role in laying out the derived type.

Interface Implementations

If the type being defined implements one or more interfaces, the type declaration has an additional clause, the *implements* clause, as shown here:

```
.namespace <dotted_name> {
   ⋮
   .class <flags> <simple_name>
      extends <class_ref>
      implements <class_refs> {
         ⋮
      }
}
```

The nonterminal symbol *<class_refs>* simply means a comma-separated list of class references:

```
<class_refs> ::= <class_ref>[,<class_ref>*]
```

For example:

```
.namespace MyNamespace {
   ⋮
   .class public MyClass
      extends MyNamespace.MyClassBase
      implements MyNamespace.IOne,
                 MyNamespace.ITwo,
                 MyNamespace.IThree {
                 ⋮
      }
}
```

The types referenced in the *implements* clause must be interfaces. A type implementing an interface must provide implementation for all of the interface's instance methods. The only exception to this rule is an abstract class.

The *implements* clause of a type declaration creates as many records in the InterfaceImpl metadata table as there are class references listed in this clause. In our preceding example, three *InterfaceImpl* records would be created.

Class Layout Information

To provide additional information regarding type layout (field alignment, total type size, or both), you need to use the *.pack* and *.size* directives, as shown in this example:

```
.namespace MyNamespace {
   ⋮
   .class public value explicit MyStruct {
      .pack 4
      .size 1024
      ⋮
   }
}
```

The *.pack* and *.size* directives appear within the scope of the type declaration, in any order. If *.pack* is not specified, the field alignment defaults to 1. If *.pack* or *.size* is specified, a *ClassLayout* record is created for this *TypeDef.*

Integer values specified in a *.pack* directive must be 0 or a power of 2, in the range 2^0 to 2^7 (1 to 128). Breaking this rule results in a compilation error. When the value is 0, the field alignment defaults to the value defined by the underlying platform.

Class layout information should not be specified for the auto-layout types. Formally, defining the class layout information for an auto-layout type represents invalid metadata. In reality, however, it is simply a waste of metadata space; when the loader encounters an auto-layout type, it never checks to see whether this type has a corresponding *ClassLayout* record.

Interfaces

An *interface* is a special kind of type, defined in Partition I of the ECMA Standard Proposal as "a named group of methods, locations and other contracts that shall be implemented by any object type that supports the interface contract of the same name." In other words, an interface is not a "real" type but merely a named descriptor of methods and properties exposed by other types. Conceptually, an interface in the common language runtime is similar to a COM interface—or at least the general idea is the same.

Not being a real type, an interface is not derived from any other type, nor can other types be derived from an interface. But an interface can "implement" other interfaces. This is not a true implementation, of course. When we say that "interface IA implements interfaces IB and IC," we mean only that the contracts defined by IB and IC are subcontracts of the contract defined by IA.

As a descriptor of items (methods, properties, events) exposed by other types, an interface cannot offer its own implementation of these items and thus is, by definition, an abstract type. When you define an interface in ILAsm, you can omit the keyword *abstract* because the compiler adds this flag automatically when it encounters the keyword *interface*.

For the same reason, an interface cannot have instance fields, because a declaration of a field is the field's implementation. However, an interface can and must offer implementation of its static members—the items shared by all instances of a type—if it has any. Bear in mind, of course, that the definition of *static* as "shared by all instances" is general for all types and does not imply that interfaces can be instantiated. They cannot be. Interfaces are inherently abstract and cannot even have instance constructors.

Static members (fields, methods) of an interface are not part of the contract defined by the interface and have no bearing on the types that implement the interface. A type implementing an interface must implement all instance members of the interface, but it has nothing to do with the static members of the interface.

The nature of an interface as a descriptor of items exposed by other types requires that the interface itself and all its members must be public, which makes perfect sense— we are, after all, talking about *exposed* items.

Interfaces have several limitations. One is obvious: because an interface is not a real type, it does not have layout. It simply doesn't make sense to talk about the packing size or total size of a contract descriptor.

Another limitation is not so obvious: interfaces should not be sealed. This might sound contradictory because, as just noted, no types can be derived from interfaces—which is precisely the definition of *sealed*. But a more general rule, applicable to all types, dictates that an abstract type cannot be sealed. Formally, an interface is a type, and it is inherently abstract; ergo, it cannot be marked as sealed, notwithstanding the fact that no type can be derived from it.

Value Types

Value types are the closest thing in the common language runtime model to C++ structures. These types are values with either trivial structure (for example, a 4-byte integer) or complex structure. When you declare a variable of a class type, you don't automatically create a class instance. You create only a reference to the class, initially pointing at nothing. But when you declare a variable of value type, the instance of this value type is created immediately, by the variable declaration itself, because a value type is primarily a *data structure*. As such, a value type must have instance fields or size defined. A zero-size value type (with no instance fields and no total size specified) represents invalid metadata; however, as in many other cases, the loader is more forgiving than the official metadata validity rules: when it encounters a zero-size value type, the loader assigns it a 1-byte size by default.

Although an instance of a value type is created at the moment a variable having this value type is declared, the instance constructor method (should it be defined for the value type in question) is not called at this moment. (See Chapter 9, "Methods," for information about the instance constructor method.) Declaring a variable creates a "blank" instance of the value type, and if this value type has an instance constructor, it should be called explicitly.

Boxed and Unboxed Values

As a data structure, a value type must sometimes be represented as an object, to satisfy the requirements of certain generic APIs, which expect object references as input parameters. The common language runtime provides the means to produce a class representation of a value type and to restore a value type (data structure) from its class representation. These operations, called *boxing* and *unboxing*, respectively, are defined for every value type.

Recall from the beginning of this chapter that types can be classified as either value types or reference types. Simply put, boxing transforms a value type into a reference type (an object reference), and unboxing does just the opposite. We can box any value type and get an object reference, but this does not mean, however, that we can unbox any object and get a value type.

When we declare a value type variable, we create a data structure. When we box this variable, an object (a class instance) is created whose data part is an exact bit copy of the data structure. Then we can deal with this instance the same way we would deal with an ordinary object—for example, we could use it in a call to a method, which takes a generic object reference as a parameter. It is important to understand that the "original" variable does not go anywhere when it is being boxed.

Instance Members of Value Types

Value types, like other types, can have static and instance members, including methods and fields. To access an instance member of a class, we need to provide the instance pointer (known in C++ as *this*). In the case of a value type, we simply use a managed reference as an instance pointer.

Let's suppose, for example, that we have a variable of type *4-byte integer*. (What can be more trivial than that, except maybe type *fewer-byte integer*?) This value type is defined as *[mscorlib]System.Int32* in the .NET Framework class library. Instead of boxing this variable and getting a reference to an instance of *System.Int32* as the class, we can simply take the reference to this variable and call the instance methods of this value type, say, *ToString( )*, which returns a string representation of the integer in question:

```
⋮
.locals init (int32 J)  // Declare variable J as value type
⋮
ldc.i4 12345
stloc J   // J = 12345
⋮
ldloca J // Get managed reference to J as instance pointer
// Call method of this instance
call instance string [mscorlib]System.Int32::ToString()
⋮
```

Derivation of Value Types

All value types are derived from the *[mscorlib]System.ValueType* class. More than that, anything derived from *[mscorlib]System.ValueType* is a value type by definition, with one important exception: the *[mscorlib]System.Enum* class, which is a parent of all enumerators (discussed in the next section).

Unlike C++, in which derivation of a structure from another structure is commonplace, the common language runtime object model does not allow any derivations from value types. All value types must be sealed. (And you probably thought I was too lazy to draw further derivation branches from value types in Figure 6-1!)

Enumerators

Enumerators (a.k.a. *enumeration types*, a.k.a. *enums*) make up a special subset of value types. All enumerators are derived from the *[mscorlib]System.Enum* class, which is the only class derived from *[mscorlib]System.ValueType*. Enumerators are possibly the most primitive of all types, and the rules regarding them are the most restrictive.

Unlike other value types in their boxed form, enumerators don't show any of the characteristics of a "true class." Enumerators can have only fields as members—no methods, properties, or events. Enumerators cannot implement interfaces; because enumerators cannot have methods, the question of implementing interfaces is moot.

Even with the fields the enumerators have no leeway: an enumerator must have exactly one instance field and at least one static field. The instance field of an enumerator represents the value of the current instance of the enumerator and must be of integer, Boolean, or string type. The type of the instance field is the underlying type of the enumerator. The enumerator itself as a value type is completely interchangeable with its underlying type in all operations except boxing. If an operation, other than boxing, expects a Boolean variable as its argument, a variable of a Boolean-based enumeration type can be used instead, and vice versa. A boxing operation, however, always results in a boxed enumerator and not in a boxed underlying type.

The static fields represent the values of the enumeration itself and have the type of the enumerator. As values of the enumeration, these fields must be not only static (shared by all instances of the enumerator) but also *literal*—they represent constants defined in the metadata. The literal fields are not true fields because they do not occupy memory allocated by the loader when the enumerator is loaded. (Chapter 8 discusses this and other aspects of fields.)

Generally speaking, you can think of an enumerator as a restriction of its underlying type to a predefined, finite set of values. As such, an enumerator obviously cannot have any specific layout requirements and must have the *auto* layout flag set.

Delegates

Delegates are a special kind of reference type, designed with the specific purpose of representing function pointers. All delegates are derived from the *[mscorlib]System.MulticastDelegate* class, which in turn is derived from the *[mscorlib]System.Delegate* class. Because delegates themselves are sealed, no types can be derived from them.

Limitations imposed on the structure of a delegate are as strict as those imposed on the enumerator structure. Delegates have no fields, events, or properties. They can have only member methods, either two or four of them, and the names and signatures of these methods are predefined.

Two mandatory methods of a delegate are the instance constructor (*.ctor*) and *Invoke*. The instance constructor returns *void* (as all instance constructors do) and takes two parameters: the object reference to the type defining the method being delegated and the integer value of the function pointer to the managed method being delegated. (See Chapter 9 for details about instance constructors.)

This leads to a question: If we can get a function pointer per se, why do we need delegates at all? Why not use the function pointers directly? We could, but then we would need to introduce fields or variables of function pointer *types* to hold these pointers—and function pointer types are considered a security risk and deemed unverifiable. If a module is unverifiable, it can be executed only from a local drive in full trust mode, when all security checks are disabled. Another drawback is that managed function pointers cannot be marshaled to unmanaged function pointers when calling unmanaged methods, whereas delegates can be. (See Chapter 15 for information on managed and unmanaged code interoperation.)

Delegates are secure, verifiable, and type-safe representations of function pointers and as such are preferable over function pointer types. Besides, delegates can offer additional useful features, as I'll describe in a moment.

The second mandatory method (*Invoke*) must have the same signature as the delegated method. Two mandatory methods (*.ctor* and *Invoke*) are sufficient to allow the delegate to be used for *synchronous calls*, which are the usual method calls when the calling thread is blocked until the called method returns. The first method (*.ctor*) creates the delegate instance and binds it to the delegated method. The *Invoke* method is used to make a synchronous call.

Delegates also can be used for *asynchronous calls*, when the called method is executed on a separate thread created by the common language runtime for this purpose and does not block the calling thread. So that it can be called asynchronously, a delegate must define two additional methods, *BeginInvoke* and *EndInvoke*.

BeginInvoke is a thread starter. It takes all the parameters of the delegated method plus two more: a delegate of type *[mscorlib]System.AsyncCallback* representing a callback method that is invoked when the call completes, and an object we choose to indicate the final status of the call thread. *BeginInvoke* returns an instance of the interface *[mscorlib]System.IAsyncResult*, carrying the object we passed as the last parameter. Remember that because interfaces, delegates, and objects are reference types, when we say "takes a delegate" or "returns an interface," we actually mean a reference.

If we want to be notified immediately when the call is completed, we must specify the *AsyncCallback* delegate. The respective callback method is called upon completion of the asynchronous call. This event-driven technique is the most widely used way to monitor the asynchronous calls.

We might choose another way to monitor the status of the asynchronous call thread: *polling* from the main thread. The returned interface has the method *bool get_IsCompleted( )*, which returns *true* when the asynchronous call is completed. We can call this method from time to time from the main thread to find out whether the call is finished.

We can also call another method of the returned interface, *get_AsyncWaitHandle*, which returns a wait handle, an instance of the *[mscorlib]System.Threading.WaitHandle* class. After we get the wait handle, we can monitor it any way we please (similar to the use of the Win32 APIs *WaitForSingleObject* and *WaitForMultipleObjects*). If you are curious, disassemble Mscorlib.dll and take a look at this class.

Of course, if we have chosen to employ a polling technique, we can forgo the callback function and specify *null* instead of the *System.AsyncCallback* delegate instance.

The *EndInvoke* method takes the *[mscorlib]System.IAsyncResult* interface, returned by *BeginInvoke*, as its single argument and returns *void*. Because this method waits for the asynchronous call to complete, blocking the calling thread, calling it immediately after *BeginInvoke* is equivalent to a synchronous call using *Invoke*. *EndInvoke* must be called eventually in order to clear the corresponding runtime threading table entry, but it should be done when we know that the asynchronous call has been completed.

All four methods of a delegate are virtual and runtime-implemented. When defining a delegate, we can simply declare the methods without providing implementation for them, as shown here:

```
.class public sealed MyDelegate
   extends [mscorlib]System.MulticastDelegate
{
   .method public hidebysig instance
      void .ctor(object MethodsClass,
                 native unsigned int MethodPtr)
         runtime managed { }

   .method public hidebysig virtual instance
      int32 Invoke(void* Arg1, void* Arg2)
         runtime managed { }

   .method public hidebysig newslot virtual instance
      class [mscorlib]System.IAsyncResult
         BeginInvoke(void* Arg1, void* Arg2,
                     class [mscorlib]System.AsyncCallback callBkPtr,
                     object) runtime managed { }

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

Nested Types

Nested types are types (classes, interfaces, value types) that are defined within other types. However, being defined within another type does not make the nested type anything like the member classes or inner classes. The instance pointers (*this*) of a nested type and its enclosing type are in no way related. A nested class does not automatically get access to the *this* pointer of its enclosing class when the instance of the enclosing class is created.

In addition, instantiation of the enclosing class does not involve instantiation of the class(es) nested in it. The nested classes must be instantiated separately. Instantiation of a nested class does not require the enclosing class to be instantiated.

Type nesting is not about membership and joint instantiation; rather, it's all about visibility. As explained earlier in "Class Attributes," nested types at any level of nesting have their own specific visibility flags. When one type is nested in another type, the visibility of the nested type is "filtered" by the visibility of the enclosing type. If, for example, a class whose visibility is set to *nestedpublic* is nested in a *private* class, this nested class will not be visible outside the assembly despite its own specified visibility.

This visibility filtering works throughout all levels of nesting. The final visibility of a nested class is defined by its own declared visibility and then is limited in sequence by the visibilities of all classes enclosing it, directly or indirectly.

Nested classes are defined in ILAsm the same way they are defined in other languages—that is, the nested classes are declared within the lexical scope of their encloser declaration:

```
.namespace MyNS {
   .class public Encl {
      ⋮
      .class nested public Nestd1 {
         ⋮
         .class nested family Nestd2 {
            ⋮
         }
      }
   }
}
```

According to this declaration, the *Nestd2* class is nested in the *Nestd1* class, which in turn is nested in *MyNS.Encl*, which is not a nested class.

Because nested classes belong to their enclosers rather than to namespaces, a nested class name is always the full name. Having said that, let's return for a moment to the experiment with dotted class names described earlier in this chapter, in the section "Full Class Names." In that case, we defined a class with a dotted name, only to find that the left part of the dotted name was moved to the namespace by the metadata emission API. The same thing will happen if we try to define a nested class with a dotted name. Although this "name redistribution" has no ill effect on the top-level classes, which are always referenced by their full names, it does have quite an effect on nested classes, which are not supposed to have namespaces and are addressed by name only. Don't use dotted names for nested classes.

While on the subject of referencing the classes, let's see how the nested classes are referenced in ILAsm:

```
<nested_class_ref> ::= <encloser_ref> / <simple_name>
```

where

```
<encloser_ref> ::= <nested_class_ref> |  <class_ref>
```

and *<class_ref>* has already been defined earlier as follows:

```
<class_ref> ::= [<resolution_scope>]<full_type_name>
```

According to these definitions, classes *Nestd1* and *Nestd2* will be referenced respectively as *MyNS.Encl/Nestd1* and *MyNS.Encl/Nestd1/Nestd2*. Names of nested classes must be unique within their nester, as opposed to the full names of top-level classes, which must be unique within the module or (for public classes) within the assembly.

Unlike Microsoft Visual C# .NET, which uses a dot delimiter for all hierarchical relationships without discrimination—so that *One.Two.Three* might mean "class *Three* of namespace *One.Two*" or "class *Three* nested in class *Two* of namespace *One*" or even "field *Three* of class *Two* nested in class *One*"—ILAsm uses different delimiters for different hierarchies. A dot is used for the full class name hierarchy; a forward slash (/) indicates the nesting hierarchy; and a double colon (::), as in C++, denotes the class-member relationship.

Thus far, the discussion has focused mainly on what nested classes are not. One more important negative to note: nested classes have no effect on the layout of their enclosers. If you want to declare a substructure of a structure, you must declare a nested value type (substructure) within the enclosing value type (structure) and then define a field of the substructure type:

```
.class public value Struct {
   ⋮
   .class nested public value Substruct {
      ⋮
   }
   .field public valuetype Struct/Substruct Substr
}
```

Now I need to say something positive about nested classes. Members of a nested class have access to all members of the enclosing class without exception, including access to private members. In this regard, the nesting relationship is even stronger than inheritance and stronger than the member class relationship in C++, where member classes don't have access to private members of their owner. Of course, to get access to the encloser's instance members, the nested type members should first obtain the instance pointer to the encloser. This "full disclosure" policy works one-way only; the encloser has no access to private members of the nested class.

Nested types can be used as base classes for other types that don't need to be nested:

```
.class public X {
   ⋮
   .class nested public Y {
      ⋮
   }
}
.class public Z extends X/Y {
   ⋮
}
```

Of course, class *Z*, derived from a nested class (*Y*), does not have any access rights to private members of the encloser (*X*). The "full disclosure" privilege is not inheritable.

A nested class can be derived from its encloser. In this case, it retains access to the encloser's private members, and it also acquires an ability to override the encloser's virtual methods. The enclosing class cannot be derived from any of its nested classes.

> **Note** A metadata validity rule states that a nested class must be defined in the same module as its encloser. In ILAsm, however, the only way to define a nested class is to declare it within the encloser's lexical scope, which means that you could not violate this validity rule in ILAsm even if you tried.

Class Augmentation

In ILAsm, as in Microsoft Visual Basic .NET and Visual C# .NET, all members, attributes, and nested classes of a class are declared within the lexical scope of that class. However, ILAsm, unlike Visual Basic .NET and Visual C# .NET, allows you to reopen a once-closed class scope and define additional items:

```
.class public X extends Y implements IX,IY {
   ⋮
}
⋮
// Later in the source, possibly in another source file...
.class X {
   ⋮ // More items defined
}
```

This reopening of the class scope is known as *class augmentation*. A class can be augmented any number of times throughout the source code, and the augmenting segments can reside in different source files. The following simple safety rules govern class augmentation:

- The class must be fully defined within the module—in other words, you cannot augment a class that is defined somewhere else.
- Class flags, the *extends* clause, and the *implements* clause must be fully defined at the lexically first opening of class scope, because these attributes are ignored in augmenting segments.
- None of the augmenting segments can contain duplicate item declarations. If you declare field *X* in one segment and then declare it in another segment, the ILAsm compiler will not appreciate the fact that you probably have the same field in mind and will read it as an attempt to define two identical fields in the same class, which is not allowed.
- The augmenting segments are not explicitly numbered, and the class is augmented according to the sequence of augmenting segments in the source code. This means that the sequence of class item declarations will change if you swap augmenting segments, which in turn might affect the class layout.

A good strategy for writing an ILAsm program is to use forward class declaration, explained in the Chapter 1 section "Forward Declaration of Classes." This strategy allows you to declare *all* classes of the current module, including nested ones, without any members and attributes and to define the members and attributes in augmenting segments. This way, the ILAsm compiler gets the full picture of the module's type declaration structure before any type is referenced. By the time locally declared types are referenced, they all are already defined and have corresponding *TypeDef* metadata records.

Manifest declarations, described in Chapter 5, followed by forward class declarations look a lot like a program header, so I would not blame you if you put them in a separate source file. Just don't forget that this file must be first on the list of source files when you assemble your module.

Metadata Validity Rules

Recall that the type-related metadata tables include the TypeDef, TypeRef, InterfaceImpl, NestedClass, and ClassLayout tables. The records of these tables contain the following entries:

- The TypeDef table contains the *Flags*, *Name*, *Namespace*, *Extends*, *FieldList*, and *MethodList* entries.
- The TypeRef table contains the *ResolutionScope*, *Name*, and *Namespace* entries.
- The InterfaceImpl table contains the *Class* and *Interface* entries.
- The NestedClass table contains the *NestedClass* and *EnclosingClass* entries.
- The ClassLayout table contains the *PackingSize*, *ClassSize*, and *Parent* entries.

TypeDef Table Validity Rules

- The *Flags* entry can have only those bits set that are defined in the enumeration *CorTypeAttr* in CorHdr.h (validity mask: 0x00173DBF).
- [run time] The *Flags* entry cannot have the *sequential* and *explicit* bits set simultaneously.
- [run time] The *Flags* entry cannot have the *unicode* and *autochar* bits set simultaneously.
- If the *rtspecialname* flag is set in the *Flags* entry, the *Name* field must be set to *_Deleted**, and vice versa.
- [run time] If the bit 0x00040000 is set in the *Flags* entry, either a *DeclSecurity* record or a custom attribute named *SuppressUnmanagedCodeSecurityAttribute* must be associated with the *TypeDef*, and vice versa.
- [run time] If the *interface* flag is set in the *Flags* entry, *abstract* must be also set.

- [run time] If the *interface* flag is set in the *Flags* entry, *sealed* must not be set.
- [run time] If the *interface* flag is set in the *Flags* entry, the *TypeDef* must have no instance fields.
- [run time] If the *interface* flag is set in the *Flags* entry, all the *TypeDef*'s instance methods must be abstract.
- [run time] The visibility flag of a nonnested *TypeDef* must be set to *private* or *public*.
- [run time] If the visibility flag of a *TypeDef* is set to *nested public*, *nested private*, *nested family*, *nested assembly*, *nested famorassem*, or *nested famandassem*, the *TypeDef* must be referenced in the *NestedClass* entry of one of the records in the NestedClass metadata table, and vice versa.
- The *Name* field must reference a nonempty string in the *#Strings* stream.
- The combined length of the strings referenced by the *Name* and *Namespace* entries must not exceed 1023 bytes.
- The TypeDef table must contain no duplicate records with the same full name (the namespace plus the name) unless the *TypeDef* is nested or deleted.
- [run time] The *Extends* entry must be nil for *TypeDef*s with the *interface* flag set and for the *TypeDef System.Object* of the Mscorlib assembly.
- [run time] The *Extends* entry of all other *TypeDef*s must hold a valid reference to the TypeDef or TypeRef table, and this reference must point at a nonsealed class (not an interface or a value type).
- [run time] The *Extends* entry must not point to the type itself or to any of the type descendants (inheritance loop).
- [run time] The *FieldList* entry can be nil or hold a valid reference to the Field table.
- [run time] The *MethodList* entry can be nil or hold a valid reference to the Method table.

Enumerator-Specific Validity Rules

If the *TypeDef* is an enumerator—that is, if the *Extends* entry holds the reference to the class *[mscorlib]System.Enum*—the following additional rules apply:

- [run time] The *interface*, *abstract*, *sequential*, and *explicit* flags must not be set in the *Flags* entry.
- The *sealed* flag must be set in the *Flags* entry.
- The *TypeDef* must have no methods, events, or properties.
- The *TypeDef* must implement no interfaces—that is, it must not be referenced in the *Class* entry of any record in the InterfaceImpl table.
- [run time] The *TypeDef* must have at least one instance field of integer type, or of type *bool* or *string*.
- [run time] All static fields of the *TypeDef* must be literal.
- The type of the static fields of the *TypeDef* must be the current *TypeDef* itself.

TypeRef Table Validity Rules

- [run time] The *ResolutionScope* entry must hold either 0 or a valid reference to the AssemblyRef, ModuleRef, Module, or TypeRef table. In the last case, *TypeRef* refers to a type nested in another type (a nested *TypeRef*).
- If the *ResolutionScope* entry is nil, the ExportedType table of the prime module of the assembly must contain a record whose *TypeName* and *TypeNamespace* entries match the *Name* and *Namespace* entries of the *TypeRef* record, respectively.
- [run time] The *Name* entry must reference a nonempty string in the *#Strings* stream.
- [run time] The combined length of the strings referenced by the *Name* and *Namespace* entries must not exceed 1023 bytes.
- The table must contain no duplicate records with the same full name (the namespace plus the name) and *ResolutionScope* value.

InterfaceImpl Table Validity Rules

A *Class* entry set to nil means a deleted *InterfaceImpl* record. If the *Class* entry is non-nil, however, the following rules apply:

- [run time] The *Class* entry must hold a valid reference to the TypeDef table.
- [run time] The *Interface* entry must hold a valid reference to the TypeDef or TypeRef table.

- If the *Interface* field references the TypeDef table, the corresponding *TypeDef* record must have the *interface* flag set in the *Flags* entry.
- The table must contain no duplicate records with the same *Class* and *Interface* entries.

NestedClass Table Validity Rules

- The *NestedClass* entry must hold a valid reference to the TypeDef table.
- [run time] The *EnclosingClass* entry must hold a valid reference to the TypeDef table, one that differs from the reference held by the *NestedClass* entry.
- The table must contain no duplicate records with the same *NestedClass* entries.
- The table must contain no records with the same *EnclosingClass* entries and *NestedClass* entries referencing *TypeDef* records with matching names—in other words, a nested class must have a unique name within its encloser.
- The table must contain no sets of records forming a circular nesting pattern—for example, A nested in B, B nested in C, C nested in A.

ClassLayout Table Validity Rules

A *Parent* entry set to nil means a deleted *ClassLayout* record. However, if the *Parent* entry is non-nil, the following rules apply:

- The *Parent* entry must hold a valid reference to the TypeDef table, and the referenced *TypeDef* record must have the *Flags* bit *explicit* or *sequential* set and must have the *interface* bit not set.
- [run time] The *PackingSize* entry must be set to 0 or to a power of 2 in the range 1 to 128.
- The table must contain no duplicate records with the same *Parent* entries.

7

Primitive Types and Signatures

Having looked at how types are defined in the common language runtime and IL assembly language (ILAsm), let's proceed to the question of how these types and their derivatives are assigned to program items—fields, variables, methods, and so on. The constructs defining the types of program items are known as the *signatures* of these items. Signatures are built from encoded references to various classes and value types; I'll discuss signatures in detail in this chapter.

But before we start analyzing the signatures of program items, let's consider the building blocks of these signatures.

Primitive Types in the Common Language Runtime

All types have to be defined somewhere. The Microsoft .NET Framework class library defines hundreds of types, and other assemblies build their own types based on the types defined in the class library. Some of the types defined in the class library are recognized by the common language runtime as primitive types and are given special encoding in the signatures. This is done only for the sake of performance—theoretically, the signatures could have been built from type tokens only, given that every type is defined somewhere and hence has a token. But resolving all these tokens simply to find that they reference trivial items such as a 4-byte integer or a Boolean value can hardly be considered a sensible way to work in the runtime.

Primitive Data Types

The term *primitive data types* refers to the types defined in the .NET Framework class library that are given specific individual type codes to be used in signatures. Because all these types are defined in the assembly Mscorlib and all belong to the namespace *System*, I have omitted the prefix *[mscorlib]System* when supplying the class library type name for a type.

The individual type codes are defined in the enumeration *CorElementType* in the header file CorHdr.h. The names of all these codes begin with *ELEMENT_TYPE_*, which I have either omitted in this chapter or abbreviated as *E_T_*.

Table 7-1 describes primitive data types and their respective ILAsm notation.

Table 7-1 Primitive Data Types Defined in the Runtime

Code	Constant Name	.NET Framework Type Name	ILAsm Notation	Comments
0x01	VOID	Void	void	
0x02	*BOOLEAN*	*Boolean*	*bool*	Single-byte value, *true* = 1, *false* = 0
0x03	*CHAR*	*Char*	*char*	2-byte unsigned integer, representing a Unicode character
0x04	*I1*	*SByte*	*int8*	Signed 1-byte integer, the same as *char* in C/C++
0x05	*U1*	*Byte*	*unsigned int8*	Unsigned 1-byte integer
0x06	*I2*	*Int16*	*int16*	Signed 2-byte integer
0x07	*U2*	*UInt16*	*unsigned int16*	Unsigned 2-byte integer
0x08	*I4*	*Int32*	*int32*	Signed 4-byte integer

Table 7-1 Primitive Data Types Defined in the Runtime *(continued)*

Code	Constant Name	.NET Framework Type Name	ILAsm Notation	Comments
0x09	*U4*	*UInt32*	*unsigned int32*	Unsigned 4-byte integer
0x0A	*I8*	*Int64*	*int64*	Signed 8-byte integer
0x0B	*U8*	*UInt64*	*unsigned int64*	Unsigned 8-byte integer
0x0C	*R4*	*Single*	*float32*	4-byte floating-point
0x0D	*R8*	*Double*	*float64*	8-byte floating-point
0x16	*TYPEDBY-REF*	*TypedReference*	*typedref*	Typed reference, carrying both reference to a type and information identifying the referenced type
0x18	*I*	*IntPtr*	*native int*	Pointer-size integer; size dependent on the underlying platform, hence use of the keyword *native*
0x19	*U*	*UIntPtr*	*native unsigned int*	Pointer-size unsigned integer

Data Pointer Types

Two data pointer types are defined in the common language runtime: the managed pointer, which is a reference, and the unmanaged pointer, which is a pointer in the conventional sense. The difference is that a *managed pointer* is managed by the runtime's garbage collection subsystem and stays valid even if the referenced item is moved in memory during the process of garbage collection, whereas an *unmanaged pointer* can be safely used only in association with "unmovable" items.

Both pointer types have no meaning per se and must be followed by the base types, which are the types to which the pointer types point. As derivatives from base types, the pointer types have no corresponding types defined in the .NET Framework class library and cannot be boxed. Table 7-2 describes the two pointer types and their ILAsm notations. Neither of them has a respective .NET Framework type associated.

Table 7-2 Pointer Types Defined in the Runtime

Code	Constant Name	ILAsm Notation	Comments
0x0F	PTR	*<type>**	Unmanaged pointer to *<type>*
0x10	*BYREF*	*<type>&*	Managed pointer to *<type>*

Note Note that although ILAsm notation places the pointer sign after the pointed type, in signatures *E_T_PTR* and *E_T_BYREF* always precede the pointed type.

Pointers of both types are subject to standard pointer arithmetic: an integer can be added to or subtracted from a pointer, resulting in a pointer; and one pointer can be subtracted from another, resulting in an integer value. The difference between pointer arithmetic in, say, C/C++ and in IL (intermediate language) is that in IL—and hence in ILAsm—the increments and decrements of pointers are always specified in bytes, regardless of the size of the item the pointer represents.

C/C++:

```
long L, *pL=&L;
⋮
pL += 4; // pL is incremented by 4*sizeof(long) = 16 bytes
```

ILAsm:

```
.locals init(int32 L, int32& pL)
ldloca L   // Load pointer to L on stack
stloc pL   // pL = &L
⋮
ldloc pL   // Load pL on stack
ldc.i4 4   // Load 4 on stack
add
stloc pL   // pL += 4, pL is incremented by 4 bytes
```

By the same token—now, this is just a common expression. I'm not referring to metadata tokens. (I think I'd better be extra careful with phrases like "by the same token" or "token of appreciation" in this book.) In the same way, the delta of two pointers in IL is always expressed in bytes, not in the items pointed at.

Using unmanaged pointers in IL is not considered nice. Because of the unlimited access that C-style pointer arithmetic gives to anybody for anything, IL code, which has unmanaged pointers dereferenced, is deemed unverifiable and can be run only from a local drive with run-time code verification disabled.

Managed pointers are tamed, domesticated pointers, fully owned by the common language runtime type control and the garbage collection subsystem. These pointers dwell in a safe but not too spacious corral, fenced along the following lines:

- Managed pointers are always references to an item in existence—a field, an array element, a local variable, a method argument.
- Managed pointer types can be used only for method attributes—local variables, parameters, or a return type.
- Array elements and fields cannot have managed pointer types. Local variables and method parameters can, and it is not a simple coincidence that all these items are stack-allocated.
- Managed pointers that point to "managed memory" (the garbage collector heap, which contains object instances and arrays) cannot be converted to unmanaged pointers.
- Managed pointers that don't point to the garbage collector heap can be converted to unmanaged pointers, but such conversion renders the IL code unverifiable.
- The underlying type of a managed pointer cannot be another pointer, but it can be an object reference.

Managed pointers are different from object references. In Chapter 6, "Namespaces and Classes," which described boxing and unboxing of the value types, we saw that it takes boxing to create an object reference to a value type. Using a simple reference—that is, a managed pointer—is not enough.

The difference is that an object reference points to the method table of an object, whereas a managed pointer points to the value (data) part of the item. When you take a managed pointer to an instance of a value type, you address the data part. You can have only this much because instances of value types, not being objects, have no method tables.

When you box a value type instance, you create an object, a class instance with its own method table and data part copied from the value type instance. This object is represented by an object reference.

Function Pointer Types

Chapter 6 briefly described the use of managed function pointers and compared them with delegate types. Managed function pointers are represented by type E_T_FNPTR, which is indicated by the value 0x1B and doesn't have a .NET Framework type associated.

Just like a data pointer type, a function pointer type does not exist by itself and must be followed by the full signature of the function to which it points. (Method signatures are discussed later in this chapter; see "Signatures.")

The ILAsm notation for a function pointer is as follows:

```
<call_conv> <return_type> * (<type>[,<type>*])
```

where *<call_conv>* is a calling convention, *<return_type>* is the return type, and the *<type>* sequence in the parentheses is the argument list. You'll find more details in the "Signatures" section.

Vectors and Arrays

The common language runtime recognizes two types of arrays: vectors and multidimensional arrays, as described in Table 7-3. Vectors are single-dimensional arrays with a zero lower bound. Multidimensional arrays, which I'll refer to as *arrays*, can have more than one dimension and nonzero lower bounds. Neither of these two types of arrays has a respective .NET Framework type associated.

Table 7-3 Arrays Supported in the Runtime

Code	Constant Name	ILAsm Notation	Comments
0x1D	SZARRAY	<type>[ ]	Vector of *<type>*
0x14	*ARRAY*	*<type>[<bounds> [,<bounds>*]]*	Array of *<type>*

All vectors and arrays are objects (class instances) derived from the abstract class *[mscorlib]System.Array*. This is a very peculiar class; in fact, it is a construct known as a *generic*.

Vector encoding is very simple: *E_T_SZARRAY* followed by the encoding of the underlying type, which can be anything except *void*. The size of the vector is not part of the encoding. Because arrays and vectors are object references, it is not enough to simply declare an array—you must create an instance of it, using the instruction *newarr* for a vector or calling an array constructor. It is at that point that the size of the vector or array instance is specified.

Array encoding is more sophisticated:

```
E_T_ARRAY<underlying_type><rank><num_sizes><size1>…<sizeN>
        <num_lower_bounds><lower_bound1>…<lower_boundM>
```

where the following is true:

```
<underlying_type> cannot be void
<rank> is the number of array dimensions (K>0)
<num_sizes> is the number of specified sizes for dimensions (N = K)
<sizen> is an unsigned integer specifying the size (n = 1,…,N)
<num_lower_bounds> is the number of specified lower bounds (M = K)
<lower_boundm> is a signed integer specifying the lower bound (m =
1,…,M)
```

All the above unsigned integer values are compressed according to the length compression formula discussed in Chapter 4, "Metadata Tables Organization." To save you a trip three chapters back, I will repeat this formula in Table 7-4.

Table 7-4 The Length Compression Formula for Unsigned Integers

Value Range	Compressed Size	Compressed Value (Big-Endian)
0–0x7F	1 byte	<value>
0x80–0x3FFF	2 bytes	0x8000 | *<value>*
0x4000–0x1FFFFFFF	4 bytes	0xC0000000 | *<value>*

Signed integer values (lower bound values) are compressed according to a different compression procedure. First the signed integer is encoded as an unsigned integer by taking the absolute value of the original integer, shifting it left by 1 bit, and setting the least significant bit according to the most significant (sign) bit of the original value. Then compression is applied according to the formula shown in Table 7-4.

If size and/or the lower bound for a dimension are not specified, they are not presumed to be 0; rather, they are marked as not specified. The specification of size and lower bound cannot have "holes"—that is, if you have an array of rank 5 and want to specify size (or lower bound) for its third dimension, you must specify size (or lower bound) for the first and second dimensions as well.

An array specification in ILAsm looks like this:

```
<type> [ <bounds>[, <bounds>*] ]
```

where

```
<bounds> ::= [<lower_bound>] … [<upper_bound>]
```

The following is an example:

```
int32[…, …] // Two-dimensional array with undefined lower bounds
            // And sizes
int32[2…5]  // One-dimensional array with lower bound 2 and size 4
int32[0…, 0…] // Two-dimensional array with zero lower bounds
              // And undefined sizes
```

If neither lower bound nor upper bound is specified for a dimension in a multidimensional array declaration, the ellipsis can be omitted. Thus *int32[…,…]* and *int32[,]* mean the same: a two-dimensional array with no lower bounds or sizes specified.

This omission does not work in the case of single-dimensional arrays, however. The notation *int32[]* indicates a vector (*<E_T_SZARRAY><E_T_I4>*), and *int32[…]* indicates an array of rank 1 whose lower bound and size are undefined (*<E_T_ARRAY><E_T_I4><1><0><0>*).

The common language runtime treats multidimensional arrays and vectors of vectors (of vectors, and so on) completely differently. The specifications *int32[,]* and *int32[ ][ ]* result in different type encoding, are created differently, and are laid out differently when created:

- ***int32[,]*** This specification has the encoding *<E_T_ARRAY><E_T_I4><1><0><0>*, is created by a single call to an array constructor, and is laid out as a contiguous two-dimensional array of *int32*.
- ***int32[ ][ ]*** This specification has the encoding *<E_T_SZARRAY><E_T_SZARRAY><E_T_I4>*, is created by a series of *newarr* instructions, and is laid out as a vector of vector references, each pointing to a contiguous vector of *int32*, with no guarantee regarding the location of each vector. Vectors of vectors are useful for describing jagged arrays, when the size of the second dimension varies depending on the first dimension index.

Modifiers

Four built-in common language runtime types, described in Table 7-5, do not denote any specific data or pointer type but rather are used as modifiers of data and pointer types. None of these modifiers have a respective .NET Framework type associated.

Table 7-5 Custom Modifiers Defined in the Runtime

Code	Constant Name	ILAsm Notation	Comments
0x1F	CMOD_REQD	*modreq(<class_ref>)*	Required C modifier
0x20	*CMOD_OPT*	*modopt(<class_ref>)*	Optional C modifier
0x41	*SENTINEL*	...	Start of optional arguments in a *vararg* method call
0x45	*PINNED*	*pinned*	Marks a local variable as unmovable by the garbage collector

The modifiers *modreq* and *modopt* indicate that the item to which they are attached—an argument, a return type, or a field, for example—must be treated in some special way. These modifiers are followed by *TypeDef* or *TypeRef* tokens, and the classes corresponding to these tokens indicate the special way the item is to be handled.

The tokens following *modreq* and *modopt* are compressed according to the following algorithm. As you might remember, an uncoded (external) metadata token is a 4-byte unsigned integer, which has the token type in its senior byte and a record index (RID) in its 3 lower bytes. It so happens that the tokens appearing in the signatures and hence requiring compression are of three types only: *TypeDef*, *TypeRef*, or *TypeSpec*. (See "Signatures" later in this chapter for information about *TypeSpec*s.) Because of that, only 2 bits, rather than a whole byte, are required for the token type: 00 denotes *TypeDef*, 01 is used for *TypeRef*, and 10 for *TypeSpec*. The token compression procedure resembles the procedure used to compress the signed integers: the RID part of the token is shifted left by 2 bits, and the 2-bit type encoding is placed in the least significant bits. The result is compressed just as any unsigned integer would be, according to the formula shown earlier in Table 7-4.

The modifiers *modreq* and *modopt* are used primarily by tools other than the common language runtime, such as compilers or program analyzers. The *modreq* modifier indicates that the modifier must be taken into account, whereas *modopt* indicates that the modifier is optional and can be ignored. The ILAsm compiler does not use these modifiers for its internal purposes.

The only use of the *modreq* and *modopt* modifiers recognized by the common language runtime is when these modifiers are applied to return types or parameters of methods subject to managed/unmanaged marshaling. For example, to specify that a managed method must have the *cdecl* calling convention when it is marshaled as unmanaged, we can use the following modifier attached to the method's return type:

```
modopt([mscorlib]System.Runtime.InteropServices.CallConvCdecl)
```

When used in the context of managed/unmanaged marshaling, the *modreq* and *modopt* modifiers are equivalent.

Although the *modreq* and *modopt* modifiers have no effect on the managed types of the items to which they are attached, signatures with and without these modifiers are considered different. The same is true for signatures differing only in classes referenced by these modifiers.

The sentinel modifier (...) was introduced in Chapter 1, "Simple Sample," when we analyzed the declaration and calling of methods with a variable-length argument list (*vararg* methods). (See "Method Declaration.") A sentinel signifies the beginning of optional arguments supplied for a *vararg* method call. This modifier can appear in only one context: at the call site, because the optional parameters of a *vararg* method are not specified when such a method is declared. The runtime treats a sentinel appearing in any other context as an error. The method arguments at the call site can contain only one sentinel, and the sentinel is used only if optional arguments are supplied:

```
// Declaration of vararg method - mandatory parameters only:
.method public static vararg int32 Print(string Format)
{
   ⋮
}
⋮
// Calling vararg method with two optional arguments:
call vararg int32 Print(string, ..., int32, int32)
⋮
// Calling vararg method without optional arguments:
call vararg int32 Print(string)
```

The *pinned* modifier is applicable to the method's local variables only. Its use means that the local variable cannot be relocated by the garbage collector and must stay put throughout the method execution. If a local variable is "pinned," it is safe to convert a managed pointer to this variable to an unmanaged pointer and then to dereference this unmanaged pointer, because the unmanaged pointer is guaranteed to still be valid when it is dereferenced:

```
.locals init(int32 A, int32 pinned B, int32* pA, int32* pB)
ldloca A
stloc pA      // pA = &A
ldloca B
stloc pB      // pB = &B
```

```
⋮
ldloc pA
ldc.i4 123
stind.i4      // *pA=123 - unsafe, A could have been moved
ldloc pB
ldc.i4 123
stind.i4      // *pB=123 - safe, B is pinned and cannot move
```

Native Types

When managed code calls unmanaged methods or exposes managed fields to unmanaged code, it is sometimes necessary to provide specific information about how the managed types should be marshaled to and from the unmanaged types. The unmanaged types recognizable by the common language runtime are referred to as *native*, and they are listed in CorHdr.h in the enumeration *CorNativeType*. All constants in this enumeration have names that begin with *NATIVE_TYPE_**; for purposes of this discussion, I have omitted this part of the names or abbreviated it as *N_T_*. The same constants are also listed in the .NET Framework class library in the enumerator *System.Runtime.InteropServices.UnmanagedType*.

Some of the native types are obsolete and are ignored by the runtime interoperability subsystem. But since these native types are not retired altogether, ILAsm must have ways to denote them—and since ILAsm denotes these types, I cannot help but list obsolete types along with others, all of which you'll find in Table 7-6.

Table 7-6 Native Types Defined in the Runtime

Code	Constant Name	.NET Framework Type Name	ILAsm Notation	Comments
0x01	VOID		void	Obsolete and thus should not be used; recognized by ILAsm but ignored by thre runtime interoperability subsystem
0x02	*BOOLEAN*	*Bool*	*bool*	4-byte Boolean value; *true* = nonzero, *false* = 0
0x03	*I1*	*I1*	*int8*	Signed 1-byte integer
0x04	*U1*	*U1*	*unsigned int8*	Unsigned 1-byte integer
0x05	*I2*	*I2*	*int16*	Signed 2-byte integer

(continued)

Table 7-6 Native Types Defined in the Runtime *(continued)*

Code	Constant Name	.NET Framework Type Name	ILAsm Notation	Comments
0x06	*U2*	*U2*	*unsigned int16*	Unsigned 2-byte integer
0x07	*I4*	*I4*	*int32*	Signed 4-byte integer
0x08	*U4*	*U4*	*unsigned int32*	Unsigned 4-byte integer
0x09	*I8*	*I8*	*int64*	Signed 8-byte integer
0x0A	*U8*	*U8*	*unsigned int64*	Unsigned 8-byte integer
0x0B	*R4*	*R4*	*float32*	4-byte floating-point
0x0C	*R8*	*R8*	*float64*	8-byte floating-point
0x0D	*SYSCHAR*		*syschar*	Obsolete
0x0E	*VARIANT*		*variant*	Obsolete
0x0F	*CURRENCY*	*Currency*	*currency*	Currency value
0x10	*PTR*		*	Obsolete; use *native int*
0x11	*DECIMAL*		*decimal*	Obsolete
0x12	*DATE*		*date*	Obsolete
0x13	*BSTR*	*BStr*	*bstr*	Unicode Visual Basic–style string
0x14	*LPSTR*	*LPStr*	*lpstr*	Pointer to a zero-terminated ANSI string
0x15	*LPWSTR*	*LPWStr*	*lpwstr*	Pointer to a zero-terminated Unicode string
0x16	*LPTSTR*	*LPTStr*	*lptstr*	Pointer to a zero-terminated ANSI or Unicode string, depending on platform
0x17	*FIXED-SYSSTRING*	*ByValTStr*	*fixed sysstring [<size>]*	Fixed-system string of size <*size*> bytes; applicable to field marshaling only
0x18	*OBJECTREF*		*objectref*	Obsolete
0x19	*IUNKNOWN*	*IUnknown*	*iunknown*	IUnknown interface pointer
0x1A	*IDISPATCH*	*IDispatch*	*idispatch*	IDispatch interface pointer

Table 7-6 Native Types Defined in the Runtime *(continued)*

Code	Constant Name	.NET Framework Type Name	ILAsm Notation	Comments
0x1B	*STRUCT*	*Struct*	*struct*	C-style structure, for marshaling the formatted managed types
0x1C	*INTF*	*Interface*	*interface*	Interface pointer
0x1D	*SAFEARRAY*	*SafeArray*	*safearray <variant_type>*	Safe array of type *<variant_type>*
0x1E	*FIXEDARRAY*	*ByValArray*	*fixed array [<size>]*	Fixed-size array, of size *<size>* bytes
0x1F	*INT*	*IntPtr*	*int*	Signed pointer-size integer
0x20	*UINT*	*UIntPtr*	*unsigned int*	Unsigned pointer-size integer
0x21	*NESTEDSTRUCT*		*nested struct*	Obsolete; use *struct*
0x22	*BYVALSTR*	*VBByRefStr*	*byvalstr*	Visual Basic–style string in a fixed-length buffer
0x23	*ANSIBSTR*	*AnsiBStr*	*ansi bstr*	ANSI Visual Basic–style string
0x24	*TBSTR*	*TBStr*	*tbstr*	*bstr* or *ansi bstr*, depending on the platform
0x25	*VARIANTBOOL*	*VariantBool*	*variant bool*	2-byte Boolean; *true* = -1, *false* = 0
0x26	*FUNC*	*FunctionPtr*	*method*	Function pointer
0x28	*ASANY*	*AsAny*	*as any*	Object; type defined at run time
0x2A	*ARRAY*	*LPArray*	*<n_type> [<sizes>]*	Fixed-size array of a native type *<n_type>*
0x2B	*LPSTRUCT*	*LPStruct*	*lpstruct*	Pointer to a C-style structure
0x2C	*CUSTOMMARSHALER*	*CustomMarshaler*	*custom (<class_str>, <cookie_str>)*	Custom marshaler
0x2D	*ERROR*	*Error*	*error*	Maps *int32* to *VT_HRESULT*

The *<sizes>* parameter in the ILAsm notation for *ARRAY*, shown in Table 7-6, can be empty or can be formatted as *<size>* + *<size_param_number>*:

```
<sizes> ::= <>
            | <size>
            | + <size_param_number>
            | <size> + <size_param_number>
```

If *<sizes>* is empty, the size of the native array is derived from the size of the managed array being marshaled.

The *<size>* parameter specifies the native array size in array items. The zero-based method parameter number *<size_param_number>* indicates which of the method parameters specifies the size of the native array. The total size of the native array is *<size>* plus the additional size specified by the method parameter that is indicated by *<size_param_number>*.

A custom marshaler declaration (shown in Table 7-6) has two parameters, both of which are quoted strings. The *<class_str>* parameter is the name of the class representing the custom marshaler, using the string conventions of *Reflection.Emit*. The *<cookie_str>* parameter is an argument string (cookie) passed to the custom marshaler at run time. This string identifies the form of the marshaling required, and its notation is specific to the custom marshaler.

Variant Types

Variant types are defined in the enumeration *VARENUM* in the Wtypes.h file, which is distributed with Microsoft Visual Studio. Not all variant types are applicable as safe array types, according to Wtypes.h, but ILAsm provides notation for all of them nevertheless, as shown in Table 7-7. It might look strange, considering that variant types appear in ILAsm only in the context of safe array specification, but we should not forget that one of ILAsm's principal applications is the generation of test programs, which contain known, preprogrammed errors.

Table 7-7 Variant Types Defined in the Runtime

Code	Constant Name	Applicable to Safe Array?	ILAsm Notation
0x00	VT_EMPTY	No	<empty>
0x01	*VT_NULL*	No	*null*
0x02	*VT_I2*	Yes	*int16*
0x03	*VT_I4*	Yes	*int32*
0x04	*VT_R4*	Yes	*float32*

Table 7-7 Variant Types Defined in the Runtime *(continued)*

Code	Constant Name	Applicable to Safe Array?	ILAsm Notation
0x05	*VT_R8*	Yes	*float64*
0x06	*VT_CY*	Yes	*currency*
0x07	*VT_DATE*	Yes	*date*
0x08	*VT_BSTR*	Yes	*bstr*
0x09	*VT_DISPATCH*	Yes	*idispatch*
0x0A	*VT_ERROR*	Yes	*error*
0x0B	*VT_BOOL*	Yes	*bool*
0x0C	*VT_VARIANT*	Yes	*variant*
0x0D	*VT_UNKNOWN*	Yes	*iunknown*
0x0E	*VT_DECIMAL*	Yes	*decimal*
0x10	*VT_I1*	Yes	*int8*
0x11	*VT_UI1*	Yes	*unsigned int8*
0x12	*VT_UI2*	Yes	*unsigned int16*
0x13	*VT_UI4*	Yes	*unsigned int32*
0x14	*VT_I8*	No	*int64*
0x15	*VT_UI8*	No	*unsigned int64*
0x16	*VT_INT*	Yes	*int*
0x17	*VT_UINT*	Yes	*unsigned int*
0x18	*VT_VOID*	No	*void*
0x19	*VT_HRESULT*	No	*hresult*
0x1A	*VT_PTR*	No	*
0x1B	*VT_SAFEARRAY*	No	*safearray*
0x1C	*VT_CARRAY*	No	*carray*
0x1D	*VT_USERDEFINED*	No	*userdefined*
0x1E	*VT_LPSTR*	No	*lpstr*
0x1F	*VT_LPWSTR*	No	*lpwstr*
0x24	*VT_RECORD*	Yes	*record*
0x40	*VT_FILETIME*	No	*filetime*
0x41	*VT_BLOB*	No	*blob*
0x42	*VT_STREAM*	No	*stream*
0x43	*VT_STORAGE*	No	*storage*
0x44	*VT_STREAMED_OBJECT*	No	*streamed_object*
0x45	*VT_STORED_OBJECT*	No	*stored_object*

(continued)

Table 7-7 Variant Types Defined in the Runtime *(continued)*

Code	Constant Name	Applicable to Safe Array?	ILAsm Notation
0x46	*VT_BLOB_OBJECT*	No	*blob_object*
0x47	*VT_CF*	No	*cf*
0x48	*VT_CLSID*	No	*clsid*
0x1000	*VT_VECTOR*	Yes	*<v_type> vector*
0x2000	*VT_ARRAY*	Yes	*<v_type> [ ]*
0x4000	*VT_BYREF*	Yes	*<v_type> &*

Representing Classes in Signatures

The classes and value types in general are represented in signatures by their *TypeDef* or *TypeRef* tokens, preceded by *E_T_CLASS* or *E_T_VALUETYPE*, respectively, as shown in Table 7-8.

Table 7-8 Representation of *CLASS* and *VALUETYPE*

Code	Constant Name	.NET Framework Type Name	ILAsm Notation	Comments
0x11	VALUETYPE		*valuetype <class_ref>*	Value type
0x12	*CLASS*		*class <class_ref>*	Class or interface, except *[mscorlib]System.Object* and *[mscorlib]System.String*
0x0E	*STRING*	*String*	*string*	*[mscorlib]System.String* class
0x1C	*OBJECT*	*Object*	*object*	*[mscorlib]System.Object* class

As you can see in Table 7-8, two classes, *String* and *Object*, are assigned their own codes and hence should have been listed along with primitive data types in Table 7-1, if it were not for their class nature. This is important: if a type (class or value type) is given its own code, it cannot be referenced in signatures other than by this code. In other words, the class *[mscorlib]System.Object* must appear in signatures as *E_T_OBJECT* and never as *E_T_CLASS<token_of_Object>*, and the value type *[mscorlib]System.Int32* must appear in signatures as *E_T_I4* and never as *E_T_VALUETYPE<token_of_Int32>*.

The JIT (just-in-time) compiler does not accept "long forms" of type encoding for types that have dedicated type codes assigned to them, and run-time signature validation procedures reject such signatures.

Important If a type (class or value type) is given its own code, it cannot be referenced in signatures other than by this code.

Signatures

Now that you know more about type encoding, let's look at how the item types are set in the common language runtime. Program items such as fields, methods, and local variables are not characterized by encoded types; rather, they are characterized by signatures. A *signature* is a binary object containing one or more encoded types and residing in the *#Blob* stream of metadata.

The following metadata tables refer to the signatures:

- **Field table** Field declaration signature
- **Method table** Method declaration signature
- **Property table** Property declaration signature
- **MemberRef table** Field or method referencing signature
- **StandAloneSig table** Local variables or indirect call signature
- **TypeSpec table** Type specification signature

Calling Conventions

The first byte of a signature defines the calling convention of the signature, which in turn identifies the type of the signature. The CorHdr.h file defines the following calling convention constants in the enumeration *CorCallingConvention*:

- ***IMAGE_CEE_CS_CALLCONV_DEFAULT*(0x0)** Default ("nomal") method with a fixed-length argument list. ILAsm has no keyword for this calling convention.
- ***IMAGE_CEE_CS_CALLCONV_VARARG* (0x5)** Method with a variable-length argument list. The ILAsm keyword is *vararg*.

- ***IMAGE_CEE_CS_CALLCONV_FIELD* (0x6)** Field. ILAsm has no keyword for this calling convention.
- ***IMAGE_CEE_CS_CALLCONV_LOCAL_SIG* (0x7)** Local variables. ILAsm has no keyword for this calling convention.
- ***IMAGE_CEE_CS_CALLCONV_PROPERTY* (0x8)** Property. ILAsm has no keyword for this calling convention.
- ***IMAGE_CEE_CS_CALLCONV_UNMGD* (0x9)** Unmanaged calling convention, not currently used by the common language runtime and not recognized by ILAsm.
- ***IMAGE_CEE_CS_CALLCONV_HASTHIS* (0x20)** Instance method that has an instance pointer (*this*) as an implicit first argument. The ILAsm keyword is *instance*.
- ***IMAGE_CEE_CS_CALLCONV_EXPLICITTHIS* (0x40)** Method call signature. The first explicitly specified parameter is the instance pointer. The ILAsm keyword is *explicit*.

The calling conventions *instance* and *explicit* are the modifiers of the default and *vararg* method calling conventions. The calling convention *explicit* can be used only in conjunction with *instance* and only at the call site, never in the method declaration.

Calling conventions for field, property, and local variables signatures don't need special ILAsm keywords because they are inferred from the context.

Field Signatures

A field signature is the simplest kind of signature. It consists of a single encoded type (SET), which of course follows the calling convention byte:

```
<field_sig> ::= <callconv_field> <SET>
```

Although this type encoding (*SET*) can be quite long, especially in the case of a multidimensional array or a function pointer, it is nevertheless a single type encoding. In a field signature, *SET* cannot have *&* or *pinned* or *sentinel* modifiers, and it cannot be *void*.

The field calling convention is always equal to *IMAGE_CEE_CS_CALLCONV_FIELD*, regardless of whether the field is static or instance. The information is inferred from the context in which the field is referenced.

Method and Property Signatures

The structures of method and property signatures (and I am talking about method and property *declarations* here) are similar:

```
<method_sig> ::= <callconv_method> <num_of_args> <return_type>
                   [<arg_type>[,<arg_type>*] ]

<prop_sig> ::= <callconv_prop> <num_of_args> <return_type>
                 [<arg_type>[,<arg_type>*] ]
```

The difference is in the calling convention. The calling convention for a method signature is the following:

```
< callconv_method > ::= <default>  // Static method, default
                                   // calling convention
                      | vararg      // Static vararg method
                      | instance    // Instance method, default
                                    // calling convention
                      | instance vararg // Instance vararg method
```

The calling convention for a property signature is always equal to *IMAGE_CEE_CS_CALLCONV_PROPERTY*.

Having noted this difference, we might as well forget about property signatures and concentrate on method signatures. The truth is that a property signature—excluding the calling convention—is a composite of signatures of the property's access methods, so it is no great wonder that method and property signatures have similar structures.

Remember that in the method calling convention, the combined calling conventions, such as *instance vararg*, are the products of bitwise OR operations performed on the respective calling convention constants.

The value <*num_of_args*>, a compressed unsigned integer, is the number of parameters, not counting the return type. The values <*return_type*> and <*arg_type*> are SETs. The difference between them and the field's SET is that the modifier *&* is allowed in both <*return_type*> and <*arg_type*>. The difference between <*return_type*> and <*arg_type*> is that <*return_type*> can be *void* and <*arg_type*> cannot.

Instance methods have the implicit first argument *this*, which is not reflected in the signature. This implicit argument is a reference to the instance of the method's parent type. It has a class reference type for classes and interfaces and a managed pointer for value types.

MemberRef Signatures

Member references, which are kept in the MemberRef metadata table, are the references to fields and methods, usually those defined outside the current module. There are no specific *MethodRefs* and *FieldRefs*, so you must look at the calling convention of a *MemberRef* signature to tell a field reference from a method reference.

MemberRef signatures for field references are the same as the field declaration signatures discussed earlier; see "Field Signatures." *MemberRef* signatures for method references are structurally similar to method declaration signatures, although you should note two differences concerning the values of signature components:

- The calling convention can contain the modifier *explicit*, which indicates that the instance pointer of the parent object (*this*) is explicitly specified in the method signature as the first parameter.
- In the argument list of a *vararg* method reference, a sentinel can precede the optional arguments. The sentinel itself does not count as an additional argument, so if you call a *vararg* method with one mandatory argument and two optional arguments, the *MemberRef* signature will have an argument count of three and an argument list structure that looks like this:

```
<mandatory_arg> <sentinel><opt_arg1><opt_arg2>
```

Indirect Call Signatures

To call methods indirectly, IL has the special instruction *calli*. This instruction takes argument values plus a function pointer from the stack and uses the *StandAloneSig* token as a parameter. The signature indexed by the token is the signature by which the call is made. Effectively, *calli* takes a function pointer and a signature and presumes that the signature is the correct one to use in calling this function:

```
ldc.i4.0       // Load first argument
ldc.i4.1       // Load second argument
ldftn   void Foo::Bar(int32, int32) // Load function pointer
calli   void(int32, int32)   // Call Foo::Bar indirectly
```

Indirect call signatures are similar to the method signatures of *MemberRefs*, but their calling convention might be one of the unmanaged calling conventions, if the method called indirectly is in fact unmanaged.

Unmanaged calling conventions are defined in CorHdr.h in the *CorUnmanagedCallingConvention* enumeration as follows:

- ***IMAGE_CEE_UNMANAGED_CALLCONV_C* (0x1)** C/C++-style calling convention. The call stack is cleaned up by the caller. The ILAsm notation is *unmanaged cdecl.*
- ***IMAGE_CEE_UNMANAGED_CALLCONV_STDCALL* (0x2)** Win32 API calling convention. The call stack is cleaned up by the callee. The ILAsm notation is *unmanaged stdcall.*
- ***IMAGE_CEE_UNMANAGED_CALLCONV_THISCALL* (0x3)** C++ member method (non-*vararg*) calling convention. The callee cleans the stack, and the *this* pointer is pushed on the stack last. The ILAsm notation is *unmanaged thiscall.*
- ***IMAGE_CEE_UNMANAGED_CALLCONV_FASTCALL* (0x4)** Arguments are passed in registers when possible. The ILAsm notation is *unmanaged fastcall.* This calling convention is not supported in the first release of the runtime.

Local Variables Signatures

Local variables signatures are the second type of signatures referenced by the StandAloneSig metadata table. Each such signature contains type encodings for all local variables used in a method. The method header can contain the *StandAloneSig* token, which identifies the local variables signature. This signature is retrieved by the loader when it prepares the method for JIT compilation.

Local variables signatures are to some extent similar to method declaration signatures, with two differences:

- The calling convention is *IMAGE_CEE_CS_CALLCONV_LOCAL_SIG.*
- Local variables signatures have no return type. The local variable count is immediately followed by the sequence of encoded local variable types:

```
<locals_sig> ::= <callconv_locals> <num_of_vars>
                 <var_type>[,<var_type>*] ]
```

- *<var_type>* is the same SET as *<arg_type>* in method declaration signatures—it can be anything except *void.*

Type Specifications

Type specifications are special metadata items residing in the TypeSpec table and representing type constructs—as opposed to *TypeDef*s and *TypeRef*s, which represent types (classes, interfaces, and value types).

A common example of a type construct is a vector or an array of classes or value types. Consider the following code snippet:

```
.locals init(int32[0...,0...] iArr) // Declare 2-dim array reference
ldc.i4 5      // Load size of first dimension
ldc.i4 10   // Load size of second dimension
// Create array by calling array constructor:
newobj instance void int32[0...,0...]::.ctor(int32,int32)
stloc iArr   // Store reference to new array in iArr
```

In the *newobj* instruction, we specified a *MemberRef* of the constructor method, parented not by a type but by a type construct, *int32[0...,0...]*. The question is, "Whose *.ctor* is it, anyway?"

You might recall that arrays and vectors are generics and can be actualized only in conjunction with some nongeneric type, producing a new class—in our case, a two-dimensional array of 4-byte integers with zero lower bounds. So the constructor we called was the constructor of this class.

And, of course, a natural way to represent such a type construct is by a signature. That's why *TypeSpec* records have only one entry, containing an offset in the *#Blob* stream, pointing at the signature. Personally, I think it's a pity the *TypeSpec* record contains only one entry; a *Name* entry could be of some use. We could go pretty far with named *TypeSpec*s.

The *TypeSpec* signature has no calling convention and consists of one SET, which, however, can be fairly long. Consider, for example, a multidimensional array of function pointers that have function pointers among their arguments.

TypeSpec tokens can be used with all IL instructions that accept *TypeDef* or *TypeRef* tokens. In addition, as you've seen, *MemberRef*s can be scoped to *TypeSpec*s as well as *TypeRef*s. The only places where *TypeSpec*s cannot replace *TypeDef*s or *TypeRef*s are the *extends* and *implements* clauses of the class declaration.

Two additional kinds of *TypeSpec*s, other than vectors and arrays, are unmanaged pointers and function pointers which are not true generics, in that no abstract class exists from which all pointers inherit. Of course, both types of pointers can be cast to the value type *int* (*[mscorlib]System.IntPtr*), but this can hardly help—the *int* value type is oblivious to the type being pointed at, so such casting results only in loss of information. Pointer kinds of *TypeSpec*s are rarely used, compared to array kinds, and have limited application.

Signature Validity Rules

Let's wrap up the basic facts discussed in this chapter:

- [run time] Signature entries of records in the Method, Field, Property, MemberRef, StandaloneSig, and TypeSpec metadata tables must hold valid offsets in the *#Blob* stream. Nil values of these entries are not acceptable.

- Signatures are built from SETs. Each SET describes the type of a field, a parameter, or other such item.
- [run time] Each SET is a sequence of primitive type codes and optional integer parameters, such as metadata tokens or array dimension sizes. A SET cannot end with the following primitive types: a sentinel, ***, *&*, *[ ]*, or *pinned*. These primitive types are modifiers for the types that follow them in the SET.
- [run time] A field signature, which is referenced from the Field or MemberRef table, consists of the calling convention *IMAGE_CEE_CS _CALLCONV_FIELD* and one valid SET, which cannot be *void* or *<type>&* and cannot contain a sentinel or a *pinned* modifier.
- A method reference signature, which is referenced from the MemberRef table, consists of a calling convention, an argument count, a return SET, and a sequence of argument SETs, corresponding in number to the argument count.
- [run time] The calling convention of a method reference signature is one of the following: the default, *vararg*, *instance*, *instance vararg*, *instance explicit*, or *instance explicit vararg*.
- [run time] The return SET of a method reference signature cannot contain a sentinel or a *pinned* modifier.
- [run time] No more than one argument SET of a method reference signature can contain a sentinel, and it can do so only if the calling convention includes *vararg*.
- [run time] The argument SETs of a method reference signature cannot be *void* and cannot contain a *pinned* modifier.
- A method declaration signature, which is referenced from the Method table, has the same structure as a method reference signature and must comply with the same requirements, plus the following restrictions: the *explicit* calling convention cannot be used, and no argument SET can contain a sentinel.
- A property declaration signature, which is referenced from the Property table, has the same structure as a method declaration signature and must comply with the same requirements except that the calling convention of a property declaration signature must be *IMAGE_CEE_CS_CALLCONV_PROPERTY*.

- An indirect call signature, which is referenced from the StandAloneSig table, has the same structure as a method reference signature and must comply with the same requirements except that the calling convention of an indirect call to an unmanaged method can be *unmanaged cdecl*, *unmanaged stdcall*, *unmanaged thiscall*, or *unmanaged fastcall*.
- A local variables signature, which is referenced from the StandAloneSig table, consists of the calling convention *IMAGE_CEE_CS_CALLCONV_LOCAL_SIG*, a local variable count, and a sequence of variable SETs, corresponding in number to the variable count.
- [run time] No variable SET can be *void* or can contain a sentinel.
- A type specification signature, which is referenced from the TypeSpec table, consists of one SET not preceded by the calling convention. The SET must represent an array, a vector, an unmanaged pointer, or a function pointer, and it cannot contain a *pinned* modifier.

8

Fields and Data Constants

Fields are one of two kinds of typed and named data locations, the second kind being method local variables, which are discussed in Chapter 9, "Methods." Fields correspond to the member variables and global variables of the C++ world. Apart from their own characteristics, fields can have additional information associated with them defining the way the fields are laid out by the loader, how they are allocated, how they are marshaled to unmanaged code, and whether they have default values. This chapter examines all aspects of member and global fields and the metadata used to describe these aspects.

Field Metadata

To define a field, you must first provide basic information: the field's name and signature and flags indicating the field's characteristics, stored in the Field metadata table. Then comes optional information, specific to certain kinds of fields: field marshaling information, found in the FieldMarshal table; field layout information in the FieldLayout table; field mapping information in the FieldRVA table; and a default value in the Constant table.

To reference a field, you must know its owner—*TypeRef*, *TypeDef*, or *ModuleRef*—as well as the field's name and signature. The references to the fields are kept in the MemberRef table. The general structure of the field metadata group is shown in Figure 8-1.

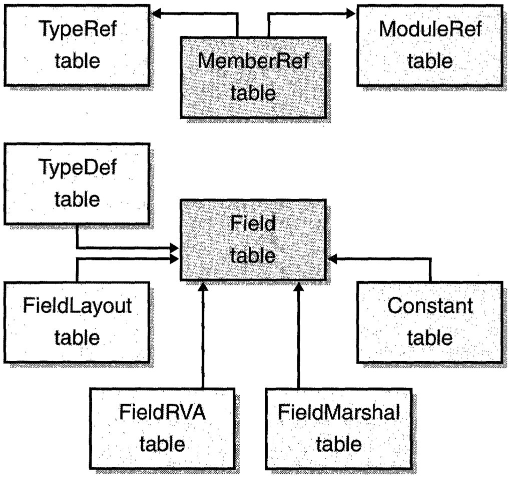


Figure 8-1 Field metadata group.

The central metadata table of the group, the Field table, has the associated token type *mdtFieldDef* (0x04000000). A record in this table has three entries:

- ***Flags* (2-byte unsigned integer)** Binary flags indicating the field's characteristics.
- ***Name* (offset in the *#Strings* stream)** The field's name.
- ***Signature* (offset in the *#Blob* stream)** The field's signature.

As you can see, a *Field* record does not contain one vital piece of information: which class or value type owns the field. The information about field ownership is furnished by the class descriptor itself: records in the TypeDef table have *FieldList* entries, which hold the RID (record index) of the Field table where the type's fields can be found.

In the simplest case, when only the Field metadata table is involved, the IL assembly language (ILAsm) syntax for a field declaration is as follows:

```
.field <flags> <type> <name>
```

The owner of a field is the class or value type in the lexical scope of which the field is defined.

A field's binary flags are defined in the CorHdr.h file in the enumeration *CorFieldAttr* and can be divided into four groups, as described in the following list. I'm using ILAsm keywords instead of the constant names from *CorFieldAttr*, as I don't think the constant names are relevant.

- Accessibility flags (mask 0x0007):
    - ❑ ***privatescope* (0x0000)** This is the default accessibility. A private scope field is exempt from the requirement of having a unique triad of owner, name, and signature and hence must always be referenced by a *FieldDef* token and never by a *MemberRef* token (0x0A000000). Otherwise, this accessibility is the same as that specified by the *private* flag.
    - ❑ ***private* (0x0001)** The field is accessible from its owner and from classes nested in the field's owner.
    - ❑ ***famandassem* (0x0002)** The field is accessible from types belonging to the owner's family—that is, the owner itself and all its descendants—defined in the current assembly.
    - ❑ ***assembly* (0x0003)** The field is accessible from types defined in the current assembly.
    - ❑ ***family* (0x0004)** The field is accessible from the owner's family.
    - ❑ ***famorassem* (0x0005)** The field is accessible from the owner's family and from all types defined in the current assembly.
    - ❑ ***public* (0x0006)** The field is accessible from any type.
- Contract flags (mask 0x02F0):
    - ❑ ***static* (0x0010)** The field is static, shared by all instances of the type.
    - ❑ ***initonly* (0x0020)** The field can only be initialized and cannot be written to later. Initialization takes place in an instance constructor (*.ctor*) for instance fields and in a class constructor (*.cctor*) for static fields.
    - ❑ ***literal* (0x0040)** The field is a compile-time constant. The

loader does not lay out this field and does not create an internal handle for it. The field cannot be directly addressed from IL and can be used only as a Reflection reference to retrieve an associated metadata-held constant. If you try to access a literal field directly—for example, through the *ldsfld* instruction—the JIT (just-in-time) compiler throws a *MissingField* exception and aborts the task.

  - ***notserialized*** **(0x0080)** The field does not have to be serialized when the owner is remoted. This flag has meaning only for instance fields of the *serializable* types.
  - ***specialname*** **(0x0200)** The field is special in some way, as defined by the name.

- Interoperability flag:
  - ***pinvokeimpl*** **(0x2000)** The field is unmanaged and is accessed from the managed code via the platform invocation mechanism (*P/Invoke*). In the first release of the Microsoft .NET common language runtime, the *P/Invoke* mechanism works for methods only, so this flag should never be set. ILAsm does not allow the flag to be set.

- Reserved flags (cannot be set explicitly; mask 0x9500):
  - ***rtspecialname*** **(0x0400)** The field has a special name that is reserved for the internal use of the common language runtime. Two field names are reserved: *value_*, for instance fields in enumerators; and *_Deleted**, for fields marked for deletion but not actually removed from metadata. The keyword *rtspecialname* is ignored by the ILAsm compiler and is displayed by the IL Disassembler for informational purposes only. This flag must be accompanied by a *specialname* flag.
  - ***marshal(<native_type>)*** **(0x1000)** The field has an associated *FieldMarshal* record specifying how the field must be marshaled when consumed by unmanaged code. The ILAsm construct *marshal(<native_type>)* defines the marshaling information emitted to the FieldMarshal table but does not set the flag directly. Rather, the flag is set behind the scenes by the metadata emission API when the marshaling information is emitted. Native types are discussed in Chapter 7, "Primitive Types and Signatures."
  - ***[no ILAsm keyword]*** **(0x8000)** The field has an associated

Constant record. The flag is set by the metadata emission API when the respective *Constant* record is emitted. See the section "Default Values," later in this chapter.

- ❑ ***[no ILAsm keyword]* (0x0100)** The field is mapped to data and has an associated *FieldRVA* record. The flag is set by the metadata emission API when the respective *FieldRVA* record is emitted. See the section "Mapped Fields," later in this chapter.

In the field declaration, the type of the field (<*type*>) is the ILAsm notation of the appropriate single encoded type, which together with the calling convention forms the field's signature. If you forgot what a field signature looks like, see the section "Field Signatures," in Chapter 7.

The name of the field (<*name*>), also included in the declaration, should be a simple name. ILAsm does not allow composite field names, although one can always cheat and put a composite name in single quotation marks, turning it into a simple name.

Examples of field declarations include the following:

```
.field public static marshal(int) int32 I
.field family string S
.field private int32& pJ   // ERROR! ByRef in field signature!
```

Field references in ILAsm have the following notation:

```
<field_ref> ::= <field_type>[<class_ref>::]<field_name>
```

where <*class_ref*>—as we know from Chapter 6, "Namespaces and Classes"—is defined as

```
<class_ref> ::= [<resolution_scope>]<full_type_name>
```

where

```
 <resolution_scope> ::= [<assembly_ref_alias>]
                      | [.module <module_ref_name>]
```

For instance, this example uses the IL instruction *ldfld*, which loads the field value on the stack:

```
ldfld  int32 [.module Another.dll]Foo.Bar::idx
```

When it is difficult to infer from the context whether the referenced member is a field or a method, <*field_ref*> is sometimes preceded by the keyword *field*. Note that the keyword does not contain a leading dot. This example uses the IL instruction *ldtoken*, which loads an item's runtime handle on the stack:

```
ldtoken field int32 [.module Another.dll]Foo.Bar::idx
```

The field references reside in the MemberRef metadata table, which has associated token type 0x0A000000. A record of this table has only three entries:

- ***Class* (coded token of type MemberRefParent)** This entry references the TypeRef or the ModuleRef table. Method references, residing in the same table, can have their *Class* entries referencing the Method and the TypeSpec tables as well.
- ***Name* (offset in the *#Strings* stream)**
- ***Signature* (offset in the *#Blob* stream)**

Instance and Static Fields

Instance fields are created every time a type instance is created, and they belong to the type instance. Static fields, which are shared by all instances of the type, are created when the type is loaded. Some of the static fields (literal and mapped fields) are never allocated. The loader simply notes where the mapped fields reside and addresses these locations whenever the fields are to be addressed. And the literal fields are replaced with the constants at compile time.

A field signature contains no indication of whether the field is static or instance. But because the loader keeps separate books for instance fields and for two out of three kinds of static fields—not for literal fields—the kind of referenced field is easily discerned from the field's token. When a field token is found in the IL stream, the JIT compiler does not have to dive into the metadata, retrieve the record, and check the field's flags; by that time, all the fields have been accounted for and duly classified by the loader.

IL has two sets of instructions for field loading and storing. The instructions for instance fields are *ldfld*, *ldflda*, and *stfld*; those for static fields are *ldsfld*, *ldsflda*, and *stsfld*. An attempt to use a static field instruction with an instance field would result in a JIT compilation failure. The inverse combination would work, but it requires loading the instance pointer on the stack, which is, of course, completely redundant for a static field.

Default Values

Default values reside in the Constant metadata table. Three kinds of metadata items can have a default value assigned and therefore can reference the Constant table: fields, method parameters, and properties. A record of the Constant table has three entries:

- ***Type* (unsigned 1-byte integer)** The type of the constant, one of the *ELEMENT_TYPE_** codes. (See Chapter 7.)

- ***Parent* (coded token of type *HasConstant*)** A reference to the owner of the constant, a record in the Field, Property, or Param table.
- ***Value* (offset in the *#Blob* stream)** A constant value blob.

The current implementation of the common language runtime and ILAsm allows the constant types described in Table 8-1. (As usual, I've dropped the *ELEMENT_TYPE_* part of the name.)

Table 8-1 Constant Types

Constant Type	ILAsm Notation	Comments
I1	Int8	Signed 1-byte integer.
I2	*int16*	Signed 2-byte integer.
I4	*int32*	Signed 4-byte integer.
I8	*int64*	Signed 8-byte integer.
R4	*float32*	4-byte floating-point.
R8	*float64*	8-byte floating-point.
CHAR	*char*	2-byte Unicode character.
BOOLEAN	*bool*	1-byte Boolean, *true* = 1, *false* = 0.
STRING	<*quoted_string*>, *bytearray*	Unicode string.
CLASS	*nullref*	Null object reference. The value of the constant of this type must be a 4-byte integer containing 0.

The ILAsm syntax for defining the default value of a field is as follows:

```
<field_def_const> ::= .field <flags> <type>
   <name> = <const_type> [( <value> )]
```

The value in parentheses is mandatory for all constant types except *nullref*. For example:

```
.field public int32 i = int32(123)
.field public static literal bool b = bool(true)
.field private float32 f = float32(1.2345)
.field public static int16 ii = int16(0xFFE0)
.field public object o = nullref
```

Defining integer and Boolean constants—not to mention *nullref*—is pretty straightforward, but floating-point constants and strings can present some difficulties.

Floating-point numbers have special cases, such as positive infinity and negative infinity, that cannot be presented textually in simple floating-point format. In these special cases, the floating-point constants can alternatively be represented as integer values with a matching byte count. The integer values are not converted to floating-point values; instead, they represent an exact *bit image* of the floating-point values. For example:

```
.field public float32 fPosInf = float32(0x7F800000)
.field public float32 fNegInf = float32(0xFF800000)
.field public float32 fNAN = float32(0xFFC00000)
```

Like all other constants, string constants are stored in the *#Blob* stream. In this regard, they differ from user-defined strings, which are stored in the *#US* stream. What both kinds of strings have in common is that they are supposed to be Unicode. I say "supposed to be" because the only Unicode-specific restrictions imposed on these strings are that their sizes are reported in Unicode characters and that their byte counts must be even. Otherwise, these strings are simply binary objects and might or might not contain invalid Unicode characters.

Notice that the type of the constant does not need to match the type of the item to which this constant is assigned—in this case, the type of the field.

In ILAsm, a string constant can be defined either as a composite quoted string or as a byte array:

```
.field public static string str1 = "Isn't" + " it " + "marvellous!"
.field public static string str2 = bytearray(00 01 FF FE 1A 00 00 )
```

When a string constant is defined as a composite quoted string, this string is converted to Unicode before being stored in the *#Blob* stream. In the case of a *bytearray* definition, the specified byte sequence is stored "as is," and padded with 1 zero byte if necessary to make the byte count even. In the example shown here, the default value for the *str2* field will be padded to bring the byte count to 8 (four Unicode characters). And if the bytes specified in the *bytearray* are invalid Unicode characters, it will surely be discovered when we try to print the string, but not before.

Assigning default values to fields (and parameters) seems to be such a compelling technique that you might wonder why we did not employ it in the simple sample discussed in Chapter 1, "Simple Sample." Really, defining the default values is a great way to initialize fields—right? Wrong. Here's a tricky question. Suppose that we define a member field as follows:

```
.field public static int32 ii = int32(12345)
```

What will the value of the field be when the class is loaded? Correct answer: 0. Why? Because default values specified in the Constant table are not used by the loader to initialize the items to which they are assigned. If we want to initialize a field to its default value, we must explicitly call the respective Reflection method to retrieve the value from metadata and then store this value in the field. This doesn't sound too nice, but, on the other hand, we should not forget that these are *default* values rather than initial values, so formally the loader might be right.

Let me remind you once again that literal fields are not true fields. They are not laid out by the loader, and they cannot be directly accessed from IL. From the point of view of metadata, however, literal fields are nevertheless valid fields having valid tokens, which allow the constant values corresponding to these fields to be retrieved by Reflection methods. The common language runtime does not provide an implicit means of accessing the Constant table, which is a pity. It would certainly be much nicer if the JIT compiler would compile the *ldsfld* instruction into the retrieval of the respective constant value instead of failing, when the *ldsfld* instruction is applied to a literal field. But such are the facts of life, and I am afraid we cannot do anything about it at the moment.

Given this situation, literal fields without associated *Constant* records are legal from the loader's point of view, but they are utterly meaningless. They serve no purpose except to inflate the Field metadata table.

But how do the compilers handle literal fields? If every time a constant from an enumerator—represented, as we know, by a literal field—was used the compiler emitted a call to the Reflection API to get this constant value, one could imagine where it would leave the performance. Most compilers are smarter than that and resolve the literal fields at compile time, replacing references to literal fields with explicit constant values of these fields, so that the literal fields never come into play at run time. So much for having the literal fields in the metadata and devising a special kind of *TypeDef* for enumerators.

ILAsm, following common language runtime functionality to the letter, allows the definition of the Constant metadata but does nothing about the symbol-to-value resolution at compile time. From the point of view of ILAsm and the runtime, the enumerators are real, as distinctive types, but the symbolic constants listed in the enumerations are not.

Mapped Fields

It is possible to provide unconditional initialization for static fields by mapping the fields to data defined in the PE file and setting this data to the initializing values. The syntax for mapping a field to data in ILAsm is the following:

```
<mapped_field_decl> ::= .field <flags> <type> <name> at <data_label>
```

Here's an example:

```
.field public static int64 ii at data_ii
```

The nonterminal symbol *<data_label>* is a simple name labeling the data segment to which the field is mapped. The ILAsm compiler allows a field to be mapped either to the "normal" data section (*.sdata*) or to the thread local storage (*.tls*), depending on the data declaration to which the field mapping refers. A field can be mapped only to data residing in the same module as the field declaration. (For information about data declaration, see the following section, "Data Constants Declaration.")

Mapping a field results in emitting a record of the FieldRVA table, which contains two entries:

- ***RVA* (4-byte unsigned integer)** The relative virtual address of the data to which the field is mapped.
- ***Field* (RID to the Field table)** The index of the *Field* record being mapped.

Two or more fields can be mapped to the same location, but each field can be mapped to one location only. Duplicate *FieldRVA* records with the same *Field* values and different *RVA* values are therefore considered invalid metadata. The loader is not particular about duplicate *FieldRVA* records, however; it simply uses the first one available for the field and ignores the rest.

The field mapping technique has some catches. The first catch (well, not much of a catch, actually) is that, obviously, only static fields can be mapped. Even if we could map instance fields, each instance would be mapped to the same physical memory, making the fields de facto static—shared by all instances—anyway. Mapping instance fields is considered invalid metadata, but it has no serious consequences for the loader—if a field is not static, the loader does not even check to see whether the field is mapped. The only real effect of mapping instance fields is a bloated FieldRVA table. The ILAsm compiler treats mapping of an instance field as an error and produces an error message.

The second catch is that a field cannot be mapped if its type contains object references (objects or arrays). Because the data sections are out of the garbage collector's reach, the validity of object references placed in the data sections cannot be guaranteed. If the loader finds object references in a mapped field type, it throws a *TypeLoad* exception and aborts the loading, even if the code is run in full trust mode from a local drive and all security-related checks are disabled. The loader checks for the presence of object references on all levels of the field type—in other words, it checks the types of all the fields that make up the type, and checks the types of fields that make up those types, and so on.

The third catch is that a field cannot be mapped if its type contains nonpublic instance fields. The reasoning behind this limitation is that if we map a field with a type containing nonpublic members, we can map another field of some all-public type to the same location and, through this second mapping, get unlimited access to nonpublic member fields of the first type. The loader checks for the presence of nonpublic members on all levels of the mapped field type and throws a *TypeLoad* exception if it finds such members. This check, unlike the check for object references, is performed only when code verification is required; it is disabled when the code is run from the local drive in full trust mode.

Note, however, that a mapped field itself can be declared nonpublic without ill consequences. This is based on the simple assumption that if developers decide to overlap their own nonpublic field and thus defy the accessibility control mechanism of the common language runtime object model, they probably know what they are doing.

The last catch worth mentioning is that the initialization data is provided "as is," exactly as it is defined in the PE file. And if you run the code on a platform other than the one on which the PE file was created, you can face some unpleasant consequences. As a trivial example, suppose that you map an *int32* field to data containing bytes 0xAA, 0xBB, 0xCC, and 0xDD. On a little endian platform (for instance, an Intel platform), the field is initialized to 0xDDCCBBAA, while on a big endian platform...well, you get the picture.

All these catches do not preclude the compilers from using field mapping for initialization.

Data Constants Declaration

A data constant declaration in ILAsm has the following syntax:

```
<data_decl> ::= .data [ tls ] [<data_label> = ] <data_items>
```

where *<data_label>* is a simple name, unique within the module, and

```
<data_items> ::= { <data_item> [ , <data_item>* ] } | <data_item>
```

where

```
<data_item> ::= <data_type> [ ( <value> ) ] [ [ <count> ] ]
```

Data constants are emitted to the *.sdata* section or the *.tls* section, depending on the presence of the *tls* keyword, in the same sequence in which they were declared in the source code. The unlabeled data declarations can be used for padding between the labeled data declarations and probably for nothing else, since without a label it's impossible to map a field to this data. Unlabeled—or, more precisely, unreferenced—data might not survive round-tripping (disassembly-reassembly) because the IL Disassembler outputs only referenced data.

The nonterminal symbol *<data_type>* specifies the data type. (See Table 8-2.) The data type is used by the ILAsm compiler exclusively for identifying the size and byte layout of <*value*> and is not emitted as any part of metadata or the data itself. Having no way to know what the type was intended to be when the data was emitted, the IL Disassembler always uses the most generic form, a byte array, for data representation.

If <*value*> is not specified, the data is initialized to a default value (usually a value with all bits set to zeros). Thus it is still "initialized data" in terms of the PE file structure—meaning that this data is part of the PE file disk image.

The optional <*count*> in square brackets indicates the repetition count of the data item. Here are some examples:

```
.data tls T_01 = int32(1234)
// 4 bytes in .tls section, value 0x000004D2
.data tls int32
// 4 bytes padding in .tls section, value doesn't matter
.data D_01 = int32(1234)[32] // 32 4-byte integers in .sdata section,
                             // Each equal to 0x000004D2
```

Table 8-2 Types Defined for Data Constants

Data Type	Size	Value	Comments
float32	4 bytes	Floating-point, single precision	If an integer value is used, it is converted to floating-point. If the value overflows *float32*, the ILAsm compiler issues a warning.
float64	8 bytes	Floating-point, double precision	If an integer value is used, it is converted to floating-point.
int64	8 bytes	8-byte signed integer	
int32	4 bytes	4-byte signed integer	If the value overflows *int32*, the ILAsm compiler issues a warning.
int16	2 bytes	2-byte signed integer	If the value overflows *int16*, the ILAsm compiler issues a warning.
int8	1 byte	1-byte signed integer	If the value overflows *int8*, the ILAsm compiler issues a warning.
bytearray	*var*	Sequence of two-digit hexadecimal numbers, without the 0x prefix	The value cannot be omitted since it defines the size. The repetition parameter ([<*count*>]) cannot be used.

(continued)

Table 8-2 Types Defined for Data Constants *(continued)*

Data Type	Size	Value	Comments
*char**	*var*	Composite quoted string	The value cannot be omitted since it defines the size. The repetition parameter ([<*count*>]) cannot be used. The string is converted to Unicode before being emitted to data.
&	4 bytes	Another data label	Data-on-data; the data containing the value of the unmanaged pointer—the virtual address—of another named data segment. The value cannot be omitted, and the repetition parameter ([<*count*>]) cannot be used. The referenced data segment must be declared before being referenced in a data-on-data declaration.

Explicit Layouts and Union Declaration

Although instance fields cannot be mapped to data, it is possible to manipulate the positioning of these fields directly. As you might remember from Chapter 6, a class or a value type can have an *explicit* flag, a special flag indicating that the metadata contains exact instructions for the loader regarding the layout of this class. This information is kept in the FieldLayout metadata table, whose records contain these two entries:

- ***OffSet* (4-byte unsigned integer)** The relative offset of the field in the class layout (*not* an RVA).
- ***Field* (RID to the Field table)** The index of the field for which the offset is specified.

In ILAsm, the field offset is specified by putting the offset value in square brackets immediately after the *.field* keyword, as shown here:

```
.class public value sealed explicit MyStruct
{
   .field [0] public int32 ii
   .field [4] public float64 dd
   .field [12] public bool bb
}
```

Only instance fields can have offsets specified. Because static fields are not part of the class instance layout, specifying explicit offsets for them is meaningless and is considered a metadata error. If an offset is specified for a static field, the loader behaves the same way it does with mapped instance fields: if the field is static, the loader does not check to see whether the field has an offset specified. Consequently, *FieldLayout* records referencing the static fields are nothing more than a waste of memory.

In a class that has an explicit layout, *all* the instance fields must have specified offsets. If one of the instance fields does not have an associated *FieldLayout* record, the loader throws a *TypeLoad* exception and aborts the loading. Obviously, a field can have only one offset, so duplicate *FieldLayout* records that have the same *Field* entry are illegal. This is not checked at run time because this metadata invalidity is not critical: the loader takes the first available *FieldLayout* record for the current field and ignores any duplicates.

The placement of object references (classes, arrays) is subject to a general limitation: the fields of object reference types must be aligned on pointer size—either 4 or 8 bytes, depending on the platform:

```
.class public value sealed explicit MyStruct
{
   .field [0] public int16 ii
   .field [2] public string str //Illegal on 32-bit and 64-bit
   .field [6] public int16 jj
   .field [8] public int32 kk
   .field [12] public object oo //Illegal on 64-bit platform
   .field [16] public int32[] iArr //Legal on both platforms
}
```

Explicit layout is a standard way to implement unions in IL. By explicitly specifying field offsets, we can make fields overlap however we want. Let's suppose, for example, that we want to treat a 4-byte unsigned integer as such, or as a pair of 2-byte words, or as 4 bytes. In C/C++ notation, the respective constructs look like this:

```
union MultiDword {
   DWORD dw;
   union {
      struct {
         WORD w1;
         WORD w2;
      };
      struct {
         BYTE b1;
         BYTE b2;
         BYTE b3;
         BYTE b4;
      };
   };
};
```

In ILAsm, the same union will be written like so:

```
.class public value sealed explicit MultiDword
{
   .field [0] public unsigned int32 dw

   .field [0] public unsigned int16 w1
   .field [2] public unsigned int16 w2

   .field [0] public unsigned int8 b1
   .field [1] public unsigned int8 b2
   .field [2] public unsigned int8 b3
   .field [3] public unsigned int8 b4
}
```

The only limitation imposed on the explicit-layout unions is that if the overlapping fields contain object references, these object references must not overlap with any other field:

```
.class public value sealed explicit StrAndIndex
{
   .field [0] public string Str // Reference, size 4 bytes
                                // on 32-bit platform
   .field [4] public unsigned int32 Index
}
.class public value sealed explicit MyUnion
{
   .field [0] public valuetype StrAndIndex str_and_index
   .field [0] public unsigned int64 whole_thing // Illegal!
   .field [0] public string str // Illegal!
   .field [2] public unsigned int32 half_and_half // Illegal!
   .field [4] public unsigned int32 index // Legal, object reference
                                          // not overlapped
}
```

Such "unionizing" of the object references would provide the means for directly modifying these references, which could thoroughly disrupt the functioning of the garbage collector. The loader checks explicit layouts for object reference overlap; if any is found, it throws a *TypeLoad* exception and aborts the loading.

A field can also have an associated *FieldLayout* record if the owner of the field has a *sequential* layout. In this case, the *OffSet* entry of the *FieldLayout* record holds a field ordinal rather than an offset. The fields belonging to a sequential-layout class needn't have associated *FieldLayout* records, but if one of the class's fields has such an associated record, all the rest must have one too.

Global Fields

Fields declared outside the scope of any class are known as *global fields*. They don't belong to a class but instead belong to the module in which they are declared. Because a module is represented by a special *TypeDef* record under the name <*Module*>, all the formalities that govern how field records are identified by reference from their parent *TypeDef* records are observed.

Global fields must be static. Since only one instance of the module exists when the assembly is loaded, and because it is impossible to create alternative instances of the module, this limitation seems obvious.

Global fields can have *public*, *private*, or *privatescope* accessibility flags—at least that's what the metadata validity rules say. As we saw in Chapter 1, however, a global item (a field or a method) can have any accessibility flag, and the loader interprets this flag only as *assembly*, *private*, or *privatescope*. The *public*, *assembly*, and *famorassem* flags are all interpreted as *assembly*, while the *family*, *famandassem*, and *private* flags are all interpreted as *private*. The global fields cannot be accessed from outside the assembly, so they don't have true *public* accessibility. And because no type can be derived from <*Module*>, the question about family-related accessibility is moot.

Global fields can be accessed from anywhere within the module, regardless of their declared accessibility. In this regard, the classes that are declared within a module and use the global fields have the same access rights as if they were nested in the module. The metadata contains no indications of such nesting, of course.

A reference to a global field declared in the same module has no <*class_ref*>:: part:

```
<global_field_ref> ::= [field] <field_type> <field_name>
```

The keyword *field* is used in particular cases when the nature of the reference cannot be inferred from the context.

A reference to a global field declared in a different module of the assembly also lacks the class name but has resolution scope:

```
<global_field_ref> ::= [field] [.module <mod_name>]::<field_name>
```

The following are two examples of such declarations:

```
ldsfld int32 globalInt
ldtoken field int32 [.module supporting.dll]::globalInt
```

Since the global fields are static, we cannot explicitly specify their layout except by mapping them to data. Thus our 4-2-1-byte union *MultiDword* would look like this if we implemented it with global fields:

```
.field public static unsigned int32 dw at D_00
.field public static unsigned int16 w1 at D_00
.field public static unsigned int16 w2 at D_02
.field public static unsigned int8 b1 at D_00
.field public static unsigned int8 b2 at D_01
.field public static unsigned int8 b3 at D_02
.field public static unsigned int8 b4 at D_03
.data D_00 = int8(0)
.data D_01 = int8(0)
.data D_02 = int8(0)
.data D_03 = int8(0)
⋮
ldc.i1.1
stsfld unsigned int8 b3 // Set value of third byte
```

Fortunately, we don't have to do that every time we need a global union. Instead, we can declare the value type *MultiDword* exactly as before and then declare a global field of this type:

```
.field public static valuetype MultiDword multi_dword
⋮
ldc.i1.1
ldsflda valuetype MultiDword multi_dword
// Load reference to the field
// As instance of MultiDword
stfld unsigned int8 MultiDword::b3 // Set value of third byte
```

Constructors vs. Data Constants

We've already taken a look at field mapping as a technique of field initialization, and I've listed the drawbacks and limitations of this technique. Field mapping has this distinct "unmanaged" scent about it, but the compilers routinely use it for field initialization nevertheless. Is there a way to get the fields initialized without mapping them? Yes, there is.

The common language runtime object model provides two special methods, the instance constructor (*.ctor*) and the class constructor (*.cctor*), a.k.a. the type initializer. We're getting ahead of ourselves a bit here; methods in general and constructors in particular are discussed in Chapter 9, so we won't concentrate on details here. For now, all we need to know about *.ctor* and *.cctor* is that *.ctor* is executed when a new instance of a type is created, and *.cctor* is executed after the type is loaded and before any one of the type members is accessed. Because class constructors are static and can deal with static members of the type only, we have a perfect setup for field initialization: *.cctor*s take care of static fields, and *.ctor*s take care of instance fields.

But how about global fields? The good news is that we can define a global *.cctor*. (Don't try this in the second beta version of the common language runtime, if you can still find a copy; global class constructors were not allowed in this beta version.) Field initialization by constructors is vastly superior to field mapping, with none of its limitations, as described earlier in the section "Mapped Fields." The catch? Unfortunately, initialization by constructors must be executed at run time, burning processor cycles, whereas mapped fields simply "are there" after the module has been loaded. The mapped fields don't require additional operations for the initialization. Whether this price is worth the increased freedom and safety regarding field initialization depends on the concrete situation, but in general I think it is.

Let me illustrate the point by building an alternative enumerator. Because all the values of an enumerator are stored in literal fields, which are inaccessible from IL directly, the compilers replace references to these fields with the respective values at compile time. We can use a very simple enumerator as a model:

```
.class public enum sealed MagicNumber
{
   .field private specialname int32 value__
   .field public static literal valuetype
      MagicNumber MagicOne = int32(123)
   .field public static literal valuetype
      MagicNumber MagicTwo = int32(456)
   .field public static literal valuetype
      MagicNumber MagicThree = int32(789)
}
```

Let's suppose that our code uses the symbolic constants of an enumerator declared in a third-party assembly. We compile the code, and the symbolic constants are replaced with their values. Forget for a moment that we must have that third-party assembly available at compile time. But we will need to recompile the code every time the enumerator changes, and we have no control over the enumerator because it is defined outside our jurisdiction. In another scenario, when we declare an enumerator in one of our own modules, we must recompile all the modules that reference this enumerator once it is changed.

Let's suppose also—for the sake of an argument—that we don't like this situation, so we decide to devise our own enumerator:

```
.class public value sealed MagicNumber
{
   .field public int32 _value_ // Specialname value__ is
                               // reserved for enums
   .field public static valuetype MagicNumber MagicOne at D_00
   .field public static valuetype MagicNumber MagicTwo at D_04
   .field public static valuetype MagicNumber MagicThree at D_08
}
.data D_00 = int32(123)
.data D_04 = int32(456)
.data D_08 = int32(789)
```

This solution looks good, except in the platform-independence department. We conquered the recompilation problem and can at last address the symbolic constants by their symbols (names), through field access instructions. This approach presents two problems, though. First, the fields representing the symbolic constants can be written to. Second, it works fine with integers, but what if we need a string enumeration?

Let's try again with a class constructor; refer to the sample MyEnums.il on the companion CD.

```
.class public value sealed MagicNumber
{
   .field private int32 _value_ // Specialname value__ is
                                // reserved for enums
   .field public static initonly valuetype MagicNumber MagicOne
   .field public static initonly valuetype MagicNumber MagicTwo
   .field public static initonly valuetype MagicNumber MagicThree
   .method public static specialname void .cctor()
   {
      ldsflda valuetype MagicNumber MagicNumber::MagicOne
      ldc.i4 123
      stfld int32 MagicNumber::_value_

      ldsflda valuetype MagicNumber MagicNumber::MagicTwo
      ldc.i4 456
      stfld int32 MagicNumber::_value_

      ldsflda valuetype MagicNumber MagicNumber::MagicThree
      ldc.i4 789
      stfld int32 MagicNumber::_value_

      ret
   }
   .method public int32 ToBase()
   {
      ldarg.0 // Instance pointer
      ldfld int32 MagicNumber::_value_
      ret
   }
}
```

All the remaining problems seem to be solved. The *initonly* flag on the static fields protects them from being overwritten outside the class constructor.

Embedding the numeric values of symbolic constants in the IL stream takes care of platform dependence. Because we are not mapping the fields, we are free to use any type as the underlying type of our enumerator. And, of course, declaring the *_value_* field private protects it from having arbitrary values assigned to it.

Alas, there is a hidden problem with this solution: the *initonly* flag does not provide full protection against arbitrary field overwriting. In the first release of the runtime, the operations *ldflda* (*ldsflda*) and *stfld* (*stsfld*) on *initonly* fields are unverifiable outside the constructors. Unverifiable but not impossible, which means that if the verification procedures are disabled, the *initonly* fields can be overwritten in any method.

Metadata Validity Rules

The field-related metadata tables include the Field, FieldLayout, FieldRVA, FieldMarshal, Constant, and MemberRef tables. The records of these tables have the following entries:

- The Field table contains the *Flags*, *Name*, and *Signature* entries.
- The FieldLayout table contains the *OffSet* and *Field* entries.
- The FieldRVA table contains the *RVA* and *Field* entries.
- The FieldMarshal table contains the *Parent* and *NativeType* (native signature) entries.
- The Constant table contains the *Type*, *Parent*, and *Value* entries.
- The MemberRef table contains the *Class*, *Name*, and *Signature* entries.

Field Table Validity Rules

- The *Flags* entry can have only those bits set that are defined in the enumeration *CorFieldAttrEnum* in CorHdr.h (validity mask: 0xB7F7).
- [run time] The accessibility flag (mask 0x0007) must be one of the following: *privatescope*, *private*, *famandassem*, *assembly*, *family*, *famorassem*, or *public*.
- The *literal* and *initonly* flags are mutually exclusive.
- If the *literal* flag is set, the *static* flag must also be set.
- If the *rtspecialname* flag is set, the *specialname* flag must also be set.
- [run time] If the flag 0x1000 (fdHasFieldMarshal) is set, the FieldMarshal table must contain a record referencing this *Field* record, and vice versa.

- [run time] If the flag 0x8000 (*fdHasDefault*) is set, the Constant table must contain a record referencing this *Field* record, and vice versa.
- [run time] If the flag 0x0100 (*fdHasFieldRVA*) is set, the FieldRVA table must contain a record referencing this *Field* record, and vice versa.
- [run time] Global fields, owned by the *TypeDef <Module>*, must have the *static* flag set.
- [run time] The *Name* entry must hold a valid reference to the *#Strings* stream, indexing a nonempty string no more than 1023 bytes long in UTF-8 encoding.
- [run time] The *Signature* entry must hold a valid reference to the *#Blob* stream, indexing a valid field signature. Validity rules for field signatures are discussed in Chapter 7.
- No duplicate records—attributed to the same *TypeDef* and having the same *Name* and *Signature* values—can exist unless the accessibility flag is *privatescope*.
- Fields attributed to enumerators must comply with additional rules, described in Chapter 6.

FieldLayout Table Validity Rules

- The *Field* entry must hold a valid reference to the Field table.
- The field referenced in the *Field* entry must not have the *static* flag set.
- [run time] If the referenced field is an object reference type and belongs to *TypeDef*s that have an explicit layout, the *OffSet* entry must hold a value that is a multiple of *sizeof(void*)*.
- [run time] If the referenced field is an object reference type and belongs to *TypeDef*s that have an explicit layout, this field must not overlap with any other field.

FieldRVA Table Validity Rules

- [run time] The *RVA* entry must hold a valid nonzero relative virtual address.
- The *Field* entry must hold a valid index to the Field table.
- No duplicate records referencing the same field can exist.

FieldMarshal Table Validity Rules

- The *Parent* entry must hold a valid reference to the Field or Param table.

- No duplicate records that contain the same *Parent* value can exist.
- The *NativeType* entry must hold a valid reference to the *#Blob* stream, indexing a valid marshaling signature. Native types that make up the marshaling signatures are described in Chapter 7.

Constant Table Validity Rules

- The *Type* entry must hold a valid *ELEMENT_TYPE_** code, one of the following: *bool*, *char*, a signed or unsigned integer of 1 to 8 bytes, *string*, or *object*.
- The *Value* entry must hold a valid offset in the *#Blob* stream.
- The *Parent* entry must hold a valid reference to the Field, Property, or Param table.
- No duplicate records that contain the same *Parent* value can exist.

MemberRef Table Validity Rules

- [run time] The *Class* entry must hold a valid reference to one of the following tables: TypeRef, TypeSpec, ModuleRef, MemberRef, or Method.
- [run time] The *Class* entry of a *MemberRef* record referencing a field must hold a valid reference to the TypeRef or ModuleRef table.
- [run time] The *Name* entry must hold a valid offset in the *#Strings* stream, indexing a nonempty string no longer than 1023 bytes in UTF-8 encoding.
- [run time] The name defined by the *Name* entry must not match the common language runtime reserved names *_Deleted** or *_VtblGap**.
- [run time] The *Signature* entry must hold a valid offset in the *#Blob* stream, indexing a valid *MemberRef* signature. Validity rules for MemberRef signatures are discussed in Chapter 7.
- No duplicate records with all three entries matching can exist.
- An item (field or method) that a *MemberRef* record references must not have the accessibility flag *privatescope*.

9

Methods

Methods are the third and the last leg of the tripod supporting the entire concept of managed programming, the first two being types and fields. When it comes down to execution, types, fields, and methods are the central players, with the rest of the metadata simply providing additional information about this triad.

Method items can appear in three contexts: a method definition, a method reference (for example, when a method is called), and a method implementation (when a method provides implementation of another method).

Method Metadata

Similar to field-related metadata, method-related metadata involves definition-specific and reference-specific metadata. In addition, method-related metadata includes method implementation, discussed later in this chapter, as well as method semantics, method interoperability, and security metadata. (Chapter 12, "Events and Properties," describes method semantics; Chapter 15, "Managed and Unmanaged Code Interoperation," examines method interoperability; and Chapter 14, "Security Attributes," includes method security.) The diagram in Figure 9-1 shows the metadata tables involved in method definition and referencing implementation and their mutual dependencies. To avoid cluttering the illustration, I have not included metadata tables involved in the other three method-related aspects: method semantics, method interoperability, and security metadata.

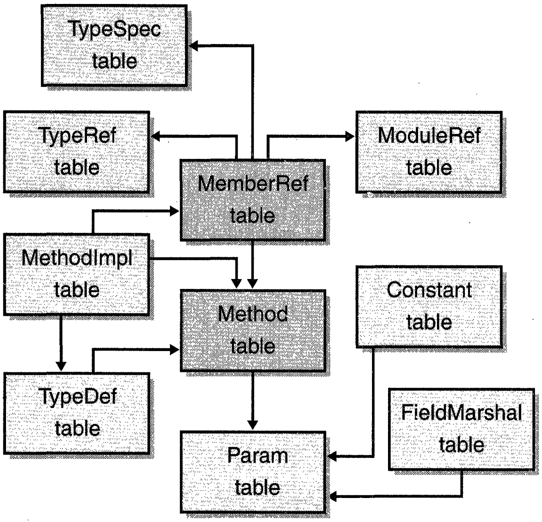


Figure 9-1 Metadata tables related to method definition and referencing.

Method Table Record Entries

The central table for method definition is the Method table, which has the associated token type *mdtMethodDef* (0x06000000). A record in the Method table has six entries:

- ***RVA* (4-byte unsigned integer)** The relative virtual address (RVA) of the method body in the module. The method body consists of header, IL code, and structured exception handling descriptors. The RVA must point to a read-only section of the PE file.

- ***ImplFlags* (2-byte unsigned integer)** Implementation binary flags indicating the specifics of the method implementation.
- ***Flags* (2-byte unsigned integer)** Binary flags indicating the method's accessibility and other characteristics.
- ***Name* (offset in the *#Strings* stream)** The name of the method. This entry must index a string of positive length no longer than 1023 bytes in UTF-8 encoding.
- ***Signature* (offset in the *#Blob* stream)** The method signature. This entry must index a blob of positive size and must comply with the method definition signature rules described in Chapter 7, "Primitive Types and Signatures."
- ***ParamList* (RID to the Param table)** The record index of the start of the parameter list belonging to this method. The end of the parameter list is defined by the start of the next method's parameter list or by the end of the Param table.

As in the case of field definition, *Method* records carry no information regarding the parent class of the method. Instead, the Method table is referenced in the *MethodList* entries of *TypeDef* records, indexing the start of *Method* records belonging to each particular *TypeDef*.

The *RVA* entry must be 0 or must hold a valid relative virtual address pointing to a read-only section of the image file. If the *RVA* value points to a read/write section, the loader will reject the method unless the application is run from a local drive with all security checks disabled. If the *RVA* entry holds 0, it means that this method is implemented somewhere else (imported from a COM library, platform-invoked from an unmanaged DLL, or simply implemented by descendants of the class owning this method). All these cases are described by special combinations of method flags and implementation flags.

The IL assembly language (ILAsm) syntax for method definition is the following:

```
<method_def> ::=
.method <flags> <call_conv> <ret_type> <name>(<arg_list>) <impl> {
   <method_body> }
```

where *<call_conv>*, *<ret_type>*, and *<arg_list>* are the method calling convention, the return type, and the argument list defining the method signature.

Method Flags

The nonterminal symbol <*flags*> identifies the method binary flags, which are defined in the enumeration *CorMethodAttr* in CorHdr.h and are described in the following list.

- Accessibility flags (mask 0x0007), which are similar to the accessibility flags of fields:
    - ❑ ***privatescope* (0x0000)** This is the default accessibility. A private scope method is exempt from the requirement of having a unique triad of owner, name, and signature and hence must always be referenced by a *MethodDef* token and never by a *MemberRef* token. Otherwise, this accessibility is the same as that specified by the *private* flag.
    - ❑ ***private* (0x0001)** The method is accessible from its owner and from classes nested in the method's owner.
    - ❑ ***famandassem* (0x0002)** The method is accessible from types belonging to the owner's family—that is, the owner itself and all its descendants—defined in the current assembly.
    - ❑ ***assembly* (0x0003)** The method is accessible from types defined in the current assembly.
    - ❑ ***family* (0x0004)** The method is accessible from the owner's family.
    - ❑ ***famorassem* (0x0005)** The method is accessible from the owner's family and from all types defined in the current assembly.
    - ❑ ***public* (0x0006)** The method is accessible from any type.
- Contract flags (mask 0x00F0):
    - ❑ ***static* (0x0010)** The method is static, shared by all instances of the type.
    - ❑ ***final* (0x0020)** The method cannot be overridden. This flag must be paired with the *virtual* flag.
    - ❑ ***virtual* (0x0040)** The method is virtual. This flag cannot be paired with the *static* flag.
    - ❑ ***hidebysig* (0x0080)** The method hides all methods of the parent classes that have a matching signature and name (as opposed to having a matching name only). This flag is ignored

by the common language runtime and is provided for the use of compilers only. The ILAsm compiler recognizes this flag but does not use it for its own purposes.

- Virtual method table (v-table) control flag (mask 0x0100):
    - ❑ ***newslot* (0x0100)** A new slot is created in the class's v-table for this virtual method so that it does not override the virtual method of the same name and signature this class inherited from its base class. This flag can be used only in conjunction with the *virtual* flag.
- Implementation flags (mask 0x2C08):
    - ❑ ***abstract* (0x0400)** The method is abstract; no implementation is provided. This method must be overridden by the non-abstract descendants of the class owning the abstract method. Any class owning an abstract method must have its own *abstract* flag set. The *RVA* entry of an abstract method record must be 0.
    - ❑ ***specialname* (0x0800)** The method is special in some way, as described by the name.
    - ❑ ***pinvokeimpl(<pinvoke_spec>)* (0x2000)** The method has unmanaged implementation and is called through the platform invocation mechanism *P/Invoke*, discussed in Chapter 15. *<pinvoke_spec>* in parentheses defines the implementation map, which is a record in the ImplMap metadata table specifying the unmanaged DLL exporting the method and the method's unmanaged calling convention. If *<pinvoke_spec>* is provided, the method's RVA must be 0, since the method is implemented externally. If *<pinvoke_spec>* is not provided—that is, the parentheses are empty—the defined method is a local *P/Invoke*, implemented in unmanaged native code embedded in the current PE file; in this case, its RVA must not be 0.
    - ❑ ***unmanagedexp* (0x0008)** The managed method is exposed as an unmanaged export. This flag is not currently used by the common language runtime.
- Reserved flags (cannot be set explicitly; mask 0xD000):
    - ❑ ***rtspecialname* (0x1000)** The method has a special name reserved for the internal use of the runtime. Four method names are reserved: *.ctor* for instance constructors, *.cctor* for

class constructors, *_VtblGap** for v-table placeholders, and *_Deleted** for methods marked for deletion but not actually removed from metadata. The keyword *rtspecialname* is ignored by the ILAsm compiler and is displayed by the IL Disassembler for informational purposes only. This flag must be accompanied by a *specialname* flag.

- **[no ILAsm keyword] (0x4000)** The method either has an associated *DeclSecurity* metadata record that holds security details concerning access to the method or has the associated custom attribute *System.Security.SuppressUnmanagedCodeSecurityAttribute.*
- ***reqsecobj* (0x8000)** Because this method calls another method containing security code, it requires an additional stack slot for a security object. This flag is formally under the *Reserved* mask, so it cannot be set explicitly. Setting this flag requires emitting the pseudocustom attribute *System.Security.DynamicSecurityMethodAttribute.* When the ILAsm compiler encounters the keyword *reqsecobj*, it does exactly that: emits the pseudo-custom attribute and thus sets this "reserved" flag.

Note I've used the word *implementation* here and there rather extensively; perhaps some clarification is in order, to avoid confusion. First, note that method implementation in the sense of one method providing implementation for another is discussed later in this chapter. Implementation-specific flags of a method are not related to that topic; rather, they indicate the features of implementation of the current method. Second, a *Method* record contains two binary flag entries: *Flags* and *ImplFlags* (implementation flags). It so happens that part of *Flags* (mask 0x2C08) is also implementation-related. Thus far, I have been talking about this part of *Flags*. For information about *ImplFlags*, see "Method Implementation Flags" later in this chapter.

Method Name

A method name in ILAsm is either a simple name or one of the two keywords *.ctor* or *.cctor*. As you already know, *.ctor* is the reserved name for instance constructors, while *.cctor* is reserved for class constructors, or type initializers.

In ILAsm, *.ctor* and *.cctor* are keywords, so they should not be single-quoted as any other irregular simple name.

The general requirements for a method name are straightforward: the name must contain 1 to 1023 bytes in UTF-8 encoding, and it should not match one of the four reserved method names—unless you really mean it. If you give a method one of these reserved names, the common language runtime treats the method according to this name.

Method Implementation Flags

The nonterminal symbol <*impl*> in the method definition form denotes the implementation flags of the method (the *ImplFlags* entry of a *Method* record). The implementation flags are defined in the enumeration *CorMethodImpl* in CorHdr.h and are described in the following list.

- Code type (mask 0x0003):
    - ***cil* (0x0000)** The default. The method is implemented in common intermediate language (CIL, a.k.a. IL, MSIL).
    - ***native* (0x0001)** The method is implemented in native platform-specific code.
    - ***optil* (0x0002)** The method is implemented in optimized IL. Because the optimized IL is not supported in the first release of the common language runtime, this flag should not be set.
    - ***runtime* (0x0003)** The method implementation is provided by the runtime itself. If this flag is set, the RVA of the method must be 0.
- Code management (mask 0x0004):
    - ***managed* (0x0000)** The default. The code is managed. In the first release of the runtime, this flag cannot be paired with the *native* flag.
    - ***unmanaged* (0x0004)** The code is unmanaged. This flag must be paired with the *native* flag.
- Implementation and interoperability (mask 0x10D8):
    - ***forwardref* (0x0010)** The method is defined, but the IL code of the method is not supplied. This flag is used primarily in edit-and-continue scenarios and in managed object files, produced by the Microsoft Managed C++ (MC++) compiler. This flag should not be set for any of the methods in a managed PE file.

- ❑ ***preservesig* (0x0080)** The method signature must not be mangled during the interoperation with classic COM code, which is discussed in Chapter 15.
- ❑ ***internalcall* (0x1000)** Reserved for internal use. This flag indicates that the method is internal to the runtime and must be called in a special way. If this flag is set, the RVA of the method must be 0.
- ❑ ***synchronized* (0x0020)** The method must be executed in single-threaded mode only. Methods belonging to value types cannot have this flag set.
- ❑ ***noinlining* (0x0008)** The runtime is not allowed to inline the method—that is, to replace the method call with explicit insertion of the method's IL code.

Take a look at the examples shown here:

```
.method public static int32 Diff(int32,int32) cil managed
{
   ⋮
}
.method public void .ctor( ) runtime internalcall {}
```

Method Parameters

Method parameters reside in the Param metadata table, whose records have three entries:

- ***Flags* (2-byte unsigned integer)** Binary flags characterizing the parameter.
- ***Sequence* (2-byte unsigned integer)** The sequence number of the parameter, with 0 corresponding to the method return.
- ***Name* (offset in the *#Strings* stream)** The name of the parameter, which can be zero-length. For the method return, it must be zero-length.

Parameter flags are defined in the enumeration *CorParamAttr* in CorHdr.h and are described in the following list.

- Input/output flags (mask 0x0013):
    - ❑ ***in* (0x0001)** Input parameter.
    - ❑ ***out* (0x0002)** Output parameter.
    - ❑ ***opt* (0x0010)** Optional parameter.

- Reserved flags (cannot be set explicitly; mask 0xF000):
  - **[no ILAsm keyword] (0x1000)** The parameter has an associated *Constant* record. The flag is set by the metadata emission API when the respective *Constant* record is emitted.
  - ***marshal(<native_type>)* (0x2000)** The parameter has an associated *FieldMarshal* record specifying how the parameter must be marshaled when consumed by unmanaged code. This is similar to the *marshal(...)* construct of a field.

To describe the ILAsm syntax of parameter definition, let me remind you of the method definition form:

```
<method_def> ::=
.method <flags> <call_conv> <ret_type> <name>(<arg_list>) <impl> {
   <method_body> }
```

where

```
<ret_type> ::=  <type> [marshal(<native_type>)],
<arg_list> ::= [ <arg> [,<arg>*] ],
<arg> ::= [ [<in_out_flag>]* ] <type> [marshal(<native_type>)]
          [<p_name>]
<in_out_flag> ::= in | out | opt
```

Obviously, *<p_name>* is the name of the parameter, which, if provided, must be a simple name.

Notice the difference in positioning of the marshaling specification in a parameter and in a field definition: in a parameter definition, the marshaling specification follows the *<type>;* in a field definition, the marshaling specification precedes the *<type>*. Here is an example of parameter definitions:

```
.method public static int32 marshal(int) Diff(
[in] int32 marshal(int) First,
[in] int32 marshal(int) Second)
{
   ⋮
}
```

The syntax just shown takes care of all the entries of a *Param* record (*Flags*, *Sequence*, *Name*) and, if needed, those of the associated *FieldMarshal* record (*Parent*, *NativeType*). To set the default values for the parameters, which are records in the Constant table, we need to add parameter specifications within the method scope:

```
<param_const_def> ::= .param [<sequence>] = <const_type> [ (<value>) ]
```

<sequence> is the parameter's sequence number. This number should not be 0, because a 0 sequence number corresponds to the return type, and a "default return value" does not make sense. *<const_type>* and *<value>* are the same as for field default value definitions, described in Chapter 8, "Fields and Data Constants." For example:

```
.method public static int32 marshal(int) Diff(
[in] int32 marshal(int) First,
[opt] int32 marshal(int) Second)
{
   .param [2] = int32(0)
   ⋮
}
```

According to the common language runtime metadata model, it is not necessary to emit a *Param* record for each return or argument of a method. Rather, it must be done only if we want to specify the name, flags, marshaling, or default value. The ILAsm compiler emits *Param* records for all arguments unconditionally and for a method return only if marshaling is specified. Name, flags, and default value are not applicable to a method return.

Referencing the Methods

Method references, like field references, translate into either *MethodDef* tokens or *MemeberRef* tokens. As a rule, a reference to a locally defined method translates into a *MethodDef* token. However, even a locally defined method can be represented by a *MemberRef* token; and in certain cases, such as references to *vararg* methods, it *must* be represented by a *MemberRef* token.

The ILAsm syntax for method referencing is as follows:

```
<method_ref> ::=
[method] <call_conv> <ret_type> <class_ref>::<name>(<arg_list>)
```

The *method* keyword, with no leading dot, is used in the following two cases in which the kind of metadata item being referenced is not clear from the context:

- When a method is referenced as an argument of the *ldtoken* instruction.
- When a method is referenced in an explicit specification of a custom attribute's owner. (See Chapter 13, "Custom Attributes," for more information.)

The same rules apply to the use of the *field* keyword in field references. The *method* keyword is used in one additional context: when specifying a function pointer as a type of field, variable, or parameter. That case, however, involves not a method reference but a signature definition.

Flags, implementation flags, and parameter-related information (names, marshaling, and so on) are not specified in a method reference. As you know, a *MemberRef* record holds only the member's parent token, name, and signature—the three elements needed to identify a method or a field unambiguously. Here are a few examples of method references:

```
call instance void Foo::Bar(int32,int32)
ldtoken method instance void Foo::Bar(int32,int32)
```

In the case of method references, the nonterminal symbol *<class_ref>* can be a *TypeDef*, *TypeRef*, *TypeSpec*, or *ModuleRef*:

```
call instance void Foo::Bar(int32,int32)
call instance void [OtherAssembly]Foo::Bar(int32,int32)
call instance void Foo[]::Bar(int32,int32)
call void [.module Other.dll]::Bar(int32,int32)
```

Method Implementation Metadata

Method implementations represent specific metadata describing method overriding, in which one method's implementation is substituted for another method's implementation. The method implementation metadata is held in the MethodImpl table, which has the following structure:

- ***Class* (RID to the TypeDef table)** The record index of the *TypeDef* implementing a method—in other words, replacing the method's implementation with that of another method.
- ***MethodBody* (coded token of type *MethodDefOrRef*)** A token indexing a record in the Method table that corresponds to the implementing method—that is, to the method whose implementation substitutes for another method's implementation. A coded token of this type can point to the MemberRef table as well, but this is illegal in the first release of the common language runtime. The method indexed by *MethodBody* must be virtual. In the first release of the runtime, the method indexed by *MethodBody* must belong to the class indexed by the *Class* entry.
- ***MethodDecl* (coded token of type *MethodDefOrRef*)** A token indexing a record in the Method table or the MemberRef table that corresponds to the implemented method—that is, to the method whose implementation is being replaced by another method's implementation. The method indexed by *MethodDecl* must be virtual.

Static, Instance, Virtual Methods

We can classify methods in many ways: global methods vs. member methods, variable argument lists vs. constant argument lists, and so on. Global and *vararg* methods are discussed in later sections. In this section, we'll focus on static vs. instance methods. Take a look at the diagram shown in Figure 9-2.

Static methods are shared by all instances of a type. They don't require an instance pointer (*this*) and cannot access instance members unless the instance pointer is provided explicitly. When a type is loaded, static methods are placed in a separate typewide table.

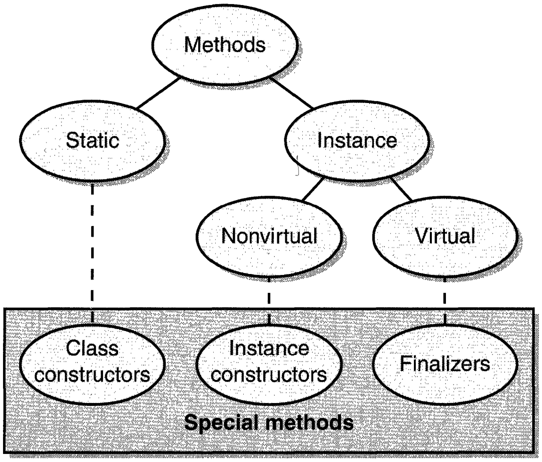


Figure 9-2 Method classification.

The signature of a static method is exactly as it is specified, with the first specified argument being number 0:

```
.method public static void Bar(int32 i, float32 r)
{
   ldarg.0 // Load int32 i on stack
   ⋮
}
```

Instance methods are instance-specific and have the *this* instance pointer as an unlisted first (number 0) argument of the signature:

```
.method public instance void Bar(int32 i, float32 r)
{
   ldarg.0 // Load instance pointer on stack
   ldarg.1 // Load int32 i on stack
   ⋮
}
```

Note Be careful about the use of the keyword *instance* in specifying the method calling convention. When a method is defined, its flags—including the *static* flag—are explicitly specified. Because of this, at definition time it's not necessary to specify an *instance* calling convention—it can be inferred from the presence or absence of the *static* flag. When a method is referenced, however, its flags are not specified, so in this case the *instance* keyword *must* be specified explicitly for instance methods; otherwise, the referenced method is presumed static. This creates a seeming contradiction: a method when declared is *instance* by default (no *static* flag specified), and the same method when referenced is *static* by default (no *instance* specified). But *static* is a flag and *instance* is a calling convention, so in fact we're dealing with two different default options here.

Instance methods are divided into virtual and nonvirtual methods, identified by the presence or absence of the *virtual* flag. The virtual methods of a class are called through the virtual method table (v-table) of this class, which adds another level of indirection to implement so-called late binding. Virtual methods can be overridden in derived classes by their own virtual methods of the same signature and name—and even of a different name, although such overriding requires an explicit declaration, as described later in this chapter. Virtual methods can be abstract or might offer some implementation.

If you have a nonvirtual method declared in a class, it does not mean that you can't declare another nonvirtual method with the same name and signature in a class derived from the first one. You can, but it will be a different method, having nothing to do with the method declared in the base class. Such a method in the derived class *hides* the respective method in the base class, but the hidden method can still be called if you explicitly specify the owning class.

If you do the same with virtual methods, however, the method declared in the derived class actually replaces, or *overrides*, the method declared in the base class. This is true unless, of course, you specify the *newslot* flag on the overriding method, in which case the overriding method will occupy a new entry of the v-table and hence will not really be overriding anything.

To illustrate this point, take a look at the following code from the sample file Virt_not.il on the companion CD:

```
.class public A
{
   .method public specialname void .ctor() { ret }
   .method public void Foo()
   {
```

(continued)

```
      ldstr "A::Foo"
      call void [mscorlib]System.Console::WriteLine(string)
      ret
   }
   .method public virtual void Bar()
   {
      ldstr "A::Bar"
      call void [mscorlib]System.Console::WriteLine(string)
      ret
   }
   .method public virtual void Baz()
   {
      ldstr "A::Baz"
      call void [mscorlib]System.Console::WriteLine(string)
      ret
   }

}

.class public B extends A
{
   .method public specialname void .ctor() { ret }
   .method public void Foo()
   {
      ldstr "B::Foo"
      call void [mscorlib]System.Console::WriteLine(string)
      ret
   }
   .method public virtual void Bar()
   {
      ldstr "B::Bar"
      call void [mscorlib]System.Console::WriteLine(string)
      ret
   }
   .method public virtual newslot void Baz()
   {
      ldstr "B::Baz"
      call void [mscorlib]System.Console::WriteLine(string)
      ret
   }
}

.method public static void Exec()
{
   .entrypoint
   newobj instance void B::.ctor() // Create instance of derived class
   castclass class A           // Cast it to base class

   dup                   // We need 3 instance pointers
   dup                    // On stack for 3 calls

   call instance void A::Foo()
   callvirt instance void A::Bar()
   callvirt instance void A::Baz()
   ret
}
```

If we compile and run the sample, we'll receive this output:

```
A:Foo
B:Bar
A:Baz
```

Because the method *A::Foo* is nonvirtual, declaring *B::Foo* does not affect *A::Foo* in any way. So when we cast *B* to *A* and call *A::Foo*, *B::Foo* does not enter the picture—it's a different method.

Because the *A::Bar* method is virtual, as is *B::Bar*, when we create an instance of *B*, *B::Bar* replaces *A::Bar* in the v-table. Casting *B* to *A* after that does not change anything: *B::Bar* is sitting in the v-table of the class instance, and *A::Bar* is gone. So when we call *A::Bar*, the "usurper" *B::Bar* is called instead.

Both the *A::Baz* and *B::Baz* methods are virtual, but *B::Baz* is marked *newslot*. Thus, instead of replacing *A::Baz* in the v-table, *B::Baz* takes a new entry and peacefully coexists with *A::Baz*. Since *A::Baz* is still present in the v-table of the instance, the situation is practically (oops, almost wrote "virtually"; should watch it; can't have puns in such a serious book) identical to the situation with *A::Foo* and *B::Foo*, except that the calls are done through the v-table. The Microsoft Visual Basic .NET compiler likes this concept and uses it rather extensively.

If we don't want a virtual method to be overridden in the class descendants, we can mark it with the *final* flag. If you try to override a final method, the loader fails and throws a *TypeLoad* exception.

Unboxed value types don't have v-tables. It is perfectly legal to declare the virtual methods as members of a value type, but these methods can be called only from a boxed instance of the value type:

```
.class public value XXX
{
   .method public void YYY( )
   {
      ⋮
   }
   .method public virtual void ZZZ( )
   {
      ⋮
   }
```

(continued)

```
}
.method public static void Exec( )
{
   .locals init(valuetype XXX xxx) // Variable xxx is an
                                   // Instance of XXX
   ldloca xxx                      // Load managed ptr to xxx
   call instance void XXX::YYY( )  // Legal: access to value
                                   // Type member
                                   // By managed ptr
   ldloca xxx
   callvirt instance void XXX::ZZZ( ) // Illegal: access to virtual
                                   // Methods possible only
                                   // By object reference.
   ldloc xxx                       // Load instance of XXX.
   box   valuetype XXX             // Convert it to object reference.
   callvirt instance void XXX::ZZZ( )   // Legal
   :
}
```

Explicit Method Overriding

Thus far, I've discussed *implicit* virtual method overriding—that is, a virtual method defined in a class overriding another virtual method of the same name and signature, defined in the class's ancestor or an interface the class implements. But implicit overriding covers only the simplest case.

Consider the following problem: class *A* implements interfaces *IX* and *IY*, and each of these interfaces defines its own virtual method *int32 Foo(int32)*. It is known that these methods are different and must be implemented separately. Implicit overriding can't help in this situation. It's time to use the MethodImpl metadata table.

The MethodImpl metadata table contains descriptors of explicit method overrides. An *explicit* override states which method overrides which other method. To define an explicit override in ILAsm, the following directive is used within the scope of the overriding method:

```
.override <class_ref>::<method_name>
```

The signature of the method need not be specified because the signature of the overriding method must match the signature of the overridden method, and the signature of the overriding method is known: it's the signature of the current method. For example:

```
.class public interface IX {
   .method public abstract virtual int32 Foo(int32) { }
}
.class public interface IY {
   .method public abstract virtual int32 Foo(int32) { }
}
```

```
.class public A implements IX,IY {
   .method public virtual int32 XFoo(int32) {
      .override IX::Foo
      ⋮
   }
   .method public virtual int32 YFoo(int32) {
      .override IY::Foo
      ⋮
   }
}
```

Not surprisingly, we can't override the same method with two different methods within the same class: there is only one slot in the v-table to be overridden. However, we can use the same method to override several virtual methods. Let's have a look at the following code from the sample file Override.il on the companion CD:

```
.class public A
{
   .method public specialname void .ctor() { ret }
   .method public void Foo()
   {
      ldstr "A::Foo"
      call void [mscorlib]System.Console::WriteLine(string)
      ret
   }
   .method public virtual void Bar()
   {
      ldstr "A::Bar"
      call void [mscorlib]System.Console::WriteLine(string)
      ret
   }
   .method public virtual void Baz()
   {
      ldstr "A::Baz"
      call void [mscorlib]System.Console::WriteLine(string)
      ret
   }

}

.class public B extends A
{
   .method public specialname void .ctor() { ret }
   .method public void Foo()
   {
      ldstr "B::Foo"
      call void [mscorlib]System.Console::WriteLine(string)
      ret
   }
```

(continued)

```
   .method public virtual void BarBaz()
   {
      .override A::Bar
      .override A::Baz
      ldstr "B::BarBaz"
      call void [mscorlib]System.Console::WriteLine(string)
      ret
   }
}
⋮
.method public static void Exec()
{
   .entrypoint
   newobj instance void B::.ctor() // Create instance of derived class
   castclass class A          // Cast it to base class

   dup                  // We need 3 instance pointers
   dup                  // On stack for 3 calls

   call instance void A::Foo()
   callvirt instance void A::Bar()
   callvirt instance void A::Baz()
   ⋮
   ret
}
```

The output of this code demonstrates that the method *B::BarBaz* overrides both *A::Bar* and *A::Baz*:

```
A::Foo
B::BarBaz
B::BarBaz
```

Virtual method overriding, both implicit and explicit, is propagated to the descendants of the overriding class, unless the descendants themselves override those methods. The second half of the sample file Override.il demonstrates this:

```
⋮
.class public C extends B
{
   .method public specialname void .ctor() { ret }
   // No overrides; let's inherit everything from B
}
.method public static void Exec()
{
   .entrypoint
   ⋮
   newobj instance void C::.ctor() // Create instance of derived class
   castclass class A          // Cast it to "grandparent"

   dup                    // We need 3 instance pointers
   dup                    // On stack for 3 calls

   call instance void A::Foo()
   callvirt instance void A::Bar()
   callvirt instance void A::Baz()
   ret
}
```

The output is the same, which proves that class *C* has inherited the overridden methods from class *B*:

```
A::Foo
B::BarBaz
B::BarBaz
```

ILAsm supports an extended form of the explicit override directive, placed within the class scope:

```
.override <class_ref>::<method_name> with <method_ref>
```

For example, the overriding effect would be the same in the preceding code if we defined class *B* like so:

```
.class public B extends A
{
   .method public specialname void .ctor() { ret }
   .method public void Foo()
   {
      ldstr "B::Foo"
      call void [mscorlib]System.Console::WriteLine(string)
      ret
   }
   .method public virtual void BarBaz()
   {
      ldstr "B::BarBaz"
      call void [mscorlib]System.Console::WriteLine(string)
      ret
   }
   .override A::Bar with instance void B::BarBaz()
   .override A::Baz with instance void B::BarBaz()
}
```

In the extended form of the *.override* directive, the overriding method must be fully specified because the extended form is used within the overriding class scope, not within the overriding method scope.

To tell the truth, the extended form of the *.override* directive is not very useful in the first version of the common language runtime because the overriding methods are restricted to those of the overriding class. Under these circumstances, the short form of the directive is sufficient, and I doubt that anyone would want to use the more cumbersome extended form. But I've noticed that in this industry the circumstances tend to change.

Method Header Attributes

The RVA value of a *Method* record—if it is nonzero and if the method is not implemented as embedded native code—points to the method body. The method body consists of a method header, IL code, and an optional structured exception handling (SEH) table, as shown in Figure 9-3.

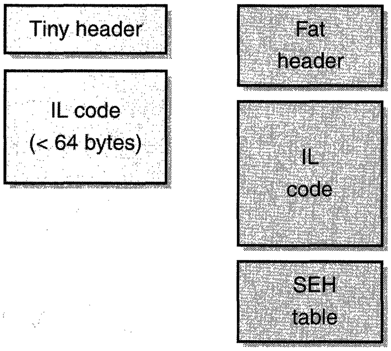


Figure 9-3 Managed method body structure.

Two types of method headers—fat and tiny—are defined in CorHdr.h. The first two bits of the header indicate its type: bits 10 stand for the tiny format, and bits 11 stand for the fat format. Why do we need two bits for a simple dichotomy? Because, speaking hypothetically, the day might come when more method header types are introduced.

A tiny method header is only 1 byte, with the first two (least significant) bits holding the type—10—and the 6 remaining bits holding the method IL code size in bytes. A method is given a tiny header if it has neither local variables nor structured exception handling, if it works fine with the default evaluation stack depth of 8 slots, and if its size is less than 64 bytes. A fat header is 12 bytes in size and has the structure described in Table 9-1. The fat headers must begin at 4-byte boundaries. The structures of both tiny and fat method headers are shown in Figure 9-4.

Table 9-1 The Fat Header Structure

Entry Size	Description
WORD	The lower 2 bits hold the fat header type code (0x3); the next 10 bits hold *Flags*. The upper 4 bits hold the size of the header in double words and must be set to 3. Currently used flags are *0x2*, which indicates that more sections follow the IL code—that is, an SEH table is present—and *0x4*, which indicates that local variables must be initialized.
WORD	*MaxStack* is the maximal evaluation stack depth in slots. Stack size in IL is measured not in bytes but in slots, with each slot able to accept one item regardless of the item's size. The default value is 8 slots, and it can be set explicitly in ILAsm by the directive *.maxstack <integer>* used inside the method scope. Be careful about trying to economize the method size by specifying *.maxstack* lower than the default: if the specified stack depth differs from the default depth, the method automatically gets a fat header even if it has no local variables, no SEH table, and a code size less than 64 bytes.
DWORD	*CodeSize* is the size of the IL code in bytes.
DWORD	*LocalVarSigTok* is the token of the local variables signature (token type 0x11000000). The structure of the local variables signature is discussed in Chapter 7. If the method has no local variables, this entry is set to 0.

Local Variables

Local variables are the typed data items that are declared within the method scope and exist from the moment the method is called until it returns. ILAsm allows us to assign names to local variables and reference them by name, but IL instructions address the local variables by their zero-based ordinals.

When the source code is compiled in debug mode, the local variable names are stored in the program database (PDB) file accompanying the module, and in this case the local variable names might survive round-tripping. In general, however, these names are not preserved because they, unlike the names of fields and method parameters, are not part of the metadata.

All the local variables, no matter when they are declared within the method scope, form a single signature, kept in the StandAloneSig metadata table (token type 0x11000000).The token referencing the respective signature is part of the method header.

Local variables are declared in ILAsm as follows:

```
.method public void Foo(int32 ii, int32 jj)
{
   .locals init (float32 ff, float64 dd, object oo, string ss)
   ⋮
}
```

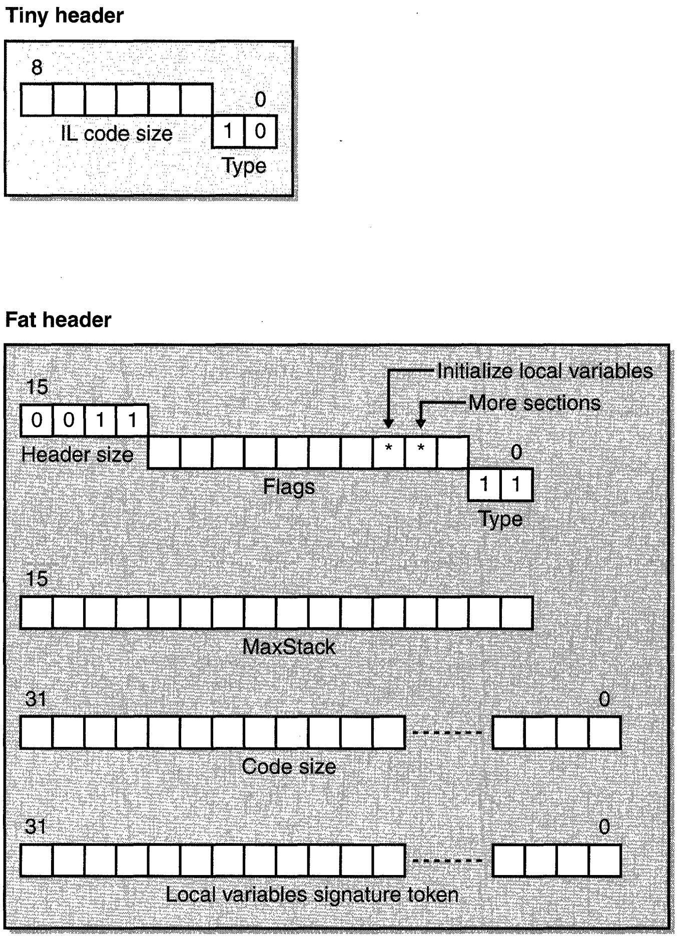


Figure 9-4 The structures of tiny and fat method headers.

The *init* keyword sets the flag *0x4* in the method header, indicating that the JIT compiler must initialize all local variables before commencing the method execution. Initialization means that for all variables of value types the corresponding default constructors are called, and all variables of object reference types are set to null. Code that contains methods without a local variable initialization flag set is deemed unverifiable and can be run from a local drive only with verification disabled.

ILAsm does not require that all local variables be declared in one place; the following is perfectly legal:

```
.method public void Foo(int32 ii, int32 jj)
{
   .locals init (float32 ff, float64 dd, object oo, string ss)
   ⋮
   {
      .locals (int32 kk, bool bb)
      ⋮
   }
   ⋮
   {
      .locals (int32 mm, float32 f)
      ⋮
   }
   ⋮

}
```

In this case, the summary local variables signature will contain the types *float32*, *float64*, *object*, *string*, *int32*, *bool*, *int32*, and *float32*. Repeating *init* in subsequent local variable declarations of the same method is not necessary because any one of the *.locals init* directives sets the local variable initialization flag.

It's obvious enough that we have a redundant local variable slot in the composite signature: by the time we need *mm*, we don't need *kk* any more, so we could reuse the slot and reduce the composite signature. In ILAsm, we can do that by explicitly specifying the 0-based slot numbers for local variables:

```
.method public void Foo(int32 ii, int32 jj)
{
   .locals init ([0]float32 ff, [1]float64 dd,
                 [2]object oo, [3]string ss)
   ⋮
   {
      .locals ([4]int32 kk, [5]bool bb)
      ⋮
   }
   ⋮
   {
      .locals ([4]int32 mm, [6]float32 f)
      ⋮
   }
   ⋮

}
```

Could we also reuse slot 5 for variable *f*? No, because the type of slot 5 is *bool*, and we need a slot of type *float32* for *f*. Only the slots holding local variables of the same type and used within nonoverlapping scopes can be reused.

Important The number of local variables declared in a method is completely unrelated to the *.maxstack* value, which depends only on how many items you might have to load simultaneously for computational purposes. For example, if you declare 20 local variables, you don't need to declare *.maxstack 20*; but if your method is calling another method that takes 20 arguments, you need to ensure that the stack has sufficient depth.

Class Constructors

Class constructors, or type initializers, are the methods specific to a type as a whole that run after the type is loaded and before any of the type's members are accessed. You've already encountered class constructors in the preceding chapter, which discussed approaches to static field initialization. That is exactly what class constructors are used for: static field initialization.

Class constructors are static, have *specialname* and *rtspecialname* flags, have neither parameters nor return value—that is, the return type is *void*—and have the name *.cctor*, which in ILAsm is a keyword rather than a name. Only one class constructor per type is permitted, and it cannot use the *vararg* calling convention.

Normally, class constructors are never called from the IL code. If a type has a class constructor, this constructor is executed automatically after the type is loaded. However, a class constructor, like any other static method, can be called explicitly. As a result of such a call, the global fields of the type are reset to their initial values. Calling *.cctor* explicitly does not lead to type reloading.

Instance Constructors

Instance constructors, unlike class constructors, are specific to an instance of a type and are used to initialize both static and instance fields of the type. Functionally, instance constructors in IL are a direct analog of C++ constructors. Instance constructors can have parameters but must return *void*, must be *instance*, must have *specialname* and *rtspecialname* flags, and have the name *.ctor*, which is also an ILAsm keyword. Like class constructors, instance constructors cannot use the *vararg* calling convention. In the first release of the common language runtime, instance constructors are not allowed to be virtual. A type can have multiple instance constructors, but they must have different parameter lists because the name (*.ctor*) and the return type (*void*) are fixed.

Usually, instance constructors are called during the execution of the *newobj* instruction, when a new type instance is created:

```
.class public Foo
{
   .field private int32 tally
   .method public void .ctor( int32 tally_init)
   {
      ldarg.0 // Load the instance reference
      ldarg.1 // Load the initializing value
      stfld int32 Foo::tally // This->tally = tally_init;
      ret
   }
   :
}
.method public static void Exec( )
{
   .locals init (class Foo foo)
   // Foo is a reference but not an instance
   ldc.i4   128 // Put 128 on stack as Foo's constructor argument
   newobj instance void Foo::.ctor(int32)
   // Instance of Foo is created
   stloc.0 // foo = new Foo(128);
   :
}
```

But, as is the case for class constructors, an instance constructor can be called explicitly. Calling the instance constructor resets the fields of the type instance and does not create a new instance. The only problem with calling class or instance constructors explicitly is that sometimes the constructors include type instantiations, if some of the fields to be initialized are of object reference type. In this case, additional care should be taken to avoid multiple type instantiations.

Warning Calling the class and instance constructors explicitly, however possible in principle, renders the code unverifiable. This limitation is imposed on the constructors of the reference types (classes) only and does not concern those of the value types. The only place where an instance constructor of a class can be called explicitly is within an instance constructor of the class's direct descendant.

Constructors of the classes cannot be the arguments of the *ldftn* instruction. In other words, you can't obtain a function pointer to a *.ctor* or *.cctor* of a class.

I repeat: all these limitations can be bypassed only if your code is run from the local drive with verification disabled. Constructors of the value types are not subject to these limitations.

Class and instance constructors are the only methods allowed to set the values of the fields marked *initonly*. If an *initonly* field is initialized by the *.cctor* of the current type, it can subsequently be modified by a *.ctor* of this type but not by any other method. Methods belonging to some other class, including *.ctor* and *.cctor*, cannot modify the *initonly* field, even if the field accessibility permits. Subsequent explicit calls to *.ctor* and *.cctor* can modify the *initonly* fields as well as the first, implicit initializing calls.

Because value types are not instantiated using the *newobj* instruction, instance constructors make less sense for them. If an instance constructor is specified for a value type, it should be called explicitly by using the *call* instruction, even though declaring a variable of a value type creates an instance of this value type. Interfaces cannot have instance constructors at all; there is no such thing as an interface instance.

Instance Finalizers

Another special method characteristic of a class instance is a finalizer, which is in many aspects similar to a C++ destructor. The finalizer must have the following signature:

```
.method family virtual instance void Finalize( )
{
   ⋮
}
```

Unlike instance constructors, which cannot be virtual, instance destructors—sorry, I mean finalizers—*must* be virtual. This requirement and the strict limitations imposed on the finalizer signature and name result from the fact that any particular finalizer is an override of the virtual method *Finalize* of the inheritance root of the class system, the *[mscorlib]System.Object* class, the ultimate ancestor of all classes in the Microsoft .NET universe. To tell the truth, the *Object*'s finalizer does exactly nothing. But *Object*, full of fatherly care, declares this virtual method anyway, so *Object*'s descendants could override it, should they desire to do something meaningful at the inevitable moment of their instances' demise.

The finalizer is executed by the garbage collection (GC) subsystem of the runtime when that subsystem decides that a class instance should be disposed of. No one knows exactly when this happens; the only solid fact is that it occurs after the instance is no longer used and has become inaccessible. But how soon after is anybody's guess.

If you prefer to execute the instance's last will and testament—that is, call the finalizer—when *you* think you don't need the instance any more, you can do exactly that by calling the finalizer explicitly. But then you should notify the GC subsystem that it does not need to call the finalizer again when in due time it decides to dispose of the abandoned class instance. You can do this by calling the .NET Framework class library method *[mscorlib]System.GC::SuppressFinalize*, which takes the object (a reference to the instance) as its sole argument—the instance is still there; you simply called its finalizer but did not destroy it—and returns *void.*

If for some reason you change your mind afterward, you can notify the GC subsystem that the finalizer must be run after all by calling the *[mscorlib]System.GC::ReRegisterForFinalize* method with the same signature, *void(object).* You needn't fear that the GC subsystem might destroy your long-suffering instance without finalization before you call *ReRegisterForFinalize*—as long as you can still reference this instance, the GC will not touch it. Both methods for controlling finalization are public and static, so they can be called from anywhere.

Variable Argument Lists

Encounters with variable argument list (*vararg*) methods in earlier chapters revealed the following information:

- The calling convention of these methods is *vararg*.
- Only mandatory parameters, if any, are specified in the *vararg* method declaration:

  ```
  .method public static vararg void Print(string Format)
  { ... }
  ```

- If and only if optional parameters are specified in a *vararg* method reference at the call site, they are preceded by a sentinel—an ellipsis in ILAsm notation—and a comma:

  ```
  call vararg void Print(string, ..., int32, float32, string)
  ```

I'm not sure that requiring the sentinel to appear as an independent comma-separated argument was a good idea. After all, a sentinel is not a true element type but is a modifier of the element type immediately following. Nevertheless, such is ILAsm notation in the first release of the common language runtime, and we'll have to live with it at least for a while.

The *vararg* method signature at the call site obviously differs from the signature specified when the method is defined, because it carries information about optional parameters. That's why the *vararg* methods are always referenced by *MemberRef* tokens and never by *MethodDef* tokens, even if the method is defined in the current module. (In that case, the *MemberRef* record corresponding to the *vararg* call site will have the respective *MethodDef* as its parent, which is slightly disturbing, but only at first sight.)

Now let's see how the *vararg* methods are implemented. IL offers no specific instructions for argument list parsing beyond the *arglist* instruction, which merely creates the argument list structure. To work with this structure and iterate through the argument list, you need to work with the .NET Framework class library value type *[mscorlib]System.ArgIterator*. This value type should be initialized with the argument list structure, which is an instance of the value type *[mscorlib]System.RuntimeArgumentHandle*, returned by the *arglist* instruction. *ArgIterator* offers such useful methods as *GetRemainingCount* and *GetNextArg*.

To make a long story short, let's review the following code snippet from the sample file Vararg.il on the companion CD:

```
// Compute sum of undefined number of arguments
.method public static vararg unsigned int64
   Sum(/* all arguments optional */)
{
   .locals init(value class [mscorlib]System.ArgIterator Args,
                unsigned int64 Sum,
                int32 NumArgs)
   ldc.i8 0
   stloc Sum

   ldloca Args
   arglist // Create argument list structure
   // Initialize ArgIterator with this structure:
   call instance void [mscorlib]System.ArgIterator::.ctor(
      value class [mscorlib]System.RuntimeArgumentHandle)

   // Get the optional argument count:
   ldloca Args
   call instance int32 System.ArgIterator::GetRemainingCount()
   stloc NumArgs

   // Main cycle:
 LOOP:
   ldloc NumArgs
   brfalse RETURN // if(NumArgs == 0) goto RETURN;

   // Get next argument:
   ldloca Args
   call instance typedref [mscorlib]System.ArgIterator::GetNextArg()

   // Interpret it as unsigned int64:
   refanyval [mscorlib]System.UInt64
   ldind.u8

   // Add it to Sum:
   ldloc Sum
   add
   stloc Sum // Sum += *((int64*)&next_arg)
   // Decrease NumArgs and go for next argument:
   ldloc NumArgs
   ldc.i4.m1
   add
   stloc NumArgs
   br LOOP

 RETURN:
   ldloc Sum
   ret
}
```

In this code, we did not specify any mandatory arguments and thus took the return value of *GetRemainingCount* for the argument count. Actually, *GetRemainingCount* returns only the number of optional arguments, which means that if we had specified N mandatory arguments, the total argument count would have been greater by N.

The *GetNextArg* method returns a typed reference, *typedref*, which is cast to a managed pointer to an 8-byte unsigned integer by the instruction *refanyval [mscorlib]System.UInt64*. If the type of the argument cannot be converted to the required type, the JIT compiler throws an *InvalidCast* exception.

Global Methods

Global methods, similar to global fields, are defined outside any class scope. Most of the features of global fields and global methods are also similar: global methods are all static, and the accessibility flags for both mean the same.

Of course, one global method worth a special mention is the global class constructor, *.cctor*. As the preceding chapter discussed, a global *.cctor* is the best way to initialize global fields. The following code snippet from the sample file Gcctor.il on the companion CD provides an example:

```
.field private static string Hello
.method private static void .cctor( )
{
   ldstr "Hi there! What's up?"
   stsfld string Hello
```

(continued)

```
   ret
}
.method public static void Exec( )
{
   .entrypoint
   ldsfld string Hello // Global fields are accessible
                       // within the module
   call void [mscorlib]System.Console::WriteLine(string)
   ret
}
```

Metadata Validity Rules

Method-related metadata tables discussed in this chapter include the Method, Param, FieldMarshal, Constant, MemberRef, and MethodImpl tables. The records in these tables have the following entries:

- The Method table: *RVA*, *ImplFlags*, *Flags*, *Name*, *Signature*, and *ParamList*.
- The Param table: *Flags*, *Sequence*, and *Names*.
- The FieldMarshal table: *Parent* and *NativeType* (native signature).
- The Constant table: *Type*, *Parent*, and *Value*.
- The MemberRef table: *Class*, *Name*, and *Signature*.
- The MethodImpl table: *Class*, *MethodBody*, and *MethodDecl*.

Chapter 8 summarized the validity rules for the FieldMarshal, Constant, and MemberRef tables. The only point to mention here regarding the MemberRef table is that, unlike field-referencing *MemberRef* records, method-referencing records can have the TypeSpec or Method table referenced in the *Parent* entry. The Method table can be referenced exclusively by the *MemberRef* records representing *vararg* call sites.

Method Table Validity Rules

- The *Flags* entry can have only those bits set that are defined in the enumeration *CorMethodAttr* in CorHdr.h (validity mask 0xFDF7).
- [run time] The accessibility flag (mask 0x0007) must be one of the following: *privatescope*, *private*, *famandassem*, *assembly*, *family*, *famorassem*, or *public*.
- The *static* flag must not be combined with any of the following flags: *final*, *virtual*, *newslot*, or *abstract*.
- The *pinvokeimpl* flag must be paired with the *static* flag (but not vice

versa).

- Methods having *privatescope* accessibility must not have the *virtual*, *final*, *newslot*, *specialname*, or *rtspecialname* flag set.
- The *abstract*, *newslot*, and *final* flags must be paired with the *virtual* flag.
- The *abstract* flag and the implementation flag *forwardref* are mutually exclusive.
- [run time] If the flag 0x4000 is set, the method must either have an associated *DeclSecurity* metadata record that holds security information concerning access to the method or have the associated custom attribute *System.Security.SuppressUnmanagedCodeSecurityAttribute*. The inverse is true as well.
- [run time] Methods belonging to interfaces must have either the *static* flag or the *virtual* flag set.
- [run time] Global methods must have the *static* flag set.
- If the *rtspecialname* flag is set, the *specialname* flag must also be set.
- The *ImplFlags* entry must have only those bits set that are defined in the enumeration *CorMethodImplAttr* in CorHdr.h (validity mask 0x10BF).
- The implementation flag *forwardref* is used only during in-memory edit-and-continue scenarios and in object files (generated by the MC++ compiler) and must not be set for any method in a managed PE file.
- [run time] The implementation flags *cil* and *unmanaged* are mutually exclusive.
- [run time] The implementation flags *native* and *managed* are mutually exclusive.
- The implementation flag *native* must be paired with the *unmanaged* flag.
- [run time] The implementation flag *synchronized* must not be set for methods belonging to value types.
- [run time] The implementation flags *runtime* and *internalcall* are for internal use only and must not be set for methods defined outside .NET Framework system assemblies.
- [run time] The *Name* entry must hold a valid reference to the *#Strings*

stream, indexing a nonempty string no more than 1023 bytes long in UTF-8 encoding.

- [run time] If the method name is *.ctor*, *.cctor*, *_VtblGap**, or *_Deleted**, the *rtspecialname* flag must be set, and vice versa.
- [run time] A method named *.ctor*—an instance constructor—must not have the *static* flag or the *virtual* flag set.
- [run time] A method named *.cctor*—a class constructor—must have the *static* flag set.
- [run time] The *Signature* entry must hold a valid reference to the *#Blob* stream, indexing a valid method signature. Validity rules for method signatures are discussed in Chapter 7.
- [run time] A method named *.ctor*—an instance constructor—must return *void* and must have the default calling convention.
- [run time] A method named *.cctor*—a class constructor—must return *void*, can take no parameters, and must have the default calling convention.
- No duplicate records—attributed to the same *TypeDef* and having the same name and signature—should exist unless the accessibility flag is *privatescope*.
- [run time] The *RVA* entry must hold 0 or a valid relative virtual address pointing to a read-only section of the PE file.
- [run time] The *RVA* entry holds 0 if and only if
    - ❑ The *abstract* flag is set, or
    - ❑ The implementation flag *runtime* is set, or
    - ❑ The implementation flag *internalcall* is set, or
    - ❑ The class owning the method has the *import* flag set, or
    - ❑ The *pinvokeimpl* flag is set, the implementation flags *native* and *unmanaged* are not set, and the ImplMap table contains a record referencing the current *Method* record.

Param Table Validity Rules

- The *Flags* entry can have only those bits set that are defined in the enumeration *CorParamAttr* in CorHdr.h (validity mask 0x3013).
- [run time] If the flag 0x2000 (*pdHasFieldMarshal*) is set, the FieldMar-

shal table must contain a record referencing this *Param* record, and vice versa.

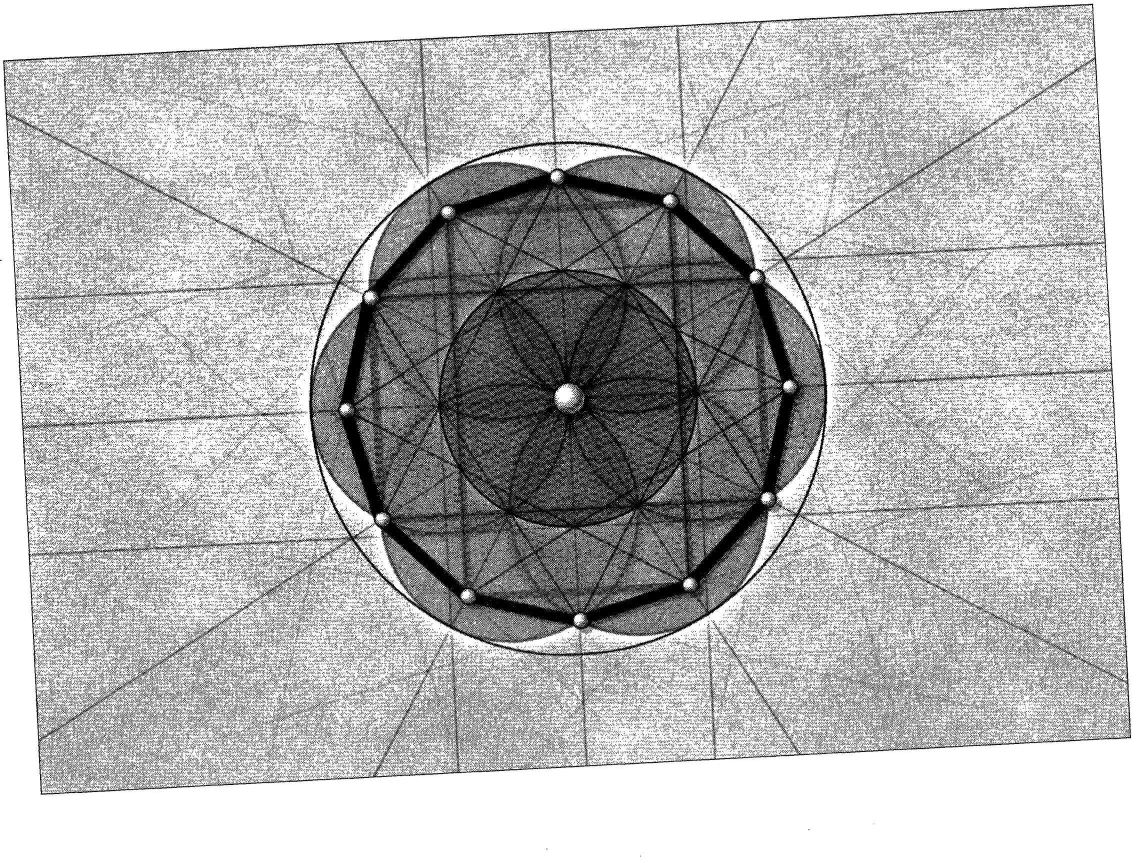
- [run time] If the flag 0x1000 (*pdHasDefault*) is set, the Constant table must contain a record referencing this *Param* record, and vice versa.
- [run time] The *Sequence* entry must hold a value no larger than the number of mandatory parameters of the method owning the *Param* record.
- If the method owning the *Param* record returns *void*, the *Sequence* entry must not hold 0.
- The *Name* entry must hold 0 or a valid reference to the *#Strings* stream, indexing a nonempty string no more than 1023 bytes long in UTF-8 encoding.

MethodImpl Table Validity Rules

- [run time] The *Class* entry must hold a valid index to the TypeDef table.
- [run time] The *MethodDecl* entry must index a record in the Method or MemberRef table.
- [run time] The method indexed by *MethodDecl* must be virtual.
- [run time] The method indexed by *MethodDecl* must not be final.
- [run time] If the parent of the method indexed by *MethodDecl* is not the *TypeDef* indexed by *Class*, the method indexed by *MethodDecl* must not be private.
- [run time] The parent of the method indexed by *MethodDecl* must not be sealed.
- [run time] The signatures of the methods indexed by *MethodDecl* and *MethodBody* must match.
- [run time] The *MethodBody* entry must index a record in the Method table.
- [run time] The method indexed by *MethodBody* must be virtual.
- [run time] The parent of the method indexed by *MethodBody* must be the *TypeDef* indexed by *Class*.

Part IV

Inside the Execution Engine



10

IL Instructions

When a method is executed, three categories of local memory plus one category of external memory are involved. All of these categories represent typed data slots, not simply an address interval as is the case in the unmanaged world. The external memory manipulated from the method is the community of the fields the method accesses. The internal memory categories include an argument table, a local variable table, and an evaluation stack. The diagram shown in Figure 10-1 describes data transitions between these categories. As you can see, all IL instructions resulting in data transfer have the evaluation stack as a source or a destination, or both.

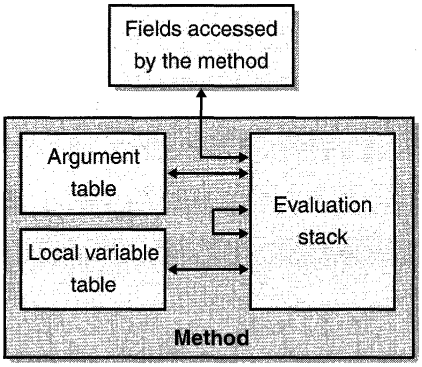


Figure 10-1 Method memory categories.

The number of slots in the argument table is inferred from the method signature at the call site (not from the method signature specified when the method is defined—remember *vararg* methods). The number of slots in the local variable table is inferred from the local variable signature whose token is specified in the method header. The number of slots in the evaluation stack is defined by the *MaxStack* value of the method header, specified in IL assembly language (ILAsm) by the *.maxstack* directive.

The slots of the argument and local variable tables have static types, which can be any of the types defined in the system. The slots of the evaluation stack have dynamic types, which change as the computations progress and the same stack slots are used for different values. The execution engine of the common language runtime implements a coarser type system for the evaluation stack: the only types a stack slot can have at a given moment are *int32*, *native int*, *int64*, *Float* (the current implementation uses 80-bit floating-point representation, which covers both *float32* and *float64* types), *&* (a managed pointer), or *ObjectRef* (an object reference, an instance pointer to an object).

The IL instruction sequences that make up the IL code of a method can be valid or verifiable, or both, or neither. The concept of validity is easy to grasp: invalid instruction sequences are rejected by the JIT (just-in-time) compiler, so nothing really bad can happen if you emit an invalid sequence—except that your code won't run.

Verifiability of the code is a security issue, not a compilation issue. Because verifiable code is not capable of any malice or hidden hacks, you can download a verifiable component from a remote location and run it without fear. If the code is deemed unverifiable—that is, if the code contains segments that just *might* contain a hack—the runtime security system will not allow it to be run except from a local disk. (I'll discuss the verifiability of IL code at the end of this chapter.) Generally, it's a good idea to check your executables with the PEVerify utility, distributed with the Microsoft .NET Framework SDK. This utility provides metadata validation and IL code verification, which includes checking both aspects—code validity and verifiability.

IL instructions consist of an *operation code* (opcode), which for some instructions is followed by an instruction parameter. Opcodes are either 1 byte or 2 bytes long; in the latter case, the first byte of the opcode is always 0xFE. In later sections of this chapter, opcodes are specified in parentheses following the instruction specification. Some instructions have synonyms, which I've also listed in parentheses immediately after the principal instruction name.

Long-Parameter and Short-Parameter Instructions

Many instructions that take an integer or an unsigned integer as a parameter have two forms. The long-parameter (original) form requires a 4-byte integer, and the short-parameter form, recognized by the suffix *.s*, requires a 1-byte integer. Short-parameter instructions are used when the value of the parameter is in the range –128 through 127 for signed parameters and in the range 0 through 255 for unsigned parameters. The long-parameter form of an instruction can also be used for parameters within these ranges, but it leads to unnecessary bloating of the IL code.

Instructions that take a metadata token as a parameter don't have short forms: metadata tokens are always used in the IL stream in uncompressed and uncoded form, as 4-byte unsigned integers.

The byte order of the integers embedded in the IL stream must be little endian—that is, the least significant byte comes first.

Labels and Flow Control Instructions

Flow control instructions include branching instructions, *switch* instructions, exiting and ending instructions used with structured exception handling (SEH) blocks, and a return instruction.

Unconditional Branching Instructions

Unconditional branching instructions take no arguments from the stack and have a signed integer parameter. The parameter specifies the offset in bytes from the current position within the IL stream. The ILAsm notation does allow you to specify the offset explicitly (for example, *br -234*), but this practice is not recommended for an obvious reason: it's difficult to calculate the offset correctly when you're writing in a programming language.

It is much safer and less troublesome to use labels instead, letting the ILAsm compiler calculate the correct offsets. Labels, which you've already encountered many times, are simple names followed by a colon:

```
⋮
Loop:
⋮
br  Loop
⋮
```

The ILAsm compiler does not automatically choose between long-parameter and short-parameter forms. Thus, if you specify a short-parameter instruction and put the target label farther than the short parameter permits, the calculated offset is truncated to 1 byte, and the ILAsm compiler issues an error message.

Unconditional branching instructions take nothing from the evaluation stack and put nothing on it.

- ***br <int32>* (0x38)** Branch *<int32>* bytes from the current position.
- ***br.s <int8>* (0x2B)** The short-parameter form of *br*.

Conditional Branching Instructions

Conditional branching instructions differ from the unconditional instructions in one aspect only: they take one *<value>* from the evaluation stack, check to see whether the condition specified by the opcode is true, and, if it is, branch according to the instruction parameter.

- ***brfalse (brnull, brzero) <int32>* (0x39)** Branch if *<value>* is 0.
- ***brfalse.s <int8>* (0x2C)** The short-parameter form of *brfalse*. Note that the synonyms *brnull.s* and *brzero.s* do not exist.
- ***brtrue (brinst) <int32>* (0x3A)** Branch if *<value>* is nonzero.
- ***brtrue.s <int8>* (0x2D)** The short-parameter form of *brtrue*. No synonyms exist.

Comparative Branching Instructions

Comparative branching instructions take two values (*<value$_1$>*, *<value$_2$>*) from the evaluation stack and compare them according to the condition specified by the opcode. Not all combinations of types of *<value$_1$>* and *<value$_2$>* are valid; Table 10-1 lists the valid combinations.

Table 10-1 Valid Type Combinations in Comparison Instructions

Type of <*value*$_i$>	Can Be Compared with Type
int32	*int32*, *native int*.
int64	*int64*.
native int	*int32*, *native int*, *&* (equality or nonequality comparisons only).
Float	*Float*. Without exception, all floating-point comparisons are formulated as <*condition*> or *unordered*. *Unordered* is true when at least one of the operands is *NaN* (not a number). A bizarre concept, if you ask me, but of course no one does.
& (managed pointer)	*native int* (equality or nonequality comparisons only), *&*. Unless the compared values are pointers to the same array or value type or pointers to pinned variables, comparing two managed pointers should be limited to equality or nonequality comparisons, because the garbage collection subsystem (GC) might move the managed pointers in an unpredictable way at unpredictable moments.
ObjectRef	*ObjectRef* (equality or nonequality comparisons only). "Greater than" unsigned comparison is also admissible and is used to compare an object reference to *null*, because objects are subject to garbage collection, and their references can be changed by the GC at will. I strongly recommend avoiding the comparison to *null*. It is much safer and more logical to use *brtrue*, and it also saves loading a *null* reference on the stack.

- ***beq <int32>* (0x3B)** Branch if <*value*$_1$> is equal to <*value*$_2$>.
- ***beq.s <int8>* (0x2E)** The short-parameter form of *beq*.
- ***bne.un <int32>* (0x40)** Branch if the two values are not equal. Integer values are interpreted as unsigned.
- ***bne.un.s <int8>* (0x33)** The short-parameter form of *bne.un*.
- ***bge <int32>* (0x3C)** Branch if greater or equal.
- ***bge.s <int8>* (0x2F)** The short-parameter form of *bge*.
- ***bge.un <int32>* (0x41)** Branch if greater or equal. Integer values are interpreted as unsigned.
- ***bge.un.s <int8>* (0x34)** The short-parameter form of *bge.un*.
- ***bgt <int32>* (0x3D)** Branch if greater.
- ***bgt.s <int8>* (0x30)** The short-parameter form of *bgt*.

- ***bgt.un <int32>* (0x42)** Branch if greater. Integer values are interpreted as unsigned.
- ***bgt.un.s <int8>* (0x35)** The short-parameter form of *bgt.un*.
- ***ble <int32>*(0x3E)** Branch if less or equal.
- ***ble.s <int8>* (0x31)** The short-parameter form of *ble*.
- ***ble.un <int32>* (0x43)** Branch if less or equal. Integer values are interpreted as unsigned.
- ***ble.un.s <int8>* (0x36)** The short-parameter form of *ble.un*.
- ***blt <int32>* (0x3F)** Branch if less.
- ***blt.s <int8>* (0x32)** The short-parameter form of *blt*.
- ***blt.un <int32>* (0x44)** Branch if less. Integer values are interpreted as unsigned.
- ***blt.un.s <int8>* (0x37)** The short-parameter form of *blt.un*.

The *switch* Instruction

The *switch* instruction implements a jump table. This instruction is unique in the sense that it has not one, not two, but N+1 parameters following it, where N is the number of cases in the switch. The first parameter is a 4-byte unsigned integer specifying the number of cases, and the following N parameters are 4-byte signed integers specifying offsets to the targets (cases). The ILAsm notation is as follows:

```
switch(Label1, Label2,…,LabelN)
:           // Default case
Label1:
:
Label2:
:
:
LabelN:
:
```

As in the case of branching instructions, ILAsm syntax allows you to replace the labels in a *switch(…)* instruction with explicit offsets, but I definitely do not recommend this.

The instruction takes one value from the stack and converts it to an unsigned integer. It then switches to the target according to the value of this unsigned integer. A 0 value corresponds to the first target offset on the list. If the value is greater than or equal to the number of targets, the *switch* instruction is ignored, and control is passed to the instruction immediately following the switch. In this sense, the default case in IL is always the first case of the switch.

- ***switch <unsigned int32> <int32>...<int32>* (0x45)** Branch to one of the *<unsigned int32>* offsets.

The *break* Instruction

This *break* instruction is *not* equivalent to the *break* statement in C, which is used as an exit from the switch cases. The *break* instruction in IL inserts a breakpoint into the IL stream and is used for debugging only. This instruction does not have parameters and does not touch the evaluation stack.

- ***break* (0x01)** Debugging breakpoint.

SEH Block Exiting Instructions

The blocks of code involved in structured exception handling cannot be entered or exited by simple branching because of the strict stack state requirements imposed on them. The *leave* instruction, or its short-parameter form, is used to exit a guarded block (a try block) or an exception handler block. You cannot use this instruction, however, to exit a filter, finally, or fault block. (For more details about these blocks, see Chapter 11, "Structured Exception Handling.")

The instruction has one integer parameter specifying the offset of the target and works the same way as an unconditional branching instruction except that it empties the evaluation stack before the branching. The ILAsm notation for this instruction is similar to the notation for unconditional branching instructions: *leave <label>* or *leave <int32>*, the latter one highly unrecommended.

- ***leave <int32>* (0xDD)** Clear the stack and branch *<int32>* bytes from the current point.
- ***leave.s <int8>* (0xDE)** The short-parameter form of *leave*.

SEH Block Ending Instructions

IL has two specific instructions to mark the end of filter, finally, and fault blocks. Unlike *leave*, these instructions mark the lexical end of a block rather than an algorithmic end, or point of exit. These instructions have no parameters.

- ***endfilter* (0xFE 0x11)** The lexical end of a filter block. The instruction takes one 4-byte integer value from the evaluation stack and signals the execution engine whether the associated exception handler should be engaged (a value of 1) or whether the exception identification should be continued (a value other than 1), because this filter doesn't know what to do with this particular exception.

- ***endfinally* (*endfault*) (0xDC)** The lexical end of a finally or fault block. This instruction clears the evaluation stack.

The *ret* Instruction

The return instruction—*ret*—returns from a called method to the call site. It has no parameters. If the called method should return a value of a certain type, exactly one value of the required type must be on the evaluation stack at the moment of return. The *ret* instruction causes this value to be removed from the evaluation stack of the called method and put on the evaluation stack of the calling method. If the called method returns *void*, its evaluation stack must be empty at the moment of return.

- ***ret* (0x2A)** Return from a method.

Arithmetical Instructions

Arithmetical operations deal with numeric data processing and include stack manipulation instructions, constant loading instructions, indirect (by pointer) loading and storing instructions, arithmetical operations, bitwise operations, data conversion operations, logical condition check operations, and block operations.

Stack Manipulation

Stack manipulation instructions work with the evaluation stack and have no parameters.

- ***nop* (0x00)** No operation; a placeholder only. The *nop* instruction is not exactly a stack manipulation instruction, since it does not touch the stack, but I've included it here rather than creating a separate category for it. The *nop* instruction is somewhat useful only in that, because it is a distinct opcode, a line of source code can be bound to it in the program database (PDB) file containing the debug information. The Microsoft Visual Basic .NET compiler introduces a lot of *nop* instructions because it wants to bind each and every line of the source code to the IL code. The reasoning behind this is not clear; perhaps the Visual Basic .NET programmers wish to be able to put breakpoints on comment lines.
- ***dup* (0x25)** Duplicate the value on the top of the stack. If the stack is empty, the JIT compiler fails because of the stack underflow.

- ***pop*** **(0x26)** Remove the value from the top of the stack. The value is lost. If the stack is empty, the JIT compiler fails. It's not healthy to invoke *dup* or *pop* on an empty stack.

Constant Loading

Constant loading instructions take at most one parameter (the constant to load) and load it on the evaluation stack. Some instructions have no parameters because the value to be loaded is specified by the opcode itself. The ILAsm syntax requires explicit specification of the constants, in decimal or hexadecimal form:

```
ldc.i4 -1
ldc.i4 0xFFFFFFFF
```

Note that the slots of the evaluation stack are either 4 or 8 bytes wide, so the constants being loaded are converted to the suitable size.

- ***ldc.i4 <int32>*** **(0x20)** Load *<int32>* on the stack.
- ***ldc.i4.s <int8>*** **(0x1F)** Load *<int8>* on the stack.
- ***ldc.i4.m1*** **(*ldc.i4.M1*) (0x15)** Load –1 on the stack.
- ***ldc.i4.0*** **(0x16)** Load 0.
- ***ldc.i4.1*** **(0x17)** Load 1.
- ***ldc.i4.2*** **(0x18)** Load 2.
- ***ldc.i4.3*** **(0x19)** Load 3.
- ***ldc.i4.4*** **(0x1A)** Load 4.
- ***ldc.i4.5*** **(0x1B)** Load 5.
- ***ldc.i4.6*** **(0x1C)** Load 6.
- ***ldc.i4.7*** **(0x1D)** Load 7.
- ***ldc.i4.8*** **(0x1E)** Load 8. (I should have listed these in reverse order so then we could imagine ourselves on Cape Canaveral.)
- ***ldc.i8 <int64>*** **(0x21)** Load *<int64>* on the stack.
- ***ldc.r4 <float32>*** **(0x22)** Load *<float32>* (single-precision) on the stack.
- ***ldc.r8 <float64>*** **(0x23)** Load *<float64>* (double-precision) on the stack. ILAsm permits the use of integer parameters in both the *ldc.r4* and *ldc.r8* instructions; in such cases, the integers are interpreted as binary images of the floating-point numbers.

Indirect Loading

An indirect loading instruction takes a managed pointer (*&*) or an unmanaged pointer (*native int*) from the stack, retrieves the value at this pointer, and puts the value on the stack. The type of the value to be retrieved is defined by the opcode. The indirect loading instructions have no parameters.

- ***ldind.i1* (0x46)** Load a signed 1-byte integer from the location specified by the pointer taken from the stack.
- ***ldind.u1* (0x47)** Load an unsigned 1-byte integer.
- ***ldind.i2* (0x48)** Load a signed 2-byte integer.
- ***ldind.u2* (0x49)** Load an unsigned 2-byte integer.
- ***ldind.i4* (0x4A)** Load a signed 4-byte integer.
- ***ldind.u4* (0x4B)** Load an unsigned 4-byte integer.
- ***ldind.i8 (ldind.u8)* (0x4C)** Load an 8-byte integer, signed or unsigned.
- ***ldind.i* (0x4D)** Load *native int*, an integer the size of a pointer.
- ***ldind.r4* (0x4E)** Load a single-precision floating-point value.
- ***ldind.r8* (0x4F)** Load a double-precision floating-point value.
- ***ldind.ref* (0x50)** Load an object reference.

Indirect Storing

Indirect storing instructions take a value and an address, in that order, from the stack and store the value at the location specified by the address. The address can be a managed or an unmanaged pointer. The type of the value to be stored is specified in the opcode. These instructions have no parameters.

- ***stind.ref* (0x51)** Store an object reference.
- ***stind.i1* (0x52)** Store a 1-byte integer.
- ***stind.i2* (0x53)** Store a 2-byte integer.
- ***stind.i4* (0x54)** Store a 4-byte integer.
- ***stind.i8* (0x55)** Store an 8-byte integer.
- ***stind.i* (0xDF)** Store a pointer-size integer.
- ***stind.r4* (0x56)** Store a single-precision floating-point value.
- ***stind.r8* (0x57)** Store a double-precision floating-point value.

Arithmetical Operations

All arithmetical operations except the negation operation take two operands from the stack and put the result on the stack. If the result value does not fit the result type, the value is truncated. Table 10-2 lists the admissible type combinations of operands and their corresponding result types.

Table 10-2 Admissible Operand Types and Their Result Types in Arithmetical Operations

Operand Type	Operand Type	Result Type
int32	int32	*int32*
int32	native int	*native int*
int32	& (addition only, unverifiable)	*&*
int64	int64	*int64*
native int	& (addition only, unverifiable)	*&*
Float	*Float* (except unsigned division)	*Float*
&	& (addition or subtraction, unverifiable)	*native int*

The arithmetical operation instructions are as follows:

- ***add* (0x58)** Addition.
- ***sub* (0x59)** Subtraction.
- ***mul* (0x5A)** Multiplication. For floating-point numbers, which have the special values *infinity* and *NaN* (not a number), the following rule applies:

  0 * infinity = NaN

- ***div* (0x5B)** Division. For integers, division by 0 results in a *DivideByZero* exception. For floating-point numbers:

  0 / 0 = NaN, infinity / infinity = NaN, x / infinity = 0

- ***div.un* (0x5C)** Unsigned division (integer types only).
- ***rem* (0x5D)** Remainder, modulo. For integers, modulo 0 results in a *DivideByZero* exception. For floating-point numbers:

infinity rem x = NaN, x rem 0 = NaN, x rem infinity = x

- ***rem.un* (0x5E)** The remainder of unsigned operands (integer operands only).
- ***neg* (0x65)** Negate—that is, invert the sign. This is the only unary arithmetical operation. It takes one operand rather than two from the evaluation stack and puts the result back. This operation is not applicable to pointer types. With integers, a peculiar situation can occur, in which the maximum negative number does not change after negation because of the overflow condition during the operation, as shown in this example:

```
ldc.i4 0x80000000 // Max. negative number for int32,
                  //-2147483648
neg
call void [mscorlib]System.Console::WriteLine(int32)
// Output: -2147483648;
// The same effect with subtraction:
ldc.i4.0
ldc.i4 0x80000000
sub
call void [mscorlib]System.Console::WriteLine(int32)
// Output: -2147483648;
```

Floating-point numbers don't have this problem. Negating *NaN* returns *NaN* because *NaN*, which is not a number, has no sign.

Overflow Arithmetical Operations

Overflow arithmetical operations are similar to the arithmetical operations described in the preceding section except that they work with integer operands only and generate an *Overflow* exception if the result does not fit the target type. The ILAsm notation for the overflow arithmetical operations contains the suffix *.ovf* following the operation kind. The type compatibility list, shown in Table 10-3, is very similar to the list shown in Table 10-2.

Table 10-3 Acceptable Operand Types and Their Result Types in Overflow Arithmetical Operations

Operand Type	Operand Type	Result Type
int32	*int32*	*int32*
int32	*native int*	*native int*

Table 10-3 Acceptable Operand Types and Their Result Types in Overflow Arithmetical Operations *(continued)*

Operand Type	Operand Type	Result Type
int32	*&* (addition only, unverifiable)	*&*
int64	*int64*	*int64*
native int	*&* (addition only, unverifiable)	*&*
&	*&* (addition or subtraction, unverifiable)	*native int*

- ***add.ovf* (0xD6)** Addition.
- ***add.ovf.un* (0xD7)** Addition of unsigned operands.
- ***sub.ovf* (0xDA)** Subtraction.
- ***sub.ovf.un* (0xDB)** Subtraction of unsigned operands.
- ***mul.ovf* (0xD8)** Multiplication.
- ***mul.ovf.un* (0xD9)** Multiplication of unsigned operands.

Bitwise Operations

Bitwise operations have no parameters and are defined for integer types only; floating-point, pointer, and object reference operands are not allowed. As a result, the related operand type compatibility list, shown in Table 10-4, is pretty simple.

Table 10-4 Acceptable Operand Types and Their Result Types in Bitwise Operations

Operand Type	Operand Type	Result Type
int32	*int32*	*int32*
int32	*native int*	*native int*
int64	*int64*	*int64*

Three of the bitwise operations are binary, taking two operands from the stack and placing one result on the stack; and one is unary, taking one operand from the stack and placing one result on the stack:

- ***and* (0x5F)** Bitwise AND (binary).
- ***or* (0x60)** Bitwise OR (binary).

- ***xor* (0x61)** Bitwise exclusive OR (binary).
- ***not* (0x66)** Bitwise inversion (unary). This operation, rather than *neg*, is recommended for integer sign inversion because *neg* has a problem with the maximum negative numbers:

```
ldc.i4 0x80000000 // Max. negative number for int32,
                  // -2147483648
not
call void [mscorlib]System.Console::WriteLine(int32)
// Output: 2147483647 (0x7FFFFFFF);
// Of course, it's not +2147483648,
// which cannot be with int32,
// but at least we have the max. positive number
```

Shift Operations

Shift operations have no parameters and are defined for integer operands only. The shift operations are binary: they take from the stack the shift count and the value being shifted, in that order, and put the shifted value on the stack. The result always has the same type as the operand being shifted, which can be of any integer type. The type of the shift count cannot be *int64* and is limited to *int32* or *native int.*

- ***shl* (0x62)** Shift left.
- ***shr* (0x63)** Shift right.
- ***shr.un* (0x64)** Shift right, treating the shifted value as unsigned.

Conversion Operations

The conversion operations have no parameters. They take a value from the stack, convert it to the type specified by the opcode, and put the result back on the stack. The specifics of the conversion obviously depend on the type of the converted value and the target type (the type to which the value is converted). If the type of the value on the stack is the same as the target type, no conversion is necessary, and the operation itself is doing nothing more than bloating the IL code.

For integer source and target types, several rules apply. If the target integer type is narrower than the source type (for example, *int32* to *int16*, or *int64* to *int32*), the value is truncated—that is, the most significant bytes are thrown away. If the situation is the opposite—if the target integer type is wider than the source—the result is either sign-extended or zero-extended, depending on the type of conversion. Conversions to signed integers use sign-extension, and conversions to unsigned integers use zero-extension.

If the source type is a pointer, it can be converted to either *unsigned int64* or *native unsigned int*. In either case, if the converted pointer was managed, it is dropped from the GC tracking and is not automatically updated when the GC rearranges the memory layout. A pointer cannot be used as a target type.

If both source and target types are floating-point, the conversion merely results in a change of precision. In float-to-integer conversions, the values are truncated toward 0—for example, the value 1.1 is converted to 1, and the value –2.3 is converted to –2. In integer-to-float conversions, the integer value is simply converted to floating-point, possibly losing fewer significant mantissa bits.

Object references cannot be subject to conversion operations either as a source or as a target.

- ***conv.i1* (0x67)** Convert the value to *int8*.
- ***conv.u1* (0xD2)** Convert the value to *unsigned int8*.
- ***conv.i2* (0x68)** Convert the value to *int16*.
- ***conv.u2* (0xD1)** Convert the value to *unsigned int16*.
- ***conv.i4* (0x69)** Convert the value to *int32*.
- ***conv.u4* (0x6D)** Convert the value to *unsigned int32*.
- ***conv.i8* (0x6A)** Convert the value to *int64*.
- ***conv.u8* (0x6E)** Convert the value to *unsigned int64*. This operation can be applied to pointers.
- ***conv.i* (0xD3)** Convert the value to *native int*.
- ***conv.u* (0xE0)** Convert the value to *native unsigned int*. This operation can be applied to pointers.
- ***conv.r4* (0x6B)** Convert the value to *float32*.
- ***conv.r8* (0x6C)** Convert the value to *float64*.
- ***conv.r.un* (0x76)** Convert an unsigned integer value to floating-point.

Overflow Conversion Operations

Overflow conversion operations differ from the conversion operations described in the preceding section in two aspects: the target types are exclusively integer types, and an *Overflow* exception is thrown whenever the value must be truncated to fit the target type. In short, the story is the same as it is with overflow arithmetical operations and arithmetical operations.

- ***conv.ovf.i1*** **(0xB3)** Convert the value to *int8*.
- ***conv.ovf.u1*** **(0xB4)** Convert the value to *unsigned int8*.
- ***conv.ovf.i1.un*** **(0x82)** Convert an unsigned integer to *int8*.
- ***conv.ovf.u1.un*** **(0x86)** Convert an unsigned integer to *unsigned int8*.
- ***conv.ovf.i2*** **(0xB5)** Convert the value to *int16*.
- ***conv.ovf.u2*** **(0xB6)** Convert the value to *unsigned int16*.
- ***conv.ovf.i2.un*** **(0x83)** Convert an unsigned integer to *int16*.
- ***conv.ovf.u2.un*** **(0x87)** Convert an unsigned integer to *unsigned int16*.
- ***conv.ovf.i4*** **(0xB7)** Convert the value to *int32*.
- ***conv.ovf.u4*** **(0xB8)** Convert the value to *unsigned int32*.
- ***conv.ovf.i4.un*** **(0x84)** Convert an unsigned integer to *int32*.
- ***conv.ovf.u4.un*** **(0x88)** Convert an unsigned integer to *unsigned int32*.
- ***conv.ovf.i8*** **(0xB9)** Convert the value to *int64*.
- ***conv.ovf.u8*** **(0xBA)** Convert the value to *unsigned int64*.
- ***conv.ovf.i8.un*** **(0x85)** Convert an unsigned integer to *int64*.
- ***conv.ovf.u8.un*** **(0x89)** Convert an unsigned integer to *unsigned int64*.
- ***conv.ovf.i*** **(0xD4)** Convert the value to *native int*.
- ***conv.ovf.u*** **(0xD5)** Convert the value to *native unsigned int*.
- ***conv.ovf.i.un*** **(0x8A)** Convert an unsigned integer to *native int*.
- ***conv.ovf.u.un*** **(0x8B)** Convert an unsigned integer to *native unsigned int*.

Logical Condition Check Operations

Logical condition check operations are similar to conditional branching operations except that they result not in branching but in putting the condition check result on the stack. The result type is *int32*, its value is equal to 1 if the condition checks and 0 otherwise. The two operands being compared are taken from the stack, and, because no branching is performed, the condition check operations have no parameters.

The admissible combinations of operand types are the same as for conditional branching instructions. (See Table 10-1.) There are, however, fewer condition check operations than conditional branching operations.

- ***ceq* (0xFE 0x01)** Check whether the two values on the stack are equal.
- ***cgt* (0xFE 0x02)** Check whether the first value is greater than the second value. It's the stack we are working with, so the "second" value is the one on the top of the stack.
- ***cgt.un* (0xFE 0x03)** Check whether the first value is greater than the second, with both values considered unsigned.
- ***clt* (0xFE 0x04)** Check whether the first value is less than the second value.
- ***clt.un* (0xFE 0x05)** Check whether the first value is less than the second, with both values considered unsigned.
- ***ckfinite* (0xC3)** This unary operation, which takes only one value from the stack, is applicable to floating-point values only. It throws an *Arithmetic* exception if the value is *+infinity*, *-infinity,* or *NaN* and puts 1 on the stack otherwise.

Block Operations

Two IL instructions deal with blocks of memory regardless of the type or types that make up this memory. Because of this type blindness, both instructions are unverifiable.

- ***cpblk* (0xFE 0x17)** Copy a block of memory. The instruction has no parameters and pops three operands from the stack in the following order: the size of the block to be copied (*unsigned int32*), the source address (a pointer or *native int*), and the destination address (a pointer or *native int*). The source and destination addresses must be aligned on the size of *native int* unless the instruction is prefixed with the *unaligned.* instruction, described in "Prefix Instructions," later in this chapter. The *cpblk* instruction puts nothing on the stack.
- ***initblk* (0xFE 0x18)** Initialize a block of memory. The instruction has no parameters and takes three operands from the evaluation stack: the size of the block (*unsigned int32*), the initialization value (*int8*), and the block start address (a pointer or *native int*). The

alignment rules mentioned above apply to the block start address. The *initblk* instruction puts nothing on the stack. As a result of this operation, each byte within the specified block is assigned the initialization value.

Addressing Arguments and Local Variables

A special group of IL instructions is dedicated to loading the values of method arguments and local variables on the evaluation stack and storing the values taken from the stack in local variables and method arguments.

Method Argument Loading

The following instructions are used for loading method argument values on the evaluation stack:

- ***ldarg <unsigned int16>* (0xFE 0x09)** Load the argument number *<unsigned int16>* on the stack. The argument enumeration is zero-based, but it's important to remember that instance methods have an "invisible" argument not specified in the method signature: the class instance pointer, *this*, which is always argument number 0. Because static methods don't have such an "invisible" argument, for them argument number 0 is the first argument specified in the method signature.
- ***ldarg.s <unsigned int8>* (0x0E)** The short-parameter form of *ldarg*.
- ***ldarg.0* (0x02)** Load argument number 0 on the stack.
- ***ldarg.1* (0x03)** Load argument number 1 on the stack.
- ***ldarg.2* (0x04)** Load argument number 2 on the stack.
- ***ldarg.3* (0x05)** Load argument number 3 on the stack.

Method Argument Address Loading

These two instructions are used for loading method argument addresses on the evaluation stack:

- ***ldarga <unsigned int16>* (0xFE 0x0A)** Load the address of argument number *<unsigned int16>* on the stack.
- ***ldarga.s <unsigned int8>* (0x0F)** The short-parameter form of *ldarga*.

Method Argument Storing

These two instructions are used for storing a value from the stack in a method argument slot:

- ***starg* <unsigned int16> (0xFE 0x0B)** Take a value from the stack and store it in argument slot number <*unsigned int16*>. The value on the stack must be of the same type as the argument slot or must be convertible to the type of the argument slot. The convertibility rules and effects are the same as those for conversion operations, discussed earlier in this chapter. With *vararg* methods, the *starg* instruction cannot target the arguments of the variable part of the signature.
- ***starg.s* <unsigned int8> (0x10)** The short-parameter form of *starg*.

Method Argument List

The following instruction is used exclusively in *vararg* methods to retrieve the method argument list and put an instance of the value type *[mscorlib]System.RuntimeArgumentHandle* on the stack. Chapter 9, "Methods," discusses the application of this instruction.

- ***arglist* (0xFE 0x00)** Get the argument list handle.

Local Variable Loading

Local variable loading instructions are similar to argument loading instructions except that no "invisible" items appear among the local variables, so local variable number 0 is always the first one specified in the local variable signature.

- ***ldloc* <unsigned int16> (0xFE 0x0C)** Load the value of local variable number <*unsigned int16*> on the stack. Local variable numbers can range from 0 to 0xFFFE.
- ***ldloc.s* <unsigned int8> (0x11)** The short-parameter form of *ldloc*.
- ***ldloc.0* (0x06)** Load the value of local variable number 0 on the stack.
- ***ldloc.1* (0x07)** Load the value of local variable number 1 on the stack.
- ***ldloc.2* (0x08)** Load the value of local variable number 2 on the stack.
- ***ldloc.3* (0x09)** Load the value of local variable number 3 on the stack.

Local Variable Reference Loading

The following instructions load references (managed pointers) to the local variables on the evaluation stack:

- ***ldloca <unsigned int16>* (0xFE 0x0D)** Load the address of local variable number *<unsigned int16>* on the stack. The local variable number can vary from 0 to 0xFFFE.
- ***ldloca.s <unsigned int8>* (0x12)** The short-parameter form of *ldloca*.

Local Variable Storing

It would be strange to have local variables and be unable to assign values to them. The following two instructions take care of this aspect of our life:

- ***stloc <unsigned int16>* (0xFE 0x0E)** Pop the value from the stack and store it in local variable slot number *<unsigned int16>*. The value on the stack must be of the same type as the argument slot or must be convertible to the type of the argument slot. The convertibility rules and effects are the same as those for the conversion operations, discussed earlier in this chapter.
- ***stloc.s <unsigned int8>* (0x13)** The short-parameter form of *stloc*. You've probably noticed that using short-parameter forms of argument and local variable manipulation instructions results in a double gain against the standard form: not only is the parameter 1 byte instead of 2, but also the opcode is shorter.

Local Block Allocation

With all due respect to the object-oriented approach, sometimes it is necessary (or just convenient) to obtain a plain, C-style chunk of memory. The IL instruction set provides an instruction for such allocation. It is to be noted, however, that the chunk of memory is allocated on the thread stack rather than on the heap. I'm talking about the native stack and heap now, used by the JIT-compiled code.

- ***localloc* (0xFE 0x0F)** Allocate a block of memory on the native thread stack. The instruction takes the block size (*native unsigned int*) from the evaluation stack and puts a managed pointer (*&*) to the allocated block on the evaluation stack. If not enough space is available on the native thread stack, a *StackOverflow* exception is thrown. This instruction must not appear within any structured exception handling block. Like any other block instruction, *localloc* is unverifiable.

Prefix Instructions

The prefix instructions listed in this section have no meaning per se but are used as prefixes for the pointer-consuming instructions—that is, instructions that take a pointer value from the stack—that immediately follow them, such as *ldind.**, *stind.**, *ldfld*, *stfld*, *ldobj*, *stobj*, *initblk*, and *stblk*. When they are used as prefixes of instructions that don't consume pointers, the prefix instructions are ignored and do *not* carry on to the nearest pointer-consuming instruction.

- ***unaligned. <unsigned int8>* (0xFE 0x12)** Indicates that the pointer(s) on the stack are *<unsigned int8>*-aligned rather than aligned on the pointer size. The *<unsigned int8>* parameter must be 1, 2, or 4.
- ***volatile.* (0xFE 0x13)** Indicates that the pointer on the stack is volatile—that is, it can be modified from another thread of execution and the results of its dereferencing therefore cannot be cached for performance considerations.

A prefix instruction affects only the immediately following instruction and does not mark the respective pointer as unaligned or volatile throughout the entire method. Both prefixes can be used with the same instruction—in other words, the pointer on the stack can be marked as both unaligned and volatile; in such a case, the order of appearance of the prefixes does not matter.

The ILAsm syntax requires the prefix instructions to be separated from the next instruction by at least a space symbol:

```
volatile. ldind.i4 // Correct

volatile.
ldind.i4 // Correct

volatile.ldind.i4 // Syntax error
```

Such a mistake is unlikely with the *unaligned.* instruction because it requires an integer parameter:

```
unaligned. 4 ldind.i4
```

Addressing Fields

Six instructions can be used to load a field value or an address on the stack or to store a value from the stack in a field. Because a field signature does not indicate whether the field is static or instance, the IL instruction set defines separate instructions for dealing with instance and static fields. Instructions dealing with instance fields take the instance pointer—an object reference if the field addressed belongs to a class, and a managed pointer if the field belongs to a value type—from the stack.

- ***ldfld* <*token*> (0x7B)** Pop the instance pointer from the stack and load the value of an instance field on the stack. <*token*> must be a valid *FieldDef* or *MemberRef* token, uncompressed and uncoded.
- ***ldsfld* <*token*> (0x7E)** Load the value of a static field on the stack.
- ***ldflda* <*token*> (0x7C)** Pop the instance pointer from the stack and load a managed pointer to the instance field on the stack.
- ***ldsflda* <*token*> (0x7F)** Load a managed pointer to the static field on the stack.
- ***stfld* <*token*> (0x7D)** Pop the value from the stack, pop the instance pointer from the stack, and store the value in the instance field.
- ***stsfld* <*token*> (0x80)** Store the value from the stack in the static field.

The ILAsm notation requires full field specification, which is resolved to <*token*> at compile time:

```
ldfld int32 Foo.Bar::ii
```

The applicable conversion rules when loading and storing values are the same as those discussed earlier. Note also that the fields cannot be of managed pointer type.

Calling Methods

Methods can be called directly or indirectly. In addition, you can also use the special case of a so-called *tail call*, discussed in this section. Because the method signature indicates whether the method is instance or static, separate instructions for instance and static methods are unnecessary. What the method signature *doesn't* hold, however, is information about whether the method is virtual. As a result, separate instructions are used for calling virtual and nonvirtual methods.

Method call instructions have one parameter: a token of the method being called, either a *MethodDef* or a *MemberRef*. The arguments of the method call should be loaded on the stack in order of their appearance in the method signature, with the last signature parameter being loaded last. Instance methods have an "invisible" first argument (an instance pointer) not present in the signature; when an instance method is called, this instance pointer should be loaded on the stack first, preceding all arguments corresponding to the method signature.

Unless the called method returns *void*, the return value is placed on the stack when the call is completed.

Direct Calls

The IL instruction set contains three instructions intended for the direct method calls:

- ***jmp <token>* (0x27)** Abandon the current method and jump to the target method, specified by *<token>*, transferring the current arguments. At the moment *jmp* is invoked, the evaluation stack must be empty, and the arguments are transferred automatically. Because of this, the signature of the target method must match the signature of the method invoking *jmp*. This instruction should not be used within SEH blocks—try, catch, filter, fault, or finally blocks, discussed in Chapter 11—or within a synchronized region. The *jmp* instruction is unverifiable.
- ***call <token>* (0x28)** Call a nonvirtual method. You can also call a virtual method, but in this case it is called not through the instance's v-table but through its type-specific method table. (If this sounds somehow vague to you, you might want to return to Chapter 9 and, more precisely, to the section "Static, Instance, Virtual Methods" and the sample file Virt_not.il.) The real difference between virtual and nonvirtual instance methods becomes obvious when you create an instance of a class, cast it to the parent type of the class, and then call instance methods on this "child-posing-as-parent" instance. Because nonvirtual methods are called through the type's method table, the parent's methods will be called in this case. Virtual methods are called through the v-table specific to the class instance, and hence the child's methods will be called. The *call* instruction works through the type's method table and ignores the instance's v-table, so the parent's methods will be called whether they are virtual or not. To confirm this, carry out a simple experiment: open the sample file Virt_not.il in a text editor and change *callvirt instance void A::Bar( )* to *call instance void A::Bar( )*. Then recompile the sample and run it.
- ***callvirt <token>* (0x6F)** Call the virtual method specified by *<token>*. This type of method call is conducted through the instance's v-table. It is possible to call a nonvirtual instance

method using *callvirt*. In this case, the method is called through the type's method table simply because the method cannot be found in the v-table. But unlike *call*, the *callvirt* instruction first checks the validity of the object reference (*this* pointer) before doing anything else, which is a very useful feature. The Microsoft Visual C# .NET compiler exploits it shamelessly, emitting *callvirt* to call both virtual and nonvirtual instance methods of classes. I say "of classes" because *callvirt* requires an object reference as the *this* pointer and will not accept a managed pointer to a value type instance.

Indirect Calls

Methods in IL can be called indirectly through the function pointer loaded on the evaluation stack. This allows us to make calls to computed targets—for example, to call a method by a function pointer returned by another method. Function pointers used in indirect calls are unmanaged pointers represented by *native int*. Two instructions load a function pointer to a specified method on the stack, and one other instruction calls a method indirectly:

- ***ldftn* <*token*> (0xFE 0x06)** Load the function pointer to the method specified by <*token*> of *MethodDef* or *MemberRef* type. The method is looked up in the class's method table.
- ***ldvirtftn* <*token*> (0xFE 0x07)** Pop the object reference (the instance pointer) from the stack and load the function pointer to the method specified by <*token*>. The method is looked up in the instance's v-table.
- ***calli* <*token*> (0x29)** Pop the function pointer from the stack, pop all the arguments from the stack, and make an indirect method call according to the method signature specified by <*token*>. <*token*> must be a valid *StandAloneSig* token. The function pointer must be on the top of the stack. If the method returns a value, it is pushed on the stack at the completion of the call. The *calli* instruction is unverifiable, which is not surprising, considering that the call is made via an unmanaged pointer, which is itself unverifiable.

It's easy enough to see that the combination *ldftn*/*calli* is equivalent to *call*, as long as we don't consider verifiability, and the combination *ldvirtftn*/*calli* is equivalent to *callvirt*.

The ILAsm notation requires full specification of the method in the *ldftn* and *ldvirtftn* instructions, similar to the *call* and *callvirt* instructions. The method signature accompanying the *calli* instruction is specified as *<call_conv> <ret_type>(<arg_list>)*. For example:

```
.locals init (native int fnptr)
⋮
ldftn void [mscorlib]System.Console::WriteLine(int32)
stloc.0 // Store function pointer in local variable
⋮
ldc.i4 12345 // Load argument
ldloc.0 // Load function pointer
calli void(int32)
⋮
```

Tail Calls

Tail calls are similar to method jumps (*jmp*) in the sense that both lead to abandoning the current method, discarding its stack frame, and passing the arguments to the tail-called (jumped-at) method. However, because the arguments of a tail call have to be loaded on the evaluation stack explicitly, a tail call—unlike a jump—does not require the entire signature of the called method to match the signature of the calling method; only the return types must be the same or compatible. In short, a jump is the equivalent of loading all the current method's arguments on the stack and invoking a tail call.

Tail calls are distinguished by the prefix instruction *tail.* immediately preceding a *call*, *callvirt*, or *calli* instruction:

- ***tail.* (0xFE 0x14)** Mark the following call instruction as a tail call. This instruction has no parameters and does not work with the stack. In ILAsm, this instruction—like the other prefix instructions *unaligned.* and *volatile.*, discussed earlier—must be separated from the call instruction that follows it by at least a space symbol.

The difference between a jump and a tail call is that the tail call instruction pair is verifiable in principle, subject to the verifiability of the call arguments, as long as it is immediately followed by the *ret* instruction. As is the case with other prefix instructions, it is illegal to bypass the prefix and branch directly to the prefixed instruction, in this case, *call*, *callvirt*, or *calli*.

Addressing Classes and Value Types

Being object-oriented in its base, IL offers quite a few instructions dedicated specifically to manipulating class and value type instances:

- ***ldnull* (0x14)** Load a null object reference on the stack.
- ***ldobj* <*token*> (0x71)** Load an instance value type specified by <*token*> on the stack. This instruction pops from the stack the managed pointer to the value type instance to be loaded. <*token*> must be a valid *TypeDef* or *TypeRef* token. The name of the instruction is somewhat misleading, for it deals with value type instances rather than objects (class instances). The ILAsm notation requires full specification of the value type so that it can be resolved to the token. For example:

  ```
  ldobj [.module other.dll]Foo.Bar
  ```

- ***stobj* <*token*> (0x81)** Pop the value type value—no, that's not a typo—from the stack, pop the managed pointer to the value type instance from the stack, and store the value type value in the instance. <*token*> indicates the value type and must be a valid *TypeDef* or *TypeRef* token. The ILAsm notation is similar to that used for *ldobj*.
- ***ldstr* <*token*> (0x72)** Load a string reference on the stack. <*token*> is a token of a user-defined string, whose RID portion is actually an offset in the *#US* blob stream. This instruction performs a lot of work: by the token, the Unicode string is retrieved from the *#US* stream, an instance of the *[mscorlib]System.String* class is created on the base of the retrieved string, and the object reference is pushed on the stack. In ILAsm, the string is specified explicitly either as a composite quoted string:

  ```
  ldstr "Hello"+" World!"
  ```

  or as a byte array:

  ```
  ldstr bytearray(A1 00 A2 00 A3 00 A4 00 A5 00 00 00)
  ```

  In the first case, at compile time the composite quoted string is converted to Unicode before being stored in the *#US* stream. In the second case, the byte array is stored "as is" without conversion. It can be padded with one 0 byte to make the byte count even. Storing a string in the *#US* stream gives the compiler the string token, which it puts into the IL stream.

- ***cpobj* <*token*> (0x70)** Copy the value of one value type instance to another instance. This instruction pops the source and the target instance pointers and pushes nothing on the stack. Both instances must be of the value type specified by <*token*>, either a *TypeDef* or a *TypeRef* token. The ILAsm notation for this instruction is similar to that used for *ldobj* or *stobj*.

- ***newobj <token>* (0x73)** Allocate memory for a new instance of a class—not a value type—and call the instance constructor method specified by <*token*>. This instruction pushes the object reference on the stack. <*token*> must be a valid *MethodDef* or *MemberRef* token. The instruction pops from the stack all the arguments explicitly specified in the constructor signature but does not pop the instance pointer. (No instance exists yet; it's being created.) The *newobj* instruction is also used for array creation:

  ```
  newobj instance void [mscorlib]System.Object::.ctor( )
  newobj instance void int32[0...,0... ]::.ctor(int32, int32)
  ```

  An array conmstructor takes as many parameters as there are undefined lower bounds and sizes of the array being created. (And hence the same number of integer values must be loaded on the stack before *newobj* is invoked.) In the example just shown, both lower bounds of the two-dimensional array are specified in the array type, so we need to specify only two sizes.

- ***initobj <token>* (0xFE 0x15)** Initialize the value type instance. This instruction takes an instance pointer—a managed pointer to a value type instance—from the stack. <*token*> specifies the value type and must be a valid *TypeDef* or *TypeRef* token. The *initobj* instruction zeroes all the fields of the value type instance, so if you need more sophisticated initialization, you might want to define *.ctor* or *.cctor* and call it instead.

- ***castclass <token>* (0x74)** Cast a class instance to the class specified by <*token*>. This instruction takes the object reference to the original instance from the stack and pushes the object reference to the cast instance on the stack. <*token*> must be a valid *TypeDef* or *TypeRef* token.

- ***isinst <token>* (0x75)** Check to see whether the object reference on the stack is an instance of the class specified by <*token*>. <*token*> must be a valid *TypeDef* or *TypeRef* token. This instruction pops the object reference from the stack and pushes the result on the stack. If the check succeeds, the result is an object reference, as if *castclass* had been invoked; otherwise, it is a null reference, as if *ldnull* had been invoked. The check succeeds under the following conditions:

  - If <*token*> indicates a class and the object reference on the stack is an instance of this class or of any class derived from it

  - ❑ If <*token*> indicates an interface and the object reference is an instance of the class implementing this interface
  - ❑ If <*token*> indicates a value type and the object reference is a boxed instance of this value type

- ***box <token>* (0x8C)** Convert a value type instance to an object reference. <*token*> specifies the value type being converted and must be a valid *TypeDef* or *TypeRef* token. This instruction pops the value type instance from the stack, creates a new instance of the type as an object, and pushes the object reference to this instance on the stack.
- ***unbox <token>* (0x79)** Revert a boxed value type instance from the object form to its value type form. <*token*> specifies the value type being converted and must be a valid *TypeDef* or *TypeRef* token. This instruction takes an object reference from the stack and puts a managed pointer to the value type instance on the stack.
- ***mkrefany <token>* (0xC6)** Pop a pointer—either managed or unmanaged—from the stack and convert it to a typed reference (*typedref*). The typed reference is an opaque handle that carries both type information and an instance pointer. The type of the created *typedref* is specified by <*token*>, which must be a valid *TypeDef* or *TypeRef* token. Typically, this instruction is used to create the *typedref* values to be passed as arguments to methods that expect *typedref* parameters. These methods split the typed references into type information and instance pointers using the *refanytype* and *refanyval* instructions.
- ***refanytype* (0xFE 0x1D)** Pop a typed reference from the stack, retrieve the type information, and push the [internal] type [handle] on the stack. This instruction has no parameters.
- ***refanyval* (0xC2)** Pop a typed reference from the stack, retrieve the instance pointer (*&* or *native int*), and push it on the stack. This instruction has no parameters.
- ***ldtoken < token>* (0xD0)** Convert <*token*> to an internal metadata handle to be used in calls to the *[mscorlib]System.Reflection* methods in the .NET Framework class library. The admissible token types are *MethodDef*, *MemberRef*, *TypeDef*, *TypeRef*, and *FieldDef*. The handle pushed on the stack is an instance of one of the following value types: *[mscorlib]System.RuntimeMethodHandle*, *[mscorlib]System.RuntimeTypeHandle*, or *[mscorlib]System.RuntimeFieldHandle*.

  The ILAsm notation requires full specification for classes (value types), methods, and fields used in *ldtoken*. This instruction is the only IL instruction that is not specific to methods only or fields only, and thus the keyword *method* or *field* must be used:

```
ldtoken [mscorlib]System.String
ldtoken method instance void [mscorlib]System.Object::.ctor(
 )
ldtoken field int32 Foo.Bar::ff
```

- ***sizeof <token>* (0xFE 0x1C)** Load the size in bytes of the value type specified by *<token>* on the stack. *<token>* must be a valid *TypeDef* or *TypeRef* token.
- ***throw* (0x7A)** Pop the object reference from the stack and throw it as a managed exception. See Chapter 11 for details about structured exception handling.
- ***rethrow* (0xFE 0x1A)** Throw the caught exception again. This instruction can be used exclusively within exception handlers.

Vector Instructions

Arrays and vectors are the only true generics implemented in the first release of the common language runtime. Vectors are "elementary" arrays, with one dimension and a zero lower bound. In signatures, vectors are represented by type *ELEMENT_TYPE_SZARRAY*, whereas "true" arrays are represented by *ELEMENT_TYPE_ARRAY*. We can, of course, declare a single-dimensional, zero-lower-bound *array* (whose ILAsm notation is *<type>[0...]*), which will be a true array, as opposed to a vector (whose ILAsm notation is *<type>[ ]*).

The IL instruction set defines specific instructions dealing with vectors but not with arrays. To handle array elements and arrays themselves, you need to call the methods of the .NET Framework class *[mscorlib]System.Array*, from which all arrays are derived.

Vector Creation

In order to work with a vector, it is necessary to create one. The IL instruction set contains special instructions for vector creation and vector length querying:

- ***newarr <token>* (0x8D)** Create a vector. *<token>* specifies the type of vector elements and must be a valid *TypeDef*, *TypeRef*, or *TypeSpec* token. This instruction pops the number of vector elements (*native int*) from the stack and pushes an object reference to the created vector on the stack. If the operation fails, an *OutOfMemory* exception is thrown. If the number of elements happens to be negative, an *Overflow* exception is thrown. The elements of the newly created vector are zero-initialized. Because *newarr* takes a token as a parameter, in ILAsm the full class names must be used for elementary types rather than the respective keywords. For example:

```
.locals init (int32[] arr)
ldc.i4 123
newarr [mscorlib]System.Int32
stloc.0
```

For specific details about array creation, see the description of the *newobj* instruction.

- ***ldlen* (0x8E)** Get the element count of a vector. This instruction takes an object reference to the vector instance from the stack and puts the element count (*native int*) on the stack.

Element Address Loading

You can obtain the managed pointer to a single vector element by using the following instruction:

- ***ldelema <token>* (0x8F)** Get the address (a managed pointer) of a vector element. *<token>* specifies the type of the element and must be a valid *TypeDef*, *TypeRef*, or *TypeSpec* token. This instruction pops the element index (*native int*) and the vector reference (an object reference) from the stack and pushes the managed pointer to the element on the stack. To get an address of an array element, the *System.Array* class provides the *Address* method.

Element Loading

Element loading instructions load a vector element of an elementary type on the stack. All of these instructions take the element index (*native int*) and the vector reference (an object reference) from the stack and put the value of the element on the stack. If the vector reference is null, the instructions throw a *NullReference* exception. If the index is negative or greater than or equal to the element count of the vector, an *IndexOutOfRange* exception is thrown. If the type of the vector element does not correspond to the type of the instruction, a *TypeMismatch* exception is thrown.

- ***ldelem.i1* (0x90)** Load a vector element of type *int8*.

- ***ldelem.u1*** **(0x91)** Load a vector element of type *unsigned int8*.
- ***ldelem.i2*** **(0x92)** Load a vector element of type *int16*.
- ***ldelem.u2*** **(0x93)** Load a vector element of type *unsigned int16*.
- ***ldelem.i4*** **(0x94)** Load a vector element of type *int32*.
- ***ldelem.u4*** **(0x95)** Load a vector element of type *unsigned int32*.
- ***ldelem.i8*** **(*ldelem.u8*) (0x96)** Load a vector element of type *int64*.
- ***ldelem.i*** **(0x97)** Load a vector element of type *native int*.
- ***ldelem.r4*** **(0x98)** Load a vector element of type *float32*.
- ***ldelem.r8*** **(0x99)** Load a vector element of type *float64*.
- ***ldelem.ref*** **(0x9A)** Load a vector element of object reference type.

Element Storing

Element storing instructions store a value from the stack in a vector element of an elementary type. All of these instructions take the value to be stored, the element index (*native int*), and the vector reference (an object reference) from the stack and put nothing on the stack. Generally, the instructions can throw the same exceptions as the *ldelem.** instructions described in the preceding section.

- ***stelem.i*** **(0x9B)** Store a value in a vector element of type *native int*.
- ***stelem.i1*** **(0x9C)** Store a value in a vector element of type *int8*.
- ***stelem.i2*** **(0x9D)** Store a value in a vector element of type *int16*.
- ***stelem.i4*** **(0x9E)** Store a value in a vector element of type *int32*.
- ***stelem.i8*** **(0x9F)** Store a value in a vector element of type *int64*.
- ***stelem.r4*** **(0xA0)** Store a value in a vector element of type *float32*.
- ***stelem.r8*** **(0xA1)** Store a value in a vector element of type *float64*.
- ***stelem.ref*** **(0xA2)** Store a value in a vector element of object reference type. This instruction involves casting of the object on the stack to the type of the vector element, so an *InvalidCast* exception can be thrown.

Special *stelem.** instructions for unsigned integer types are missing for an obvious reason: the *stelem.i** instructions are equally applicable to signed and unsigned integer types.

Code Verifiability

The verification algorithm associates IL instructions with valid evaluation stack states and, first of all, with the number of stack slots occupied and available at each moment. Stack overflows and underflows render the code not only unverifiable but invalid as well. The verification algorithm also presumes all local variables are zero-initialized before the method execution begins. As a result, the *.locals* directive—at least one, if several of these are used throughout the method—must have the *init* clause in order for the method to be verifiable.

The verification algorithm simulates all possible control flow paths and branchings, checking to see whether legal stack states correspond to every reachable instruction. It is impossible, of course, to predict the actual values stored on the evaluation stack at every moment, but the number of stack slots occupied and the types of the slots can be assessed.

As mentioned, the evaluation stack type system is coarser than the metadata type system used for field, argument, and local variable types. Hence, the type validity of instructions transferring data between the stack and other typed memory categories depends on the type conversion performed during such transfers. Table 10-5 lists type conversions between different type systems.

Table 10-5 Evaluation Stack Type Conversions

Metadata Type	**Stack Type**	**Managed Pointer to Type**
[unsigned] int8, bool	*int32*	*int8&*
[unsigned] int16, char	*int32*	*int16&*
[unsigned] int32	*int32*	*int32&*
[unsigned] int64	*int64*	*int64&*
native [unsigned] int, function pointer	*native int*	*native int&*
float32	*Float*	*float32&*
float64	*Float*	*float64&*
Value type	Same type (see substitution rules in this section)	Same type&
Object	Same type (see substitution rules in this section)	Same type&

According to verification rules, type A can be replaced with type B in the following cases only:

- If A is a class and B is the same class or any class derived from A
- If A an interface and B is a class implementing this interface
- If both A and B are interfaces and the implementation of B requires the implementation of A
- If A is a class or an interface and B is a null reference
- If both A and B are vectors and their element types can be respectively substituted
- If both A and B are arrays of the same rank and their element types can be respectively substituted
- If both A and B are function pointers and the signatures of their respective methods match

These substitution rules set the limits of "type leeway" allowed for the IL code to remain verifiable. As the verification algorithm proceeds from one instruction to another along every possible path, it checks the simulated stack types against the types expected by the next instruction. Failure to comply with the substitution rules results in verification failure and possibly indicates invalid IL code.

A few verification rules, rather heuristic than formal, are based on the question, "Is it possible in principle to do something unpredictable using this construct?":

- Any code containing embedded native code is unverifiable.
- Any code using unmanaged pointers is unverifiable.
- Any code containing calls to methods that return managed pointers is unverifiable. The reason: theoretically, the called method might return a managed pointer to one of its local variables or another "perishable" item.
- An instance constructor of a class must call the instance constructor of the base class. The reason: until the base class *.ctor* is called, the instance pointer (*this*) is considered uninitialized; and until *this* is initialized, no instance methods should be called.
- When a delegate is being instantiated—its constructor takes a function pointer as the last argument—the *newobj* instruction must be immediately preceded by the *ldftn* or *ldvirtftn* instruction, which loads the function pointer. If anything appears between these two instructions, the code becomes unverifiable.

A great many additional rules regulate structured exception handling, but the place to discuss them is the next chapter.

11

Structured Exception Handling

Usually the exception handling model of a programming language is considered the domain of that particular language's runtime. Under the hood, each language has its own way of detecting exceptions and locating an appropriate exception handler. Some languages perform exception handling completely within the language runtime, whereas others rely on the structured exception handling (SEH) mechanism provided by the operating system—which in our case is Win32.

In the world of managed code, exception handling is a fundamental feature of the common language runtime execution engine. The execution engine is fully capable of handling exceptions without regard to language, allowing exceptions to be raised in one language and caught in another. At that, the runtime does not dictate any particular syntax for handling exceptions. The exception mechanism is language-neutral in that it is equally efficient for all languages.

No special metadata is captured for exceptions other than the metadata for the exception classes themselves. No association exists between a method of a class and the exceptions that the method might throw. Any method is permitted to throw any exception at any time.

Although we talk about managed exceptions thrown and caught within managed code, a common scenario involves a mix of both managed and unmanaged code. Execution threads routinely traverse managed and unmanaged blocks of code through the use of the common language runtime's platform invocation mechanism (*P/Invoke*) and other interoperability mechanisms. (See Chapter 15, "Managed and Unmanaged Code Interoperation.") Consequently, during execution, exceptions can be thrown or caught in either managed or unmanaged code.

The runtime exception handling mechanism integrates seamlessly with the Win32 SEH mechanism so that exceptions can be thrown and caught within and between the two exception handling systems.

SEH Clause Internal Representation

Structured exception handling tables are located immediately after a method's IL code, with the beginning of the table aligned on a double word boundary. It would be more accurate to say that "additional sections" are located after the method IL code, but the first release of the common language runtime allows only one kind of additional section—the exception handling section.

This additional section begins with the section header, which contains two entries, *Kind* and *DataSize*. In a *small* header, *DataSize* is represented by 1 byte, whereas in a *fat* header, *DataSize* is 3 bytes long. A *Kind* entry can contain the following binary flags:

- ***Reserved* (0x00)**
- ***EHTable* (0x01)** The section contains an exception handling table. This bit must be set.
- ***OptILTable* (0x02)** Not used in the first release of the runtime. This bit must not be set.
- ***FatFormat* (0x40)** The section header has a fat format—that is, *DataSize* is represented by 3 bytes.
- ***MoreSects* (0x80)** More sections follow this one.

The section header—padded with 2 bytes if small—is followed by a sequence of exception handling (EH) clauses, which can also have small or fat format. Each EH clause describes a single triad made up of a guarded block, an exception identification, and an exception handler. The entries of small and fat EH clauses have the same names and meanings but different sizes, as shown in Table 11-1.

Table 11-1 EH Clause Entries

EH Clause Entry	Size in Small Clause (bytes)	Size in Fat Clause (bytes)	Description
Flags	2	4	Binary flags specifying the type of the EH clause, which is the type of the exception identification method.
TryOffset	2	4	Offset, in bytes, of the beginning of the guarded code block from the beginning of the method IL code. The guarded block can begin only at code points where the evaluation stack is empty.
TryLength	1	4	Length, in bytes, of the guarded block.
HandlerOffset	2	4	Offset of the exception handler block.
HandlerLength	1	4	Length of the exception handler block.
ClassToken/FilterOffset	4	4	Exception type token or offset of the exception filtering block, depending on the type of the EH clause.

Branching into or out of guarded blocks and handler blocks is illegal. A guarded block must be entered "through the top"—that is, through the instruction located at *TryOffset*—and handler blocks are entered only when they are engaged by the exception handling subsystem of the execution engine. To exit guarded and handler blocks, you must use the instruction *leave* (or *leave.s*). You might recall that in Chapter 2, "Enhancing the Code," this principle was formulated as "leave only by *leave*." Another way to leave any block is to throw an exception using the *throw* or *rethrow* instruction.

Types of SEH Clauses

Exception handling clauses are classified by the algorithm of the handler engagement. Four mutually exclusive EH clause types are available, and because of that the *Flags* entry must hold one of the following values:

- **0x0000** The handler must be engaged if the type of the exception object matches the type identified by the token specified in the *ClassToken* entry or any of its descendants. Theoretically, any object can be thrown as an exception, but it's strongly recommended that all exception types be derived from the *[mscorlib]System.Exception* class. This is because throughout Microsoft .NET Framework classes the construct *catch [mscorlib]System.Exception* is used in the sense of "catch any exception"—it is an analog of *catch(...)* in C++. In other words, *[mscorlib]System.Exception* is presumed to be the ultimate base class for all exceptions. This type of EH clause is called a *catch* type.
- **0x0001** A dedicated block of the IL code, called filter, will process the exception and define whether the handler should be engaged. The offset of the filter block is specified in the *FilterOffset* entry. Since we cannot specify the filter block length—the EH clause structure contains no entry for it—a positioning limitation is associated with the filter block: the respective handler block must immediately follow the filter block, allowing the length of the filter block to be inferred from the offset of the handler. The filter block must end with the *endfilter* instruction, described in Chapter 10, "IL Instructions." At the moment *endfilter* is invoked, the evaluation stack must hold a single *int32* value, equal to 1 if the handler is to be engaged and equal to 0 otherwise. This EH clause type is called a *filter* type. Branching into or out of the filter block is illegal.
- **0x0002** The handler will be engaged whether or not an exception has occurred. The EH clause entry *ClassToken/FilterOffset* is ignored. This EH clause type is called a *finally* type. The *finally* handlers are not meant to process an exception but rather to perform any cleanup that might be needed when leaving the guarded block. The *finally* handlers must end with the *endfinally* instruction. If no exception has occurred within the guarded block, the *finally* handler is executed at the moment of leaving that block. If an exception has been thrown within the guarded block, the *finally* handler is executed after any preceding handler is executed or, if no preceding handler was engaged, before any following handler is executed. If no *catch*

or *filter* handlers are engaged—that is, the exception is uncaught—the *finally* handler has no chance to be engaged either, because it does not catch exceptions by itself.

Figure 11-1 illustrates this process. If an exception of type A is thrown within the guarded block, it is caught and processed by the first handler (*catch* A), and the *finally* handler is engaged when the first handler invokes the *leave* instruction. If an exception of type B is thrown, it is caught by the third handler (*catch* B), and the *finally* handler is executed before the third handler. If no exception is thrown within the guarded block, the *finally* handler is engaged when the guarded block invokes the *leave* instruction.

- **0x0004** The handler will be engaged if any exception occurs. This type of EH clause is called a *fault* type. A *fault* handler is similar to a *finally* handler in all aspects except one: the *fault* handler is not engaged if no exception has been thrown within the guarded block and everything is nice and quiet. The *fault* handler must also end with the *endfinally* instruction, which for this specific purpose has been given the synonym *endfault*.

Label Form of SEH Clause Declaration

The most generic form of IL assembly language (ILAsm) notation of an EH clause is as follows:

```
.try <label> to <label> <EH_type_specific> handler <label> to <label>
```

where *<EH_type_specific>* ::=

```
   catch <class_ref> |
   filter <label> |
   finally |
   fault
```

Take a look at this example:

```
BeginTry:
   ⋮
   leave KeepGoing
BeginHandler:
   ⋮
   leave KeepGoing
KeepGoing:
   ⋮
   ret
   .try BeginTry to BeginHandler catch [mscorlib]System.Exception
      handler BeginHandler to KeepGoing
```

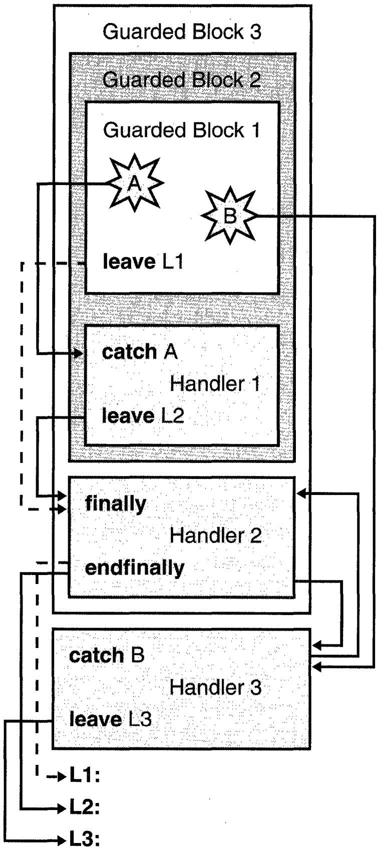


Figure 11-1 Engagement of the *finally* exception handler.

In the final lines of the example, the code *.try <label> to <label>* defines the guarded block, and *handler <label> to <label>* defines the handler block. In both cases, the second *<label>* is exclusive, pointing at the first instruction after the respective block. ILAsm imposes a limitation on the positioning of the EH clause declaration directives: all labels used in the directives must have already been defined. Thus, the best place for EH clause declarations in the label form is at the end of the method scope.

In the case just presented, the handler block immediately follows the guarded block, but we could put the handler block anywhere within the method, provided it does not overlap with the guarded block or other handlers:

```
   ⋮
   br AfterHandler // Can't enter the handler block on our own
BeginHandler:
   ⋮
   leave KeepGoing
AfterHandler:
   ⋮
BeginTry:
   ⋮
   leave KeepGoing
KeepGoing:
   ⋮
   ret
   .try BeginTry to KeepGoing catch [mscorlib]System.Exception
      handler BeginHandler to AfterHandler
```

A single guarded block can have several handlers:

```
   ⋮
   br AfterHandler2 // Can't enter the handler block(s) on our own
BeginHandler1:
   ⋮
   leave KeepGoing
AfterHandler1:
   ⋮
BeginHandler2:
   ⋮
   leave KeepGoing
AfterHandler2:
   ⋮
BeginTry:
   ⋮
   leave KeepGoing
KeepGoing:
   ⋮
   ret
   .try BeginTry to KeepGoing
      catch [mscorlib]System.StackOverflowException
         handler BeginHandler1 to AfterHandler1
   .try BeginTry to KeepGoing catch [mscorlib]System.Exception
      handler BeginHandler2 to AfterHandler2
```

In the case of multiple handlers—*catch* or *filter*, but not *finally* or *fault*—the guarded block declaration need not be repeated:

```
   .try BeginTry to KeepGoing
      catch [mscorlib]System.StackOverflowException
         handler BeginHandler1 to AfterHandler1
      catch [mscorlib]System.Exception
         handler BeginHandler2 to AfterHandler2
```

The lexical order of handlers belonging to the same guarded block is the order in which the ILAsm compiler emits the EH clauses, and hence is the same order in which the execution engine of the runtime processes these clauses. We must be careful about ordering the handlers. For instance, if we swap the handlers in the preceding example, the handler for *[mscorlib]System.Exception* will always work and the handler for *[mscorlib]System.StackOverflowException* will never work. This is because all exceptions are derived, eventually, from *[mscorlib]System.Exception*, and hence all exceptions are caught by the first handler, leaving the other handlers unemployed.

The *finally* and *fault* handlers cannot peacefully coexist with other handlers, so if a guarded block has a *finally* or *fault* handler, it cannot have anything else. To combine a *finally* or *fault* handler with other handlers, we need to nest the guarded and handler blocks within other guarded blocks, as shown in Figure 11-1, so that each *finally* or *fault* handler has its own personal guarded block.

Scope Form of SEH Clause Declaration

The label form of the EH clause declaration is universal, ubiquitous, and close to the actual representation of the EH clauses in the EH table. The only quality the label form lacks is convenience. In view of that, ILAsm offers an alternative form of EH clause description: a scope form. You've already encountered the scope form in Chapter 2, which discussed protecting the code against possible surprises in the unmanaged code being invoked. Just to remind you, here's what the protected part of the method (from the sample file Simple2.il on the companion CD) looks like:

```
  ⋮
  .try {
      // Guarded block begins
      call string [mscorlib]System.Console::ReadLine()
      // pop
      // ldnull
      ldstr "%d"
      ldsflda int32 Odd.or.Even::val
      call vararg int32 sscanf(string,string,...,int32*)
      stloc.0
      leave.s DidntBlowUp
      // Guarded block ends
  }
  catch [mscorlib]System.Exception
    { // Exception handler begins
      pop
      ldstr "KABOOM!"
      call void [mscorlib]System.Console::WriteLine(string)
      leave.s Return
  } // Exception handler ends
DidntBlowUp:
  ⋮
```

The scope form can be used only for a limited subset of all possible EH clause configurations: the handler blocks must immediately follow the previous handler block or the guarded block. If the EH clause configuration is different, we must resort to the label form or a mixed form:

```
   ⋮
   br AfterHandler
HandlerBegins:
   // The exception handler code
   ⋮
   leave KeepGoing
AfterHandler:
   ⋮
   .try {
      // Guarded code
      ⋮
      leave KeepGoing
   }
   catch [mscorlib]System.Exception
      handler HandlerBegins to AfterHandler
   ⋮
KeepGoing:
   ⋮
```

The IL Disassembler by default outputs the EH clauses in the scope form—at least those clauses that can be represented in this form. However, we have the option to suppress the scope form and output all EH clauses in their generic label form. But let's suppose for the sake of convenience that we can shape the code in such a way that the contiguity condition is satisfied, allowing us to use the scope form.

A single guarded block with multiple handlers in scope form will look like this:

```
.try {
   // Guarded code
   ⋮
   leave KeepGoing
}
catch [mscorlib]System.StackOverflowException {
   // The exception handler #1 code
   ⋮
   leave KeepGoing
}
catch [mscorlib]System.Exception {
   // The exception handler #2 code
   ⋮
      leave KeepGoing
}
   ⋮
KeepGoing:
   ⋮
```

Much more readable, isn't it? The nested EH configuration shown earlier in Figure 11-1 is easily understandable when written in scope form:

```
.try {
   .try {
      .try {
         // Guarded code
         ⋮
         leave L1
      }
      catch A {
         // This code works when exception A is thrown
         ⋮
         leave L2
      }
   } // No need for leave here!
   finally {
      // This code works in any case
      ⋮
      endfinally
   }
} // No need for leave here either!
catch B {
   // This code works when exception B is thrown in guarded code
   ⋮
   leave L3
}
```

The *filter* EH clauses in scope form are subject to the same limitation: the handler block must immediately follow the guarded block. But because in a *filter* clause the handler block includes first the filter block and then, immediately following it, the actual handler, the scope form of a *filter* clause looks like this:

```
.try {
   // Guarded code
   ⋮
   leave KeepGoing
}
filter {
   // Here we decide whether we should invoke the actual handler
   ⋮
   ldc.i4.1 // OK, let's invoke the handler
   endfilter
} {
   // Actual handler code
   ⋮
   leave KeepGoing
}
```

And, of course, we can easily switch between scope form and label form within a single EH clause declaration. The general ILAsm syntax for an EH clause declaration is as follows:

```
<EH_clause> ::= .try <guarded_block>
                     <EH_type_specific> <handler_block>
Where
<guarded_block> ::= <label> to <label> | <scope>
<EH_type_specific> ::= catch <class_ref> |
        filter <label> | filter <scope> |
        finally |
        fault
<handler_block> ::= handler <label> to <label> | <scope>
```

The nonterminals *<label>* and *<class_ref>* must be familiar by now, and the meaning of *<scope>* is obvious: "code enclosed in curly braces."

Processing the Exceptions

The execution engine of the runtime processes an exception in two passes. The first pass determines which, if any, of the managed handlers will process the exception. Starting at the top of the EH table for the current method frame, the execution engine compares the address where the exception occurred to the *TryOffset* and *TryLength* entries of each EH clause. If it finds that the exception happened in a guarded block, the execution engine checks to see whether the handler specified in this clause will process the exception. (The "rules of engagement" for *catch* and *filter* handlers were discussed in previous sections.) If this particular handler can't be engaged—for example, the wrong type of exception has been thrown—the execution engine continues traversing the EH table in search of other clauses that have guarded blocks covering the exception locus. The *finally* and *fault* handlers are ignored during the first pass.

If none of the clauses in the EH table for the current method are suited to handle the exception, the execution engine steps up the call stack and starts checking the exception against EH tables of the method that called the method where the exception occurred. In these checks, the call site address serves as the exception locus. This process continues from method frame to method frame up the call stack, until the execution engine finds a handler to be engaged or until it exhausts the call stack. The latter case is the end of the story: the execution engine cannot continue with an unhandled exception on its conscience, and the runtime either aborts the application execution or offers the user a choice between aborting the execution and invoking the debugger, depending on the runtime configuration.

If a suitable handler is found, the execution engine swings into the second pass. The execution engine again walks the EH tables it worked with during the first pass and invokes all relevant *finally* and *fault* handlers. Each of these handlers ends with the *endfinally* instruction (or *endfault*, its synonym), signaling the execution engine that the handler has finished and that it can proceed browsing the EH tables. Once the execution engine reaches the *catch* or *filter* handler it found on its first pass, it engages the actual handler.

What happens to the method's evaluation stack? When a guarded block is exited in any way, the evaluation stack is discarded. If the guarded block is exited peacefully, without raising an exception, the *leave* instruction discards the stack; otherwise, the evaluation stack is discarded the moment the exception is thrown.

During the first pass, the execution engine puts the exception object on the evaluation stack every time it invokes a filter block. The filter block pops the exception object from the stack and analyzes it, deciding whether this is a job for its actual handler block. The decision, in the form of *int32* having the value 1 or 0, is the only thing that must be on the evaluation stack when the *endfilter* instruction is reached; otherwise, the IL verification will fail. The *endfilter* instruction takes this value from the stack and passes it to the execution engine.

During the second pass, the *finally* and *fault* handlers are invoked with an empty evaluation stack. Because these handlers do nothing about the exception itself and work only with method arguments and local variables, the execution engine doesn't bother providing the exception object. If anything is left on the evaluation stack by the time the *endfinally* (or *endfault*) instruction is reached, it is discarded by *endfinally* (or *endfault*).

When the actual handler is invoked, the execution engine puts the exception object on the evaluation stack. The handler pops this object from the stack and handles it to the best of its abilities. When the handler is exited by using the *leave* instruction, the evaluation stack is discarded.

Table 11-2 summarizes the stack evolutions.

Table 11-2 Changes in the Evaluation Stack

When the block	is entered, the stack...	is exited, the stack...
try	must be empty	is discarded
filter	holds the exception object	must hold a single *int32* value, equal to 1 or 0, consumed by *endfilter*
handler	holds the exception object	is discarded
finally, fault	is empty	is discarded

Two IL instructions are used for raising an exception explicitly: *throw* and *rethrow*. The *throw* instruction takes the exception object (*ObjectRef*) from the stack and raises the exception. This instruction can be used anywhere, within or outside any EH block.

The *rethrow* instruction can be used within actual handlers only (*not* within the filter block), and it does not work with the evaluation stack. This instruction signals the execution engine that the handler that was supposed to take care of the caught exception has for some reason changed its mind and that the exception should therefore be offered to the higher-level EH clauses. The only blocks where the words "caught exception" mean something are the actual handler block and the filter block, but invoking *rethrow* within a filter block, though theoretically possible, is illegal. It is legal to throw the caught exception from the filter block, but it doesn't make much sense to do so: the effect is the same as if the filter simply refused to handle the exception, by loading 0 on the stack and invoking *endfilter*.

Rethrowing an exception is not the same as throwing the caught exception, which we have on the evaluation stack upon entering an actual handler. The *rethrow* instruction preserves the call stack trace of the original exception so that the exception can be tracked down to its point of origin. The *throw* instruction starts the call stack trace anew, giving us no way to determine where the original exception came from.

Exception Types

Chapter 10 mentioned some of the exception types that can be thrown during the execution of IL instructions. Earlier chapters mentioned some of the exceptions thrown by the loader and the JIT (just-in-time) compiler. Now it's time to review all these exceptions in an orderly manner.

All managed exceptions defined in the .NET Framework classes are descendants of the *[mscorlib]System.Exception* class. This base exception type, however, is never thrown by the common language runtime. In the following sections, I've listed the exceptions the runtime *does* throw, classifying them by major runtime subsystems. Enjoying the monotonous repetition no more than you do, I've omitted the *[mscorlib]System.* part of the names, common to all exception types. As you can see, many of the exception type names are self-explanatory.

Loader Exceptions

The loader represents the first line of defense against erroneous applications, and the exceptions it throws are related to the file presence and integrity.

- ***AppDomainUnloadedException***
- ***CannotUnloadAppDomainException***

- ***BadImageFormatException*** Corrupt file headers or segments that belong in read-only sections (such as the runtime header, metadata, and IL code) are located in writeable sections of the PE file.
- ***ArgumentException*** This exception is also thrown by the JIT compiler and the interoperability services.
- ***Security.Cryptography.CryptographicException***
- ***FileLoadException***
- ***MissingFieldException***
- ***MissingMethodException***
- ***TypeLoadException*** This exception, which is most frequently thrown by the loader, indicates that the type metadata is illegal.
- ***UnauthorizedAccessException*** A user application is attempting to directly manipulate the system assembly Mscorlib.dll.
- ***OutOfMemoryException*** This exception, which is also thrown by the execution engine, indicates memory allocation failure.

JIT Compiler Exceptions

The JIT compiler throws only two exceptions. The second one can be thrown only when the security services are engaged.

- ***InvalidProgramException*** This exception, which is also thrown by the execution engine, indicates an error in IL code.
- ***VerificationException*** This exception, which is also thrown by the execution engine, indicates that IL code verification has failed.

Execution Engine Exceptions

The execution engine throws a wide variety of exceptions, most of them related to the operations on the evaluation stack. A few exceptions are thrown by the thread control subsystem of the engine.

- ***ArithmeticException***
- ***ArgumentOutOfRangeException***
- ***ArrayTypeMismatchException*** This exception is also thrown by the interoperability services.
- ***DivideByZeroException***
- ***DuplicateWaitObjectException***

- ***ExecutionEngineException*** This is the generic exception, indicating that some sequence of IL instructions has brought the execution engine into a state of complete perplexity—as a rule, by corrupting the memory. Verifiable code cannot corrupt the memory and hence does not raise exceptions of this type.
- ***FieldAccessException*** This exception indicates, for example, an attempt to load from or store to a private field of another class.
- ***FormatException***
- ***IndexOutOfRangeException***
- ***InvalidCastException***
- ***InvalidOperationException***
- ***MethodAccessException*** This exception indicates an attempt to call a method to which the caller does not have access—for example, a private method of another class.
- ***NotSupportedException***
- ***NullReferenceException*** This exception indicates an attempt to dereference a null pointer (a managed or unmanaged pointer, or an object reference).
- ***OverflowException***
- ***RankException*** This exception is thrown when a method specific to an array is being called on a vector instance.
- ***RemotingException***
- ***Security.SecurityException***
- ***StackOverflowException***
- ***Threading.SynchronizationLockException*** This exception is thrown when an application tries to manipulate or release a lock it has not acquired—for example, by calling the *Wait*, *Pulse*, or *Exit* method before calling the *Enter* method of the *[mscorlib]System.Threading.Monitor* class.
- ***Threading.ThreadAbortException***
- ***Threading.ThreadInterruptedException***
- ***Threading.ThreadStateException***
- ***Threading.ThreadStopException***
- ***TypeInitializationException*** This exception is thrown when a type—a class or a value type—failed to initialize.

Interoperability Exceptions

The following exceptions are thrown by the interoperability services of the common language runtime, which are responsible for managed and unmanaged code interoperation:

- ***DllNotFoundException*** This exception is thrown when an unmanaged DLL specified as a location of the unmanaged method being called cannot be found.
- ***ApplicationException***
- ***EntryPointNotFoundException***
- ***InvalidComObjectException***
- ***Runtime.InteropServices.InvalidOleVariantTypeException***
- ***MarshalDirectiveException*** This exception is thrown when data cannot be marshaled between managed and unmanaged code in the specified way.
- ***Runtime.InteropServices.SafeArrayRankMismatchException***
- ***Runtime.InteropServices.SafeArrayTypeMismatchException***
- ***Runtime.InteropServices.COMException***
- ***Runtime.InteropServices.SEHException*** This is the generic managed exception type for unmanaged exceptions.

Subclassing the Exceptions

In addition to the plethora of exception types already defined in the .NET Framework classes, you can always devise your own types tailored to your needs. The best way to do this is to derive your exception types from the "standard" types listed in the preceding sections.

The following exception types are sealed and can't be subclassed. Again, I've omitted the *[mscorlib]System.* portion of the names.

- ***InvalidProgramException***
- ***TypeInitializationException***
- ***Threading.ThreadAbortException***
- ***StackOverflowException***

Caution As mentioned earlier, I must warn you against devising your own exception types not derived from *[mscorlib]System.Exception* or some other exception type of the .NET Framework classes.

Unmanaged Exception Mapping

When an unmanaged Win32 exception occurs within a native code segment, the execution engine maps it to a managed exception that is thrown in its stead. The different types of unmanaged exceptions, identified by their status code, are mapped to the managed exceptions as described in Table 11-3.

Table 11-3 Mapping Between the Managed and Unmanaged Exceptions

Unmanaged Exception Status Code	Mapped to Managed Exception
STATUS_FLOAT_INEXACT_RESULT	*ArithmeticException*
STATUS_FLOAT_INVALID_OPERATION	*ArithmeticException*
STATUS_FLOAT_STACK_CHECK	*ArithmeticException*
STATUS_FLOAT_UNDERFLOW	*ArithmeticException*
STATUS_FLOAT_OVERFLOW	*OverflowException*
STATUS_INTEGER_OVERFLOW	*OverflowException*
STATUS_FLOAT_DIVIDE_BY_ZERO	*DivideByZeroException*
STATUS_INTEGER_DIVIDE_BY_ZERO	*DivideByZeroException*
STATUS_FLOAT_DENORMAL_OPERAND	*FormatException*
STATUS_ACCESS_VIOLATION	*NullReferenceException*
STATUS_ARRAY_BOUNDS_EXCEEDED	*IndexOutOfRangeException*
STATUS_NO_MEMORY	*OutOfMemoryException*
STATUS_STACK_OVERFLOW	*StackOverflowException*
All other status codes	*Runtime.InteropServices.SEHException*

SEH Clause Structuring Rules

The rules for structuring EH clauses within a method are neither numerous nor overly complex:

All the blocks—try, filter, handler, finally, and fault—of each EH clause must be fully contained within the method code. No block can protrude from the method.

Blocks belonging to the same EH clause or different EH clauses can't partially overlap. A block either is fully contained within another block or is completely outside it. If one guarded block (A) is contained within another guarded block (B) but is not equal to it, all handlers assigned to A must also be fully contained within B.

A handler block of an EH clause can't be contained within a guarded block of the same clause, and vice versa. Neither can a handler block be contained in another handler block that is assigned to the same guarded block.

A filter block can't contain any guarded blocks or handler blocks.

All blocks must start and end on instruction boundaries—that is, at offsets corresponding to the first byte of an instruction. Prefixed instructions must not be split, meaning that you can't have constructs such as *tail. .try { call ... }*.

A guarded block must start at a code point where the evaluation stack is empty.

The same handler block can't be associated with different guarded blocks:

```
.try Label1 to Label2 catch A handler Label3 to Label4
.try Label4 to Label5 catch B handler Label3 to Label4 // Illegal!
```

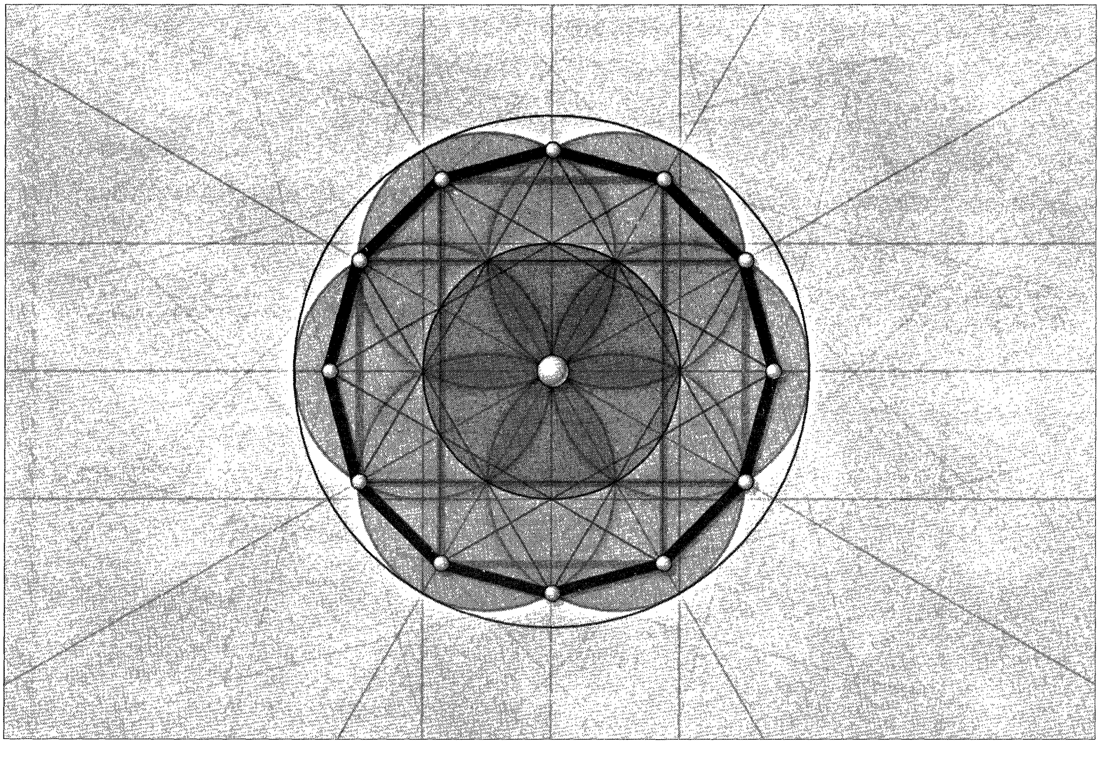
If the EH clause is a *filter* type, the filter's actual handler must immediately follow the filter block. Since the filter block must end with the *endfilter* instruction, this rule can be formulated as "the actual handler starts with the instruction after *endfilter*."

If a guarded block has a *finally* or *fault* handler, the same block can have no other handler. If you need other handlers, you must declare another guarded block, encompassing the original guarded block and the handler:

```
.try {
   .try {
      .try {
         // Code that needs finally, catch, and fault handlers
         ⋮
         leave KeepGoing
      }
      finally {
         ⋮
         endfinally
      }
   }
   catch [mscorlib]System.StackOverflowException
   {
      ⋮
      leave KeepGoing
   }
}
fault {
   ⋮
   endfault
}
```

Part V

Special Components



12

Events and Properties

Events and properties are special metadata components that are intended to make life easier for the high-level language compilers. The most intriguing feature of events and properties is that the JIT compiler and the execution engine are completely unaware of them. Can you recall any IL instruction that deals with an event or a property? That's because none exist.

To understand the indifference of the JIT compiler and the execution engine toward events and properties, you need to understand the way these items are implemented.

Events and Delegates

The managed events I'm talking about here are not synchronization elements, similar to Win32 event objects. Rather, they more closely resemble Microsoft Visual Basic events and *On<event>* functions. Managed events provide a means to describe asynchronous execution of methods, initiated by certain other methods.

The general sequence of activities is illustrated in Figure 12-1. A program unit—known as the *publisher,* or *source,* of an event—defines the event. We can think of this program unit as a class, for the sake of simplicity. Other program units (classes)—known as *subscribers*, or event *listeners*, or event *sinks*—define the methods to be executed when the event occurs and pass this information to the event publisher. When the event publisher raises, or *fires,* the event by calling a special method, all the subscriber's methods associated with this event are executed.

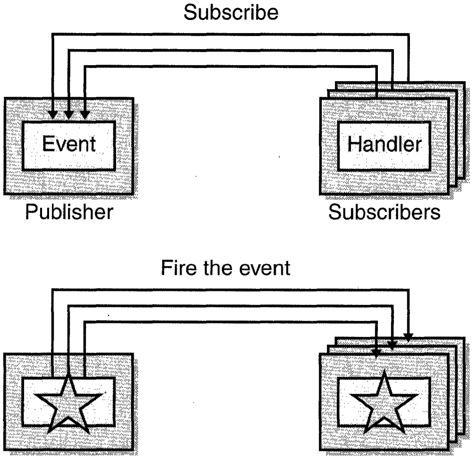


Figure 12-1 The interaction of an event publisher and subscribers.

In a nutshell, to implement a managed event we need an entity that can collect the callback methods (event handlers) from the event subscribers and execute these methods when the publisher executes a method that signifies the event.

We have a type (a class) designed to do exactly that: the delegate type, which is discussed in Chapter 6, "Namespaces and Classes." As you might remember, delegates are classes derived from the class *[mscorlib]System.MulticastDelegate* and play the role of "politically correct" function pointers in the managed world. The actual function pointer to a delegated method is passed to the delegate as an argument of its constructor, and the delegated method can subsequently be executed by calling the delegate's *Invoke* method.

What Chapter 6 doesn't mention is that several delegates can be aggregated into one delegate. Calling the *Invoke* method of such an aggregated delegate invokes all the delegated methods that make up the aggregate—which is exactly what we need to implement an event.

The *[mscorlib]System.MulticastDelegate* class defines the virtual methods *CombineImpl* and *RemoveImpl*, adding a delegate to the aggregate and removing a delegate from the aggregate, respectively. These methods are defined in Mscorlib.dll as follows. (I have omitted the resolution scope *[mscorlib]* of delegate types here because the methods are defined in the same assembly; this doesn't mean you can omit the resolution scope when you refer to these types in your assemblies.)

```
.method family hidebysig final virtual
        instance class System.Delegate
        CombineImpl(class System.Delegate follow) cil managed
{ ... }
.method family hidebysig final virtual
        instance class System.Delegate
        RemoveImpl(class System.Delegate 'value') cil managed
{ ... }
```

The methods take the object reference to the delegate being added (or removed) as the argument and return the object reference to the aggregated delegate. The parameter and return type of both methods is *System.Delegate* rather than *System.MulticastDelegate*, but this isn't contradictory: *System.MulticastDelegate* is derived from *System.Delegate* and hence can be used in its stead.

The principle of the delegate-based implementation of an event is more or less clear. Each event subscriber creates a delegate representing its event handler and then subscribes to the event by combining the handler delegate with the aggregate delegate, held by the event publisher. To raise the event, the publisher simply needs to call the *Invoke* method of the aggregate delegate, and everybody's happy.

One question remains, though: what does the publisher's aggregate delegate look like before any event subscriber has subscribed to the event? The answer is, it doesn't exist at all. The aggregate delegate is a result of combining the subscribers' handler delegates. As long as there are no subscribers, the publisher's aggregate delegate does not exist. This poses a certain problem: *CombineImpl* is an instance method, which has to be called on the instance of the aggregated delegate, and hence each subscriber must worry about whether it is the first in line (in other words, whether the aggregated delegate exists yet). That's why the subscribers usually use the static methods *Combine* and *Remove*, inherited by *System.MulticastDelegate* from *System.Delegate*:

```
.method public hidebysig static class System.Delegate
        Combine(class System.Delegate a,
                class System.Delegate b)
{ ... }
.method public hidebysig static class System.Delegate
        Remove(class System.Delegate source,
               class System.Delegate 'value')
{ ... }
```

If one of the arguments of these methods is a null reference, the methods simply return the non-null argument. If both arguments are null references, the methods return a null reference. If the arguments are incompatible—that is, if the delegated methods have different signatures—*Combine*, which internally calls *CombineImpl*, throws an *Argument* exception and *Remove*, which internally calls *RemoveImpl*, simply returns the aggregated delegate unchanged.

In general, delegates are fascinating types, with more features than this book can discuss. The best way to learn more about delegates first-hand is to disassemble Mscorlib.dll and have a look at how *System.Delegate* and *System.MulticastDelegate* are implemented and used. The same advice is applicable to other Microsoft .NET Framework classes you happen to be interested in: when in doubt, disassemble the respective DLL and see for yourself.

Events, of course, can be implemented without delegates. But given the functionality needed to implement events, I don't see why anyone would waste time on an alternative implementation when the delegates offer a complete and elegant solution.

Managed Synchronization Elements

You're probably wondering whether managed code has any elements equivalent to the synchronization events and APIs of the unmanaged world. It does, although this aspect is unrelated to the events discussed in this chapter. The synchronization elements of the managed world are implemented as classes of the .NET Framework class library. You can learn a lot about them by disassembling Mscorlib.dll and having a look at the namespace *System.Threading*—and especially at the *WaitHandle* class of this namespace. (You've already encountered the *System.Threading.WaitHandle* class in the discussion of asynchronous invocation of delegates in Chapter 6.) The *WaitHandle* class is central to the entire class system of the *System.Threading* namespace and implements such methods as *WaitOne*, *WaitAll*, and *WaitAny*. Sounds familiar, doesn't it? The *AutoResetEvent*, *ManualResetEvent*, and *Mutex* classes, derived from the *WaitHandle* class, are also worth a glance.

Event Metadata

To define an event, we need to know the event type, which, as a rule, is derived from *[mscorlib]System.MulticastDelegate*; the methods associated with the event (methods to subscribe to the event, to unsubscribe, to fire the event, and perhaps to carry out other tasks we might define); and, of course, the class defining the event. Because events are never referenced in IL instructions, we needn't worry about the syntax for referencing the events.

The event metadata group includes the Event, EventMap, TypeDef, TypeRef, Method, and MethodSemantics tables. Figure 12-2 shows the mutual references between the tables of this group.

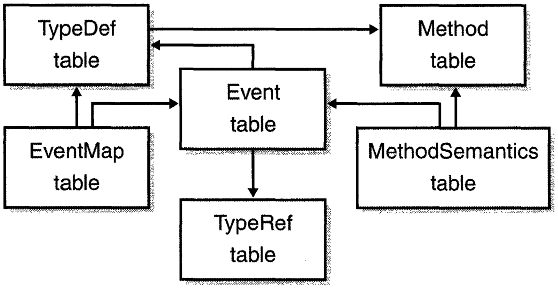


Figure 12-2 The event metadata group.

The Event Table

The Event table has the associated token type *mdtEvent* (0x14000000). An *Event* record has three entries:

- ***EventFlags* (2-byte unsigned integer)** Binary flags of the event characteristics.
- ***Name* (offset in the *#Strings* stream)** The name of the event, which must be a simple name no longer than 1023 bytes in UTF-8 encoding.
- ***EventType* (coded token of type *TypeDefOrRef*)** The type associated with the event. The coded token indexes a *TypeDef* or *TypeRef* record. The class indexed by this token is either a delegate or a class providing the necessary functionality similar to that of a delegate.

Only two flag values are defined for events, and only one of them can be set explicitly:

- ***specialname* (0x0200)** The event is special in some way, as specified by the name.
- ***rtspecialname* (0x0400)** The event has a special name reserved for the internal use of the common language runtime. This flag can't be set explicitly. The IL Disassembler (ILDASM) outputs this flag for information purposes, but the IL assembly language (ILAsm) compiler ignores the keyword.

To my knowledge, the primary use of these event flags is to mark deleted events in edit-and-continue scenarios. When an event record is marked as deleted, both flags are set and its name is changed to *_Deleted*. Some compilers, however, might find certain uses for the *specialname* flag. After all, an event as a metadata item exists solely for the benefit of the compilers.

The EventMap Table

The EventMap table provides mapping between the classes defining the events (the TypeDef table) and the events themselves (the Event table). An *EventMap* record has two entries:

- ***Parent* (record index [RID] to the TypeDef table)** The type declaring the events.
- ***EventList* (RID to the Event table)** The beginning of the events declared by the type indexed to the *Parent* entry. The mechanism of addressing the events in this case is identical to the mechanism used by *TypeDef* records to address the *Method* and *Field* records belonging to a certain *TypeDef.* In the optimized metadata model (the #~ stream), the records in the Event table are ordered by the declaring type. In the unoptimized model (the #- stream), the event records are not ordered and an intermediate lookup metadata table, EventPtr, is used. (The metadata models and intermediate tables are described in Chapter 4, "Metadata Tables Organization.")

The MethodSemantics Table

The MethodSemantics metadata table connects events and properties with their associated methods and provides information regarding the type of this association. A record in this table has three entries:

- ***Semantic* (2-byte unsigned integer)** The type of method association.
- ***Method* (RID to the Method table)** The index of the associated method.
- ***Association* (coded token of type *HasSemantics*)** A token indexing an event or a property the method is associated with.

The *Semantic* entry can have the following values, which look like binary flags but in fact are mutually exclusive:

- ***msSetter* (0x0001)** The method sets a value of a property.
- ***msGetter* (0x0002)** The method retrieves a value of a property.

- ***msOther* (0x0004)** The method has another meaning for a property or an event
- ***msAddOn* (0x0008)** The method subscribes to an event.
- ***msRemoveOn* (0x0010)** The method removes the subscription to an event.
- ***msFire* (0x0020)** The method fires an event.

The same method can be associated in different capacities with different events or properties. An event must have one subscribing method and one unsubscribing method. These methods return *void* and have one parameter of the same type as the event's associated type (the *EventType* entry of the respective *Event* record). The Microsoft Visual C# .NET and Visual Basic .NET compilers use uniform naming for subscribing and unsubscribing methods: *add_*<event_name> and *remove_<event_name>*, respectively. In addition, these compilers mark these methods with the *specialname* flag.

An event can have at most one firing method. The firing method usually boils down to an invocation of the delegate implementing the event. The Visual C# .NET and Visual Basic .NET compilers, for example, never bother to define a firing method for an event—that is, the method invoking the delegate is there, but it is never associated with the event as a firing method. Such an approach contains a certain logic: the firing method is a purely internal affair of the event publisher and need not be exposed to the event subscribers. And because the compilers, as a rule, use the event metadata to facilitate subscription and unsubscription, associating a firing method with an event is not necessary. If an event does have an associated firing method, however, this method must return *void.*

Event Declaration

In ILAsm, the syntax for an event declaration is as follows:

```
.event <class_ref> <name> { <method_semantics_decl>* }
```

where *<class_ref>* represents the type associated with the event, *<name>* is a simple name, and

```
<method_semantics_decl> ::= <semantics> <method_ref>
<semantics> ::= .addon | .removeon | .fire | .other
```

The following is an example of an event declaration:

```
// The delegate implementing the event
.class public sealed MyEventImpl
       extends [mscorlib]System.MulticastDelegate
{
```

(continued)

```
        .method public hidebysig specialname
                void   .ctor(object 'object',
                             native int 'method') runtime managed
        { }
           .method public hidebysig virtual instance void
              Invoke(int32 EventCode, string Msg) runtime managed
              { }
}
// The event publisher
.class public A
{
        .field private class MyEventImpl evImpl
                             // Aggregate delegate
        .method public specialname void .ctor()
        {
              ldnull
              ldarg.0
              stfld class MyEventImpl A::evImpl
              ret
        }
        .method public void Subscribe(class MyEventImpl aHandler)
        {
              ldarg.0
              ldfld class MyEventImpl A::evImpl
              ldarg.1
              call class [mscorlib]System.Delegate
                         A::Combine(class [mscorlib]System.Delegate,
                         class [mscorlib]System.Delegate)
              ldarg.0
              stfld class MyEventImpl A::evImpl
              ret
        }
        .method public void Unsubscribe(class MyEventImpl aHandler)
        {
              ldarg.0
              ldfld class MyEventImpl A::evImpl
              ldarg.1
              call class [mscorlib]System.Delegate
                         A::Remove(class [mscorlib]System.Delegate,
                         class [mscorlib]System.Delegate)
              ldarg.0
              stfld class MyEventImpl A::evImpl
              ret
        }
        .method public void Raise(int32 EventCode, string Msg)
        {
              ldarg.0
              ldfld class MyEventImpl A::evImpl
```

```
            ldarg.1
            ldarg.2
            call void MyEventImpl::Invoke(int32, string)
            ret
      }
      .method public bool HasSubscribers()
      {
            ldc.i1.0
            ldarg.0
            ldfld class MyEventImpl A::evImpl
            brnull L1
            pop
            ldc.i1.1
          L1:     ret
      }
      .event MyEventImpl MyEvent
      {
         .addon instance void A::Subscribe(class MyEventImpl)
         .removeon instance void A::Unsubscribe(class MyEventImpl)
         .fire instance void A::Raise(int32, string)
         .other instance bool A::HasSubscribers()
      }
      // Other class members
      :
} // The end of the publisher class
// The event subscriber
.class public B
{
   .method public void MyEvtHandler(int32 EventCode, string Msg)
   {
         // If EventCode > 100, print the message
         ldarg.1
         ldc.i4 100
         ble.s Return
         ldarg.2
         call void [mscorlib]System.Console::WriteLine(string)
     Return:
         ret
   }
   .method private void SubscribeToMyEvent(class A Publisher)
      {
// Publisher->Subscribe(new MyEventImpl
// (this,(int)(this->MyEvtHandler)))
         ldarg.1
         ldarg.0
         dup
         ldftn instance void MyEvtHandler(int32, string)
         newobj instance void MyEventImpl::.ctor(object, native
```

(continued)

```
int)
        call instance void A::Subscribe(class MyEventImpl)
        ret
}
  // Other class members
   ⋮
} // The end of the subscriber class
```

Property Metadata

Properties are considerably less fascinating than events. Typically, a property is some characteristic of the class that declares it—for example, the value of a private field—accessible only through the so-called *accessor methods*. Because of this, the only aspects of a property the common language runtime is concerned with are the property's accessors.

Let's suppose that a property is based on a private field. Let's also suppose that both read and write accessors are defined. What is the sense in declaring such a property, when we could simply make the field public and be done with it? At least two reasons argue for declaring it: the accessors can run additional checks to ensure that the field has valid values at all times, and the accessors can fire events signaling that the property has been changed or accessed. I'm sure you can think of other reasons for implementing properties, even leaving aside cases in which the property is not field-based or has only a read accessor or only a write accessor.

A property's read and write accessors are referred to as *getters* and *setters*, respectively. The Visual C# .NET and Visual Basic .NET compilers follow these naming conventions for the property accessors: setters are named *set_<property_name>*, and getters are named *get_<property_name>*. Both methods are marked with the *specialname* flag.

The property metadata group includes the following tables: Property, PropertyMap, TypeDef, Method, MethodSemantics, and Constant. The structure of the property metadata group is shown in Figure 12-3. The following sections describe the Property and PropertyMap tables. The MethodSemantics table was discussed in the preceding section of this chapter, and Chapter 8, "Fields and Data Constants," contains information about the Constant table.

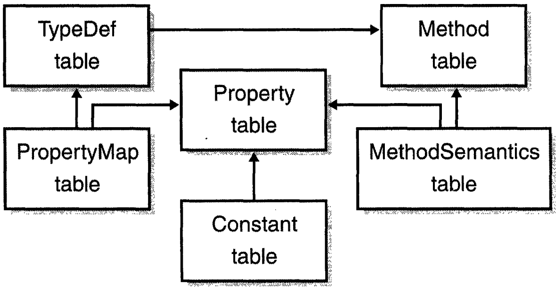


Figure 12-3 The property metadata group.

The Property Table

The Property table has the associated token type *mdtProperty* (0x17000000), and its records have three entries:

- ***PropFlags* (2-byte unsigned integer)** Binary flags of the property characteristics.
- ***Name* (offset in the *#Strings* stream)** The name of the property, which must be a simple name no longer than 1023 bytes in UTF-8 encoding.
- ***Type* (offset in the *#Blob* stream)** The property signature.

The *Type* entry holds an offset to the property signature residing in the *#Blob* metadata stream. The structure of the property signature is similar to that of the method signature, except that the calling convention is *IMAGE_CEE_CS_CALLCONV_PROPERTY* (0x08). The return type and the parameter types of the property should correspond to those of the getter. The runtime, of course, pays no attention to what the property signature looks like, but the compilers do care.

Three flag values are defined for properties, and, as in the case of events, only one of them can be set explicitly:

- ***specialname* (0x0200)** The property is special in some way, as specified by the name.
- ***rtspecialname* (0x0400)** The event has a special name reserved for the internal use of the common language runtime. This flag can't be set explicitly.
- **[no ILAsm keyword] (0x1000)** The property has a default value, which resides in the Constant table.

Like the event flags, the *specialname* and *rtspecialname* flags are used by the runtime for marking deleted properties in edit-and-continue scenarios. The deleted property name is changed to *_Deleted*. The flag 0x1000 is set by the metadata emission API when a *Constant* record is emitted for this property, signifying the property's default value.

The PropertyMap Table

The PropertyMap table serves the same purpose for properties as the EventMap table does for events: it provides mapping between the TypeDef table and the Property table. A *PropertyMap* record has two entries:

- ***Parent* (RID to the TypeDef table)** The type declaring the properties.
- ***PropertyList* (RID to the Property table)** The beginning of the properties declared by the type referenced by the *Parent* entry.

In the unoptimized model (the #- stream), an intermediate lookup metadata table, PropertyPtr, is used to remap the properties so that they are ordered by parent.

Property Declaration

The ILAsm syntax for a property declaration is as follows:

```
.property <flags> <ret_type> <name>(<param_type>[,<param_type>*] )

          [ <const_decl> ]   { <method_semantics_decl>* }
```

where

```
<method_semantics_decl> ::= <semantics> <method_ref>
<semantics> ::= .set | .get | .other
<const_decl> ::=   = <const_type> [ ( <value> ) ]
```

The *<ret_type>* and the sequence of *<param_type>* nonterminals define the property's signature. *<semantics>* defines the type of the associated methods: *.set* for the setter, *.get* for the getter, and *.other* for any other method defined for this property. The optional *<const_decl>* is the declaration of the property's default value, similar to that of a field or a method parameter. The parent of the property is the class in whose scope the property is declared, as is the case for other class members (fields, methods, and events).

Now, as an exercise, let's declare a simple property:

```
.class public A
{
   .field private unsigned int32 theTally = int32(0xFFFFFFFF)
   // Constructor: set Tally to 0xFFFFFFFF (not used yet)
   .method public void .ctor()
   {
      ldarg.0
      call instance void [mscorlib]System.Object::.ctor()
      ldc.i4 0xFFFFFFFF
      ldarg.0
      stfld unsigned int32 A::theTally
      ret
   }
   // Setter: set Tally to Val if Val is not 0xFFFFFFFF
   .method public void set_Tally(unsigned int32 Val)
   {
      ldarg.1
      ldc.i4 0xFFFFFFFF
      beq.s Return
      ldarg.1
      ldarg.0
      stfld unsigned int32 A::theTally
   Return:
      ret
   }
   // Getter: return the value of Tally
   .method public unsigned int32 get_Tally()
   {
      ldarg.0
      ldfld unsigned int32 A::theTally
      ret
   }
   // Other method: reset the value of Tally to 0xFFFFFFFF
   .method public void reset_Tally()
   {
      ldc.i4 0xFFFFFFFF
      ldarg.0
      stfld unsigned int32 A::theTally
      ret
   }
   .property unsigned int32 Tally(unsigned int32)
     = int32(0xFFFFFFFF)
   {
      .set instance void A::set_Tally(unsigned int32)
      .get instance unsigned int32 A::get_Tally()
      .other instance void A::reset_Tally()
   }
} // The end of class A
```

Metadata Validity Rules

The event-related and property-related metadata tables are Event, EventMap, Property, PropertyMap, Method, MethodSemantics, TypeDef, TypeRef, and Constant. Earlier chapters have discussed the validity rules for Method, TypeDef, TypeRef, and Constant tables. The records of the remaining tables have the following entries:

- The Event table: *EventFlags*, *Name*, and *EventType*.
- The EventMap table: *Parent* and *EventList*.
- The Property table: *PropFlags*, *Name*, and *Type*.
- The PropertyMap table: *Parent* and *PropertyList*.
- The MethodSemantics table: *Semantic*, *Method*, and *Association*.

Event Table Validity Rules

- The *EventFlags* entry must contain 0, or must have the *specialname* flag set (0x0200), or must have both the *specialname* and *rtspecialname* flags set (0x0600).
- The *Name* entry must contain a valid offset in the *#Strings* stream, indexing a string no longer than 1023 bytes in UTF-8 encoding.
- If the *specialname* and *rtspecialname* flags are set, the event name must be *_Deleted**.
- No duplicate records—those with the same name belonging to the same class—can exist unless the event name is *_Deleted**.
- The *EventType* entry must contain a valid reference to the TypeDef or TypeRef table.

EventMap Table Validity Rules

- The *Parent* entry must hold a valid reference to the TypeDef table.
- The *EventList* entry must hold a valid reference to the Event table.

Property Table Validity Rules

- The *PropFlags* entry must contain 0 or a combination of the binary flags *specialname* (0x0200), *rtspecialname* (0x0400), and 0x1000.
- If the *rtspecialname* flag is set, the *specialname* flag must also be set.

- If the 0x1000 flag is set, the Constant table must contain a valid record whose *Parent* entry holds the reference to this *Property* record, and vice versa.
- The *Name* entry must contain a valid offset in the *#Strings* stream, indexing a string no longer than 1023 bytes in UTF-8 encoding.
- If the *specialname* and *rtspecialname* flags are set, the property name must be *_Deleted**.
- No duplicate records—those with the same name and signature belonging to the same class—can exist unless the property name is *_Deleted**.
- The Type entry must contain a valid offset in the *#Blob* stream, indexing a valid property signature. The validity rules for property signatures were discussed in Chapter 7, "Primitive Types and Signatures."

PropertyMap Table Validity Rules

- The *Parent* entry must hold a valid reference to the TypeDef table.
- The *PropertyList* entry must hold a valid reference to the Property table.

MethodSemantics Table Validity Rules

- The *Semantic* entry must contain one of the following values: *msSetter* (0x0001), *msGetter* (0x0002), *msOther* (0x0004), *msAddOn* (0x0008), *msRemoveOn* (0x0010), *msFire* (0x0020).
- The *Method* entry must contain a valid index to the Method table.
- The *Association* entry must contain a valid reference to the Event or Property table.
- If the *Semantic* entry contains *msSetter* or *msGetter*, the *Association* entry must reference the Property table.
- If the *Semantic* entry contains *msAddOn*, *msRemoveOn*, or *msFire*, the *Association* entry must reference the Event table.
- No duplicate records that have the same *Method* and *Association* entries can exist.
- No duplicate records that have the same *Association* and *Semantic* entries can exist unless the *Semantic* entry contains *msOther*.

- For each *Event* record referenced in the *Association* entry, the table can contain one and only one *MethodSemantics* record with a *Semantic* entry equal to *msAddOn*.
- For each *Event* record referenced in the *Association* entry, the table can contain one and only one *MethodSemantics* record with a *Semantic* entry equal to *msRemoveOn*.
- For each *Event* record referenced in the *Association* entry, the table can contain no more than one *MethodSemantics* record with a *Semantic* entry equal to *msFire*.

13

Custom Attributes

Every system worth its name needs extensibility. The languages that describe an extensible system and their compilers need extensibility as well; otherwise, they are describing not the system but rather its glorious past.

A system and the associated languages can be extended in three ways. The first way is to tinker with the system itself, changing its inner structure and changing the languages accordingly. This approach is good as long as the system has a negligible number of users, because each new version of the system (and hence the languages) is basically different from the previous version. This approach is characteristic of the early stages of the life cycle of a complex system.

The second way is to leave the system and the languages as they are and build a parallel system (and parallel languages and their compilers) providing additional functionality. A classic example of this approach was the introduction of the remote procedure call (RPC) standard and the interface description language (IDL) in parallel with existing C runtime and C/C++ compilers.

The third way is to build into the system (and the languages) some formal means of extensibility and then employ these means when needed. This approach allows the system developers to sneak in new features and subsystems without changing the basic characteristics of the system. The only challenge is to devise a means of extensibility that is both efficient and universal—efficient because we need productivity, and universal because we don't know what we'll need tomorrow or a year from now. These requirements are contradictory, and usually universality wins out. If efficiency wins at universality's expense, sooner or later the designers run out of options and must switch to the second way.

The Microsoft .NET platform, including the common language runtime, the .NET Framework, and the compilers, has a universal extensibility mechanism built in. This mechanism is known as custom attributes.

Concept of a Custom Attribute

A *custom attribute* is a metadata item specifically designed as a universal tool for metadata extension. Custom attributes do not, of course, change the metadata schema, which is hard-coded and a sacrosanct part of the common language runtime. Neither do custom attributes play any role similar to the generics, creating new types based on some "templates." Rather, custom attributes provide a way to specify additional information about metadata items, information not represented by a metadata item itself.

The information carried by custom attributes is intended mostly for various tools such as compilers, linkers, and debuggers. The runtime recognizes only a small subset of custom attributes.

Custom attributes are also a lifesaver for compilers. If the designers of compilers and languages discover, to their surprise, that more features are required to describe a problem area than were initially built into a language or its compiler, they can easily extend the descriptive power of the language by introducing new custom attributes. Of course, the language and its compiler must recognize the concept of a custom attribute to begin with, but it's hardly a problem—all managed languages and their compilers do this.

I've heard some slanderous statements to the effect that the number of custom attributes used by a tool is in direct proportion to the degree of wisdom acquired by the tool designers after the fact. But of course this can't be true.

Jokes aside, custom attributes are an extremely useful tool. Think of the following simple example. If we want managed code to interoperate with classic COM applications, we need to play by the classic COM rules. One of these rules is that every exported interface must have a globally unique identifier, a GUID, assigned to it. The runtime generates GUIDs on the fly, but we might need not just *any* GUID but rather a specific GUID assigned to a class. What do we do? Add another field to the *TypeDef* record to store an offset in the *#GUID* stream? This would surely help to reduce the size of the metadata tables, especially when we consider that only a small fraction of all *TypeDef*s might ever be used in COM interoperation. To solve the problem, we can introduce a GUID-carrying custom attribute—actually, we have one already: *System.Runtime.InteropServices.GuidAttribute*—and assign this attribute to any *TypeDef* participating in the COM interoperation.

The problem with custom attributes is that they are very expensive in terms of resources. They bloat the metadata. Because they represent metadata add-ons, the IL code has no means of accessing them directly. As a result, custom attributes must be resolved through Reflection methods, an approach that approximates having a lively chat by means of mailing letters written in Morse code—fun if you have an eternity at your disposal.

There's good news regarding custom attributes, and there's also bad news. The bad news is that custom attributes keep breeding at an astonishing rate as new tools and new features are introduced. And sometimes custom attributes are invented not because of need but because "I can" or because someone wonders, "Why should I do it the hard way?" It's so easy to use, no wonder.... Ahem! The good news, however, is that most custom attributes are specific to certain tools and only a small fraction are actually used at run time.

CustomAttribute Metadata Table

The CustomAttribute table contains data that can be used to instantiate custom attributes at run time. A record in this CustomAttribute table has three entries:

- ***Parent* (coded token of type *HasCustomAttribute*)** This entry references the metadata item to which the attribute is assigned.
- ***Type* (coded token of type *CustomAttributeType*)** This entry defines the type of the custom attribute itself.
- ***Value* (offset in the *#Blob* stream)** This entry is the binary representation of the custom attribute's parameters.

Given their nature, as informational add-ons to other metadata items, custom attributes can be attached to any metadata item that has a specific token type assigned to it. The one exception is that custom attributes cannot be attached to another custom attribute. Chapter 4, "Metadata Tables Organization," described the 21 token types. The token type *mdtString* (0x70000000) is assigned to user strings, which are not part of the metadata tables, and *mdtCustomAttribute* (0x0C000000) belongs to the custom attributes themselves. This leaves us with 19 tables providing potential owners of custom attributes: Module, TypeRef, TypeDef, Field, Method, Param, InterfaceImpl, MemberRef, DeclSecurity, StandAloneSig, Event, Property, ModuleRef, TypeSpec, Assembly, AssemblyRef, File, ExportedType, and ManifestResource. No metadata table references the CustomAttribute table. Note that a custom attribute can be assigned to a specific type (*TypeRef*, *TypeDef*), but not to an instance of the type.

The *Type* entry of a custom attribute is a coded token of type *CustomAttributeType* and hence theoretically can be one of the following: *TypeRef*, *TypeDef*, *Method*, *MemberRef*, or *String*. (See the section "Coded Tokens" in Chapter 4.) In fact, in the first release of the common language runtime, the choice is limited to *Method* or *MemberRef* because of the requirement that the type of a custom attribute must be an instance constructor and nothing else. The class whose instance constructor represents the custom attribute type should be derived from the abstract class *[mscorlib]System.Attribute.*

The *Value* entry of a custom attribute is a blob whose contents depend on the nature of the custom attribute. If we were allowed to use a user-defined string as the custom attribute type, *Value* would contain the Unicode text. But because the custom attribute type is limited to instance constructors, the *Value* blob contains the encoded arguments of the constructor. If the constructor has no parameters, because the mere presence of the custom attribute is considered sufficiently informational, the *Value* entry can hold 0.

Custom Attribute Value Encoding

The *Value* blob of a custom attribute might contain two categories of data: encoded argument values of the instance constructor, and additional encoded name/value pairs specifying the initialization values of the fields and properties of the custom attribute class.

The *Value* blob encoding is based on serialization type codes enumerated in *CorSerializationType* in the CorHdr.h file. The serialization codes for the primitive types, strings, and vectors are the same as the respective *ELEMENT_TYPE_** codes—that is, *ELEMENT_TYPE_BOOLEAN*, and so on, as described in Chapter 7, "Primitive Types and Signatures." Additional serialization codes include *TYPE* (0x50), *TAGGED_OBJECT* (0x51), *FIELD* (0x53), *PROPERTY* (0x54), and *ENUM* (0x55). All the constant names include the prefix *SERIALIZATION_TYPE_*, which I omit because of my inherent laziness.

The encoded blob begins with the prolog, which is always the 2-byte value 0x0001. The prolog is followed by the values of the constructor arguments. Size and byte layout of these values are inferred from the constructor's signature. For example, the value 0x1234 supplied as an argument of type *int32* is encoded as the following sequence of bytes:

```
0x34 0x12 0x00 0x00
```

If the argument is a vector, its encoding begins with a 4-byte element count, followed by the element values. For example, a vector of the three *unsigned int16* values 0x1122, 0x3344, and 0x5566 is encoded as follows:

```
0x03 0x00 0x00 0x00 0x22 0x11 0x44 0x33 0x66 0x55
```

If the argument is a string, its encoding begins with the compressed string length, followed by the string itself in UTF-8 encoding, without the terminating 0 byte. The length compression formula was discussed in Table 4-1. For example, the string *Common Language Runtime* is encoded as the following byte sequence, with the leading byte (0x17) representing the string length (23 bytes):

```
0x17 0x43 0x6F 0x6D 0x6D 0x6F 0x6E 0x20 0x4C 0x61 0x6E 0x67 0x75 0x61
0x67 0x65 0x20 0x52 0x75 0x6E 0x74 0x69 0x6D 0x65
```

If the argument is an object reference to a boxed primitive value type—*bool*, *char*, one of the integer types, or one of the floating-point types—the encoding consists of 1-byte primitive type encoding, followed by the value of the primitive value type.

Finally, if the argument is a type (class), its encoding is similar to that of a string, with the type's fully qualified name playing the role of the string constant. The rules of the fully qualified type name formatting applied in the custom attribute blob encoding are those of Reflection, which differ from IL assembly language (ILAsm) conventions. The full class name is formed in Reflection and ILAsm almost identically, except for the separator symbols that denote the class nesting. ILAsm notation uses a forward slash:

```
MyNamespace.MyEnclosingClass/MyNestedClass
```

whereas the Reflection standard uses a plus sign:

```
MyNamespace.MyEnclosingClass+MyNestedClass
```

We find greater difference, however, in the way resolution scope is designated. In ILAsm, the resolution scope is expressed as the external assembly's alias in square brackets preceding the full class name. In Reflection notation, the resolution scope is specified after the full class name, comma-separated from it. In addition, the concept of the external assembly alias is specific to ILAsm, and Reflection does not recognize it. Thus, if the version, public key token, or culture must be specified, it is done explicitly as a part of the resolution scope specification. The following is an ILAsm example:

```
.assembly extern OtherAssembly as OtherAsm2
{
   .ver 1:2:3:4
   .publickeytoken = (01 02 03 04 05 06 07 08)
   .locale "fr-CA"
}
⋮
[OtherAsm2]MyNamespace.MyEnclosingClass/MyNestedClass
```

In contrast, here is a Reflection example:

```
MyNamespace.MyEnclosingClass+MyNestedClass, OtherAssembly,
   Version=1.2.3.4, PublicKeyToken=0102030405060708, Culture=fr-CA
```

According to Reflection conventions, the resolution scope specification can be omitted if the referenced class is defined in the current assembly or in Mscorlib.dll. In ILAsm, as you know, the resolution scope is omitted only if the class is defined in the current module.

The byte sequence representing the prolog and the constructor arguments is followed by the 2-byte count of the name/value pairs. A name/value pair specifies which particular field or property must be initialized to a certain value.

The name/value pair encoding begins with the serialization code of the target: *FIELD* or *PROPERTY*. The next byte is the serialization code of the target type, which is limited to the primitive types, string, and *TYPE*. After the target type comes the name of the target, encoded the same way a string argument would be: the compressed length, followed by the string itself in UTF-8 encoding, without the 0 terminator. Immediately after the target name is the target initialization value, encoded similarly to the arguments. For example, the name/value pair initializing a field (0x53) of type *bool* (0x02) named *Inherited* (length 0x09) to *true* (0x01) is encoded as this byte sequence:

```
0x53 0x02 0x09 0x49 0x6E 0x68 0x65 0x72 0x69 0x74 0x65 0x64 0x01
```

Custom Attribute Declaration

The ILAsm syntax for declaring a custom attribute is as follows:

```
.custom <attribute_type> [ = ( <hexbytes> ) ]
```

or, considering the limitation imposed on *<attribute_type>* in the first release of the common language runtime:

```
.custom instance void <class_ref>::.ctor(<arg_list>)
   [ = ( <hexbytes> ) ]
```

where *<class_ref>* is a fully qualified class reference, *<arg_list>* is an argument list of the instance constructor, and *<hexbytes>* is the sequence of two-digit hexadecimal numbers representing the bytes in the custom attribute's blob.

You might have noticed a regrettable omission in the first release of ILAsm: the language does not allow you to specify the custom attribute value in "civilized" terms, requiring an explicit specification of the blob contents. For example, instead of writing

```
.custom instance void MyAttribute::.ctor(bool) = (true)
```

you must write

```
.custom instance void MyAttribute::.ctor(bool) = (01 00 01 00 00)
```

I expect this omission to be remedied in later releases of ILAsm and its compiler.

The owner of the custom attribute, or the metadata item to which the attribute is attached, is defined by the positioning of the custom attribute declaration. At first glance, the rule regarding the declaration of metadata items is simple: If the item declaration has a scope (for example, an assembly, a class, or a method), the custom attributes declared within this scope belong to the item. Otherwise—that is, if the item declaration has no scope (such items as a file, a module, or a field)—the custom attributes declared immediately *after* the item declaration belong to the item. For example, take a look at these excerpts from the disassembly of Mscorlib.dll:

```
.assembly mscorlib
{
   .custom instance void System.CLSCompliantAttribute::.ctor(bool)
      = ( 01 00 01 00 00 )
   .custom instance void
      System.Resources.NeutralResourcesLanguageAttribute::.ctor(string
)
      = ( 01 00 05 65 6E 2D 55 53 00 00 )   // ...en-US..
   ⋮
}
⋮
.module CommonLanguageRuntimeLibrary
.custom instance void
   System.Security.UnverifiableCodeAttribute::.ctor()
   = ( 01 00 00 00 )
⋮
.class interface public abstract auto ansi IEnumerable
{
   .custom instance void
      System.Runtime.InteropServices.GuidAttribute::.ctor(string)
      = ( 01 00 24 34 39 36 42 30 41 42 45 2D 43 44 45 45
          2D 31 31 64 33 2D 38 38 45 38 2D 30 30 39 30 32
          37 35 34 43 34 33 41 00 00 )
   .method public hidebysig newslot virtual abstract
      instance class System.Collections.IEnumerator
      GetEnumerator() cil managed
   {
      .custom instance void
         System.Runtime.InteropServices.DispIdAttribute::.ctor(int32)
         = ( 01 00 FC FF FF FF 00 00 )
      ⋮
   } // End of method IEnumerable::GetEnumerator
   ⋮
} // End of class IEnumerable
⋮
```

This is in stark contrast to the way custom attributes are declared, for instance, in Microsoft Visual C# .NET, where a custom attribute belonging to an item immediately *precedes* the item declaration. For example, the following is an excerpt showing the Visual C# .NET declaration of the interface *IEnumerable* mentioned in the preceding code:

```
[Guid("496B0ABE-CDEE-11d3-88E8-00902754C43A")]
public interface IEnumerable
{
   [DispId(-4)]
   IEnumerator GetEnumerator();
}
```

The ILAsm rule specifying that custom attribute ownership is defined by the position of the attribute declaration can play tricks on you if you don't pay attention, however. Don't forget that when a nonscoped item is declared within the scope of another item, the custom attribute's ownership immediately switches to this newly declared item. Because of that, the custom attributes belonging to a scoped item cannot be declared just anywhere within the item's scope. The following code snippet illustrates the point:

```
.class public MyClass
{
   .custom instance void MyClassAttribute::.ctor()=(01 00 00 00)
   .field int32 MyField
   .custom instance void MyFieldAttribute::.ctor()=(01 00 00 00)
   .method public int32 MyMethod([opt]int32 J)
   {
      .custom instance void MyMethodAttribute::.ctor()=(01 00 00 00)
      .param[1] = int32(123456)
      .custom instance void MyParamAttribute::.ctor()=(01 00 00 00)
      ⋮
   }
}
```

To avoid possible confusion about the ownership of a custom attribute, it's better to declare an item's custom attributes as soon as the item scope is opened.

The preceding discussion covers the rules for assigning custom attributes to items that are declared explicitly. Obviously, these rules cannot be applied to metadata items, which are declared implicitly, simply by their appearance in ILAsm directives and instructions. After all, such metadata items as *TypeRef*s, *TypeSpec*s, and *MemberRef*s might want their fair share of custom attributes, too.

To resolve this problem, ILAsm offers another form of the custom attribute declaration, with explicit specification of the custom attribute owner:

```
.custom ( <owner_spec> ) instance void <class_ref>::.ctor(<arg_list>)
       [ = ( <hexbytes> ) ]
```

where

```
<owner_spec> ::= <class_ref> | <type_spec>
                 | method <method_ref> | field <field_ref>
```

For example:

```
.custom ([mscorlib]System.String)
       instance void MyTypeRefAttribute::.ctor()=(01 00 00 00)
.custom ([mscorlib]System.String[])
       instance void MyTypeSpecAttribute::.ctor()=(01 00 00 00)
.custom (method instance void Foo::Bar(int32,int32))
       instance void MyMemberRefAttribute1::.ctor()=(01 00 00 00)
.custom (field int32 Foo::Baz)
       instance void MyMemberRefAttribute2::.ctor()=(01 00 00 00)
```

Custom attribute declarations in their full form can appear anywhere within the ILAsm source code, because the owner of a custom attribute is specified explicitly and doesn't have to be inferred from the positioning of the custom attribute declaration. The IL Disassembler (ILDASM) puts the custom attribute declarations in full form at the end of the source code dump, before the data dump.

Two additional categories of metadata items can in principle own custom attributes: *InterfaceImpl*s and *StandAloneSig*s. The first release of ILAsm offers no language means to define custom attributes belonging to these items, another omission to be corrected in future revisions of ILAsm and its compiler. Of course, so far no compiler or other tool has generated custom attributes for these items, but you never know. The tools develop quickly, and the custom attributes proliferate even more quickly, so sooner or later somebody will manage to assign a custom attribute to an interface implementation or a stand-alone signature.

On the other hand, many custom attributes are intended for the compilers' internal consumption and never make it past the complete compilation process. ILAsm needs to be concerned only with those custom attributes that are recognized by the common language runtime or the tools operating with managed PE files, such as the assembly linker (AL) or the debuggers.

Classification of Custom Attributes

Having decided to concentrate on the custom attributes recognized by the common language runtime or the tools dealing with managed PE files, let's see which custom attributes are intended for various subsystems of the runtime and tools.

Before proceeding, however, I must mention one custom attribute that stands apart from any classification and is truly unique. It is the attribute *System.AttributeUsageAttribute*, which can (and should) be owned only by the custom attribute types. Make no mistake—custom attributes can't own custom attributes, but the type of a custom attribute is always an instance constructor of some class. This class should own the custom attribute *System.AttributeUsage-Attribute*, which identifies what kinds of metadata items can own the custom attributes typed after this class, whether these custom attributes are inheritable, and whether multiple custom attributes of this type can be owned by the same metadata item. Because all operations concerning custom attributes are performed through Reflection, *AttributeUsageAttribute* can be considered the only custom attribute intended exclusively for Reflection itself. The instance constructor of the *AttributeUsageAttribute* type has one *int32* parameter, representing the binary flags for various metadata items as potential owners of the custom attribute typed after the attributed class. The flags are defined in the enumeration *System.AttributeTargets*.

The following should save you the time of looking up this enumeration in the disassembly of Mscorlib.dll:

```
.class public auto ansi serializable sealed AttributeTargets
       extends System.Enum
{
   // The following custom attribute is intended for the compilers
   // And indicates that the values of the enum are binary flags
   // And hence can be bitwise OR'ed
   .custom instance void System.FlagsAttribute::.ctor()
          = ( 01 00 00 00 )
   .field public specialname rtspecialname int32 value__
   .field public static literal valuetype System.AttributeTargets
          Assembly = int32(0x00000001)
   .field public static literal valuetype System.AttributeTargets
          Module = int32(0x00000002)
   .field public static literal valuetype System.AttributeTargets
          Class = int32(0x00000004)
   .field public static literal valuetype System.AttributeTargets
          Struct = int32(0x00000008) // Value type
   .field public static literal valuetype System.AttributeTargets
          Enum = int32(0x00000010)
   .field public static literal valuetype System.AttributeTargets
          Constructor = int32(0x00000020)
   .field public static literal valuetype System.AttributeTargets
          Method = int32(0x00000040)
   .field public static literal valuetype System.AttributeTargets
          Property = int32(0x00000080)
   .field public static literal valuetype System.AttributeTargets
          Field = int32(0x00000100)
```

```
    .field public static literal valuetype System.AttributeTargets
           Event = int32(0x00000200)
    .field public static literal valuetype System.AttributeTargets
           Interface = int32(0x00000400)
    .field public static literal valuetype System.AttributeTargets
           Parameter = int32(0x00000800)
    .field public static literal valuetype System.AttributeTargets
           Delegate = int32(0x00001000)
    .field public static literal valuetype System.AttributeTargets
           ReturnValue = int32(0x00002000)
    .field public static literal valuetype System.AttributeTargets
           All = int32(0x00003FFF)
} // End of class AttributeTargets
```

As you can see, Reflection's list of potential custom attribute owners is somewhat narrower than the metadata's list of 19 tables. Perhaps we needn't worry about the custom attributes of the interface implementations and standalone signatures just yet.

The remaining two characteristics of *AttributeUsageAttribute*—the Boolean properties *Inherited* and *AllowMultiple*—must be defined through the name/value pairs. The defaults are *All* for the potential custom attribute owners, *true* for *Inherited*, and *false* for *AllowMultiple*.

You'll find this information useful when (note that I'm not saying "if") you decide to invent your own custom attributes. And now, back to our classification scheme.

Execution Engine and JIT Compiler

The execution engine and the JIT (just-in-time) compiler of the common language runtime recognize three custom attributes:

- ***System.Diagnostics.DebuggableAttribute*** This attribute, which can be owned by the assembly or the module, sets a special debug mode for the JIT compiler. The instance constructor has two Boolean parameters, the first enabling the JIT compiler tracking the extra information about the generated code, and the second disabling JIT compiler optimizations. The ILAsm compiler automatically emits this custom attribute when the */DEBUG* command-line option is specified. The ILDASM outputs this attribute but comments it out.

- ***System.Security.UnverifiableCodeAttribute*** This attribute, which can be owned by the module, indicates that the module contains unverifiable code. Thus, because the result is known, IL code verification procedures don't have to be performed. The instance constructor has no parameters.

- ***System.ThreadStaticAttribute*** This attribute, which can be owned by a field, indicates that the static field is not shared between the threads. Instead, the common language runtime creates an individual copy of the static field for each thread. The effect is approximately the same as mapping the static field to the thread local storage (TLS) data, but this effect is achieved on the level of the runtime rather than that of the operating system.

Interoperation Subsystem

All the custom attribute types in this group belong to the namespace *System.Runtime.InteropServices*. The following list refers to them by their class names only:

- ***ClassInterfaceAttribute*** This attribute, which can be owned by the assembly or a *TypeDef* (class), specifies whether a COM class interface is generated for the attributed type. This attribute type has two instance constructors, each having a single parameter. The first constructor takes a value of enumerator *ClassInterfaceType*; the second takes an *int16* argument. The acceptable argument values are 0 (no automatic interface generation), 1 (automatic *IDispatch* interface generation), or 2 (automatic dual interface generation).
- ***ComAliasNameAttribute*** This attribute, which can be owned by a parameter (including the return value), a field, or a property, indicates the COM alias for the attributed item. The instance constructor has a single *string* parameter.
- ***ComConversionLossAttribute*** This attribute, which can be owned by any item, indicates that information about a class or an interface was lost when it was imported from a type library to an assembly. The instance constructor has no parameters.
- ***ComRegisterFunctionAttribute*** This attribute, which can be owned by a method, indicates that the method must be called when an assembly is registered for use from COM. This allows for the execution of user-defined code during the registration process. The instance constructor has no parameters.
- ***ComUnregisterFunctionAttribute*** This attribute, which can be owned by a method, indicates that the method must be called when an assembly is unregistered from COM. The instance constructor has no parameters.
- ***ComSourceInterfacesAttribute*** This attribute, which can be owned by a *TypeDef*, identifies a list of interfaces that are exposed

as COM event sources for the attributed type. This attribute type has five instance constructors; the most useful one has a single *string* parameter, the value of which should contain a space-separated list of all interface types in Reflection notation. (See "Custom Attribute Value Encoding," earlier in this chapter.)

- ***ComVisibleAttribute*** This attribute, which can be owned by the assembly, a *TypeDef*, a method, a field, or a property, indicates whether the attributed item is visible to classic COM. The instance constructor has one Boolean parameter having a value of *true* if the item is visible.

- ***DispIdAttribute*** This attribute, which can be owned by a method, a field, a property, or an event, specifies the COM DispId of the attributed item. The instance constructor has one *int32* parameter, the value of the DispId.

- ***GuidAttribute*** This attribute, which can be owned by the assembly or a *TypeDef*, specifies an explicit GUID if the GUID automatically generated by the runtime is for some reason not guid—I mean, good—enough. The instance constructor has one *string* parameter, which should contain the GUID value in standard literal representation without the surrounding curly braces.

- ***ImportedFromTypeLibAttribute*** This attribute, which can be owned by the assembly, indicates that the types defined within the assembly were originally defined in a COM type library. The attribute is set automatically by the TlbImp.exe utility. The instance constructor has one *string* parameter, which should contain the filename of the imported type library.

- ***InterfaceTypeAttribute*** This attribute, which can be owned by a *TypeDef* (interface), indicates the COM-specific interface type this interface is exposed as. The instance constructor has one *int16* parameter. A value of 0 indicates a dual interface, a value of 1 indicates *IUnknown*, and a value of 2 indicates *IDispatch*.

- ***ProgIdAttribute*** This attribute, which can be owned by a *TypeDef* (class), explicitly specifies the COM ProgId of the attributed class. Normally, the ProgId strings are generated automatically as a full class name (namespace plus name), but the ProgId length is limited to 39 bytes plus a 0 terminator. The namespaces and class names in .NET are rather long-winded, so there's a good chance 39 bytes won't even cover the namespace. The instance constructor has one *string* parameter, which should contain the ProgId string.

- ***TypeLibFuncAttribute*** This attribute, which can be owned by a method, specifies the COM function flags that were originally imported from the type library. (The COM function flags are described in COM literature and on the Microsoft Developer Network [MSDN].) This attribute is generated automatically by the TlbImp.exe utility. The instance constructor has one *int16* parameter, the flags' value.
- ***TypeLibTypeAttribute*** This attribute, which can be owned by a *TypeDef*, is similar to *TypeLibFuncAttribute* except that COM type flags are specified instead of COM function flags.
- ***TypeLibVarAttribute*** This attribute, which can be owned by a field, is similar to *TypeLibFuncAttribute* and *TypeLibTypeAttribute* except that the flags in question are COM variable flags.

Security

Security-related custom attributes are special attributes that are converted to *DeclSecurity* metadata records. Usually, the security custom attributes don't make it past the compilation stage—they are converted and cease to exist. In one scenario, however, the security custom attributes do "survive" the compilation and are emitted into the PE file. This happens when the security attributes owned by the assembly are specified in the assembly modules, further linked to the assembly by the assembly linker tool. In this case, the assembly-owned security attributes are converted to *DeclSecurity* metadata records by the assembly linker, but they remain in the assembly modules, although they play no role.

One of the security custom attributes belongs to the namespace *System.Security*:

- ***SuppressUnmanagedCodeSecurityAttribute*** This attribute, which can be owned by a method or a *TypeDef*, indicates that the security check of the unmanaged code invoked through the *P/Invoke* mechanism must be suppressed. The instance constructor has no parameters. This custom attribute differs from other security attributes in that it is not converted to *DeclSecurity* metadata and hence stays intact once emitted.

The rest of the security custom attributes belong to the namespace *System.Security.Permissions*. The ownership of all security custom attributes is limited to the assembly, a *TypeDef* (class or value type), and a method. The instance constructors of these attributes have one *int16* parameter, the action type code. Chapter 14, "Security Attributes," discusses the security action types and their respective codes as well as various types of permissions.

The following list offers a brief description of the security custom attributes; you can find further details in Chapter 14.

- ***SecurityAttribute*** This generic security attribute is the base class of all other security attributes.
- ***CodeAccessSecurityAttribute*** This attribute is the base class of the code access security attributes. Other attributes derived from this one are used to secure access to the resources or securable operations.
- ***EnvironmentPermissionAttribute*** This attribute sets the security action for the environment permissions that are to be applied to the code.
- ***FileDialogPermissionAttribute*** This attribute sets the security action for file open/save dialog permissions.
- ***FileIOPermissionAttribute*** This attribute sets the security action for the file input/output permissions (read, write, append and so on).
- ***IsolatedStorageFilePermissionAttribute*** This attribute sets the security action for the permissions related to the isolated storage files (available storage per user, the kind of isolated storage containment).
- ***PermissionSetAttribute*** This attribute sets the security action not for one permission but for a whole permission set, specified in a string or an XML file or a named permission set.
- ***PrincipalPermissionAttribute*** This attribute sets the security action for the principal security permissions (security checks against the active principal).
- ***PublisherIdentityPermissionAttribute*** This attribute sets the security action for the security permissions related to the software publisher's identity.
- ***ReflectionPermissionAttribute*** This attribute sets the security action for the Reflection permissions.
- ***RegistryPermissionAttribute*** This attribute sets the security action for the registry access permissions (read, write, create a key).
- ***SecurityPermissionAttribute*** This attribute sets the security action for the security permissions.
- ***SiteIdentityPermissionAttribute*** This attribute sets the security action for the site identity permissions.

- ***StrongNameIdentityPermissionAttribute*** This attribute sets the security action for the assembly's strong name manipulation permissions.
- ***UIPermissionAttribute*** This attribute sets the security action for the user interface permissions (window flags, Clipboard manipulation flags).
- ***UrlIdentityPermissionAttribute*** This attribute sets the security action for the URL permissions.
- ***ZoneIdentityPermissionAttribute*** This attribute sets the security action for the security zone (MyComputer, Intranet, Internet, Trusted, Untrusted).

Remoting Subsystem

The following custom attributes are recognized by the remoting subsystem of the common language runtime and can be owned by a *TypeDef*:

- ***System.Runtime.Remoting.Contexts.ContextAttribute*** This custom attribute class, which sets the remoting context, is a base class of all context attribute classes. It provides the default implementations of the interfaces *IContextAttribute* and *IContextProperty*. The instance constructor has one *string* parameter, the attribute name.
- ***System.Runtime.Remoting.Contexts.SynchronizationAttribute*** This custom attribute specifies the synchronization requirement and the re-entrance capability of the attributed class. It defines the class behavior in the synchronized contexts (contexts having the *Synchronization* property). The presence of an instance of this property in a context enforces a synchronization domain for the context (and all contexts that share the same instance). This means that at any instant, at most one thread could be executing in all contexts that share this property instance. The synchronization requirement flags are as follows:

  1 The class should not be instantiated in a context that has synchronization

  2 It is irrelevant to the class whether or not the context has synchronization

  4 The class should be instantiated in a context that has synchronization

  8 The class should be instantiated in a context with a new instance of the *Synchronization* property

  This attribute type has four instance constructors, as follows:

Constructor with no parameters	Defaults the synchronization requirement to 1 and the re-entrancy flag to *false*
Constructor with one *int32* parameter	Sets the synchronization requirement and defaults the re-entrancy flag
Constructor with one Boolean parameter	Sets the re-entrancy flag and defaults the synchronization requirement
Constructor with *int32* and Boolean parameters	Sets both values

- ***System.Runtime.Remoting.Activation.UrlAttribute*** This attribute is used at the call site to specify the URL of the site where the activation will happen. The instance constructor has one *string* parameter, which contains the target URL.

The information provided here is rather brief, but a protracted discussion of the topics related to remoting implementation goes far beyond the scope of this book. This is one of those occasions when one has to remember that modesty is a virtue.

Visual Studio .NET Debugger

The following two custom attributes are recognized by the Microsoft Visual Studio .NET debugger. They are not recognized by the .NET Framework debugger (Cordbg.exe). Both of these custom attributes belong to the namespace *System.Diagnostics*.

- ***DebuggerHiddenAttribute*** This attribute, which can be owned by a method or a property, signals the debugger not to stop in the attributed method and not to allow a breakpoint to be set in the method. The instance constructor has no parameters.
- ***DebuggerStepThroughAttribute*** This attribute is the same as *DebuggerHiddenAttribute* except that it *does* allow a breakpoint to be set in the method.

Assembly Linker

The five custom attributes listed in this section are intended for the assembly linker tool (Al.exe). They specify the characteristics of the assembly that the assembly linker is about to create from several modules.

The most fascinating aspect of these attributes is their ownership. Think about it: when the attributes are declared, no assembly exists yet; if it did, we wouldn't need these attributes in the first place. Hence, the attributes are declared in one or more of the modules that will make up the future assembly. What in a module might serve as an owner of these attributes? The solution is straightforward: the .NET Framework class library defines the *System.Runtime.CompilerServices.AssemblyAttributesGoHere* class, and the assembly-specific attributes are assigned to the *TypeRef* of this class. Ownership of the assembly-specific attributes is the only reason this class exists.

All of the assembly-specific attributes, described in the following list, belong to the namespace *System.Reflection*:

- ***AssemblyCultureAttribute*** This attribute specifies the culture of the assembly. The instance constructor has one *string* parameter, which contains the culture identification string.
- ***AssemblyVersionAttribute*** This attribute specifies the version of the assembly. The instance constructor has one *string* parameter, which contains the text representation of the version: dot-separated decimal values of the major version, the minor version, the build, and the revision. Everything beyond the major version can be omitted. If major and minor versions are specified, the build and/or the revision can be omitted or specified as an asterisk, which leads to automatic computation of these values at the assembly linker run time. The build number is computed as the present day's number counting since January 1, 2000. The revision number is computed as the number of seconds that have elapsed since midnight, local time, modulo 2.
- ***AssemblyKeyFileAttribute*** This attribute specifies the name of the file containing the key pair used to generate the strong name signature. The instance constructor has one *string* parameter.
- ***AssemblyKeyNameAttribute*** This attribute specifies the name of the key container holding the key pair used to generate the strong name signature. The instance constructor has one *string* parameter.
- ***AssemblyDelaySignAttribute*** This attribute specifies whether the assembly is signed immediately at the time of generation or delay-signed—in other words, fully prepared to be signed later by the strong name signing utility (Sn.exe). The instance constructor has one Boolean parameter, *true*, indicating that the assembly is delay-signed.

Common Language Specification (CLS) Compliance

The following two custom attributes are intended for the compilers and similar tools. Types of both of the custom attributes belong to the *System* namespace.

- ***ObsoleteAttribute*** This attribute, which can be owned by a *TypeDef*, a method, a field, a property, or an event, indicates that the item is not to be used any more. The attribute holds two characteristics: a *string* message to be produced when the obsolete item is used, and a Boolean flag indicating whether use of the item should be treated as an error. This attribute type has three instance constructors, as follows:

Constructor with no parameters	Produces no message and no error
Constructor with a *string* parameter	Produces a message but no error
Constructor with *string* and Boolean parameters	Produces a message and an error flag

- ***CLSComplianceAttribute*** This attribute, which can be owned by anything, indicates the (claimed) CLS compliance or noncompliance of the attributed item. Assigning this attribute to an assembly doesn't make the assembly CLS-compliant or noncompliant; it's simply an expression of your opinion on the matter. The instance constructor has one Boolean parameter; a value of *true* indicates CLS compliance.

Pseudocustom Attributes

As mentioned earlier, custom attributes are a lifesaver for compilers. Once a language is given the syntax to express a custom attribute, it's free to use this syntax to describe various metadata oddities its principal syntax can't express. The parallel evolution of the common language runtime and the managed compilers, with the runtime getting ahead now and then, created the concept of the so-called pseudocustom attributes. These attributes are perceived and treated by the compilers as other custom attributes are, but they are never emitted as such. Instead of emitting these attributes, the metadata emission API sets specific values of the metadata.

The following are the 13 pseudocustom attributes:

- ***System.Runtime.InteropServices.ComImportAttribute*** This attribute sets the *import* flag of a type definition. The instance constructor has no parameters.
- ***System.Runtime.InteropServices.DllImportAttribute*** This attribute sets the method flag *pinvokeimpl*, the implementation flag *preservesig*, and the name of the unmanaged library from which the method is imported. The instance constructor has one

string parameter, the name of the unmanaged library. The entry point name and the marshaling flags are specified through the name/value pairs of the *EntryPoint*, *CharSet*, *SetLastError*, *ExactSpelling*, and *CallingConvention* properties.

- ***System.SerializableAttribute*** This attribute sets the *serializable* flag of a type definition. The instance constructor has no parameters.
- ***System.NonSerializedAttribute*** This attribute sets the *notserialized* field flag. The instance constructor has no parameters.
- ***System.Runtime.InteropServices.InAttribute*** This attribute sets the parameter flag *in*. The instance constructor has no parameters.
- ***System.Runtime.InteropServices.OutAttribute*** This attribute sets the parameter flag *out*. The instance constructor has no parameters.
- ***System.Runtime.InteropServices.OptionalAttribute*** This attribute sets the parameter flag *opt*. The instance constructor has no parameters.
- ***System.Runtime.CompilerServices.MethodImplAttribute*** This attribute sets the method implementation flags. The instance constructor has one *int16* parameter, the implementation flags.
- ***System.Runtime.InteropServices.MarshalAsAttribute*** This attribute is used on fields and method parameters for managed/unmanaged marshaling. The instance constructor has one *int16* parameter, the native type.
- ***System.Runtime.InteropServices.PreserveSigAttribute*** This attribute sets the *preservesig* method implementation flag. The instance constructor has no parameters.
- ***System.Runtime.InteropServices.StructLayoutAttribute*** This attribute sets the layout flags of a type definition (*auto*, *sequential*, or *explicit*), the string marshaling flags (*ansi*, *unicode*, or *autochar*), and the characteristics *.pack* and *.size*. The instance constructor has one *int16* parameter, the layout flag. The *.pack* and *.size* characteristics and the string marshaling flags are specified through the name/value pairs of the *Pack*, *Size*, and *CharSet* properties, respectively.
- ***System.Runtime.InteropServices.FieldOffsetAttribute*** This attribute sets the field offset (ordinal) in explicit or sequential class layouts. The instance constructor has one *int32* parameter, the offset or ordinal value.
- ***System.Security.DynamicSecurityMethodAttribute*** This attribute sets the method flag *reqsecobj*. The instance constructor has no parameters.

ILAsm syntax is adequate to describe all the parameters and characteristics listed here and does not use the pseudocustom attributes.

> **Caution** As a matter of fact, I should warn you against using the pseudocustom attributes instead of ILAsm keywords and constructs. Using pseudocustom attributes rather than keywords is not a bright idea in part because the keywords are shorter than the custom attribute declarations. In addition, you should not forget that the ILAsm compiler, which has no use for custom attributes, treats them with philosophical resignation—in other words, it emits them just as they are, without analysis. Hence, if you specify important flags through pseudocustom attributes, the compiler will not see these flags and as a result could come to the wrong conclusions.

Metadata Validity Rules

A record of the CustomAttribute table has three entries: *Parent*, *Type*, and *Value*. The metadata validity rules for the custom attributes are rather simple:

- The *Parent* entry must hold a valid index to one of the following tables: Module, TypeRef, TypeDef, Field, Method, Param, InterfaceImpl, MemberRef, DeclSecurity, StandAloneSig, Event, Property, ModuleRef, TypeSpec, Assembly, AssemblyRef, File, ExportedType, or ManifestResource.
- The *Type* entry must hold a valid index to the Method or MemberRef table, and the indexed method must be an instance constructor.
- The *Value* entry must hold either 0 or a valid offset in the *#Blob* stream.
- The blob indexed in the *Value* entry must be encoded according to the rules described earlier in this chapter; see "Custom Attribute Value Encoding."
- The fields and properties listed in the name/value pairs of the *Value* blob must be accessible from the custom attribute owner (referenced in the *Parent* entry).

14

Security Attributes

As a platform for massively distributed operations, the Microsoft .NET Framework must have an adequate security mechanism. We all know that distributed platforms, especially those exposed to the Internet, are the favorite targets of all sorts of pranks and mischief, which can sometimes be very destructive.

The security system of the .NET Framework includes two major components: security policies and embedded security requirements. Security policies are part of the .NET Framework setup and reflect the opinions of the system administrator and the system user regarding what managed applications can and cannot do. Which policies are established can depend in part on the general origin of the application (for example, whether the application resides on the local drive of a machine, is taken from a closed intranet, or comes from the Internet), on the software publisher (for example, whether the system administrator feels differently about applications published by Microsoft or IBM and those published by tailspintoys.com), on the URL specifying the application's origin, on a particular application, and so forth. Important as they are, these security policies and their definition are beyond the scope of this book, and, with regret, I will forgo a detailed discussion of this topic.

Embedded security requirements are embedded in the applications themselves. Effectively, the embedded security requirements tell the common language runtime which rights an application needs in order to execute. The runtime checks the application's security requirements against the policy under which the application is executed and decides whether it's a go or a no-go.

Embedded security requirements are of two kinds: imperative security, which is part of the application's code; and declarative security, which is part of the application's metadata. *Imperative security* explicitly describes the operations necessary to perform a security check—for example, calling a method to query the runtime whether the application is given a certain right. *Declarative security* is a set of security attributes assigned to certain metadata items (the assembly as a whole or a certain class or method). Each of these attributes describes the rights that the corresponding item needs in order to be loaded and executed.

This chapter concentrates on declarative security because it is an important part of metadata and because you need to know how it is defined in IL assembly language (ILAsm). Besides, I have a feeling that many aspects of imperative security, and even security policies, can be deduced from an analysis of declarative security.

Declarative Security

Compared to imperative security, declarative security has two main advantages:

- Being part of the metadata, declarative security can be identified and assessed without exhaustive analysis of the application's IL code.
- Declarative security can be developed and modified independent of the functional code. As a result, a division of labor is possible: developer X the functionality guru writes the application, and developer Y the security guru tinkers with the security attributes.

A disadvantage of declarative security is its coarse targeting. Declarative security can be attributed to a class as a whole but not to the parts of the class and not to specific instances. Declarative security can be attributed to a method as a whole, without exact specification of when and under what circumstances the special rights might be needed. Imperative security, in contrast, allows the method to behave more flexibly—"...can I do this? No? OK, then I'll do it some other way. Let's see. Can I do that?..."

Declarative Actions

A declarative security attribute has three characteristics: the *target*, the metadata item to which it is attributed; the *permission*, a description of the rights that interest the target; and the *action*, a description of the precise way the target is interested in these rights.

The nine declarative security actions are intended for different targets and take effect at different stages of the application execution. The earliest stage of execution is the initial loading of the assembly's prime module and analysis of its manifest. Three declarative actions, targeting the assembly, take effect at this stage:

- ***Request Minimum*** This action specifies that the permission is a minimum requirement for the assembly to be executed. If the minimal permissions are not specified, the assembly is granted all rights according to the existing security policy. These rights, however, might be reduced by other already running parts of the application, by means of a *Deny* or *Permit Only* action.
- ***Request Optional*** This action specifies that the permission is useful to have but is not vital for the assembly execution.
- ***Request Refuse*** This action specifies that the permission should not be granted even if the security policy is willing to grant it. This action might be used to ensure that the assembly does not have rights it does not need, thus providing a shield against possible bugs in the assembly itself and against malicious code that might try to coerce the assembly to do something it shouldn't.

The next stage of the application execution is the loading of its classes and their members. Only one declarative action, targeting classes and methods, plays a role at this stage:

- ***Inheritance Demand*** For classes, this action specifies the permission that all classes descending from this one must have. For methods, this action specifies the permission that all methods overriding this one must have. Obviously, this action makes sense for virtual methods only.

After the classes and their members have been loaded, the IL code of the methods is JIT (just-in-time) compiled. The declarative action targeting classes and methods takes effect at this stage:

- ***Link Demand*** This action specifies the permission that all callers of this method must have—or, if the target is a class, the permission that any method of this class must have. For example, if you have a method that formats the system drive, you want to ensure that this method cannot be successfully called from some rogue code that has no right to do so. This action is limited to the immediate caller only. If method *A* link-demands permission *P*, and method *B* calling *A* has this permission, but method *C* calling *B* does not, the call will go through.

The last stage of the application execution is the run time, when the JIT-compiled code is actually executed. The declarative actions taking effect at this last stage and targeting classes and methods are as follows:

- ***Demand*** This action is similar to *Link Demand*, but it demands that all callers in the call chain have the specified permission.
- ***Assert*** This action specifies the permission that *any* caller on the call stack must have. If any caller at any level has the specified permission, the security check succeeds. This action obviously weakens the declarative security model and should be applied with caution. You cannot apply this action unless the code has the access permission *SecurityPermission*, which is discussed later in this chapter.
- ***Deny*** This action specifies the permission that must be disabled for all callers along the call stack for the duration of the called method. If a caller never had the specified permission in the first place, the action has no effect on it.
- ***Permit Only*** This action specifies the permission that must *not* be disabled for all callers along the call stack, presuming that the rest of the permissions must be disabled. The action seems excessively cruel (to strip the poor callers of all their privileges except one), but you must not forget that the target might have multiple security attributes. Using a series of *Permit Only* actions, you can create a set of permissions that remain for the callers to enjoy while all other permissions are temporarily revoked. To clarify this, consider the following example. If the called method has security attributes *Deny P* and *Deny Q*, all callers will have their permissions *P* and *Q* suspended. If the called method has security attributes *Permit Only P* and *Permit Only Q*, all permissions except *P* and *Q* of all callers will be suspended.

And now, let's see what these *P*s and *Q*s stand for.

Security Permissions

Security permissions define the kinds of activities the code requests (or demands, or denies, and so on) the right to perform. The same permissions are used in security policy definitions, specifying what sort of applications have the right to perform these activities and under what circumstances.

These permissions are represented by special classes of the .NET Framework class library. Each permission class is accompanied by a permission attribute class, a class whose instance constructor is used as a type of security custom attribute. Applying a security custom attribute to a metadata item leads to instantiation of the security object targeting the associated metadata item.

In some sense, it's easier to describe the permissions in terms of the accompanying attribute classes, because the permission classes have instance constructors of different signatures, whereas the instance constructors of the security attribute classes invariably have one parameter—the security action code—and all the parameters of the instance constructor(s) of the respective permission class are represented by the attribute's properties, set through name/value pairs.

The permissions form three groups. The first group includes the permissions related to access rights to certain resources. The second group consists of permissions related to identity characteristics of the code, including its origin. The third group represents custom permissions, invented by .NET Framework users for their particular purposes. It seems to be a general principle of the .NET Framework that if you can't find something satisfactory within the Framework, it at least provides you with the means to build your own better mousetrap.

Because most of the permission classes belong to the namespace *System.Security.Permissions*, of the Mscorlib.dll assembly, I've specified the assembly and namespace in the following sections only when they are different.

Access Permissions

The access permissions control access rights to various resources. The group includes the following nine permissions:

- ***[System.DirectoryServices]System.DirectoryServices.DirectoryServicesPermission*** This permission defines access to the Active Directory. The attribute class has two properties:
    - ❑ *Path* (type *string*) indicates the path for which the permission is specified.
    - ❑ *PermissionAccess* (type *int32*) specifies the type of access: a value of 0 indicates no access, a value of 2 indicates browse access, and a value of 6 indicates write access.

- ***[System]System.Net.DnsPermission*** This permission defines the right to use the Domain Name Services (DNS). The attribute class has no properties because there are no details to specify: either you can use DNS or you can't.

- ***EnvironmentPermission*** This permission defines the right to access the environment variables. The attribute class has three properties, all of type *string*, which specify the names of the environment variables affected:

  - □ *All* specifies the name of the environment variable that can be accessed in any way.
  - □ *Read* specifies the name of the environment variable that can be read.
  - □ *Write* specifies the name of the environment variable that can be written to.

- ***FileDialogPermission*** This permission defines the right to access a file selected through the standard Open or Save As dialog. The attribute class has two properties, both of type *bool*, for which *true* indicates that the access is to be granted and *false* indicates that it is to be denied:

  - □ *Open* grants or denies the right to read the file.
  - □ *Save* grants or denies the right to write to the file.

- ***FileIOPermission*** This permission defines the right to access specified directories or individual files. The attribute class has five properties, all of type *string*, which contain a path or a full-path file specification. If the path is specified, the permission is propagated to the whole directory subtree starting at this path. The attribute class properties are as follows:

  - □ *All* indicates full access to the specified path or file.
  - □ *Read* indicates read access to the specified path or file.
  - □ *Write* indicates write access, including file overwriting and new file creation.
  - □ *Append* indicates append access—in other words, the existing file can be appended but not overwritten, and a new file can be created.
  - □ *PathDiscovery* indicates browse access—for example, querying the current directory, getting a filename back from the file dialog, and so on.

- ***IsolatedStorageFilePermission*** This permission defines the right to access the isolated storage. Briefly, the *isolated storage* is a storage space allocated specifically for the user's application, providing a configurationless data store independent of the structure of the local file system. Data compartments within the isolated storage are defined by the identity of the application or component code. Thus, there's no need to work magic with the file paths to ensure that the data storages specific to different applications don't overlap. The attribute class has two properties:

  - ❑ *UsageAllowed* (*int32*-based enumeration *IsolatedStorageContainment*) indicates the isolated storage type.
  - ❑ *UserQuota* (type *int64*) indicates the maximum size in bytes of the isolated storage that can be allocated for one user.

  The *UsageAllowed* property can be assigned the following *int32* values:

  None (0x00)
  DomainIsolationByUser (0x10)
  AssemblyIsolationByUser (0x20)
  DomainIsolationByRoamingUser (0x50)
  AssemblyIsolationByRoamingUser (0x60)
  AdministerIsolatedStorageByUser (0x70)
  UnrestrictedIsolatedStorage (0xF0)

- ***ReflectionPermission*** This permission defines the right to invoke Reflection methods on nonpublic class members and to create dynamic assemblies at run time using the methods of *Reflection.Emit*. The attribute class has four properties:

  - ❑ *MemberAccess* (type *bool*) grants or denies the right to access the nonpublic members through Reflection methods.
  - ❑ *TypeInformation* (type *bool*) grants or denies the right to invoke Reflection methods to retrieve information about the class, including information about the nonpublic members.
  - ❑ *ReflectionEmit* (type *bool*) grants or denies the right to invoke *Reflection.Emit* methods.
  - ❑ *Flags* (*int32*-based enumeration *ReflectionPermissionFlag*) summarizes the three preceding properties, using a binary OR combination of flags 0x01 for *TypeInformation*, 0x02 for *MemberAccess*, and 0x04 for *ReflectionEmit*.

- ***RegistryPermission*** This permission defines the right to manipulate the registry keys and values. This permission is analogous in all ways to *FileIOPermission* except that it specifies the access rights to the registry rather than to the file system. The attribute class has four properties, all of type *string*, which contain the registry path:
  - ❑ *Create* grants the right to create the keys and values anywhere in the registry subtree, starting with the node specified in the property.
  - ❑ *Read* grants the right to read the keys and values.
  - ❑ *Write* grants the right to change the existing keys and values.
  - ❑ *All* grants all of the three preceding rights.
- ***SecurityPermission*** This permission defines a set of 13 essential rights to modify the behavior of the common language runtime security subsystem itself. The attribute class has 13 properties of type *bool* (one for each right) plus one property (*int32*-based enumeration *SecurityPermissionFlag*) representing an OR combination of binary flags corresponding to the Boolean properties:
  - ❑ *Assertion* defines the right to override a security check for any granted permission. The respective binary flag is 0x0001.
  - ❑ *UnmanagedCode* defines the right to invoke the native unmanaged code, for example, through *P/Invoke* or COM interoperation (flag 0x0002). If this right is granted, it is asserted every time the unmanaged code is invoked, which results in a significant performance hit. To avoid this, the custom attribute *System.Security.SuppressUnmanagedCodeSecurityAttribute* can be used. The presence of this attribute suppresses the repetitive security checks when the unmanaged code is invoked.
  - ❑ *SkipVerification* defines the right to run the code without the IL verification procedures at JIT compilation time (flag 0x0004). *This is an extremely dangerous right.* To avoid inviting trouble, this right should be granted only to code that is known to be safe and that comes from a trusted source.
  - ❑ *Execution* defines the right to run the code (flag 0x0008). This right, which is granted to almost any code, is the opposite of *SkipVerification*. The right can be revoked by the administrator or by user security policies regarding specific applications or specific sources that are known for or suspected of being the purveyors of malicious code.

- *ControlThread* defines the right to perform thread control operations, such as suspending, interrupting, stopping a thread, changing the thread priority, and so on (flag 0x0010).
- *ControlEvidence* defines the right of the domain host to give evidence to the applications loaded in the domains created by this host (flag 0x0020). The evidence in question usually includes information about the origin and strong name signature of the loaded assembly. If the domain host does not have this right, it gives its own evidence instead.
- *ControlPolicy* defines the right to access and modify security policies, both user-specific and machinewide (flag 0x0040). *This is another extremely dangerous right that must be granted with great caution.*
- *SerializationFormatter* defines the right to perform the serialization formatting operations and to retrieve and change the characteristics of any nontransient members of the serializable types, regardless of the accessibility of these members (flag 0x0080). This permission resembles *ReflectionPermission* in the sense that both are of very low opinion about the accessibility rules and allow you to access and invoke private class members at will.
- *ControlDomainPolicy* defines the right of the domain host to specify domainwide security policy (flag 0x0100).
- *ControlPrincipal* defines the right to replace the *Principal* object (carrying the user's identity characteristics) for a given thread, for example, in order to implement role-based security (flag 0x0200). In the role-based security model, the security actions depend on the identity (*Principal* object) of the "code runner" and the role in which the "code runner" operates.
- *ControlAppDomain* defines the right to create and manipulate the application domains (flag 0x0400).
- *RemotingConfiguration* defines the right to configure the remoting types and channels (flag 0x0800).
- *Infrastructure* defines the right to plug the code into the common language runtime infrastructure, such as adding remoting context sinks, envoy sinks, and dynamic sinks (flag 0x1000).
- *Flags* is a summary binary representation of the 13 rights just listed. The validity mask is 0x1FFF.

Identity Permissions

The access permissions describe the resources to be accessed and the actions to be performed. The identity permissions describe the identities of the agents who are accessing these resources and performing these actions. As a trivial example, suppose that you've created a method or a component so atrocious that you want only components written by your company to be able to access it, because you can't trust anyone else to keep the beast in check.

It's a good practice to use identity permissions to extend rather than limit the rights granted to the code of a specific origin. Limiting the rights on the basis of the code's identity is a poor protection technique because the identity information of the code can easily be suppressed. A software publisher you particularly dislike can simply neglect to sign its malicious software, for instance, and you'll never know that this particular code must be treated with extra caution. Or the obnoxious snooping marketing site you'd love to block can start operating through a different Web server or spoof its IP address.

The five identity permissions all belong to the namespace *System.Security.Permissions* and are defined in the Mscorlib.dll assembly:

- ***ZoneIdentityPermission*** This permission identifies the zone from which the calling code originates. The zones are defined and mapped from the URLs by APIs of *IInternetSecurityManager* and related interfaces. The zones are not overlapping, and any particular URL can belong to only one zone. The attribute class has one property, *Zone* (*int32*-based enumeration *[mscorlib]System.Security.SecurityZone*). The values of the enumeration are as follows:
  - ❑ *MyComputer* (0x0) means that the application is run from the local drive.
  - ❑ *Intranet* (0x1) means that the application is run from a closed intranet.
  - ❑ *Trusted* (0x2) means that the application is run from a trusted server.
  - ❑ *Internet* (0x3) means that the application originates from the Internet.
  - ❑ *Untrusted* (0x4) means that the application's origin is suspicious and that a high level of security is required.
  - ❑ *NoZone* (0xFFFFFFFF) means that no zone information is available.

- ***StrongNameIdentityPermission*** This permission identifies an assembly by its strong name attributes—namely, by the assembly name, the assembly version, and the public encryption key. The public encryption key of the assembly must exactly match the one specified in the permission.

  The assembly name, however, might only partially match the one specified in the permission because a wildcard character (*) can be used in the assembly name specification in the permission. The name of the assembly is usually a dotted name, such as *System.DirectoryServices*, and any right part of the name can be replaced with the wildcard character. Thus, *System.DirectoryServices* denotes this specific assembly only, *System.** denotes any assembly whose name starts with *System.* (including the assembly *System*), and * denotes any assembly. If, for example, the permission includes the Microsoft private encryption key and the assembly name is given as *System.DirectoryServices*, the permission identifies the assembly *System.DirectoryServices* from the .NET Framework. If the assembly name included is *System.**, the permission identifies it as any Microsoft assembly whose name begins with *System.* If the assembly name is given simply as *, the permission identifies it as any assembly produced and signed by Microsoft. It is illegal to replace the left part of the name with the wildcard character (for example, **.DirectoryServices*).

  The assembly version includes four components: the major version, the minor version, the build number, and the revision number. The fourth or both the third and the fourth components can be omitted, but the first two components must be specified, unless the version is not specified at all. The attribute class has three properties, all of type *string*:

  - ❑ *Name* is the name of the assembly, possibly with a wildcard character in the right part.
  - ❑ *PublicKey* is the encoded hexadecimal representation of the public encryption key.
  - ❑ *Version* is the literal representation of the version—for example, 1.12.123.1 or 1.12.

- ***PublisherIdentityPermission*** This permission specifies the software publisher's identity, based on the public key defined by an X509v3 certificate. This certificate is issued by a trusted certification authority and contains encrypted information authenticating the publisher's public encryption key. The name of the publisher is ignored. The associated attribute class has three properties, all of type *string*. Only one of the properties can be set, because they represent alternate ways of obtaining the certificate:

- ❑ *X509Certificate* contains the explicit X509v3 certificate in a coded form.
- ❑ *CertFile* contains the name of the file containing the certificate.
- ❑ *SignedFile* contains the name of the file strong name signed with this certificate, so that the certificate can be obtained from the file's strong name signature.

- ***SiteIdentityPermission*** This permission identifies the Web site from which the code originates. The attribute class has one property, *Site*, of type *string*, which contains part of the Web site's URL with a stripped protocol specification at the start and the filename at the end—for example, *www.microsoft.com* in the URL *http://www.microsoft.com/ms.htm*. The protocol is presumed to be HTTP, HTTPS, or FTP. The wildcard character (*) is allowed in the site specifications, this time as the left part of the specification.
- ***UrlIdentityPermission*** This permission identifies the full URL of the site from which the code originates. The attribute class has one property, *Url*, of type *string*, which contains the full URL specification, including the protocol specification and file specification—for example, *http://oursite.microsoft.com/apps/foo/zzz.html*. The wildcard character is permitted, this time as the right part of the specification—for example, *http://oursite.microsoft.com/apps/foo/**.

Custom Permissions

The custom permissions, similar to those already defined in the .NET Framework class library, describe access rights to various resources. Once defined, a custom permission can be used in exactly the same way as any "standard" permission. Custom permissions are introduced, as a rule, when it's necessary to describe access to some new kind of resource not covered by existing permissions, such as a new input or output device.

Tip It's a bad practice to try to redefine existing permissions as custom permissions. It is possible to do so, but having multiple permissions pertaining to the same resource can only create pain for the system administrators, who must then keep an eye on all alternative "doors" leading to the resource. As a matter of practical advice: don't make system administrators any unhappier than they already are; it might cost you.

To define a custom permission, you'll need to do the following:

1. Define the new permission class.
2. Define constructors and methods of the permission class according to the permission semantics.
3. Define the methods implementing the *[mscorlib]System.Security.IPermission* interface: *Copy*, *Intersect*, *Union*, *IsSubsetOf*, and *Demand*.
4. If, in principle, full access to the resource can be granted, define the *IsUnrestricted* method implementing *[mscorlib]System.Security.IUnrestrictedPermission*.
5. Define the methods implementing the *[mscorlib]System.Security.ISecurityEncodable* interface that provide the XML encoding and decoding of the permission: *FromXml*, *ToXml*.
6. If necessary, define the *GetObjectData* method implementing the *[mscorlib]System.Runtime.Serialization.ISerializable* interface.
7. Define the accompanying attribute class.
8. Add support for declarative security.
9. Add the mechanism enforcing the permission wherever the associated resource is exposed.
10. Modify the security policies to take your new permission into account.

Needless to say, the preceding list is meant to discourage you from defining custom permissions....

The best way to define a custom permission is to pick a standard permission whose semantics resemble your intended semantics most closely and use it as an example. It's always a good idea to derive the custom permission classes from *[mscorlib]System.Security.CodeAccessPermission* and the accompanying attribute classes from *[mscorlib]System.Security.Permissions.CodeAccessSecurityAttribute.*

One of the major design problems in defining a custom permission is the question of the granularity of the resource access description. In other words, what level of detail is adequate to describe the protected resource? If you were designing *RegistryPermission*, for example, your choice of granularity could range from a 1-bit indication of whether or not full access to the registry is granted to a detailed description of a specific kind of access to a specific registry node.

Generally, four basic principles should guide your approach to permission granularity:

- **Total Boolean**, which grants or denies access to the resource.
- **Total enumerated**, which grants one of the specified (enumerated) forms of access to the resource.
- **Listed Boolean**, which grants or denies access to the resource components listed in the permission declaration.
- **Listed enumerated**, which grants one of the specified forms of access to the resource components listed in the permission.

Although additional questions might arise about the level of detail involved in the access form enumeration and the resource components list, the four basic principles, I think, stand. You are welcome to introduce a fifth and put me to shame.

A custom permission class must implement the *ISecurityEncodable* interface with its methods *ToXml* and *FromXml*, to encode the permission in XML form and restore the permission object from the XML text. The outermost tag of the XML encoding is *Permission*:

```
<Permission class="MyPermission, MyOtherAssembly.dll"  version="1">
      ⋮
</Permission>
```

To support the declarative security mechanism built into the common language runtime, the custom permission class must be accompanied by the attribute class. The attribute class must have properties that correspond to the parameters of the permission class's constructors. The attribute class must also implement at least one variant of the *CreatePermission* method. The custom attribute *System.AttributeUsageAttribute* must be assigned to the attribute class, defining its possible targets, inheritance, and multiplicity, as described in the section "Classification of Custom Attributes," in Chapter 13, "Custom Attributes."

Enforcing a newly created custom permission is the easy part; the items dealing with the new resource must create security objects from the custom permission and also security actions, such as *Demand*, *Assert*, and so on. The simplest way to do this is to assign the security custom attribute to the respective item.

The last step in creating a custom permission is updating the security policies to include the permission. This is done by writing an XML descriptor of the custom permission and invoking the code access security policy tool, Caspol.exe:

```
caspol  -addset  cust_perm.xml  cust_perm_name
```

Then, again by using the Caspol.exe utility, a new code group must be added or the existing one changed, to specify the code identities that will be granted the custom permission. Operation of the Caspol utility is rather complicated and well beyond the scope of this book; for information, you can refer to the documentation on Caspol and security administration included in the .NET Framework SDK.

Permission Sets

Individual permission objects (the instances of the permission classes) can be combined into permission sets. A permission set is an instance of the *[mscorlib]System.Security.PermissionSet* class or of the *[mscorlib]System.Security.NamedPermissionSet* class, which is derived from the former. A permission set can be constructed, for example, by combining all permissions relevant to a certain resource or to a certain metadata item (the assembly, a class, or a method).

The *PermissionSet* class, after its constituent permission classes, implements the interface *IPermission* with its methods *Copy*, *Intersect*, *Union*, *IsSubsetOf*, and *Demand*.

The unnamed permission sets constructed on a per-data-item basis, pertinent to a certain type of security action, form the metadata representation of the declarative security.

Declarative Security Metadata

The declarative security metadata resides in the metadata table DeclSecurity. A record in this table has the three entries described in the following list.

- ***Action* (2-byte unsigned integer)** The security action code.
- ***Parent* (coded token of type *HasDeclSecurity*)** The index to the Assembly, TypeDef, or Method metadata table, indicating the metadata item with which the *DeclSecurity* record is associated.
- ***PermissionSet* (offset in the *#Blob* stream)** Encoded representation of the permission set associated with a specific security action and a specific metadata item.

The following security action codes and their respective ILAsm keywords are defined for the security actions listed in the "Declarative Actions" section of this chapter and for special-purpose security actions:

- *Request*: code 0x0001, ILAsm keyword *request*.
- *Demand*: code 0x0002, ILAsm keyword *demand*.
- *Assert*: code 0x003, ILAsm keyword *assert*.
- *Deny*: code 0x0004, ILAsm keyword *deny*.
- *Permit Only*: code 0x0005, ILAsm keyword *permitonly*.
- *Link Demand*: code 0x0006, ILAsm keyword *linkcheck*.
- *Inheritance Demand*: code 0x0007, ILAsm keyword *inheritcheck*.
- *Request Minimum*: code 0x0008, ILAsm keyword *reqmin*.
- *Request Optional*: code 0x0009, ILAsm keyword *reqopt*.
- *Request Refuse*: code 0x000A, ILAsm keyword *reqrefuse*.
- *Pre-JIT Grant* (persisted grant, set at pre-JIT compilation time by the Ngen.exe utility): code 0x000B, ILAsm keyword *prejitgrant*.
- *Pre-JIT Deny* (persisted denial, set at pre-JIT compilation time): code 0x000C, ILAsm keyword *prejitdeny*. This security action is not supported in the first release of the common language runtime.
- *Non-CAS Demand*: code 0x000D, ILAsm keyword *noncasdemand*. This action is similar to *Demand*, but the permission classes that make up the permission set must not be derived from *System.Security.Permissions.CodeAccessPermission*.
- *Non-CAS Link Demand*: code 0x000E, ILAsm keyword *noncaslinkdemand*. This action is similar to *Link Demand* but has the same limitation as *Non-CAS Demand*.
- *Non-CAS Inheritance Demand*: code 0x000F, ILAsm keyword *noncasinheritance*. This action is similar to *Inheritance Demand* but has the same limitation as *Non-CAS Demand*.

The blob indexed in the *PermissionSet* entry of the *DeclSecurity* record contains an encoded representation of the permission set object. In the first release of the common language runtime, the blob contains a Unicode-encoded XML description of the permission set.

Security Attribute Declaration

ILAsm syntax offers two forms of security attribute declaration: as separate permissions and as permission sets. The owner of the security attribute is the item whose scope contains the security attribute declaration. The syntax for the permission declaration is as follows:

```
.permission <sec_action> <class_ref> [(<name_value_pairs>)]
```

where *<sec_action>* is one of the security action keywords listed in the preceding section, *<class_ref>* is a class reference to the attribute class associated with the permission class, and optional *<name_value_pairs>* define the values of the attribute class's properties, as shown here:

```
<name_value_pairs> ::= <nv_pair>[,<nv_pair>*]
<nv_pair> ::= <prop_name> = <prop_value>
```

<prop_name> is the property name of the attribute class, specified as a quoted string. The form of *<prop_value>* depends on the type of property:

```
<prop_value> ::= true | false // For Boolean properties
     | <int32> | int32(<int32>) // For integer properties
   | <class_ref> (<int32>) // For enumerated properties,
                 // <class_ref> specifies the enumerator
   | <class_ref>(<int_type> : <int32>) // <int_type>::=int8
                                       // | int16 | int32
   | <quoted_string> // For string properties
```

For example:

```
.method private void WriteToSystemDrive(string Str2BWritten)
{
     .permission demand
     [mscorlib]System.Security.Permissions.FileIOPermissionAttribute

          = ("Write"="C:\\")
     ⋮
}
```

The ILAsm compiler combines separate *.permission* declarations into permission sets before emitting the DeclSecurity metadata. However, a permission set can be declared explicitly using

```
.permissionset <sec_action> = ( <hexbytes> )
```

where *<hexbytes>* is a byte array representing the *PermissionSet* blob. This byte array is usually fairly long—a "live" example would take a couple of pages. To see such an example, you can simply disassemble any .NET Framework assembly (Mscorlib.dll or System.dll, for instance) and have a look.

Given that the *PermissionSet* blob is in fact a Unicode-encoded XML representation of the permission set, use of *<hexbytes>* in the permission set declaration is another obvious shortcoming of ILAsm, which should be corrected in future releases.

The IL Disassembler always uses the *.permissionset* directive to reflect the *DeclSecurity* metadata records.

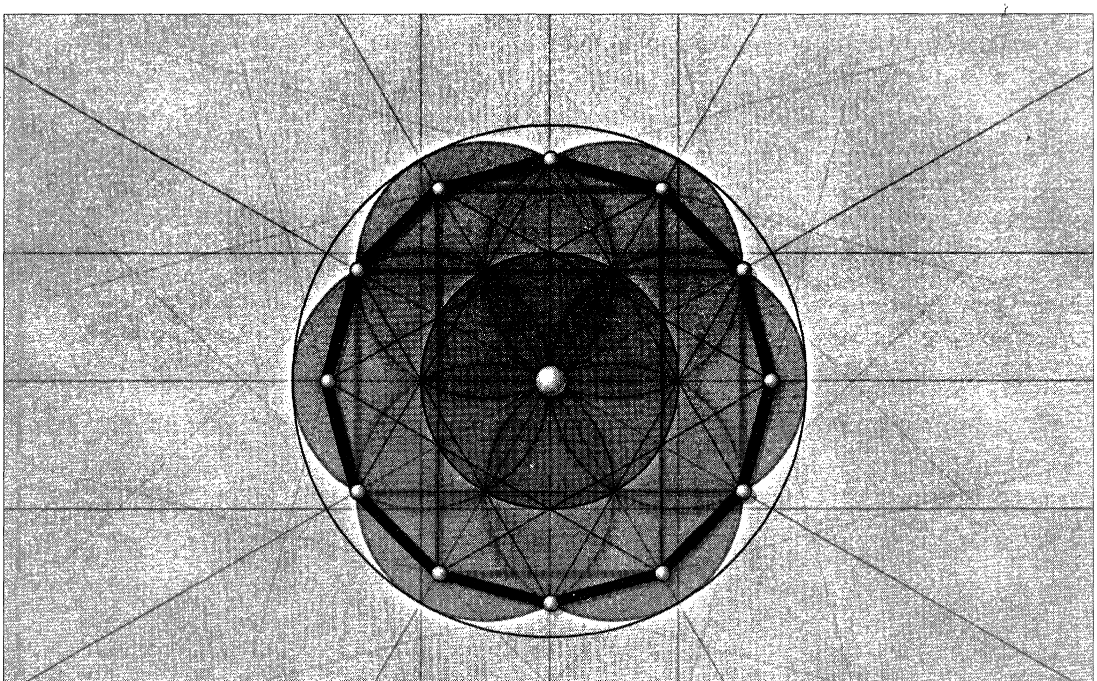
Metadata Validity Rules

A record of the DeclSecurity metadata table has three entries: *Action*, the security action code; *Parent*, the metadata item to which the security record is attached; and *PermissionSet*, the blob containing the XML descriptor of the permission set. The metadata validity rules for the *DeclSecurity* metadata records are as follows:

- [run time] The *Action* entry must hold a valid security action code in the range 0x1 through 0xF.
- The *Parent* entry must hold a valid reference to the Assembly, TypeDef, or Method tables.
- If the *Parent* entry references a *TypeDef* record, this record must not define an interface.
- If the *Parent* entry references a *TypeDef* or *Method* record, the metadata item referenced in the *Parent* entry must have its respective *HasSecurity* flag set (0x00040000 for *TypeDef* records, 0x4000 for *Method* records).
- [run time] The *PermissionSet* entry must hold a valid offset in the *#Blob* heap. The blob at this offset must contain a legal XML representation of the permission set, Unicode-encoded.

Part VI

Interoperation



15

Managed and Unmanaged Code Interoperation

There can be no question about the need to provide seamless interoperation between managed and unmanaged code, and I'm not going to waste time discussing this obvious point.

Depending on the type and the role of the unmanaged code, managed and unmanaged code can interoperate in several different scenarios. First of all, the unmanaged code participating in the interoperation can be either "traditional" code, exposed as a set of functions, or classic COM code, exposed as a set of COM interfaces. Second, the unmanaged code can play the role of either a server, with the managed code initiating the interaction, or a client, with the unmanaged code initiating the interaction. Third, the unmanaged code can reside in a separate executable, or it can be embedded in the managed module. The embedding option exists only for a "traditional" unmanaged server, and its use is limited to the specifics of the Microsoft Managed C++ (MC++) compiler implementation.

These three dichotomies result in the classification of the interoperation scenarios shown in Figure 15-1.

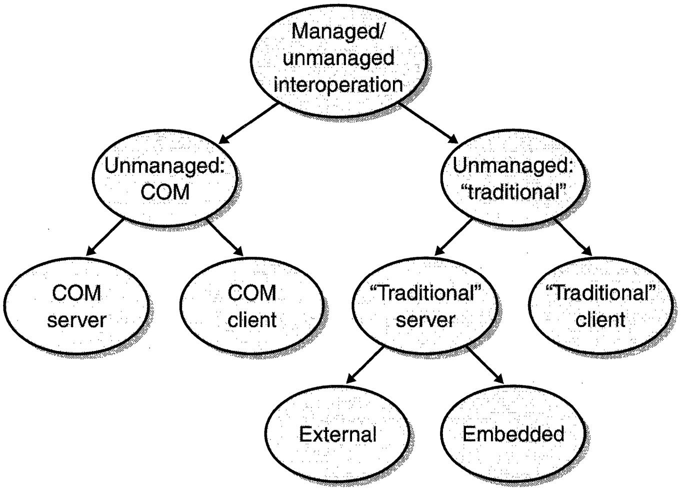


Figure 15-1 A classification of interoperation scenarios.

We have five basic scenarios here:

- An external COM server, implemented through the COM interoperability subsystem of the common language runtime and runtime callable wrappers (RCW)
- An external COM client, implemented through the same subsystem and COM callable wrappers (CCW)
- An external "traditional" server, implemented through the platform invocation (*P/Invoke*) subsystem of the runtime
- An embedded "traditional" server, implemented through a special case of *P/Invoke* known as IJW ("it just works")
- An external "traditional" client, implemented through the unmanaged export of the managed methods

Thunks and Wrappers

The interoperation between managed and unmanaged code requires the common language runtime to build special interface elements that provide the target identification and necessary data conversion, or marshaling. These runtime-generated interface elements are referred to as *thunks*, or *stubs*, in interoperation with "traditional" unmanaged code; in COM interoperation, they are referred to as *wrappers*.

P/Invoke Thunks

In order to build a client thunk for managed code to call unmanaged code, the common language runtime needs the following information:

- The name of the module exporting the unmanaged method—for example, Kernel32.dll
- The exported method's name or ordinal in the export table of the unmanaged module
- Binary flags reflecting specifics of how the unmanaged method is called and marshaled

All these items constitute the metadata item known as an *implementation map*, discussed in the following section.

The binary flag values and the respective IL assembly language (ILAsm) keywords are as follows:

- ***nomangle* (0x0001)** The exported method's name must be matched literally.
- ***ansi* (0x0002)** The method parameters of type *string* must be marshaled as ANSI zero-terminated strings unless explicitly specified otherwise.
- ***unicode* (0x0004)** The method parameters of type *string* must be marshaled as Unicode strings.
- ***autochar* (0x0006)** The method parameters of type *string* must be marshaled as ANSI or Unicode strings, depending on the underlying platform.
- ***lasterr* (0x0040)** The native method supports the last error querying by the Win32 API *GetLastError*.
- ***winapi* (0x0100)** The native method uses the calling convention standard for the underlying platform.
- ***cdecl* (0x0200)** The native method uses the C/C++–style calling convention, and the call stack is cleaned up by the caller.
- ***stdcall* (0x0300)** The native method uses the standard Win32 API calling convention, and the call stack is cleaned up by the callee.
- ***thiscall* (0x0400)** The native method uses the C++ member method (non-*vararg*) calling convention. The call stack is cleaned up by the callee, and the instance pointer (*this*) is pushed on the stack last.
- ***fastcall* (0x0500)** The arguments are passed to the native method in registers when possible.

The name of the exported method can be replaced with the method's ordinal in the unmanaged module's export table. The ordinal is specified as a decimal number, preceded by the # character—for example, *#10*.

If the specified name is a regular name rather than an ordinal, it is matched to the entries of the Export Name table of the unmanaged module. If the *nomangle* flag is set, the name is matched literally. Otherwise, things get more interesting.

Let's suppose, for example, that the name is specified as *Hello*. If the strings are marshaled to ANSI and the Export Name table does not contain *Hello*, the *P/Invoke* mechanism tries to find *HelloA*. If the strings are marshaled as Unicode, the *P/Invoke* mechanism looks for *HelloW*; only if *HelloW* is not found does *P/Invoke* look for *Hello*. If it still can't find a match, it tries the mangled name *_Hello@N*, where *N* is a decimal representation of the size of the method's destination buffer in bytes. The destination buffer is the buffer holding all method parameters. For example, if method *Hello* has two 4-byte parameters (either integer or floating-point), the mangled name would be *_Hello@8*. Because this kind of function name mangling is characteristic only of the *stdcall* functions, if the calling convention is different and the name is mangled in some other way, the *P/Invoke* mechanism will not find the exported method.

The thunk is perceived by the managed code as simply another method, and hence it must be declared as any method would be. The presence of the *pinvokeimpl* flag in the respective *Method* record signals the runtime that this method is indeed a client thunk and not a true managed method. You have already encountered the following declaration of a *P/Invoke* thunk in Chapter 1, "Simple Sample":

```
.method public static pinvokeimpl("msvcrt.dll" cdecl)
   vararg int32 sscanf(string,int8*) cil managed { }
```

The parameters within the parentheses of the *pinvokeimpl* clause represent the implementation map data. The string marshaling flag is not specified, and the marshaling defaults to ANSI. The method name need not be specified because it is the same as the declared thunk name. If you want to use *sscanf* but would rather call it *Foo* (*sscanf* is such a reptilian name!), you could declare the thunk as follows:

```
.method public static pinvokeimpl("msvcrt.dll" as "sscanf" cdecl)
   vararg int32 Foo(string,int8*) cil managed { }
```

Because the unmanaged method resides somewhere else and the thunk is generated by the runtime, the *Method* record of a "true" *P/Invoke* thunk has its *RVA* entry set to 0.

Implementation Map Metadata and Validity Rules

The implementation map metadata resides in the ImplMap metadata table. A record in this table has four entries:

- ***MappingFlags* (unsigned 2-byte integer)** Binary flags, which were described in the previous section. The validity mask is 0x0747.
- ***MemberForwarded* (coded token of type *MemberForwarded*)** An index to the Method table, identifying the *Method* record of the *P/Invoke* thunk. This must be a valid index. The indexed method must have the *pinvokeimpl* and *static* flags set. The token type *MemberForwarded* can, in principle, index the Field table as well; but the first release of the common language runtime does not implement the *P/Invoke* mechanism for fields, and ILAsm syntax does not permit you to specify *pinvokeimpl(...)* in field definitions.
- ***ImportName* (offset in the *#Strings* stream)** The name of the unmanaged method as it is defined in the export table of the unmanaged module. The name must be nonempty and fewer than 1024 bytes long in UTF-8 encoding.
- ***ImportScope* (record index [RID] to the ModuleRef table)** The index of the *ModuleRef* record containing the name of the unmanaged module. It must be a valid RID.

IJW Thunks

IJW thunks, similar in structure and function to "true" *P/Invoke* thunks, are created without the implementation map information. The information regarding the identity of the unmanaged method is not needed because the method is embedded in the same PE file and can be identified by its relative virtual address (RVA). IJW thunks *cannot* have an RVA value of 0, as opposed to *P/Invoke* thunks, which *must* have an RVA value of 0.

The calling convention of the unmanaged method is defined by the thunk signature rather than by the binary flags of the implementation map. The IJW thunk signature usually has the modifier *modopt* or *modreq*—for example, *modopt([mscorlib]System.Runtime.InteropServices.CallConvCdecl)*. The string marshaling default is *ansi*.

To distinguish IJW thunks from *P/Invoke* thunks, the loader first looks at the implementation flags; IJW thunk declarations should have the flags *native* and *unmanaged* set. If the loader doesn't see these flags, it presumes that this is a "true" *P/Invoke* thunk and tries to find its implementation map. If the map is not found, the loader realizes that this is an IJW thunk after all and proceeds accordingly. That's why I noted that the *native* and *unmanaged* flags *should* be set rather than specified that they *must* be set. The loader will discover the truth even without these flags, but not before it tries to find the implementation map and fails.

The following is a typical example of an IJW thunk declaration; it is a snippet from a disassembly of an MC++-generated mixed-code PE file:

```
.method public static pinvokeimpl(/* No map */)
   unsigned int32  _mainCRTStartup() native unmanaged preservesig
{
   .entrypoint
   .custom instance void [mscorlib]
      System.Security.SuppressUnmanagedCodeSecurityAttribute::.ctor()
      = ( 01 00 00 00 )
   // Embedded native code
   // Disassembly of native methods is not supported
   // Managed TargetRVA = 0x106f
}  // End of global method _mainCRTStartup
```

As you can see, a thunk can be declared as an entry point, and custom attributes and security attributes can be assigned to it. In these respects, a thunk has the same privileges as any other method.

As you can also see, neither the IL Disassembler nor ILAsm can handle the embedded native code. The mixed-code PE files, employing the IJW interoperation, cannot be round-tripped (disassembled and reassembled).

COM Callable Wrappers

Classic COM objects are allocated from the standard operating system heap and contain internal reference counters. The COM objects must self-destruct when they are not referenced any more—in other words, when their reference counters reach 0.

Managed objects are allocated from the common language runtime internal heap, which is controlled by the garbage collection subsystem (the GC heap). Managed objects don't have internal reference counters. Instead, the runtime traces all the object references, and the GC automatically destroys unreferenced objects. But the references can be traced only if the objects are being referenced by managed code. Hence, it would be a bad idea to allow unmanaged COM clients to access managed objects directly.

Instead, for each managed object, the runtime creates a COM callable wrapper, which serves as a proxy for the object. Because a CCW is not subject to the GC mechanism, it can be referenced from unmanaged code without causing any ill effects.

In addition to lifetime control of the managed object, a CCW provides data marshaling for method calls and handles managed exceptions, converting them to *HRESULT* returns, which is standard for COM. If, however, a managed method is designed to return *HRESULT* (in the form of *unsigned int32*) rather than throw exceptions, it must have the implementation flag *preservesig* set. In this case, the method signature is exported exactly as defined.

The runtime carefully maintains a one-to-one relationship between a managed object and its CCW, not allowing an alternative CCW to be created. This guarantees that all interfaces of the same object relate to the same *IUnknown* and that the interface queries are consistent.

Any CCW generated by the runtime implements *IDispatch* for late binding. For early binding, which is done directly through the v-table, the runtime must generate the type information in a form consumable by COM clients—namely, in the form of a COM type library. The Microsoft .NET Framework SDK includes the type library exporting utility TlbExp.exe, which generates an accompanying COM type library for any specified assembly. Another tool, RegAsm.exe, also included in the .NET Framework SDK, registers the types exposed by an assembly as COM classes and generates the type library.

When managed classes and their members are exposed to COM, their exposed names might differ from the originals. First, the type library exporters consider all names that differ only in case to be a single form—for example, *Hello*, *hello*, *HELLO*, and *hElLo* are exported as *Hello*. Second, classes are exported by name only, without the namespace part, except in the case of a name collision. If a collision exists—if, for example, an assembly has classes *A.B.IHello* and *C.D.IHello* defined—the classes are exported by their full names, with underscores replacing the dots: *A_B_IHello*, *C_D_IHello*.

Other COM parameters characterizing the CCW for each class are defined by the COM interoperability custom attributes, listed in the section "Custom Attribute Classification" in Chapter 13, "Custom Attributes." Because all information pertinent to exposing managed classes as COM servers is defined through custom attributes, ILAsm does not have any linguistic constructs specific to this aspect of the interoperation.

Runtime Callable Wrappers

A runtime callable wrapper is created by the common language runtime as a proxy of a classic COM object that the managed code wants to consume. The reasons for creating an RCW are roughly the same as those for creating a CCW: the managed objects know nothing about reference counting and expect their counterparts to belong to the GC heap. An RCW is allocated from the GC heap and caches the reference-counted interface pointers to a single COM object. In short, from the runtime point of view, an RCW is a "normal" managed server; and from the COM point of view, RCW is a "normal" COM client. So everyone's happy.

An RCW is created when a COM object is instantiated—for example, by a *newobj* instruction. There are two approaches to binding to the COM classes: early binding, which requires a so-called interop assembly, and late binding by name, which is performed through Reflection methods.

An *interop assembly* is a managed assembly either produced from a COM type library by means of running the utility TlbImp.exe (included in the .NET Framework SDK) or, at run time, produced by calling methods of the class *[mscorlib]System.Runtime.InteropServices.TypeLibConverter*. From the point of view of the managed code, the interop assembly is simply another assembly, all classes of which happen to carry the *import* flag. This flag is the signal for the runtime to instantiate an RCW every time it is commanded to instantiate such a class.

Late binding through Reflection works in much the same way as *IDispatch* does, but it has nothing to do with the interface itself. The COM classes that implement *IDispatch* can be early-bound as well. Neither is late binding restricted to imported classes only. "Normal" managed types can also be late-bound by using the same mechanism.

Late binding is achieved by consecutive calls to the *[mscorlib]System.Type::GetTypeFromProgID* and *[mscorlib]System.Activator::CreateInstance* methods, followed when necessary by calls to the *[mscorlib]System.Type::InvokeMember* method. For example, if you want to instantiate a COM class *Bar* residing in the COM library Foo.dll and then call its *Baz* method, which takes no arguments and returns an integer, you could write the following code:

```
⋮
.locals init (class [mscorlib]System.Type Typ,
              object Obj,
              int32 Ret)
// Typ = Type::GetTypeFromProgID("Foo.Bar");
ldstr "Foo.Bar"
call class [mscorlib]System.Type
     [mscorlib]System.Type::GetTypeFromProgID(string)
stloc Typ

// Obj = Activator::CreateInstance(Typ);
ldloc Typ
call instance object [mscorlib]System.Activator::CreateInstance(
     class [mscorlib]System.Type)
stloc Obj
⋮
// Ret = (int)Typ->InvokeMember("Baz",BindingFlags::InvokeMethod,
//                              NULL,Obj,NULL);
```

```
ldloc Typ
ldstr "Baz"
ldc.i4 0x100  // System.Reflection.BindingFlags::InvokeMethod
ldnull        // Reflection.Binder - don't need it
ldloc Obj
ldnull        // Parameter array - don't need it
call instance object [mscorlib]System.Type::InvokeMember(string,
              valuetype System.Reflection.BindingFlags,
              class System.Reflection.Binder,
              object,
              object[])
unbox valuetype [mscorlib]System.Int32
stloc Ret
⋮
```

An RCW converts the *HRESULT* returns of COM methods to managed exceptions. The only problem with this is that the RCW throws exceptions only for failing *HRESULT* values, so subtleties such as *S_FALSE* go unnoticed. The only way to deal with this situation is to set the implementation flag *preservesig* on the methods that might return *S_FALSE* and revert their signatures to the original form.

Another problem arises when the COM method has a variable-length array as one parameter and the array length as another. The type library carries no information about which parameter is the length, and the runtime is thus unable to marshal the array correctly. In this case, the signature of the method must be modified to include explicit marshaling information.

Yet another problem requiring manual intervention involves unions with overlapped reference types. Perfectly legal in the unmanaged world, such unions are outlawed in managed code. Therefore, these unions are converted into value types with *.pack* and *.size* parameters specified but without the member fields.

The manual intervention mentioned usually involves disassembling the interop assembly, editing the text, and reassembling it. Because the interop assemblies don't contain embedded native code, this operation can easily be performed.

Data Marshaling

All thunks and wrappers provide data conversions between managed and unmanaged data types, which is referred to as *marshaling*. Marshaling information is kept in the FieldMarshal metadata table, which is described in Chapter 8, "Fields and Data Constants." The marshaling information can be associated with *Field* and *Param* metadata records.

Blittable Types

One significant subset of managed data types directly corresponds to unmanaged types, requiring no data conversion on managed and unmanaged code boundaries. These types, which are referred to as *blittable*, include pointers (*not* references), function pointers, signed and unsigned integer types, and floating-point types. Formatted value types (the value types having sequential or explicit class layout) that contain only blittable elements are also blittable.

The nonblittable managed data types that might require conversion during marshaling because of different or ambiguous unmanaged representation are as follows:

- *bool* (1-byte, *true* = 1, *false* = 0) can be converted either to native type *bool* (4-byte, *true* = 1, *false* = 0) or to *variant bool* (2-byte, *true* = 0xFFFF, *false* = 0).
- *char* (Unicode character, unsigned 2-byte integer) can be converted either to *int8* (an ANSI character) or to *unsigned int16* (a Unicode character).
- *string* (class *System.String*) can be converted either to an ANSI or a Unicode zero-terminated string (an array of characters) or to *bstr* (a Unicode Visual Basic–style string).
- *object* (class *System.Object*) can be converted either to a structure or to an interface pointer.
- *class* can be converted either to an interface pointer or, if the class is a delegate, to a function pointer.
- *valuetype* (nonblittable) is converted to a structure with a fixed layout.
- An array and a vector can be converted to a safe array or a C-style array.

The references (managed pointers) are marshaled as unmanaged pointers. The managed objects and interfaces are references in principle, so they are marshaled as unmanaged pointers as well. Consequently, references to the objects and interfaces (*class IFoo&*) are marshaled as double pointers (*IFoo***).

In/Out Parameters

The method parameter flags *in* and *out* can be (but are not necessarily) taken into account by the marshaler. When that happens, the marshaler can optimize the process by abandoning the marshaling in one direction. By default, parameters passed by reference (including references to objects but excluding the objects) are presumed to be *in/out* parameters, whereas parameters passed by value (including the objects, even though managed objects are in principle references) are presumed to be *in* parameters. The exceptions to this rule are the *[mscorlib]System.Text.StringBuilder* class, which is always marshaled as *in/out*, and classes and arrays containing the blittable types that can be pinned—which, if the *in* and *out* flags are explicitly specified, can be two-way marshaled even when passed by value.

Considering that managed objects don't necessarily stay in one place and can be moved any time the garbage collector does its job, it is vital to ensure that the arguments of an unmanaged call don't wander around while the call is in progress. This can be accomplished in the following two ways:

- Pin the object for the duration of the call, preventing the garbage collector from moving it. This is done for the instances of formatted, blittable classes that have fixed layout in memory, invariant to managed or unmanaged code.
- Allocate some unmovable memory. If the parameter has an *in* flag, marshal the data from the argument to this unmovable memory. Call the method, passing this memory as the argument. If the parameter has an *out* flag, marshal this memory back to the original argument upon completion of the call.

The ILAsm syntax for explicit marshaling definition of fields and method parameters is described in Chapter 8 and in Chapter 9, "Methods." Chapter 7, "Primitive Types and Signatures," discusses the native types used in explicit marshaling definitions. Rather than reviewing that information here, let's discuss some interesting marshaling cases instead.

String Marshaling

String marshaling is defined in at least three places: a string conversion flag of a *TypeDef* (*ansi*, *unicode*, or *autochar*), a similar flag of a *P/Invoke* implementation map, and, explicitly, in *marshal(…)* clauses.

As method arguments, managed strings (instances of the *System.String* class) can be marshaled as the following native types:

- *lpstr*, a pointer to a zero-terminated ANSI string
- *lpwstr*, a pointer to a zero-terminated Unicode string
- *lptstr*, a pointer to a zero-terminated ANSI or Unicode string, depending on the platform
- *bstr*, a Unicode Visual Basic–style string with a prepended length

- *ansi bstr*, an ANSI Visual Basic–style string with a prepended length
- *tbstr*, an ANSI or Unicode Visual Basic–style string, depending on the platform

The COM wrappers marshal the string arguments as *lpstr*, *lpwstr*, or *bstr* only. Other unmanaged string types are not COM-compatible.

At times, a string buffer must be passed to an unmanaged method in order to be filled with some particular contents. Passing a string by value does not work in this case because the called method cannot modify the string contents. Passing the string by reference does not initialize the buffer to the required length. The solution, then, is to pass not a *string* (an instance of *System.String*) but rather an instance of *System.Text.StringBuilder*, initialized to the required length:

```
.method public static pinvokeimpl("user32.dll" stdcall)
   int32 GetWindowText(int32 hndl,
                    class [mscorlib]System.Text.StringBuilder s,
                    int32 nMaxLen) { }
.method public static string GetWText(int32 hndl)
{
   .locals init(class [mscorlib]System.Text.StringBuilder sb)
   ldc.i4 1024 // Buffer size
   newobj instance void
      [mscorlib]System.Text.StringBuilder::.ctor(int32)
   stloc.0
   ldarg.0   // Load hndl on stack
   ldloc.0   // Load StringBuilder instance on stack
   ldc.i4 1024 // Buffer size again
   call int32 GetWindowText(int32,
            class [mscorlib]System.Text.StringBuilder,
            int32)
   pop     // Discard the result
   ldloc.0  // Load StringBuilder instance (filled in) on stack
   call instance string
        [mscorlib]System.Text.StringBuilder::ToString()
   ret
}
```

The string fields of the value types are marshaled as *lpstr*, *lpwstr*, *lptstr*, *bstr*, or *fixed sysstring[<size>]*, which is a fixed-length array of ANSI or Unicode characters, depending on the string conversion flag of the field's parent *TypeDef*.

Object Marshaling

The fields and method parameters of an object type are marshaled as *struct* (converted to a COM-style variant), *interface* (converted to *IDispatch* if possible and otherwise to *IUnknown*), *iunknown* (converted to *IUnknown*), or *idispatch* (converted to *IDispatch*). The default marshaling is as *struct.*

When an object is marshaled as *struct* to a COM variant, the type of the variant can be explicitly set by those object types that implement the *[mscorlib]System.IConvertible* interface. The types that do not implement this interface are marshaled to and from variants as shown in Table 15-1. All listed types belong to the *System* namespace.

Table 15-1 Marshaling of Managed Objects to and from COM Variants

Type of object marshaled to...	...COM variant type...	...marshaled to type of object
Null reference	*VT_EMPTY*	Null reference
DBNull	*VT_NULL*	*DBNull*
Runtime.InteropServices.ErrorWrapper	*VT_ERROR*	*UInt32*
Reflection.Missing	*VT_ERROR* with *E_PARAMNOTFOUND*	*UInt32*
Runtime.InteropServices.IdispatchWrapper	*VT_DISPATCH*	*__ComObject* or null reference if the variant value is null
Runtime.InteropServices.IunknownWrapper	*VT_UNKNOWN*	*__ComObject* or null reference if the variant value is null
Runtime.InteropServices.CurrencyWrapper	*VT_CY*	*Decimal*
Boolean	*VT_BOOL*	*Boolean*
Sbyte	*VT_I1*	*Sbyte*
Byte	*VT_UI1*	*Byte*
Int16	*VT_I2*	*Int16*
UInt16	*VT_UI2*	*UInt16*
Int32	*VT_I4*	*Int32*
UInt32	*VT_UI4*	*UInt32*
Int64	*VT_I8*	*Int64*
UInt64	*VT_UI8*	*UInt64*
Single	*VT_R4*	*Single*
Double	*VT_R8*	*Double*
Decimal	*VT_DECIMAL*	*Decimal*

(continued)

Table 15-1 Marshaling of Managed Objects to and from COM Variants *(continued)*

Type of object marshaled to...	...COM variant type...	...marshaled to type of object
DateTime	*VT_DATE*	*DateTime*
String	*VT_BSTR*	*String*
IntPtr	*VT_INT*	*Int32*
UintPtr	*VT_UINT*	*UInt32*
Array	*VT_ARRAY*	*Array*

If you wonder why, for example, *System.Int16* and *System.Boolean* should be used instead of *int16* and *bool*, respectively, I should remind you that our discussion concerns the conversion of the *objects*.

When a managed object is passed to unmanaged code by reference, the marshaler creates a new variant and copies the contents of the object reference into this variant. The unmanaged code is free to tinker with the variant contents, and these changes are propagated back to the referenced object when the method call is completed. If the type of the variant has been changed within the unmanaged code, the back-propagation of the changes can result in a change of the object type, so you might find yourself with a different type of object after the call. The same story happens (in reverse order) when unmanaged code calls a managed method, passing a variant by reference: the type of the variant can be changed during the call.

The variant can contain a pointer to its value rather than the value itself. (In this case, the variant has its type flag *VT_BYREF* set.) Such a "reference variant," passed to the managed code by value, is marshaled to a managed object, and the marshaler automatically dereferences the variant contents and retrieves the actual value. Despite its reference type, the variant is nonetheless passed by value, so any changes made to the object in the managed code are not propagated back to the original variant.

If a "reference variant" is passed to the managed code by reference, it is marshaled to an object reference, with the marshaler dereferencing the variant contents and copying the value into a newly constructed managed object. But in this case, the changes made in the managed code are propagated back to the unmanaged code only if they did not lead to a change in the variant type. If the changes did affect the variant type, the marshaler throws an *InvalidCast* exception.

Class Marshaling

Managed classes are always marshaled by COM wrappers as the interfaces. Every managed class can be seen as implementing an implicit interface that contains all nonprivate members of the class.

When a type library is generated from an assembly, a class interface and a coclass are produced for each accessible managed class. The class interface is marked as a default interface for the coclass.

A CCW generated by the common language runtime for each instance of the exposed managed class also implements other interfaces not explicitly implemented by the class. In particular, a CCW automatically implements *IUnknown* and *IDispatch*.

When an interop assembly is generated from a type library, the coclasses of the type library are converted to the managed classes. The member sets of these classes are defined by the default interfaces of the coclasses.

An RCW generated by the runtime for a specific instance of a COM class represents this instance and not a specific interface exposed by this instance. Hence, an RCW must implement all interfaces exposed by the COM object. This means that the identity of the COM object itself must be determined by one of its interfaces because COM objects are not passed as method arguments but their interfaces are. In order to do this, the runtime queries the passed interface for *IProvideClassInfo2*. If this interface is unavailable, the runtime queries the passed interface for *IProvideClassInfo*. If either of the interfaces is available, the runtime obtains the CLSID (class identifier) of the COM class exposing the interface—by calling the *IProvideClassInfo2::GetGUID()* or *IProvideClassInfo::GetClassInfo()* method—and uses it to retrieve full information about the COM class from the registry. If this action sequence fails, the runtime instantiates a generic wrapper, *System.__ComObject.*

Array Marshaling

Unmanaged arrays can be either C-style arrays of fixed or variable length or COM-style safe arrays. Both kinds of arrays are marshaled to managed vectors, with the unmanaged element type of the array marshaled to the respective managed element type of the vector. For example, a safe array of BSTR is marshaled to *string[].*

The rank and bound information carried by a safe array is lost in the transition. If this information is vital for correct interfacing, manual intervention is required again: the interop assembly produced from the COM type library must be disassembled, the array definitions must be manually edited, and the assembly must be reassembled. For example, if a three-dimensional safe array of

BSTR is marshaled as *string[]*, the respective type must be manually edited to *string[0...,0...,0...]* in order to restore the rank of the array.

C-style arrays can have either a fixed length or a length specified by another parameter of the method. Both values, the length and the length parameter's zero-based ordinal, can be specified for the marshaler so that a vector of appropriate size can be allocated. The ILAsm syntax for specifying the array length is described in Chapter 7. For example:

```
// Fixed array length
.method public static pinvokeimpl("unmanaged.dll" stdcall)
   void Foo(string[] marshal(bstr[128]) StrArray) {}

// Array length is specified by arrLen (parameter #1)
.method public static pinvokeimpl("unmanaged.dll" stdcall)
   void Boo(string[] marshal(bstr[+1]) StrArray, int32 arrLen) {}

// Base length is 128, additional length specified by moreLen
.method public static pinvokeimpl("unmanaged.dll" stdcall)
   void Goo(int32 moreLen, string[] marshal(bstr[128+0]) StrArray) {}
```

The managed vectors and arrays can be marshaled as safe arrays or as C-style arrays. Marshaling as safe arrays preserves the rank and boundary information of the managed arrays. This information is lost when the managed arrays are marshaled as C-style arrays. Vectors of vectors—for example, *int32[][]*—cannot be marshaled.

Delegate Marshaling

Delegates are marshaled as interfaces by COM wrappers and as unmanaged function pointers by *P/Invoke* thunks. The type library Mscorlib.tlb defines the *_Delegate* interface, which represents delegates in the COM world. This interface exposes the *DynamicInvoke* method, which allows the COM code to call a delegated managed method.

Marshaling a delegate as an unmanaged function pointer represents a certain risk. Because the common language runtime does not count this as a live reference to the delegate, the delegate might be destroyed by the garbage collector before the call to the unmanaged method is completed. The calling managed code must take steps to ensure the delegate's survival for the duration of the method call.

Providing Managed Methods as Callback for Unmanaged

In a *P/Invoke* interaction, the initiative must come from the managed code's side. The process starts in managed mode and makes calls to the unmanaged functions. However, the exchange can't always go in only one direction; that model would be too simplistic to be usable.

Many unmanaged methods require callback functions, and the managed code must have the means to provide those functions. Thus, it's necessary to have a way to pass a managed method pointer to an unmanaged function, permitting the unmanaged function to call the managed method. The managed method in question might be simply a *P/Invoke* thunk of another unmanaged method, but that changes nothing—it's still a managed method.

The way to pass managed methods as callbacks to unmanaged functions involves the use of delegates. The delegates are marshaled by *P/Invoke* thunks as unmanaged function pointers, which makes them suitable for the task.

Let's look at a sample to review the way delegates are used for callback specifications. You can find this sample, Callback.il, on the companion CD. The sample implements a simple program that sorts 15 integer values in ascending order, employing the well-known C function *qsort*, called through *P/Invoke*. The difference between the *P/Invoke* calls you've encountered so far and this one is that *qsort* requires a callback function, which performs the comparison of two elements of the array being sorted, thus defining the sorting order.

Let the sample speak for itself:

```
// Necessary preliminaries
.assembly extern mscorlib {}
.assembly callback { }
.module callback.exe

// Here is the unsorted data we want to sort. With minor
// improvements, I could get the data from a file or the console,
// but it would not serve any illustrative purpose.
// Feel free to try modifying the code to include this feature.
.class private value explicit sealed SixtyBytes { .pack 1 .size 60 }
.field public static valuetype SixtyBytes DataToSort at D_0001
.data D_0001 = {int32(10), int32(32), int32(-1), int32(567),
   int32(3), int32(18), int32(1), int32(-51), int32(789), int32(2345),
   int32(-43), int32(788), int32(-345), int32(345), int32(0)}

// To show that the sorting really happens, I am going to print
// out data before and after the sorting. Rather than using the
// [mscorlib]System.Console::WriteLine, I will use P/Invoke
// to call the C function int printf(char* Format,...).
.method public hidebysig static pinvokeimpl("msvcrt.dll" ansi cdecl)
```

(continued)

```
   vararg int32 printf(string) preservesig {}

// And this is a managed printing method, invoking printf.
.method public static void printInt32(void* pBuff, int32 N)
{
   .locals init(int32 i, void* pb)
   // i = 1;
   ldc.i4.1
   stloc.0
   // pb = pBuff;
   ldarg.0
   stloc.1

Next:   // if(i > N) goto Return;
   ldloc.0
   ldarg.1
   bgt Return

   // printf("%2.2d : %d\n",i,*(int*)pb);
   ldstr "%2.2d : %d\n"
   ldloc.0
   ldloc.1
   ldind.i4
   call vararg int32 printf(string,...,int32,int32)
   pop

   // i += 1;
   ldloc.0
   ldc.i4.1
   add
   stloc.0

   // pb += 4; In C, this would be an illegal operation because
   // pb is void*, and the absolute pointer increment could not be
   // calculated. In IL, however, the pointer increment is always
   // absolute.
   ldloc.1
   ldc.i4.4
   add
   stloc.1
   br Next

Return:
   ret
}

// I can't pass the managed method pointer to the unmanaged function,
// and even the ldftn instruction will not help me.
// This delegate will serve as an appropriate vehicle.
```

```
.class public sealed CompareDelegate
       extends [mscorlib]System.MulticastDelegate
{
   .method public hidebysig specialname
           instance void  .ctor(object Object,
                                 native unsigned int MethodPtr)
                                 runtime managed {}

   // Note the modopt modifier of the Invoke signature--it's very
   // important. Without it, the calling convention of the callback
   // function is marshaled as stdcall (callee cleans the stack).
   // But qsort expects the callback function to have the cdecl
   // calling convention (caller clears the stack). If we supply the
   // callback with the stdcall calling convention, qsort blows
   // the stack away and causes a memory access violation. You are
   // welcome to comment out the modopt line and see what happens.
   // Note also that the modopt modifier is placed on the delegate's
   // Invoke signature, not on the signature of the delegated method.
   .method public hidebysig virtual instance int32
      modopt([mscorlib]System.Runtime.CompilerServices.CallConvCdecl)
             Invoke(void*, void*) runtime managed {}

   // Well, I don't really need asynchronous invocation here,
   // but, you know, dura lex sed lex.
   .method public hidebysig newslot virtual
           instance class [mscorlib]System.IAsyncResult
              BeginInvoke(object,
                          class [mscorlib]System.AsyncCallback,
                          object) runtime managed {}

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

// The hero of the occasion: the qsort function.
.method public hidebysig static pinvokeimpl("msvcrt.dll" ansi cdecl)
   void qsort(void*,int32,int32,class CompareDelegate) preservesig {}

// This is the comparison method I'm going to offer as
// a callback to qsort. What can be simpler than comparing
// two integers?
.method public static int32 compInt32(void* arg1,void* arg2)
{
   // return(*arg1 - *arg2);
   ldarg.0
   ldind.i4
   ldarg.1
   ldind.i4
```

(continued)

```
   sub
   ret
}

// And now, let's put this show on the road.
.method public static void Exec()
{
   .entrypoint
   .locals init(class CompareDelegate)

   // Print the unsorted values.
   ldstr "Before Sorting:\n"
   call vararg int32 printf(string)
   pop
   ldsflda valuetype SixtyBytes DataToSort
   ldc.i4 15
   call void printInt32(void*, int32)

   // Create the delegate.
   // Null object ref indicates the global method.
   ldnull
   ldftn int32 compInt32(void*,void*)
   newobj instance void
      CompareDelegate::.ctor(object,native unsigned int)
   stloc.0

   // Invoke qsort.
   ldsflda valuetype SixtyBytes DataToSort // Pointer to data
   ldc.i4 15   // Number of items to sort
   ldc.i4 4   // Size of an individual item
   ldloc.0   // Callback function pointer
   call void qsort(void*,int32,int32,class CompareDelegate)

   // Print the sorted values.
   ldstr "After Sorting:\n"
   call vararg int32 printf(string)
   pop
   ldsflda valuetype SixtyBytes DataToSort
   ldc.i4 15
   call void printInt32(void*, int32)

   ret
}
```

Managed Methods as Unmanaged Exports

Exposing managed methods as unmanaged exports provides a way for unmanaged, non-COM clients to consume the managed services. In fact, this technique opens the managed world in all its glory—with its secure and type-safe computing and with all the wealth of its class libraries—to unmanaged clients.

Of course, the managed methods are not exposed as such. Instead, the inverse *P/Invoke* thunks, automatically created by the common language runtime, are exported. These thunks provide the same marshaling functions as "conventional" *P/Invoke* thunks, but in the opposite direction.

In order to expose managed methods as unmanaged exports, the ILAsm compiler builds a v-table, a v-table fixup (VTableFixup) table, and a group of unmanaged export tables, which include the Export Address table, the Name Pointer table, the Ordinal table, the Export Name table, and the Export Directory table. All of these tables, their structure, and their positioning within a managed PE file are discussed in Chapter 3, "The Structure of a Managed Executable File."

The VTableFixup table is an array of VTableFixup descriptors, with each descriptor carrying the RVA of a v-table entry, the number of slots in the entry, and the binary flags indicating the size of each slot (32-bit or 64-bit) and any special features of the entry. One special feature is creation of the marshaling thunk to be exposed to the unmanaged client.

The v-table and the VTableFixup table of a managed module serve two purposes. One purpose—relevant only to the MC++ compiler, the only compiler that produces mixed-code modules—is to provide the managed/unmanaged linking capabilities for mixed-code modules. Another purpose is to facilitate the unmanaged export of managed methods.

Each slot of a v-table in a PE file carries the token of the method the slot represents. At run time, the v-table fixups are executed, replacing the method tokens with actual method addresses.

The ILAsm syntax for a v-table fixup definition is as follows:

```
.vtfixup [<num_slots>] <flags> at <data_label>
```

where square brackets are part of the definition and do not mean that *<num_slots>* is optional. *<num_slots>* is an integer constant, indicating the number of v-table slots grouped into one entry because their flags are identical. This grouping has no effect other than saving some space—you can emit a single slot per entry, but then you'll have to emit as many v-table fixups as there are slots.

The flags specified in the definition can be those that are described in the following list.

- ***int32*** Each slot in this v-table entry is 4 bytes wide.
- ***int64*** Each slot in this v-table entry is 8 bytes wide. The *int32* and *int64* flags are mutually exclusive.
- ***fromunmanaged*** The entry is to be called from the unmanaged code, so the marshaling thunk must be created by the runtime.
- ***callmostderived*** This flag is not currently used.

The order of appearance of *.vtfixup* declarations defines the order of the respective VTableFixup descriptors in the VTableFixup table.

The v-table entries are defined simply as data entries. Note that the v-table must be contiguous—in other words, the data definitions for the v-table entries must immediately follow one another.

For example:

```
⋮
.vtfixup [1] int32 fromunmanaged at VT_01
⋮
.vtfixup [1] int32 at VT_02
⋮
.data VT_01 = int32(0x0600001A)
.data VT_02 = int32(0x0600001B)
⋮
```

The actual data representing the method tokens is automatically generated by the ILAsm compiler and placed in designated v-table slots. To achieve that, it is necessary to indicate which method is represented by which v-table slot. ILAsm provides the *.vtentry* directive for this purpose:

```
.vtentry <entry_number> : <slot_number>
```

where *<entry_number>* and *<slot_number>* are one-based integer constants. The *.vtentry* directive is placed within the respective method's scope, as shown in the following code:

```
⋮
.vtfixup [1] int32 fromunmanaged at VT_01
⋮
.method public static void Foo()
{
   .vtentry 1:1 // Entry 1, slot 1
   ⋮
}
⋮
.data VT_01 = int32(0) // The slot will be filled automatically.
⋮
```

The export table group consists of five tables:

- The Export Address table (EAT), containing the RVA of the exported unmanaged functions
- The Export Name table (ENT), containing the names of the exported functions
- The Name Pointer table (NPT) and the Ordinal table (OT), together forming a lookup table that rearranges the exported functions in lexical order of their names
- The Export Directory table, containing the location and size information about the other four tables

Location and size information concerning the Export Directory table itself resides in the first of 16 data directories in the PE header. The structure of the export table group is shown in Figure 15-2.

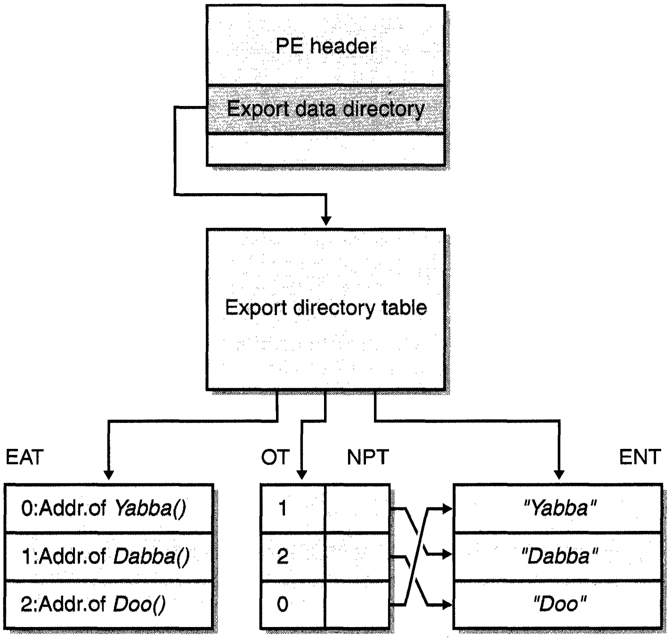


Figure 15-2 The structure of the export table group.

In an unmanaged PE file, the EAT contains the RVA of the exported unmanaged methods. In a managed PE file, the picture is more complicated. The EAT cannot contain the RVA of the managed methods because it's not the managed methods that are exported—rather, it's their marshaling thunks, generated at run time.

The only way to address a yet-to-be-created thunk is to define a slot in a v-table entry for the exported managed method and a VTableFixup descriptor for this entry, carrying the *fromunmanaged* flag. In this case, the contents of the v-table slot (a token of the exported method) are replaced at run time with the address of the marshaling thunk. (If the *fromunmanaged* flag is not specified, the thunk is not created, and the method token is replaced with this method's address; but this is outside the scenario being discussed.)

For each exported method, the ILAsm compiler creates a tiny native stub—yes, you've caught me: the ILAsm compiler *does* produce embedded native code after all—consisting of the x86 command *jump indirect* (0x25FF) followed by the RVA of the v-table slot allocated for the exported method. The EAT contains the RVA of these tiny stubs.

The tiny stubs are necessary because the EAT must contain solid addresses of the exported methods as soon as the operating system loads the PE file. Otherwise, the unmanaged client won't be able to match the entries of its Import Address table (IAT) to the entries of the managed module's EAT. The addresses of the methods or their thunks don't exist at the moment the file is loaded. But the tiny stubs exist and have solid addresses. It's true that at that moment they cannot perform any meaningful jumps, because the v-table slots they are referencing contain method tokens instead of addresses. But by the time the stubs are called, the methods and thunks will have been generated and the v-table slots will be fixed up, the method tokens replaced with thunk addresses.

The diagram shown in Figure 15-3 illustrates this scenario.

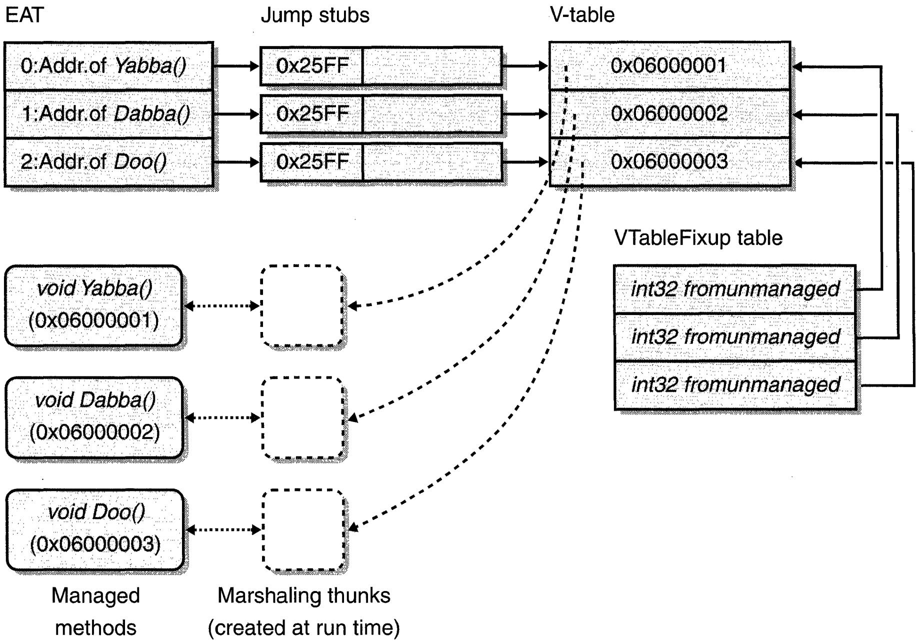


Figure 15-3 Indirect referencing of v-table entries from the EAT.

The unmanaged exports require that relocation fixups are executed at the module load time. When a program runs under the Microsoft Windows XP operating system, this requirement can create a problem similar to those encountered with thread local storage (TLS) data and data-on-data. As described in Chapter 3, if the common language runtime header flag *COMIMAGE_FLAGS_ILONLY* is set, the loader of Windows XP ignores the *.reloc* section, and the fixups are not executed. To avoid this, the ILAsm compiler automatically replaces the *COMIMAGE_FLAGS_ILONLY* flag with *COMIMAGE_FLAGS_32BITREQUIRED* whenever the source code specifies TLS data or data-on-data. Unfortunately, the compiler neglects to do this automatically when unmanaged exports are specified in the source code, and it is thus necessary to explicitly set the runtime header flags using the directive *.corflags* 0x00000002.

The ILAsm syntax for declaring a method as an unmanaged export is very simple:

```
.export [<ordinal>] as <export_name>
```

where *<ordinal>* is an integer constant. The *<export_name>* provides an alias for the exported method. It is necessary to specify *<export name>* even if the method is exported under its own name.

The *.export* directive is placed within the scope of the respective method together with the *.vtentry* directive, as shown in this example:

```
⋮
.corflags 0x00000002
⋮
.vtfixup [1] int32 fromunmanaged at VT_01
⋮
.method public static void Foo()
{
   .vtentry 1:1     // Entry 1, slot 1
   .export [1] as Bar // Export #1, Name="Bar"
   ⋮
}
⋮
.data VT_01 = int32(0) // The slot will be filled automatically.
⋮
```

The source code for the small sample described earlier in Figure 15-2 could look like the following, which is taken from the sample file YDD.il on the companion CD:

```
.assembly extern mscorlib { }
.assembly YDD { }
.module YDD.dll
```

(continued)

```
.corflags 0x00000002
.vtfixup [1] int32 fromunmanaged at VT_01 // First v-table fixup
.vtfixup [1] int32 fromunmanaged at VT_02 // Second v-table fixup
.vtfixup [1] int32 fromunmanaged at VT_03 // Third v-table fixup
.data VT_01 = int32(0)         // First v-table entry
.data VT_02 = int32(0)         // Second v-table entry
.data VT_03 = int32(0)         // Third v-table entry
.method public static void Yabba()
{
   .vtentry 1:1
   .export [1] as Yabba
   ldstr "Yabba"
   call void [mscorlib]System.Console::WriteLine(string)
   ret
}
.method public static void Dabba()
{
   .vtentry 2:1
   .export [2] as Dabba
   ldstr "Dabba"
   call void [mscorlib]System.Console::WriteLine(string)
   ret
}
.method public static void Doo()
{
   .vtentry 3:1
   .export [3] as Doo
   ldstr "Doo!"
   call void [mscorlib]System.Console::WriteLine(string)
   ret
}
```

Now you can compile the sample to a managed DLL, remembering to use the */DLL* command-line option of the ILAsm compiler, and then write a small unmanaged program that calls the methods from this DLL. This unmanaged program can be built with any unmanaged compiler—for example, Microsoft Visual C++ 6—but don't forget that YDD.dll cannot run unless the .NET Framework is installed. It's still a managed assembly, even if your unmanaged program does not know about it.

As you've probably noticed, all *.vtfixup* directives of the sample sport identical flags. This means that three single-slot v-table entries can be grouped into one three-slot entry:

```
.vtfixup [3] int32 fromunmanaged at VT_01
.data VT_01 = int32[3]
```

Then the *.vtentry* directives of the *Dabba* and *Doo* methods must be changed to *.vtentry 1:2* and *.vtentry 1:3*, respectively.

It's worth making a few additional points about the sample. First, it's a good practice to define all VTableFixup and v-table entries in the beginning of the source code, before any methods or other data constants are defined. This ensures that you will not attempt to assign a nonexistent v-table slot to a method and that the v-table will be contiguous.

Second, in the sample, the export ordinals correspond to v-table entry numbers. In fact, no such correspondence is necessary. But if you're using the v-table only for the purpose of unmanaged export, it might not be a bad idea to maintain this correspondence simply to keep track of your v-table slots. It won't do you any good to assign the same v-table slot or the same export ordinal to two different methods.

Third, you should remember that the export ordinals are relative. The Export Directory table has a *Base* entry, which contains the base value for the export ordinals. The ILAsm compiler simply finds the lowest ordinal used in the *.export* directives throughout the source code and assigns this ordinal to the *Base* entry. If you start numbering your exports from 5, it does not mean that the first four entries in the EAT will be undefined. The common practice is to use one-based export ordinals.

16

Multilanguage Projects

The Microsoft .NET paradigm is in principle multilanguage. You can derive your class from another class that has been declared in an assembly produced by someone else, and you don't need to worry about how the language you are using relates to the language used to write the other assembly. You can create a multimodule assembly, each module of which is written in a different language.

What you can't do so easily, however, is to build a single-module assembly using different languages. This means that once you've selected a development language for your single-module assembly, you must accept all the limitations of the selected language.

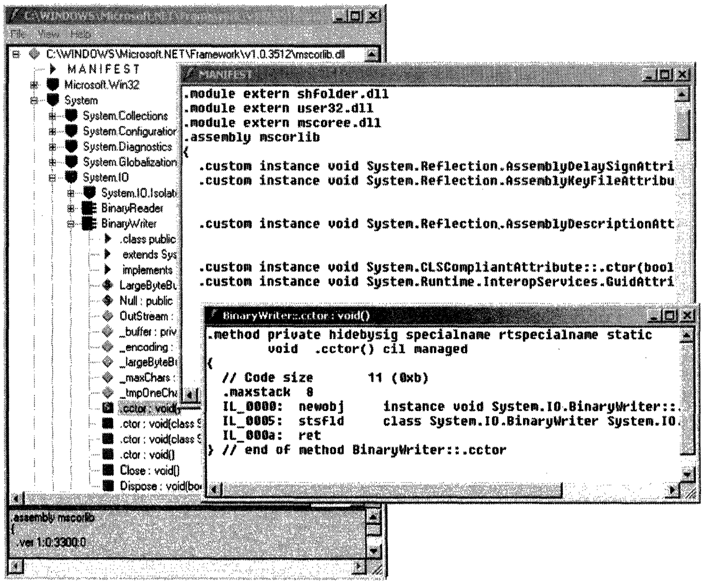
IL assembly language (ILAsm) offers a way to resolve this problem. ILAsm, as a platform-oriented and ideologically neutral language, provides a natural common base for the high-level, pure-IL languages. Because of this, ILAsm can be used as an intermediate stage for multilanguage projects. Most of the high-level language compilers don't actually use ILAsm as their base language, but this can be easily helped by the use of the IL Disassembler (ILDASM).

IL Disassembler

The IL Disassembler tool, Ildasm.exe, is distributed with the .NET Framework SDK and is one of the most popular tools among developers working on .NET-based programs. Virtually every book dedicated to .NET themes at least mentions ILDASM and briefly describes its features.

ILDASM is a dual-mode application—that is, it can run either as a console or as a GUI application. Two ILDASM command-line options—*/OUT:<file_name>* and */TEXT*—set the disassembler mode. If either */TEXT* or */OUT:CON* is specified, ILDASM outputs the disassembly text to the console window from which it was started. If */OUT:<file_name>* is specified, ILDASM dumps the disassembly text into the specified file. If neither */TEXT* nor */OUT* is specified, ILDASM switches to graphical mode.

The graphical user interface of ILDASM is rather modest and strictly functional. The disassembled module is represented as a tree. The module itself is shown as the root, namespaces and classes as tree nodes, and members—methods, fields, events, and properties—as tree leaves. Double-clicking a tree leaf displays a disassembly window containing the ILAsm source text of the corresponding item of the module, as shown in Figure 16-1.



```
.module extern shfolder.dll
.module extern user32.dll
.module extern mscoree.dll
.assembly mscorlib
{
  .custom instance void System.Reflection.AssemblyDelaySignAttri
  .custom instance void System.Reflection.AssemblyKeyFileAttribu


  .custom instance void System.Reflection.AssemblyDescriptionAtt


  .custom instance void System.CLSCompliantAttribute::.ctor(bool
  .custom instance void System.Runtime.InteropServices.GuidAttri
```

```
.method private hidebysig specialname rtspecialname static
        void  .cctor() cil managed
{
  // Code size       11 (0xb)
  .maxstack  8
  IL_0000:  newobj     instance void System.IO.BinaryWriter::.
  IL_0005:  stsfld     class System.IO.BinaryWriter System.IO.
  IL_000a:  ret
} // end of method BinaryWriter::.cctor
```

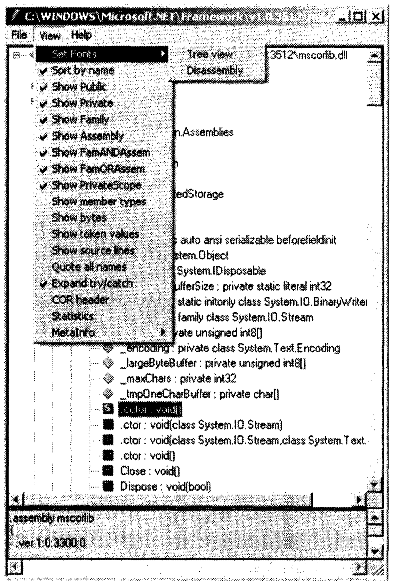
```
.assembly mscorlib
{
  .ver 1:0:3300:0
```

Figure 16-1 The IL Disassembler in graphical mode.

The tree leaf MANIFEST corresponds to all module-level information, including manifest metadata, module metadata, and v-table fixups.

Each tree node representing a type has special leaves providing information about the type: a *class* leaf, an *extends* leaf (if the type is derived from another type), and one *implements* leaf for each interface the type implements. Double-clicking a *class* leaf displays a disassembly window containing full class information except for the disassembly of the class members. Double-clicking an *extends* leaf or an *implements* leaf moves the cursor in the tree view to the respective class or interface if it is defined in the current module.

The disassembler provides numerous viewing options that allow you to control the disassembly text presentation. In graphical mode, these options are listed on the View menu, as shown in Figure 16-2.



```
3512\mscorlib.dll
n.Assemblies
n
edStorage
auto ansi serializable beforefieldinit
stem.Object
System.IDisposable
ufferSize : private static literal int32
static initonly class System.IO.BinaryWriter
family class System.IO.Stream
vate unsigned int8[]
_encoding : private class System.Text.Encoding
_largeByteBuffer : private unsigned int8[]
_maxChars : private int32
_tmpOneCharBuffer : private char[]
.cctor : void()
.ctor : void(class System.IO.Stream)
.ctor : void(class System.IO.Stream,class System.Text.
.ctor : void()
Close : void()
Dispose : void(bool)
```

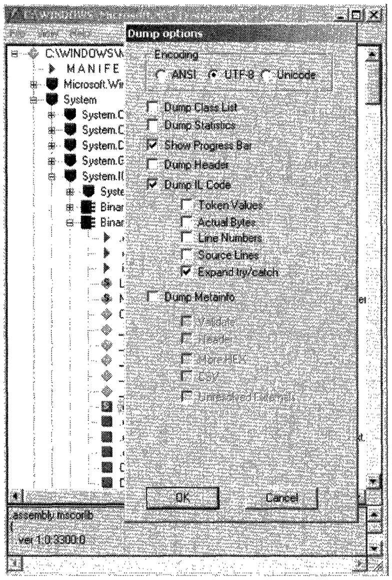
```
.assembly mscorlib
{
  .ver 1:0:3300:0
```

Figure 16-2 Disassembler viewing options.

The module opened in ILDASM's graphical mode can be dumped to a file but not to a console window. To dump the module to a file, choose Dump from the File menu, set the dump options as shown in Figure 16-3, and click OK. In the Save As dialog box displayed, specify a directory and the name of an output file. To dump a text representation of the fully expanded tree view to a specified file, choose DumpTree from the File menu.

Note For reasons I won't discuss here, the disassembler does not offer all possible viewing options by default. To access all the options, you must use the */ADVANCED* (or */ADV*, because ILDASM options are recognized by their first three characters) command-line option. I strongly recommend that you make it a habit to invoke the disassembler as *ildasm /adv <...other options...>* to avoid the frustration of being unable to access the option you need and being forced to close and restart ILDASM. And, yes, I know it's inconvenient.

Certain options are available only in advanced mode. Among them, the group of */METAINFO* options, which provide various summaries of the module metadata, are very useful. Two others are rarely used: */STATISTICS*, a summary of the PE file characteristics; and */CLASSLIST*, a list of types defined in the module, available for file or console dump only.



Dump options

Encoding
○ ANSI ● UTF-8 ○ Unicode

□ Dump Class List
□ Dump Statistics
☑ Show Progress Bar
□ Dump Header
☑ Dump IL Code
  □ Token Values
  □ Actual Bytes
  □ Line Numbers
  □ Source Lines
  ☑ Expand try/catch
□ Dump Metainfo
  □ Validate
  □ Header
  □ MoreHEX
  □ CSV
  □ Unresolved Externals

OK Cancel

Figure 16-3 Selecting file dump options.

All of the disassembly options shown in Figure 16-3 are available as command-line options in ILDASM, but the inverse is not quite true. Appendix D, "IL Assembler and Disassembler Command-Line Options," contains a complete list of all the command-line options. The following list focuses only on the most important of these options:

- The */ADVANCED* option is the first item you should specify when invoking the disassembler.
- The */UTF8* and */UNICODE* options set the encoding of the output file. The default encoding is ANSI.
- The */TOKENS* option includes hexadecimal token values as comments in the disassembly text.
- The */BYTES* option includes the hexadecimal representation of IL instructions as comments in the disassembly text.
- The */ITEM=<item_description>* option limits the disassembly to the specified item: a class or a member method. For example, */ITEM="Foo"* dumps the *Foo* class and all its members, */ITEM="Foo::Bar"* dumps all member methods named *Bar* in the *Foo* class, and */ITEM="Foo::Bar(int32(int32, string))"* dumps the method *int32 Foo::Bar(int32, string)*. This option has no effect if the disassembler is invoked in graphical mode.
- The */VISIBILITY=<vis>[+<vis>*]* option limits the disassembly to the items that have the specified visibility and accessibility flags. The *<vis>* suboptions are three-letter abbreviations of all possible visibility and accessibility flags:
    - ***PUB*** Public
    - ***PRI*** Private
    - ***FAM*** Family
    - ***ASM*** Assembly
    - ***FAA*** Family and assembly
    - ***FOA*** Family or assembly
    - ***PSC*** Private scope

    For example, */VIS=PUB+FAM+FOA* limits the disassembly output to only those items that can be accessed from outside the assembly.
- The */NOIL* option suppresses the ILAsm source text output. You can use this option when you are interested not in a disassembly but in file statistics, a metadata summary, and so on. This option has no effect if the disassembler is invoked in graphical mode.
- The */RAWEH* option forces all structured exception handling clauses to be dumped in canonic form at the end of each method scope.

- The */LINENUM* option includes the *.language* and *.line* directives in the disassembly text, to allow the reassembled code to be bound to the original source files rather than the ILAsm source file. (The section "Compiling in Debug Mode," later in this chapter, discusses the use of these directives in detail.) This option has no effect if the PE file being disassembled is not accompanied by a program database (PDB) file that contains all the debug information.

- The */NOBAR* option suppresses the pop-up window showing the disassembly progress. This option is useful if the disassembler is invoked from batch files as part of an automatic process running in the background.

- The */METAINFO[=<met_opt>]* option dumps the metadata summary. The *<met_opt>* suboptions indicate the specifics of this summary:

  - ❑ ***HEX*** Add hexadecimal representation of the signatures
  - ❑ ***CSV*** Provide the sizes of string, blob, and *GUID* heaps and sizes of the metadata tables and their records
  - ❑ ***MDH*** Provide the metadata header details
  - ❑ ***UNR*** Provide a list of unresolved method references and method definitions without implementation
  - ❑ ***VAL*** Run metadata validation

  The metadata suboptions can't be concatenated using the plus character, as the visibility suboptions can be. Instead, multiple occurrences of */METAINFO* options are permitted in order to set multiple suboptions. For example, *ildasm /adv /noil /met=hex /met=mdh MyModule.dll /out:MyModule.txt.*

All of these options are recognized by their first three letters (*/NOBAR* means the same as */NOB*, for instance) and are case-insensitive (*/NOB* means the same as */nob*). The colon character (:) and the equality character (=) are interchangeable; for example, */vis=pub* means the same as */vis:pub*.

When a PE file is disassembled in full to a file, the managed and unmanaged resources are automatically saved to respective files so that they can be picked up by the assembler and incorporated into a new PE file during the reassembly. "In full" means that neither */NOIL* nor */ITEM* nor */VIS* options are specified, because these options result in a partial disassembly, whose text is not suitable for reassembling. The unmanaged resources are saved in a file that has the same name as the output file and has the extension RES. The managed resources are saved in files named according to the managed resource names specified in the metadata. The resource files are not saved when a PE file is disassembled to a console window using the option */TEXT* or */OUT:CON*.

Principles of Round-Tripping

The round-tripping of managed PE files includes two steps. The first step is to disassemble the PE file into an ILAsm source file and the managed and unmanaged resource files:

```
ildasm MyModule.dll /out:MyModule.il
```

You can forgo the */ADV* option in this case. You usually don't need advanced presentation options for round-tripping, since the output is intended for the ILAsm compiler.

The second step of round-tripping is to invoke the ILAsm compiler to produce a new PE file from the results of the disassembler's activities:

```
ilasm /dll MyModule.il /out:MyModuleRT.dll /res:MyModule.res
```

The command-line options of the ILAsm compiler are listed in full in Appendix D. The most important of these options are the following:

- The */OUT:<file_name>* option specifies the name of the resulting PE file. The default name is the name of the first source file with the extension DLL or EXE.
- The */DLL* option creates a dynamic-link library module. The default is to create an executable (EXE) PE file. The file extension of the output file does not matter: if you specify */OUT:MyModule.dll* and neglect to specify */DLL*, the result is an executable PE file (EXE) named MyModule.dll. You can try to sell such a PE file to Barnum, but you won't be able to do much more than that.
- The */RES:<unmanaged_resource_file_name>* option indicates that the compiler must incorporate the specified unmanaged resource file into the PE file. The managed resources are specified in the ILAsm source code and are picked up by the compiler automatically, whereas an unmanaged resource has no metadata representation and hence must be explicitly specified in a command-line option.
- The */DEBUG* option has two effects: a PDB file is created, and the *[mscorlib]System.Diagnostics.DebuggableAttribute* custom attribute is assigned to the *Assembly* or *Module* metadata record. See "Compiling in Debug Mode," later in this chapter, for more details.
- The */KEY:<private_key_file_name>* or */KEY:@<private_key_source_name>* option generates the strong name signature of the PE file. Only the prime module of an assembly can carry a strong name signature. If you are round-tripping a strong name signed prime module and don't have the private key—if, in other words, it's someone

else's assembly—you can leave the module unsigned. In this case, you'll be able to use it as a private assembly only. If you decide not to sign the module, you must delete or comment out the *.publickey* directive in the *.assembly* scope. Otherwise, you will produce a delayed-signed assembly—that is, an assembly that must be strong name signed at some moment after the compilation and before it can be used. (Alternatively, you can sign the prime module with your own private key and say that it was your own assembly all along. Do this only if you find true joy in litigation.)

A few items that might be present in a managed PE file don't survive round-tripping. For example, any embedded native code is lost. The exceptions to this rule are the tiny pieces of native code that are automatically generated during the compilation: the common language runtime startup stub and the unmanaged export stubs. Strictly speaking, even these tiny pieces don't really round-trip: they are generated anew rather than reproduced from the disassembly.

Another item that does not survive round-tripping is data-on-data, which is a data constant containing the address of another data constant. Fortunately, this kind of data is rather rare and not very useful, thanks to the strict limitations the runtime imposes on operations with unmanaged pointers. Among the compilers producing pure-IL modules, only the ILAsm compiler is capable of generating such data.

Local variable names survive round-tripping only if the PDB file accompanying the original PE file is available. The local variable names are part of the debug information rather than the metadata.

Creative Round-Tripping

Simple two-step round-tripping, involving only disassembly and reassembly, is not very interesting, unless you are testing the round-tripping capabilities of the IL assembler and disassembler. A more creative scheme involves three steps: disassembly, *tinkering with the ILAsm source code*, and reassembly.

Generally speaking, you can alter the ILAsm source code during this creative round-tripping in only three ways:

- You can change the code emitted by a high-level compiler or a tool in a way the compiler (the tool) would not allow you to do. From the section "Thunks and Wrappers" in Chapter 15, "Managed and Unmanaged Code Interoperation," you might recall mention of the "manual intervention" necessary to correct the interop assemblies produced by the Tlbimp.exe tool. Other scenarios can also call for editing original code. For example, let's suppose that you don't believe me when I say that the common language runtime does not permit overriding the final virtual methods. You write a test program in Microsoft Visual Basic .NET, only to discover that the compiler will not let you explicitly override a final method. Without explicit overriding, the compiler automatically sets the *newslot* flag of the overriding method, and, alas, there goes your experiment. Then you recall that the ILAsm compiler doesn't have such inhibitions. You disassemble your test application, remove the *newslot* flag, reassemble the test application, run it, and find out that I was right. As another example, let's suppose that you have a nice assembly written in Microsoft Visual C# .NET that can do a lot of nice things, but your retrograde colleagues insist that in order to be useful your assembly must expose its functionality to the unmanaged legacy components. And those components are so far on the legacy side that they don't even use COM. Then you recall that ILAsm allows you to export the managed methods as unmanaged entry points, and... I don't think I need to continue.

- You can add the items written in ILAsm to extend your application's functionality beyond the capabilities of a high-level compiler.
- Finally, you can disassemble several modules and reassemble them into one module.

Using Class Augmentation

The ILAsm-specific technique of class augmentation can be useful when you want to add new components written in ILAsm to your application written in a high-level language. If you need to add new types, an obvious solution is to declare these classes in a separate ILAsm source file, disassemble your application, and reassemble it with this additional .il file. Class augmentation allows you to apply the same approach if you need to add new members to some of the types defined in your application. In other words, you don't need to edit the disassembly text of your application, inserting new members in the type definitions, because you can augment the respective type definitions in a separate source file.

For example, suppose that you would like to have a thread local storage (TLS) mapped field in class *X* and a *vararg* method in class *Y*, but the high-level language of your choice does not allow you to specify such items. You can write the following amendment file, Amend.il:

```
.class X
{
   .field public static int32 tlsField at TLSD001
}
```

(continued)

```
.data tls TLSD001 = int32(1234)
.class Y
{
   .method public vararg int64 Sum()
   {
      ⋮
   }
}
```

Then you can disassemble your original (incomplete) module and reassemble it with an amendment:

```
ildasm MyApp.exe /out:MyApp.il
ilasm MyApp Amend
```

The last line is so laconic because it uses three defaults: the default source file extension (IL), the default output file type and extension (EXE), and the default output file name (the same as the name of the first source file).

Module Linking Through Round-Tripping

Now let's assume that instead of writing the amendment file in ILAsm, you wrote it in another high-level language, compiled it to a module, and then disassembled it. Can you do that? Yes, you can, and it means that round-tripping can be used for linking several modules together to form one. The original language used to write each module does not matter as long as all the modules are pure-IL. The modules must be pure-IL simply because any mixed-code module will fail to round-trip.

Brad Abrams, a program manager I work with, has written a small tool called "Lame Linker," which performs managed module linking through round-tripping. You can have a look at this tool at GotDotNet, *http://www.gotdotnet.com/userarea/keywordsrch.aspx?keyword=Lame%20Link.* As Brad explains it, he calls his linker "lame" because it doesn't have many of the features of a good linker. Lame or not, this linker is used rather extensively and has proven to be a useful tool.

The basic problem with linking multiple modules through round-tripping is that you inevitably run into duplicate declarations. When you write amendment files in ILAsm, you don't need to make sure that these files compile per se; they must compile together with the disassembly of the original module. But each module you link *has* been compiled per se, and a significant part of its metadata overlaps with the metadata of other modules being linked.

Let's review the potential effects of multiple declarations of different metadata items.

Multiple *Assembly* declarations (*.assembly*) should be avoided. The ILAsm compiler ignores repetitive *Assembly* declarations as long as the assembly name is the same, but if one of any subsequent declarations specifies a name that differs from that of the first declaration, the compiler diagnoses an error.

Multiple *AssemblyRef* declarations (*.assembly extern*) are harmless. The ILAsm compiler ignores them. The same is true for *Module* declarations (*.module*), *ModuleRef* declarations (*.module extern*), *File* declarations (*.file*), and *ExportedType* declarations (*.class extern*).

Duplicate *ManifestResource* declarations (*.mresource*) should be avoided. The ILAsm compiler will not emit a new *ManifestResource* record for each declaration encountered, but it will incorporate a copy of the respective managed resource for each *.mresource* declaration in the output PE file. The resulting PE file will perform as expected, but it will be bloated.

Duplicate member declarations (*.field, .method, .event, .property*) must be avoided because their presence leads to compilation failure. Duplicate member declarations can happen in two cases only: if you declare a type in one module and amend it in another, or if you declare global fields or methods. I can't say how likely the first scenario is—somehow, the feasibility of declaring part of a type in one module and another part in another module, with the parts overlapping, escapes me. But the second scenario is very likely indeed: you usually don't pay much attention to naming global fields and methods because they are an "internal affair" of the module. But when you link several modules together to form one, all global fields and methods from each module wind up as globals in the resulting module.

Multiple declarations of the module entry point (*.entrypoint*) must be avoided as well, for they also cause compilation failure.

If several of your original modules use mapped fields, you should watch for duplicate data declarations. ILDASM automatically generates the data labels—*D_<data_RVA>* for usual data and *T_<data_RVA>* for TLS data—when it disassembles each original module, so the data labels are almost guaranteed to overlap. Duplicate data labels cause compilation failure.

The Lame Linker I mentioned earlier eliminates multiple *Assembly* declarations, but nothing else.

The list of hazards to watch for in the process of linking through round-tripping looks endless, but in fact all these limitations are reasonable, and their analogs can be found in the traditional linking of object files. Actually, traditional linking is even less tolerant of duplicate definitions. And avoiding (or getting rid of) the dangerous duplications is not rocket science.

Module linking is necessary whenever you want to create a single-module assembly from a multimodule assembly. And it does not matter how you came into possession of the multimodule assembly in the first place. Perhaps you developed different modules using different languages. Or perhaps you split your application into subsystems to be developed independently. Or perhaps you split your application for independent development, and the developers of each subsystem chose their own development language.

Compiling in Debug Mode

When a managed compiler compiles source code in debug mode, you can expect at least two occurrences. First, the resulting module has the custom attribute *[mscorlib]System.Diagnostics.DebuggableAttribute* attached to the *Module* record or, if it is a prime module, to the *Assembly* record. Second, the compiler produces a PDB file containing data about the source files and the compiler, the local variable names, and the tables binding source lines and columns to the code offsets. Of course, the compiler can perform other tasks as well in debug mode—for example, emitting different IL code.

When a module is round-tripped, or when a high-level compiler produces the ILAsm source code as an intermediate step, it is usually desirable to preserve the debug information binding the original source code to the final IL code. ILAsm provides two directives facilitating this:

- The *.language <Language_GUID>[,<Vendor_GUID>[,<Document_GUID>]]* directive defines the source language and, optionally, the compiler vendor and the source document type.
- The *.line <line_num>[:<column_num> [<file_name>]]* directive identifies the line and column in the original source file that are "responsible" for the IL code that follows the *.line* directive.

For example, the following Visual C# .NET code

```
using System;

public class arr
{
   private static int[,] MakeArray() {
      return (int[,])Array.CreateInstance(typeof(int),
         new int[]{2,3}, new int[]{-1, 0});
   }

   private static void Main() {
      int[,] _aTgt = MakeArray();
      foreach (int i in _aTgt) {
         Console.Write(i + " ");
      }
   }
}
```

compiled in debug mode, is disassembled, using the option */LINENUM*, into the following ILAsm code:

```
⋮
.class public auto ansi beforefieldinit arr
       extends [mscorlib]System.Object
{
   .method private hidebysig static int32[0...,0...]
           MakeArray() cil managed
   {
      // Code size       53 (0x35)
      .maxstack  5
      .locals init ([0] int32[0...,0...] CS$00000003$00000000,
             [1] int32[] CS$00000002$00000001,
             [2] int32[] CS$00000002$00000002)
      .language '{3F5162F8-07C6-11D3-9053-00C04FA302A1}',
                '{994B45C4-E6E9-11D2-903F-00C04FA302A1}',
                '{5A869D0B-6611-11D3-BD2A-0000F80849BD}'
      .line 6:3 'C:\\MyDirectory\\arr.cs'
      IL_0000:  ldtoken    [mscorlib]System.Int32
      IL_0005:  call class [mscorlib]System.Type
                [mscorlib]System.Type::GetTypeFromHandle(
                   valuetype [mscorlib]System.RuntimeTypeHandle)
      IL_000a:  ldc.i4.2
      IL_000b:  newarr     [mscorlib]System.Int32
      IL_0010:  stloc.1
      IL_0011:  ldloc.1
      IL_0012:  ldc.i4.0
      IL_0013:  ldc.i4.2
      IL_0014:  stelem.i4
      IL_0015:  ldloc.1
      IL_0016:  ldc.i4.1
      IL_0017:  ldc.i4.3
      IL_0018:  stelem.i4
      IL_0019:  ldloc.1
      IL_001a:  ldc.i4.2
      IL_001b:  newarr     [mscorlib]System.Int32
      IL_0020:  stloc.2
      IL_0021:  ldloc.2
      IL_0022:  ldc.i4.0
      IL_0023:  ldc.i4.m1
      IL_0024:  stelem.i4
      IL_0025:  ldloc.2
      IL_0026:  call class [mscorlib]System.Array
                [mscorlib]System.Array::CreateInstance(
                          class [mscorlib]System.Type,
                          int32[],
                          int32[])
      IL_002b:  castclass  int32[0...,0...]
```

(continued)

```
    IL_0030:  stloc.0
    IL_0031:  br.s       IL_0033

    .line 7:2
    IL_0033:  ldloc.0
    IL_0034:  ret
} // End of method arr::MakeArray

.method private hidebysig static void  Main() cil managed
{
    .entrypoint
    // Code size       103 (0x67)
    .maxstack  3
    .locals init ([0] int32[0...,0...] _aTgt,
            [1] int32 i,
            [2] int32[0...,0...] CS$00000007$00000000,
            [3] int32 CS$00000264$00000001,
            [4] int32 CS$00000265$00000002,
            [5] int32 CS$00000008$00000003,
            [6] int32 CS$00000009$00000004)
    .line 10:3
    IL_0000:  call       int32[0...,0...] arr::MakeArray()
    IL_0005:  stloc.0
    .line 11:21
    IL_0006:  ldloc.0
    IL_0007:  stloc.2
    IL_0008:  ldloc.2
    IL_0009:  ldc.i4.0
    IL_000a:  callvirt instance int32
              [mscorlib]System.Array::GetUpperBound(int32)
    IL_000f:  stloc.3
    IL_0010:  ldloc.2
    IL_0011:  ldc.i4.1
    IL_0012:  callvirt   instance int32
              [mscorlib]System.Array::GetUpperBound(int32)
    IL_0017:  stloc.s    CS$00000265$00000002
    IL_0019:  ldloc.2
    IL_001a:  ldc.i4.0
    IL_001b:  callvirt   instance int32
              [mscorlib]System.Array::GetLowerBound(int32)
    IL_0020:  stloc.s    CS$00000008$00000003
    IL_0022:  br.s       IL_0061

    IL_0024:  ldloc.2
    IL_0025:  ldc.i4.1
    IL_0026:  callvirt   instance int32
              [mscorlib]System.Array::GetLowerBound(int32)
    IL_002b:  stloc.s    CS$00000009$00000004
    IL_002d:  br.s       IL_0055
```

```
      .line 11:12
      IL_002f:  ldloc.2
      IL_0030:  ldloc.s    CS$00000008$00000003
      IL_0032:  ldloc.s    CS$00000009$00000004
      IL_0034:  call
                instance int32 int32[0...,0...]::Get(int32, int32)
      IL_0039:  stloc.1
      .line 12:8
      IL_003a:  ldloc.1
      IL_003b:  box        [mscorlib]System.Int32
      IL_0040:  ldstr      " "
      IL_0045:  call       string
            [mscorlib]System.String::Concat(object, object)
      IL_004a:  call void [mscorlib]System.Console::Write(string)
      .line 11:3
      IL_004f:  ldloc.s    CS$00000009$00000004
      IL_0051:  ldc.i4.1
      IL_0052:  add
      IL_0053:  stloc.s    CS$00000009$00000004
      IL_0055:  ldloc.s    CS$00000009$00000004
      IL_0057:  ldloc.s    CS$00000265$00000002
      IL_0059:  ble.s      IL_002f

      IL_005b:  ldloc.s    CS$00000008$00000003
      IL_005d:  ldc.i4.1
      IL_005e:  add
      IL_005f:  stloc.s    CS$00000008$00000003
      IL_0061:  ldloc.s    CS$00000008$00000003
      IL_0063:  ldloc.3
      IL_0064:  ble.s      IL_0024

      .line 14:2
      IL_0066:  ret
   } // End of method arr::Main

   .method public hidebysig specialname rtspecialname
           instance void  .ctor() cil managed
   {
      // Code size       7 (0x7)
      .maxstack  1
      IL_0000:  ldarg.0
      IL_0001:  call
                instance void [mscorlib]System.Object::.ctor()
      IL_0006:  ret
   } // End of method arr::.ctor

} // End of class arr
```

The *.language* directive sets the GUIDs for all following code until it is superseded by another *.language* directive.

You'll encounter a slight problem with the *.line* directive in the first release of the ILAsm compiler and disassembler: the directive specifies the starting line and column of the original source statement that has been compiled into ILAsm code following the *.line* directive. This doesn't bode well for the Microsoft Visual Studio .NET debugger, which wants to see the line/column interval (starting line and column and ending line and column) for each original source statement. This problem will be corrected in future releases of the ILAsm compiler and disassembler.

In short, if you want the resulting code bound to the original source code, you need to do the following:

- If your compiler generates ILAsm source code, it must insert *.language* and *.line* directives at appropriate points.
- If you are round-tripping a module compiled from a high-level language, use the disassembler option */LINENUM* (or */LIN*).
- In any case, don't forget to use the option */DEBUG* (or */DEB*) of the ILAsm compiler.

V-Table Fixup Declaration

```
<vtfixupDecl> ::= .vtfixup [<int32>] <vtfixupAttr> at <id>

<vtfixupAttr> ::= /* EMPTY */
                | <vtfixupAttr> int32
                | <vtfixupAttr> int64
                | <vtfixupAttr> fromunmanaged
                | <vtfixupAttr> callmostderived
```

Namespace and Type Declarations

```
<nameSpaceHead> ::= .namespace <compName>

<classHead> ::= .class <classAttrs> <id> <extendsClause>
                       <implClause>

<classAttrs> ::= /* EMPTY */ | <classAttrs> <classAttr>

<classAttr> ::= <classAttr> public
              | <classAttr> private
              | <classAttr> nested public
              | <classAttr> nested private
              | <classAttr> nested family
              | <classAttr> nested assembly
              | <classAttr> nested famandassem
              | <classAttr> nested famorassem
              | <classAttr> value
              | <classAttr> enum
              | <classAttr> interface
              | <classAttr> sealed
              | <classAttr> abstract
              | <classAttr> auto
              | <classAttr> sequential
              | <classAttr> explicit
              | <classAttr> ansi
              | <classAttr> unicode
              | <classAttr> autochar
              | <classAttr> import
              | <classAttr> serializable
              | <classAttr> beforefieldinit
              | <classAttr> specialname
              | <classAttr> rtspecialname

<extendsClause> ::= /* EMPTY */ | extends <classRef>

<implClause> ::= /* EMPTY */ | implements <classRefs>

<classRefs> ::= <classRefs>,<classRef> | <classRef>
```

(continued)

```
<classRef> ::= [ <compName> ] <slashedName>
             | [.module <compName> ] <slashedName>
             | <slashedName>

<slashedName> ::= <compName>
                | <slashedName>/<compName>

<classDecls> ::= /* EMPTY */ | <classDecls> <classDecl>

<classDecl> ::= <methodHead>  <methodDecls> }
              | <classHead> { <classDecls> }
              | <eventHead> { <eventDecls> }
              | <propHead> { <propDecls> }
              | <fieldDecl>
              | <dataDecl>
              | <secDecl>
              | <extSourceSpec>
              | <customAttrDecl>
              | .size <int32>
              | .pack <int32>
              | .override <typeSpec>::<methodName>
                   with <callConv> <type> <typeSpec>::<methodName>(
                                                           <sigArgs>)
              | <languageDecl>
```

Signature Type Specifications

```
<type> ::= class <classRef>
         | object
         | string
         | value class <classRef>
         | valuetype <classRef>
         | <type> [ ]
         | <type> [ <bounds> ]
         | <type> &
         | <type> *
         | <type> pinned
         | <type> modreq(<classRef>)
         | <type> modopt(<classRef>)
         | method <callConv> <type>*(<sigArgs>)
         | typedref
         | char
         | void
         | bool
         | int8
         | int16
         | int32
```

```
               | int64
               | float32
               | float64
               | unsigned int8
               | unsigned int16
               | unsigned int32
               | unsigned int64
               | native int
               | native unsigned int

<bounds> ::= <bound>
           | <bounds>,<bound>

<bound> ::= /* EMPTY */
          | ...
          | <int32>
          | <int32> ... <int32>
          | <int32> ...

<callConv> ::= instance <callConv>
             | explicit <callConv>
             | <callKind>

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

Native Type Declarations

```
<nativeType> ::= /* EMPTY */
               | custom( <compQstring>, <compQstring> )
               | fixed sysstring[<int32>]
               | fixed array[<int32>]
               | variant
               | currency
               | syschar
               | void
               | bool
               | int8
               | int16
               | int32
               | int64
               | float32
               | float64
```

(continued)

```
                | error
                | unsigned int8
                | unsigned int16
                | unsigned int32
                | unsigned int64
                | <nativeType>*
                | <nativeType>[ ]
                | <nativeType>[<int32>]
                | <nativeType>[<int32> + <int32>]
                | <nativeType>[ + <int32>]
                | decimal
                | date
                | bstr
                | lpstr
                | lpwstr
                | lptstr
                | objectref
                | iunknown
                | idispatch
                | struct
                | interface
                | safearray <variantType>
                | safearray <variantType>,<compQstring>
                | int
                | unsigned int
                | nested struct
                | byvalstr
                | ansi bstr
                | tbstr
                | variant bool
                | method
                | as any
                | lpstruct

<variantType> ::=  /* EMPTY */
                 | null
                 | variant
                 | currency
                 | void
                 | bool
                 | int8
                 | int16
                 | int32
                 | int64
                 | float32
                 | float64
                 | unsigned int8
                 | unsigned int16
                 | unsigned int32
```

```
                        | unsigned int64
                        | *
                        | <variantType> [ ]
                        | <variantType> vector
                        | <variantType> &
                        | decimal
                        | date
                        | bstr
                        | lpstr
                        | lpwstr
                        | iunknown
                        | idispatch
                        | safearray
                        | int
                        | unsigned int
                        | error
                        | hresult
                        | carray
                        | userdefined
                        | record
                        | filetime
                        | blob
                        | stream
                        | storage
                        | streamed_object
                        | stored_object
                        | blob_object
                        | cf
                        | clsid
```

Field Declarations

```
<fieldDecl> ::= .field <repeatOpt> <fieldAttr> <type> <id>
                      <atOpt> <initOpt>

<repeatOpt> ::= /* EMPTY */
              | [<int32>]

<fieldAttr> ::= /* EMPTY */
              | <fieldAttr> public
              | <fieldAttr> private
              | <fieldAttr> family
              | <fieldAttr> assembly
              | <fieldAttr> famandassem
              | <fieldAttr> famorassem
              | <fieldAttr> privatescope
              | <fieldAttr> static
              | <fieldAttr> initonly
```

(continued)

```
            | <fieldAttr> rtspecialname
            | <fieldAttr> specialname
            | <fieldAttr> marshal( <nativeType> )
            | <fieldAttr> literal
            | <fieldAttr> notserialized

<atOpt> ::= /* EMPTY */
        | at <id>

<initOpt> ::= /* EMPTY */
          | = <fieldInit>

<fieldInit> ::= float32(<float64>)
            | float64(<float64>)
            | float32(<int64>)
            | float64(<int64>)
            | int64(<int64>)
            | int32(<int64>)
            | int16(<int64>)
            | char(<int64>)
            | int8(<int64>)
            | bool(<truefalse>)
            | <compQstring>
            | bytearray( <bytes> )
            | nullref
```

Data Declarations

```
<dataDecl> ::= <dataHead> <dataBody>

<dataHead> ::= .data <tls> <id> =
           | .data <tls>

<tls> ::= /* EMPTY */
      | tls

<dataBody> ::= { <dataItemList> }
           | <dataItem>

<dataItemList> ::= <dataItem>,<dataItemList>
               | <dataItem>

<dataItem> ::=char*(<compQstring>)
           | &(<id>)
           | bytearray = ( <bytes> )
           | float32(<float64>) <repeatOpt>
           | float64(<float64>) <repeatOpt>
           | int64(<int64>) <repeatOpt>
```

```
            | int32(<int32>) <repeatOpt>
            | int16(<int32>) <repeatOpt>
            | int8(<int32>) <repeatOpt>
            | float32 <repeatOpt>
            | float64 <repeatOpt>
            | int64 <repeatOpt>
            | int32 <repeatOpt>
            | int16 <repeatOpt>
            | int8 <repeatOpt>
```

Method Header Declarations

```
<methodHead> ::= .method <methAttr> <callConv> <paramAttr> <type>
                      <methodName>(<sigArgs>) <implAttr> {
               | .method <methAttr> <callConv> <paramAttr> <type>
                      marshal(<nativeType>)
                           <methodName>(<sigArgs>) <implAttr> {

<methAttr> ::= /* EMPTY */
             | <methAttr> static
             | <methAttr> public
             | <methAttr> private
             | <methAttr> family
             | <methAttr> assembly
             | <methAttr> famandassem
             | <methAttr> famorassem
             | <methAttr> privatescope
             | <methAttr> final
             | <methAttr> virtual
             | <methAttr> abstract
             | <methAttr> hidebysig
             | <methAttr> newslot
             | <methAttr> reqsecobj
             | <methAttr> specialname
             | <methAttr> rtspecialname
             | <methAttr> unmanagedexp

             | <methAttr> pinvokeimpl(<compQstring>
                                   as <compQstring> <pinvAttr>)
             | <methAttr> pinvokeimpl(<compQstring> <pinvAttr>)
             | <methAttr> pinvokeimpl(<pinvAttr>)

<pinvAttr> ::= /* EMPTY */
             | <pinvAttr> nomangle
             | <pinvAttr> ansi
             | <pinvAttr> unicode
             | <pinvAttr> autochar
```

(continued)

```
              | <pinvAttr> lasterr
              | <pinvAttr> winapi
              | <pinvAttr> cdecl
              | <pinvAttr> stdcall
              | <pinvAttr> thiscall
              | <pinvAttr> fastcall

<methodName> ::= .ctor
               | .cctor
               | <compName>

<paramAttr> ::= /* EMPTY */
              | <paramAttr> [in]
              | <paramAttr> [out]
              | <paramAttr> [opt]

<implAttr> ::= /* EMPTY */
             | <implAttr> native
             | <implAttr> cil
             | <implAttr> optil
             | <implAttr> managed
             | <implAttr> unmanaged
             | <implAttr> forwardref
             | <implAttr> preservesig
             | <implAttr> runtime
             | <implAttr> internalcall
             | <implAttr> synchronized
             | <implAttr> noinlining

<sigArgs> ::= /* EMPTY */
            | <sigArgList>

<sigArgList> ::= <sigArg>
               | <sigArgList>,<sigArg>

<sigArg> ::= ...
           | <paramAttr> <type>
           | <paramAttr> <type> <id>
           | <paramAttr> <type> marshal( <nativeType> )
           | <paramAttr> <type> marshal( <nativeType> ) <id>
```

Method Body Declarations

```
<methodDecls> ::= /* EMPTY */
                | <methodDecls> <methodDecl>

<methodDecl> ::= .emitbyte <int32>
               | .maxstack <int32>
```

```
                 | .locals( <sigArgs> )
                 | .locals init ( <sigArgs> )
                 | .entrypoint
                 | .zeroinit
                 | .export [ <int32> ]
                 | .export [ <int32> ] as <id>
                 | .vtentry <int32>:<int32>
                 | .override <typeSpec>::<methodName>
                 | <scopeBlock>
                 | .param [ <int32> ] <initOpt>
                 | <id>:
                 | <sehBlock>
                 | <instr>
                 | <secDecl>
                 | <extSourceSpec>
                 | <languageDecl>
                 | <customAttrDecl>
                 | <dataDecl>

<typeSpec> ::= <classRef>
             | [ <compName> ]
             | [.module <compName> ]
             | <type>

<scopeBlock> ::= { <methodDecls> }

<sehBlock> ::= <tryBlock> <sehClauses>

<tryBlock> ::= .try <scopeBlock>
             | .try <id> to <id>
             | .try <int32> to <int32>

<sehClauses> ::= <sehClause> <sehClauses>
               | <sehClause>

<sehClause> ::= catch <classRef> <handlerBlock>
              | finally <handlerBlock>
              | fault <handlerBlock>
              | <filterClause> <handlerBlock>

<filterClause> ::= filter <scopeBlock>
                 | filter <id>
                 | filter <int32>

<handlerBlock> ::= <scopeBlock>
                 | handler <id> to <id>
                 | handler <int32> to <int32>

<instr> ::= INSTR_NONE       // nop
```

(continued)

```
        | INSTR_VAR <int32> // ldarg,ldarga,starg,ldloc,
                            // ldloca,stloc
        | INSTR_VAR <id>
        | INSTR_I <int32>   // ldc.i4
        | INSTR_I8 <int64>  // ldc.i8
        | INSTR_R <float64> // ldc.r8
        | INSTR_R <int64>
        | INSTR_R ( <bytes> )
        | INSTR_BRTARGET <int32> // br,brtrue,brfalse,beq,bne,...
        | INSTR_BRTARGET <id>
        | INSTR_METHOD <methodRef>
                  // call,callvirt,jmp,newobj,ldftn,ldvirtftn
        | INSTR_FIELD <type> <typeSpec>::<id>
                    // ldfld,ldsfld,ldflda,ldsflda,stfld,stsfld
        | INSTR_FIELD <type> <id>
        | INSTR_TYPE <typeSpec> // ldobj,stobj,box,unbox,newarr,...
        | INSTR_STRING <compQstring>                // ldstr
        | INSTR_STRING bytearray = ( <bytes> )
        | INSTR_SIG <callConv> <type> ( <sigArgs> ) // calli
        | INSTR_TOK <ownerType>                     // ldtoken
        | INSTR_SWITCH ( <labels> )                 // switch

<methodRef> ::=
   <callConv> <type> <typeSpec>::<methodName>( <sigArgs> )
         | <callConv> <type> <methodName>( <sigArgs> )

<labels> ::= /* EMPTY */
         | <id>,<labels>
         | <int32>,<labels>
         | <id>
         | <int32>

<ownerType> ::= <typeSpec>
            | <memberRef>

<memberRef> ::= method <methodRef>
            | field <type> <typeSpec>::<id>
            | field <type> <id>
```

Event Declarations

```
<eventHead> ::= .event <eventAttr> <typeSpec> <id>
            | .event <eventAttr> <id>

<eventAttr> ::= /* EMPTY */
            | <eventAttr> rtspecialname
            | <eventAttr> specialname
```

```
<eventDecls> ::= /* EMPTY */
             | <eventDecls> <eventDecl>

<eventDecl> ::= .addon <methodRef>
            | .removeon <methodRef>
            | .fire <methodRef>
            | .other <methodRef>
            | <customAttrDecl>
            | <extSourceSpec>
            | <languageDecl>
```

Property Declarations

```
<propHead> ::= .property <propAttr> <type>
                         <id>(<sigArgs>) <initOpt>

<propAttr> ::= /* EMPTY */
           | <propAttr> rtspecialname
           | <propAttr> specialname

<propDecls> ::= /* EMPTY */
            | <propDecls> <propDecl>

<propDecl> ::= .set <methodRef>
           | .get <methodRef>
           | .other <methodRef>
           | <customAttrDecl>
           | <extSourceSpec>
           | <languageDecl>
```

Custom Attribute Declarations

```
<customAttrDecl> ::= .custom <customType>
                 | .custom <customType> = <compQstring>
                 | .custom <customType> = ( <bytes> )
                 | .custom <customType> = <compQstring>
                 | .custom (<ownerType>) <customType>
                 | .custom (<ownerType>) <customType> =
                                        <compQstring>
                 | .custom (<ownerType>) <customType> = ( <bytes> )

<customType> ::= <callConv> <type> <typeSpec>::.ctor( <sigArgs> )
             | <callConv> <type> .ctor( <sigArgs> )
```

Security Declarations

```
<secDecl> ::= .permission <secAction> <typeSpec> ( <nameValPairs> )
          | .permission <secAction> <typeSpec>
          | .permissionset <secAction> = ( <bytes> )

<nameValPairs> ::= <nameValPair>
                | <nameValPair>,<nameValPairs>

<nameValPair> ::= <compQstring> = <caValue>

<caValue> ::= <truefalse>
          | <int32>
          | int32( <int32> )
          | <compQstring>
          | <classRef> ( int8: <int32> )
          | <classRef> ( int16: <int32> )
          | <classRef> ( int32: <int32> )
          | <classRef> ( <int32> )

<secAction> ::= request
            | demand
            | assert
            | deny
            | permitonly
            | linkcheck
            | inheritcheck
            | reqmin
            | reqopt
            | reqrefuse
            | prejitgrant
            | prejitdeny
            | noncasdemand
            | noncaslinkdemand
            | noncasinheritance
```

Manifest Declarations

```
<moduleHead> ::= .module
             | .module <compName>
             | .module extern <compName>


<fileDecl> ::= .file <fileAttr> <compName> <fileEntry>
                     .hash = ( <bytes> ) <fileEntry>
           | .file <fileAttr> <compName> <fileEntry>
```

```
<fileAttr> ::= /* EMPTY */
           | <fileAttr> nometadata

<fileEntry> ::= /* EMPTY */
            | .entrypoint

<assemblyHead> ::= .assembly <asmAttr> <compName>

<asmAttr> ::= /* EMPTY */
          | <asmAttr> noappdomain
          | <asmAttr> noprocess
          | <asmAttr> nomachine

<assemblyDecls> ::= /* EMPTY */
                | <assemblyDecls> <assemblyDecl>

<assemblyDecl> ::= .hash algorithm <int32>
               | <secDecl>
               | <asmOrRefDecl>

<asmOrRefDecl> ::= .publickey = ( <bytes> )
               | .ver <int32>:<int32>:<int32>:<int32>
               | .locale <compQstring>
               | .locale = ( <bytes> )
               | <customAttrDecl>

<assemblyRefHead> ::= .assembly extern <compName>
                  | .assembly extern <compName> as <compName>

<assemblyRefDecls> ::= /* EMPTY */
                   | <assemblyRefDecls> <assemblyRefDecl>

<assemblyRefDecl> ::= .hash = ( <bytes> )
                  | <asmOrRefDecl>
                  | .publickeytoken = ( <bytes> )

<expTypeHead> ::= .class extern <exptAttr> <compName>

<exptAttr> ::= /* EMPTY */
           | <exptAttr> private
           | <exptAttr> public
           | <exptAttr> nested public
           | <exptAttr> nested private
           | <exptAttr> nested family
           | <exptAttr> nested assembly
           | <exptAttr> nested famandassem
           | <exptAttr> nested famorassem
```

(continued)

```
<expTypeDecls> ::= /* EMPTY */
                 | <expTypeDecls> <expTypeDecl>

<expTypeDecl> ::= .file <compName>
                | .class extern <compName>
                | .class <int32>
                | <customAttrDecl>

<manifestResHead> ::= .mresource <manresAttr> <compName>

<manresAttr> ::= /* EMPTY */
               | <manresAttr> public
               | <manresAttr> private

<manifestResDecls> ::= /* EMPTY */
                     | <manifestResDecls> <manifestResDecl>

<manifestResDecl> ::= .file <compName> at <int32>
                    | .assembly extern <compName>
                    | <customAttrDecl>
```

Appendix B

Metadata Tables Reference

Entry Types

BYTE	Unsigned 1-byte integer
SHORT	Signed 2-byte integer
USHORT	Unsigned 2-byte integer
ULONG	Unsigned 4-byte integer
RID: <*table*>	Record index to <*table*>
STRING	Offset in the *#Strings* stream
GUID	Offset in the *#GUID* stream
BLOB	Offset in the *#Blob* stream
<*coded_token_type*>	Coded token (see the Coded Token Types table at the end of the Appendix)

Module; RID type: 00; Token type: 0x00000000; Metadata (MD) streams: #~, #-

Entry Name	Entry Type	Comments
Generation	*USHORT*	For edit-and-continue
Name	*STRING*	No longer than 512 bytes
Mvid	*GUID*	Generated automatically
EncId	*GUID*	For edit-and-continue
EncBaseId	*GUID*	For edit-and-continue

TypeRef; RID type: 01; Token type: 0x01000000; MD streams: #~, #-

Entry Name	Entry Type	Comments
ResolutionScope	*ResolutionScope*	
Name	*STRING*	
Namespace	*STRING*	

TypeDef; RID type: 02; Token type: 0x02000000; MD streams: #~, #-

Entry Name	Entry Type	Comments
Flags	*ULONG*	Validity mask: 0x001173DBF
Name	*STRING*	
Namespace	*STRING*	
Extends	*TypeDefOrRef*	Base type
FieldList	RID: Field	
MethodList	RID: Method	

FieldPtr; RID type: 03; Token type: none; MD stream: #-

Entry Name	Entry Type	Comments
Field	RID: Field	

Field; RID type: 04; Token type: 0x04000000; MD streams: #~, #-

Entry Name	Entry Type	Comments
Flags	*USHORT*	Validity mask: 0xB7F7
Name	*STRING*	No longer than 1023 bytes
Signature	*BLOB*	Cannot be 0

MethodPtr; RID type: 05; Token type: none; MD stream:#-

Entry Name	Entry Type	Comments
Method	RID: Method	

Method; RID type: 06; Token type: 0x06000000; MD streams: #~, #-

Entry Name	Entry Type	Comments
RVA	*ULONG*	Must be 0 or point at read-only section
ImplFlags	*USHORT*	Validity mask: 0x10BF
Flags	*USHORT*	Validity mask: 0xFDF7
Name	*STRING*	No longer than 1023 bytes

(continued)

Method; RID type: 06; Token type: 0x06000000; MD streams: #~, #- *(continued)*

Entry Name	Entry Type	Comments
Signature	*BLOB*	Cannot be 0
ParamList	RID: Param	

ParamPtr; RID type: 07; Token type: none; MD stream: #-

Entry Name	Entry Type	Comments
Param	RID: Param	

Param; RID type: 08; Token type: 0x08000000; MD streams: #~, #-

Entry Name	Entry Type	Comments
Flags	*USHORT*	Validity mask: 0x3013
Sequence	*USHORT*	0 means return value
Name	*STRING*	

InterfaceImpl; RID type: 09; Token type: 0x09000000; MD streams: #~, #-

Entry Name	Entry Type	Comments
Class	RID: TypeDef	Class implementing the interface
Interface	*TypeDefOrRef*	Implemented interface

MemberRef; RID type: 10; Token type: 0x0A000000; MD streams: #~, #-

Entry Name	Entry Type	Comments
Class	*MemberRefParent*	Cannot be *TypeDef*
Name	*STRING*	No longer than 1023 bytes
Signature	*BLOB*	Cannot be 0

Constant; RID type: 11; Token type: none; MD streams: #~, #-

Entry Name	Entry Type	Comments
Type	*BYTE*	
Parent	*HasConstant*	
Value	*BLOB*	

CustomAttribute; RID type: 12; Token type: 0x0C000000; MD streams: #~, #-

Entry Name	Entry Type	Comments
Parent	*HasCustomAttribute*	
Type	*CustomAttributeType*	
Value	*BLOB*	

FieldMarshal; RID type: 13; Token type: none; MD streams: #~, #-

Entry Name	Entry Type	Comments
Parent	*HasFieldMarshal*	
NativeType	*BLOB*	Cannot be 0

DeclSecurity; RID type: 14; Token type: 0x0E000000; MD streams: #~, #-

Entry Name	Entry Type	Comments
Action	*SHORT*	
Parent	*HasDeclSecurity*	
PermissionSet	*BLOB*	

ClassLayout; RID type: 15; Token Type: none; MD streams: #~, #-

Entry Name	Entry Type	Comments
PackingSize	*USHORT*	Power of 2, from 1 through 128
ClassSize	*ULONG*	
Parent	RID: TypeDef	

FieldLayout; RID type: 16; Token type: none; MD streams: #~, #-

Entry Name	Entry Type	Comments
OffSet	ULONG	Offset in bytes or ordinal
Field	RID: Field	

StandAloneSig; RID type: 17; Token type: 0x11000000; MD streams: #~, #-

Entry Name	Entry Type	Comments
Signature	*BLOB*	Cannot be 0

EventMap; RID type: 18; Token type: none; MD streams: #~, #-

Entry Name	Entry Type	Comments
Parent	RID: TypeDef	
EventList	RID: Event	

EventPtr; RID type: 19; Token type: none; MD stream: #-

Entry Name	Entry Type	Comments
Event	RID: Event	

Event; RID type: 20; Token type: 0x14000000; MD streams: #~, #-

Entry Name	Entry Type	Comments
EventFlags	*USHORT*	0x0000, 0x0200, or 0x0600
Name	*STRING*	
EventType	*TypeDefOrRef*	

PropertyMap; RID type: 21; Token type: none; MD streams: #~, #-

Entry Name	Entry Type	Comments
Parent	RID: TypeDef	
PropertyList	RID: Property	

PropertyPtr; RID type: 22; Token type: none; MD stream: #-

Entry Name	Entry Type	Comments
Property	RID: Property	

Property; RID type: 23; Token type: 0x17000000; MD streams: #~, #-

Entry Name	Entry Type	Comments
PropFlags	*USHORT*	Validity mask: 0x1600
Name	*STRING*	
Type	*BLOB*	Property signature

MethodSemantics; RID type: 24; Token type: none; MD streams: #~, #-

Entry Name	Entry Type	Comments
Semantic	*USHORT*	
Method	RID: Method	
Association	*HasSemantic*	

MethodImpl; RID type: 25; Token type: none; MD streams: #~, #-

Entry Name	Entry Type	Comments
Class	RID: TypeDef	
MethodBody	*MethodDefOrRef*	Overriding method
MethodDeclaration	*MethodDefOrRef*	Overridden method

ModuleRef; RID type: 26; Token type: 0x1A000000; MD streams: #~, #-

Entry Name	Entry Type	Comments
Name	*STRING*	No longer than 512 bytes

TypeSpec; RID type: 27; Token type: 0x1B000000; MD streams: #~, #-

Entry Name	Entry Type	Comments
Signature	*BLOB*	Cannot be 0

Table: ENCLog; RID type: 28; Token type: none; MD stream: #-

Entry Name	Entry Type	Comments
Token	*ULONG*	
FuncCode	*ULONG*	

Table: ImplMap; RID type: 29; Token type: none; MD streams: #~, #-

Entry Name	Entry Type	Comments
MappingFlags	*USHORT*	Validity mask: 0x0747
MemberForwarded	*MemberForwarded*	Method only
ImportName	*STRING*	Entry point name
ImportScope	RID: ModuleRef	*ModuleRef* to unmanaged DLL

ENCMap; RID type: 30; Token type: none; MD stream: #-

Entry Name	Entry Type	Comments
Token	*ULONG*	

FieldRVA; RID type: 31; Token type: none; MD streams: #~, #-

Entry Name	Entry Type	Comments
RVA	*ULONG*	
Field	RID: Field	

Assembly; RID type: 32; Token type: 0x20000000; MD streams: #~, #-

Entry Name	Entry Type	Comments
HashAlgId	*ULONG*	
MajorVersion	*USHORT*	
MinorVersion	*USHORT*	
BuildNumber	*USHORT*	
RevisionNumber	*USHORT*	
Flags	*ULONG*	Validity mask: 0x0000C031
PublicKey	*BLOB*	
Name	*STRING*	No path, no extension
Locale	*STRING*	

AssemblyProcessor; RID type: 33; Token type: none; Unused

Entry Name	Entry Type	Comments
Processor	*ULONG*	

Table: AssemblyOS; RID type: 34; Token type: none; Unused

Entry Name	Entry Type	Comments
OSPlatformId	*ULONG*	
OSMajorVersion	*ULONG*	
OSMinorVersion	*ULONG*	

AssemblyRef; RID type: 35; Token type: 0x23000000; MD streams: #~, #-

Entry Name	Entry Type	Comments
MajorVersion	*USHORT*	
MinorVersion	*USHORT*	
BuildNumber	*USHORT*	
RevisionNumber	*USHORT*	
Flags	*ULONG*	0x00000000 or 0x00000001
PublicKeyOrToken	*BLOB*	
Name	*STRING*	No path, no extension

(continued)

AssemblyRef; RID type: 35; Token type: 0x23000000; MD streams: #~, #- *(continued)*

Entry Name	Entry Type	Comments
Locale	*STRING*	
HashValue	*BLOB*	

AssemblyRefProcessor; RID type: 36; Token type: none; Unused

Entry Name	Entry Type	Comments
Processor	*ULONG*	
AssemblyRef	RID: AssemblyRef	

AssemblyRefOS; RID type: 37; Token type: none; Unused

Entry Name	Entry Type	Comments
OSPlatformId	*ULONG*	
OSMajorVersion	*ULONG*	
OSMinorVersion	*ULONG*	
AssemblyRef	RID: AssemblyRef	

File; RID type: 38; Token type: 0x26000000; MD streams: #~, #-

Entry Name	Entry Type	Comments
Flags	*ULONG*	0x00000000 or 0x00000001
Name	*STRING*	No path
HashValue	*BLOB*	

ExportedType; RID type: 39; Token type: 0x27000000; MD streams: #~, #-

Entry Name	Entry Type	Comments
Flags	*ULONG*	Validity mask: 0x00000007
TypeDefId	*ULONG*	*TypeDef* token in another module
TypeName	*STRING*	
TypeNamespace	*STRING*	
Implementation	*Implementation*	*File*, *ExportedType*

ManifestResource; RID type: 40; Token type: 0x28000000; MD streams: #~, #-

Entry Name	Entry Type	Comments
Offset	*ULONG*	
Flags	*ULONG*	0x000001 or 0x000002
Name	*STRING*	
Implementation	*Implementation*	0, *File*, *AssemblyRef*

NestedClass; RID type: 41; Token type: none; MD streams: #~, #-

Entry Name	Entry Type	Comments
NestedClass	RID: TypeDef	
EnclosingClass	RID: TypeDef	

Coded Token Types

Type	Tag
***TypeDefOrRef* (64): 3 referenced tables, tag size 2**	
TypeDef	0
TypeRef	1
TypeSpec	2
***HasConstant* (65): 3 referenced tables, tag size 2**	
Field	0
Param	1
Property	2
***HasCustomAttribute* (66): 19 referenced tables, tag size 5**	
Method	0
Field	1
TypeRef	2
TypeDef	3
Param	4
InterfaceImpl	5
MemberRef	6
Module	7
DeclSecurity	8

(continued)

Coded Token Types *(continued)*

Type	Tag
Property	9
Event	10
StandAloneSig	11
ModuleRef	12
TypeSpec	13
Assembly	14
AssemblyRef	15
File	16
ExportedType	17
ManifestResource	18
***HasFieldMarshal* (67): 2 referenced tables, tag size 1**	
Field	0
Param	1
***HasDeclSecurity* (68): 3 referenced tables, tag size 2**	
TypeDef	0
Method	1
Assembly	2
***MemberRefParent* (69): 5 referenced tables, tag size 3**	
TypeDef	0
TypeRef	1
ModuleRef	2
Method	3
TypeSpec	4
***HasSemantics* (70): 2 referenced tables, tag size 1**	
Event	0
Property	1
***MethodDefOrRef* (71): 2 referenced tables, tag size 1**	
Method	0
MemberRef	1
***MemberForwarded* (72): 2 referenced tables, tag size 1**	
Field	0
Method	1

(continued)

Coded Token Types *(continued)*

Type	Tag
***Implementation* (73): 3 referenced tables, tag size 2**	
File	0
AssemblyRef	1
ExportedType	2
***CustomAttributeType* (74): 5 referenced tables, tag size 3**	
TypeRef	0
TypeDef	1
Method	2
MemberRef	3
String	4
***ResolutionScope* (75): 4 referenced tables, tag size 2**	
Module	0
ModuleRef	1
AssemblyRef	2
TypeRef	3

Appendix C

IL Instruction Set Reference

Instruction Parameter Types

Type	Description
int8	Signed 1-byte integer
uint8	Unsigned 1-byte integer
int32	Signed 4-byte integer
uint32	Unsigned 4-byte integer
int64	Signed 8-byte integer
float32	4-byte floating point number
float64	8-byte floating point number
<Method>	*MethodDef* or *MemberRef* token
<Field>	*FieldDef* or *MemberRef* token
<Type>	*TypeDef*, *TypeRef*, or *TypeSpec* token
<Signature>	*StandAloneSig* token
<String>	User-defined string token

Evaluation Stack Types

Type	Description
int32	Signed 4-byte integer
int64	Signed 8-byte integer
Float	80-bit floating point number
&	Managed or unmanaged pointer
o	Object reference
*	Unspecified type

IL Instructions

Opcode	Name	Parameter(s)	Pop	Push
00	nop	-	-	-
01	break	-	-	-
02	ldarg.0	-	-	*
03	ldarg.1	-	-	*
04	ldarg.2	-	-	*
05	ldarg.3	-	-	*
06	ldloc.0	-	-	*
07	ldloc.1	-	-	*
08	ldloc.2	-	-	*
09	ldloc.3	-	-	*
0A	stloc.0	-	*	-
0B	stloc.1	-	*	-
0C	stloc.2	-	*	-
0D	stloc.3	-	*	-
0E	ldarg.s	*uint8*	-	*
0F	ldarga.s	*uint8*	-	&
10	starg.s	*uint8*	*	-
11	ldloc.s	*uint8*	-	*
12	ldloca.s	*uint8*	-	&
13	stloc.s	*uint8*	*	-
14	ldnull	-	-	&=0
15	ldc.i4.m1 ldc.i4.M1	-	-	*int32*=–1
16	ldc.i4.0	-	-	*int32*=0
17	ldc.i4.1	-	-	*int32*=1
18	ldc.i4.2	-	-	*int32*=2
19	ldc.i4.3	-	-	*int32*=3
1A	ldc.i4.4	-	-	*int32*=4

(continued)

IL Instructions *(continued)*

Opcode	Name	Parameter(s)	Pop	Push
1B	ldc.i4.5	-	-	*int32*=5
1C	ldc.i4.6	-	-	*int32*=6
1D	ldc.i4.7	-	-	*int32*=7
1E	ldc.i4.8	-	-	*int32*=8
1F	ldc.i4.s	*int8*	-	*int32*
20	ldc.i4	*int32*	-	*int32*
21	ldc.i8	*int64*	-	*int64*
22	ldc.r4	*float32*	-	*Float*
23	ldc.r8	*float64*	-	*Float*
25	dup	-	*	*,*
26	pop	-	*	-
27	jmp	*<Method>*	-	-
28	call	*<Method>*	N arguments	Ret.value
29	calli	*<Signature>*	N arguments	Ret.value
2A	ret	-	*	-
2B	br.s	*int8*	-	-
2C	brfalse.s brnull.s brzero.s	*int8*	*int32*	-
2D	brtrue.s brinst.s	*int8*	*int32*	-
2E	beq.s	*int8*	*,*	-
2F	bge.s	*int8*	*,*	-
30	bgt.s	*int8*	*,*	-
31	ble.s	*int8*	*,*	-
32	blt.s	*int8*	*,*	-
33	bne.un.s	*int8*	*,*	-
34	bge.un.s	*int8*	*,*	-
35	bgt.un.s	*int8*	*,*	-

(continued)

IL Instructions *(continued)*

Opcode	Name	Parameter(s)	Pop	Push
36	ble.un.s	*int8*	*,*	-
37	blt.un.s	*int8*	*,*	-
38	br	*int32*	-	-
39	brfalse brnull brzero	*int32*	*int32*	-
3A	brtrue brinst	*int32*	*int32*	-
3B	beq	*int32*	*,*	-
3C	bge	*int32*	*,*	-
3D	bgt	*int32*	*,*	-
3E	ble	*int32*	*,*	-
3F	blt	*int32*	*,*	-
40	bne.un	*int32*	*,*	-
41	bge.un	*int32*	*,*	-
42	bgt.un	*int32*	*,*	-
43	ble.un	*int32*	*,*	-
44	blt.un	*int32*	*,*	-
45	switch	(*uint32*=N) + N(*int32*)	*,*	-
46	ldind.i1	-	*&*	*int32*
47	ldind.u1	-	*&*	*int32*
48	ldind.i2	-	*&*	*int32*
49	ldind.u2	-	*&*	*int32*
4A	ldind.i4	-	*&*	*int32*
4B	ldind.u4	-	*&*	*int32*
4C	ldind.i8 ldind.u8	-	*&*	*int64*
4D	ldind.i	-	*&*	*int32*
4E	ldind.r4	-	*&*	*Float*
4F	ldind.r8	-	*&*	*Float*
50	ldind.ref	-	*&*	*&*
51	stind.ref	-	*&,&*	-
52	stind.i1	-	*int32,&*	-

IL Instructions *(continued)*

Opcode	Name	Parameter(s)	Pop	Push
53	stind.i2	-	*int32,&*	-
54	stind.i4	-	*int32,&*	-
55	stind.i8	-	*int32,&*	-
56	stind.r4	-	*Float,&*	-
57	stind.r8	-	*Float,&*	-
58	add	-	*,*	*
59	sub	-	*,*	*
5A	mul	-	*,*	*
5B	div	-	*,*	*
5C	div.un	-	*,*	*
5D	rem	-	*,*	*
5E	rem.un	-	*,*	*
5F	and	-	*,*	*
60	or	-	*,*	*
61	xor	-	*,*	*
62	shl	-	*,*	*
63	shr	-	*,*	*
64	shr.un	-	*,*	*
65	neg	-	*	*
66	not	-	*	*
67	conv.i1	-	*	*int32*
68	conv.i2	-	*	*int32*
69	conv.i4	-	*	*int32*
6A	conv.i8	-	*	*int64*
6B	conv.r4	-	*	*Float*
6C	conv.r8	-	*	*Float*
6D	conv.u4	-	*	*int32*
6E	conv.u8	-	*	*int64*
6F	callvirt	<*Method*>	N arguments	Ret.value
70	cpobj	< *Type* >	*&,&*	-
71	ldobj	<*Type*>	*&*	*

(continued)

IL Instructions *(continued)*

Opcode	Name	Parameter(s)	Pop	Push
72	ldstr	*<String>*	-	*o*
73	newobj	*<Method>*	N arguments	*o*
74	castclass	*<Type>*	*o*	*o*
75	isinst	*<Type>*	*o*	*int32*
76	conv.r.un	-	*	*Float*
79	unbox	*<Type>*	*o*	*&*
7A	throw	-	*o*	-
7B	ldfld	*<Field>*	*o/&*	*
7C	ldflda	*<Field>*	*o/&*	*&*
7D	stfld	*<Field>*	*o/&,**	-
7E	ldsfld	*<Field>*	-	*
7F	ldsflda	*<Field>*	-	*&*
80	stsfld	*<Field>*	*	-
81	stobj	*<Type>*	*&,**	-
82	conv.ovf.i1.un	-	*	*int32*
83	conv.ovf.i2.un	-	*	*int32*
84	conv.ovf.i4.un	-	*	*int32*
85	conv.ovf.i8.un	-	*	*int64*
86	conv.ovf.u1.un	-	*	*int32*
87	conv.ovf.u2.un	-	*	*int32*
88	conv.ovf.u4.un	-	*	*int32*
89	conv.ovf.u8.un	-	*	*int64*
8A	conv.ovf.i.un	-	*	*int32*
8B	conv.ovf.u.un	-	*	*int64*
8C	box	*<Type>*	*	*o*
8D	newarr	*<Type>*	*int32*	*o*
8E	ldlen	-	*o*	*int32*
8F	ldelema	*<Type>*	*int32,o*	*&*
90	ldelem.i1	-	*int32,o*	*int32*
91	ldelem.u1	-	*int32,o*	*int32*
92	ldelem.i2	-	*int32,o*	*int32*
93	ldelem.u2	-	*int32,o*	*int32*
94	ldelem.i4	-	*int32,o*	*int32*
95	ldelem.u4	-	*int32,o*	*int32*

IL Instructions _(continued)_

Opcode	Name	Parameter(s)	Pop	Push
96	ldelem.i8	-	_int32_,o	_int64_
	ldelem.u8			
97	ldelem.i	-	_int32,o_	_int32_
98	ldelem.r4	-	_int32,o_	_Float_
99	ldelem.r8	-	_int32,o_	_Float_
9A	ldelem.ref	-	_int32,o_	_o/&_
9B	stelem.i	-	_int32,int32,o_	-
9C	stelem.i1	-	_int32,int32,o_	-
9D	stelem.i2	-	_int32,int32,o_	-
9E	stelem.i4	-	_int32,int32,o_	-
9F	stelem.i8	-	_int64,int32,o_	-
A0	stelem.r4	-	_Float,int32,o_	-
A1	stelem.r8	-	_Float,int32,o_	-
A2	stelem.ref	-	_o/&,int32,o_	-
B3	conv.ovf.i1	-	*	_int32_
B4	conv.ovf.u1	-	*	_int32_
B5	conv.ovf.i2	-	*	_int32_
B6	conv.ovf.u2	-	*	_int32_
B7	conv.ovf.i4	-	*	_int32_
B8	conv.ovf.u4	-	*	_int32_
B9	conv.ovf.i8	-	*	_int64_
BA	conv.ovf.u8	-	*	_int64_
C2	refanyval	_<Type>_	*	_&_
C3	ckfinite	-	*	_Float_
C6	mkrefany	_<Type>_	_&_	*
D0	ldtoken	_<Type>/<Field>/<Method>_	-	_&_
D1	conv.u2	-	*	_int32_
D2	conv.u1	-	*	_int32_
D3	conv.i	-	*	_int32_
D4	conv.ovf.i	-	*	_int32_
D5	conv.ovf.u	-	*	_int32_
D6	add.ovf	-	*,*	*
D7	add.ovf.un	-	*,*	*
D8	mul.ovf	-	*,*	*

(continued)

IL Instructions *(continued)*

Opcode	Name	Parameter(s)	Pop	Push
D9	mul.ovf.un	-	*,*	*
DA	sub.ovf	-	*,*	*
DB	sub.ovf.un	-	*,*	*
DC	endfinally	-	-	-
	endfault			
DD	leave	_int32_	-	-
DE	leave.s	_int8_	-	-
DF	stind.i	-	_int32,&_	-
E0	conv.u	-	*	_int32_
FE 00	arglist	-	*	_&_
FE 01	ceq	-	*,*	_int32_
FE 02	cgt	-	*,*	_int32_
FE 03	cgt.un	-	*,*	_int32_
FE 04	clt	-	*,*	_int32_
FE 05	clt.un	-	*,*	_int32_
FE 06	ldftn	_<Method>_	-	_&_
FE 07	ldvirtftn	_<Method>_	_o_	_&_
FE 09	ldarg	_uint32_	-	*
FE 0A		_uint32_	-	_&_
	ldarga			
FE 0B	starg	_uint32_	*	-
FE 0C	ldloc	_uint32_	-	*
FE 0D	ldloca	_uint32_	-	_&_
FE 0E	stloc	_uint32_	*	-
FE 0F	localloc	-	_int32_	_&_
FE 11	endfilter	-	_int32_	-
FE 12	unaligned.	_uint8_	-	-
FE 13	volatile.	-	-	-
FE 14	tail.	-	-	-
FE 15	initobj	_<Type>_	_&_	-
FE 17	cpblk	-	_int32,&,&_	-
FE 18	initblk	-	_int32,int32,&_	-
FE 1A	rethrow	-	-	-
FE 1C	sizeof	_<Type>_	-	_int32_
FE 1D	refanytype	-	*	_&_

Appendix D

IL Assembler and Disassembler Command-Line Options

IL Assembler

The command-line structure of IL Assembler is as follows:

```
ilasm [<options>] <sourcefile> [<options>][<sourcefile>*]
```

The default source file extension is IL. Multiple source files are parsed in the order of their appearance on the command line. Because options do not need to appear in a prescribed order, options and names of source files can be intermixed. All options specified on the command line are pertinent to the entire set of source files.

All options are recognized by the first three characters following the option key, and all are case-insensitive. The option key can be a forward slash (/) or a hyphen (-). In options that specify parameters, the equality character (=) is interchangeable with the colon character (:). The following option notations are equivalent:

/OUTPUT=MyModule.dll

-OUTPUT:MyModule.dll

/out:MyModule.dll

-Outp:MyModule.dll

The following command-line options are defined for IL Assembler:

- ***/LISTING*** Type a formatted listing of the compilation result.
- ***/NOLOGO*** Suppress typing the logo and copyright statement.
- ***/QUIET*** Suppress reporting the compilation progress.
- ***/DLL*** Compile to a dynamic-link library.
- ***/EXE*** Compile to a runnable executable (the default).
- ***/DEBUG*** Include debug information and create a program database (PDB) file.

- ***/CLOCK*** Measure and report the compilation times.
- ***/RESOURCE=<res_file>*** Link the specified unmanaged resource file (*.res) into the resulting PE file. *<res_file>* must be a full filename, including the extension.
- ***/OUTPUT=<targetfile>*** Compile to the file whose name is specified. The file extension must be specified explicitly; there is no default. If this option is omitted, IL Assembler sets the name of the output file to that of the first source file and sets the extension of the output file to DLL if the */DLL* option is specified and to EXE otherwise.
- ***/KEY=<keyfile>*** Compile with a strong name signature. *<keyfile>* specifies the file containing the private encryption key.
- ***/KEY=@<keysource>*** Compile with a strong name signature. *<keysource>* specifies the name of the source of the private encryption key.
- ***/SUBSYSTEM=<int>*** Set the *Subsystem* value in the PE header. The most frequently used *<int>* values are 3 (Microsoft Windows console application) and 2 (Microsoft Windows GUI application).
- ***/FLAGS=<int>*** Set the *Flags* value in the common language runtime header. The most frequently used *<int>* values are 1 (pure-IL code) and 2 (mixed code). The third bit of the *<int>* value, indicating that the PE file is strong name signed, is ignored.
- ***/ALIGNMENT=<int>*** Set the *FileAlignment* value in the PE header. The *<int>* value must be a power of 2, in the range 512 to 65536.
- ***/BASE=<int>*** Set the *ImageBase* value in the PE header.
- ***/ERROR*** Attempt to create the PE file even if compilation errors have been reported.

  Using the */ERROR* option does not guarantee that the PE file will be created: some errors are abortive, and others lead specifically to a failure to create the PE file. This option also disables the following IL Assembler autocorrection features:

  - ❑ An unsealed value type is marked *sealed.*
  - ❑ A method declared as both *static* and *instance* is marked *static.*
  - ❑ A nonabstract, nonvirtual instance method of an interface is marked *abstract* and *virtual.*
  - ❑ A global *abstract* method is marked *nonabstract.*
  - ❑ Nonstatic global fields and methods are marked *static.*

> **Warning** Don't use the */ERROR* command-line option unless you're positive that you know what you're doing. It is very dangerous! You can create a monster that will crash your system.

IL Disassembler

The command-line structure of IL Disassembler is as follows:

```
ildasm [<options>] [<in_filename>] [<options>]
```

If no filename is specified, the disassembler starts in graphical mode. You can then open a specific file by using the File Open menu command or by dragging the file to the disassembler's tree view window.

All options are recognized by the first three characters following the option key, and all are case-insensitive. The option key can be a forward slash (/) or a hyphen (-). In options that specify parameters, the equality character (=) is interchangeable with the colon character (:).

The */ADVANCED* (*/ADV*) option sets the advanced mode of the disassembler, which offers additional viewing and dumping options. The */ADV* option must be specified before any of the advanced-only options. I recommend that you place the */ADV* option ahead of all other options.

Options for Output Redirection

- ***/OUT=<out_filename>*** Direct the output to a file rather than to a GUI.
- ***/OUT=CON*** Direct the output to a console window rather than to a GUI.
- ***/TEXT*** A shortcut for */OUT=CON*.

If the */OUT* option or the */TEXT* option is specified, the *<in_filename>* must be specified as well.

ILAsm Code Formatting Options (PE Files Only)

- ***/BYTES*** Show the actual IL stream bytes (in hexadecimal notation) as instruction comments.
- ***/RAWEH*** Show structured exception handling clauses in canonical (label) form.

- ***/TOKENS*** Show metadata token values as comments.
- ***/SOURCE*** Show original source lines as comments. This requires the presence of the PDB file accompanying the PE file being disassembled and the original source files. If the original source files cannot be found at the location specified in the PDB file, the disassembler tries to find them in the current directory.
- ***/LINENUM*** Include references to original source lines (*.line* directives). This requires the presence of the PDB file accompanying the PE file being disassembled.
- ***/VISIBILITY=<vis>[+<vis>...]*** Disassemble only the items with specified visibility. Visibility suboptions (*<vis>*) include the following:
  - ❑ ***PUB*** Public
  - ❑ ***PRI*** Private
  - ❑ ***FAM*** Family
  - ❑ ***ASM*** Assembly
  - ❑ ***FAA*** Family and assembly
  - ❑ ***FOA*** Family or assembly
  - ❑ ***PSC*** Private scope
- ***/PUBONLY*** A shortcut for */VIS=PUB*.
- ***/QUOTEALLNAMES*** Enclose all names in single quotation marks. By default, only names that don't match the ILAsm definition of a simple name are quoted.
- ***/NOBAR*** Suppress the pop-up window showing the disassembly progress bar.

Options for File Output (PE Files Only)

- ***/UTF8*** Use UTF-8 encoding for output. The default is ANSI.
- ***/UNICODE*** Use Unicode encoding for output.

Options for File or Console Output (PE Files Only)

- ***/NOIL*** Suppress ILAsm code output.
- ***/HEADER*** Include PE header information and runtime header information in the output.

- ***/ITEM=<class>[::<method>[(<sig>)]*** Disassemble the specified item only. If <*sig*> is not specified, all methods named <*method*> of <*class*> are disassembled. If <*method*> is not specified, all members of <*class*> are disassembled. For example, */ITEM="Foo"* produces the full disassembly of the *Foo* class and all its members; */ITEM="Foo::Bar"* produces the disassembly of all methods named *Bar* in the *Foo* class; */ITEM="Foo::Bar(void(int32,string))"* produces the disassembly of a single method, *void Foo::Bar(int32,string).*
- ***/STATS*** Include statistics of the image file; an advanced option.
- ***/CLASSLIST*** Include the list of classes defined in the module; an advanced option.
- ***/ALL*** Combine the */HEADER*, */BYTES*, and */TOKENS* options and, in advanced mode, the */CLASSLIST and /STATS* options.

Metadata Summary Option

The metadata summary option is available in advanced mode only. It is suitable for file or console output, and it is the only option that works for both PE and COFF managed files. If an object file or an object library file is specified as an input file, the IL Disassembler in advanced mode automatically invokes the metadata summary, ignoring all other options. In nonadvanced mode, the disassembler does nothing.

- ***/METAINFO[=<specifier>]*** Show the metadata summary. The optional <*specifier*> is one of the following:
  - ❑ ***MDH*** Show the metadata header information and sizes.
  - ❑ ***HEX*** Show the hexadecimal representation of the signatures.
  - ❑ ***CSV*** Show the sizes of the *#Strings*, *#Blob*, *#US*, and *#GUID* streams and the sizes of the metadata tables and their records.
  - ❑ ***UNR*** Show the list of unresolved method references and unimplemented method definitions.
  - ❑ ***VAL*** Invoke the metadata validator and show its output.
- ***/OBJECTFILE=<obj_file_name>*** Show the metadata summary of a single object file in the object library. This option is valid for managed LIB files only.

Appendix E

Offline Verification Tool Reference

An offline verification tool for managed PE files, PEVerify.exe, is distributed with the Microsoft .NET Framework SDK. The tool includes two components: the metadata validator, MDValidator, and the IL verifier, ILVerifier.

MDValidator works on the module level, running validity checks of the metadata of a specified managed module (PE file). It does not matter whether the specified module is a prime module or an auxiliary. If the specified module is a prime module of an assembly, MDValidator does not automatically check other modules of the same assembly.

ILVerifier works on the assembly level, loading the assembly in full in memory, resolving internal references, and verifying the IL code of the methods contained in the assembly. Consequently, ILVerifier fails if the specified PE file is not the prime module of the assembly.

The result of this discrepancy in the approaches taken by MDValidator and ILVerifier is that only single-module assemblies can be fully validated and verified in one pass of the verification tool.

The PEVerify tool sets the exit code to 1 if errors are found during the PE file verification and sets the code to 0 otherwise.

The command-line format is as follows:

```
peverify <PE_file> [<option>*]
```

Unlike the IL Assembler and the IL Disassembler, the PEVerify tool does not allow arbitrary positioning of filename and options on the command line; rather, the name of the PE file being verified must be the first command-line parameter. Also, unlike the assembler and disassembler options, which are recognized by their first three characters only, PEVerify options must be fully spelled out.

PEVerify options are case-insensitive, and the option key can be a forward slash (/) character or a hyphen (-) character. The equality character (=) cannot be replaced with the colon character (:).

The command-line options include the following:

- ***/IL*** Check the PE structure and verify the IL code.
- ***/MD*** Check the PE structure and validate the metadata. If neither */MD* nor */IL* is specified, the metadata validation is performed first; then, if no metadata errors were found, the IL verification is performed. If either the */MD* or */IL* option is specified, only the metadata validation or the IL verification, respectively, is performed. If both */MD* and */IL* options are specified, the metadata validation is performed, followed by the IL verification, regardless of whether errors were found during the metadata validation phase.
- ***/UNIQUE*** Disregard repeating error codes; report only the first occurrence of each error type.
- ***/HRESULT*** Display error codes in hexadecimal format.
- ***/CLOCK*** Measure and report validation and verification times.
- ***/IGNORE=<err_code>[,<err_code>...]*** Ignore the specified error codes. Error codes must be specified in hexadecimal format.
- ***/IGNORE=@<err_code_file>*** Ignore the error codes specified in *<err_code_file>*, which is a text file containing comma-separated and/or line-separated hexadecimal error codes.
- ***/BREAK=<maxErrorCount>*** Abort verification after *<maxErrorCount>* errors. The value of *<maxErrorCount>* is a decimal number; if it is negative or unspecified, *<maxErrorCount>* is set to 1.
- ***/QUIET*** Suppress reporting the errors; report only the file being verified and the end result of the verification.

The following example shows verification of an exceptionally buggy PE file, created using IL Assembler with the */ERROR* option:

```
D:\MTRY>peverify mtry.exe /md /il /hresult /unique

Microsoft (R) .NET Framework PE Verifier  Version 1.0.3304.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

[MD](0x8013121D): Error: TypeDef is marked ValueType but not marked
Sealed. [token:0x02000002]
[MD](0x80131256): Error: TypeDef is not marked Nested but has an
encloser type. [token:0x02000006]
[MD](0x8013126D): Error: Global item (field,method) must be Public,
Private, or PrivateScope. [token:0x04000002]
[MD](0x8013126E): Error: Global item (field,method) must be Static.
[token:0x04000002]
```

```
[MD](0x8013126A): Error: Field name value__ is reserved for Enums only.
[token:0x04000008]
[MD](0x80131B24): Error: Illegal use of type 'void' in signature.
[token:0x06000001]
[MD](0x801312DB): Error: Constructor, initializer must return void.
[token:0x06000005]
[MD](0x801312DF): Error: ELEMENT_TYPE_SENTINEL is only allowed in
MemberRef signatures. [token:0x06000009]
[MD](0x801312E2): Error: Trailing ELEMENT_TYPE_SENTINEL in signature.
[token:0x06000009]
[MD](0x80131239): Error: Signature has invalid calling convention=0x000
00023. [token:0x0600000D]
[MD](0x801312E0): Error: Signature containing ELEMENT_TYPE_SENTINEL
must be VARARG. [token:0x0A000006]
[MD](0x801312E1): Error: Multiple ELEMENT_TYPE_SENTINEL in signature.
[token:0x0A000006]
[MD](0x80131230): Error: FieldLayout2 record has Field token=0x04000003
marked Static. [token:0x00000001]
13 Errors Verifying mtry.exe
```

Error Codes and Messages

In the following tables, 0xff, 0xffff, and 0xffffffff denote hexadecimal numbers, and 99 denotes a decimal number.

Metadata Validation Error Codes and Messages

HRESULT	Error Message
0x80131203	Error (Structural): Table=0xffffffff, Col=0xffffffff, Row=0xffffffff, has rid out of range.
0x80131204	Error (Structural): Table=0xffffffff, Col=0xffffffff, Row=0xffffffff, has coded token type out of range.
0x80131205	Error (Structural): Table=0xffffffff, Col=0xffffffff, Row=0xffffffff, has coded rid out of range.
0x80131206	Error (Structural): Table=0xffffffff, Col=0xffffffff, Row=0xffffffff, has an invalid String offset.
0x80131207	Error (Structural): Table=0xffffffff, Col=0xffffffff, Row=0xffffffff, has an invalid GUID offset.
0x80131208	Error (Structural): Table=0xffffffff, Col=0xffffffff, Row=0xffffffff, has an invalid BLOB offset.
0x80131209	Error: Multiple module records found.
0x8013120A	Error: Module has no MVID.
0x8013120B	Error: TypeRef has no name.
0x8013120C	Error: TypeRef has a duplicate, token=0xffffffff.

(continued)

Metadata Validation Error Codes and Messages *(continued)*

HRESULT	Error Message
0x8013120D	Error: TypeDef has no name.
0x8013120E	Error: TypeDef has a duplicate based on name+namespace, token=0xffffffff.
0x8013120F	Warning: TypeDef has a duplicate based on GUID, token=0xffffffff.
0x80131210	Error: TypeDef that is not an Interface and not the Object class extends Nil token.
0x80131211	Error: TypeDef for Object class extends token=0xffffffff which is not nil.
0x80131212	Error: TypeDef extends token=0xffffffff which is marked Sealed.
0x80131213	Error: TypeDef is a Deleted record but not marked RTSpecialName.
0x80131214	Error: TypeDef is marked RTSpecialName but is not a Deleted record.
0x80131215	Error: MethodImpl overrides private method (token=0xffffffff).
0x80131216	Error: Assembly name contains path and/or extension.
0x80131217	Error: File has a reserved system name.
0x80131218	Error: MethodImpl has static overriding method (token=0xffffffff).
0x80131219	Error: TypeDef is marked Interface but not Abstract.
0x8013121A	Error: TypeDef is marked Interface but extends non-Nil token=0xffffffff.
0x8013121B	Warning: TypeDef is marked Interface but has no GUID.
0x8013121C	Error: MethodImpl overrides final method (token=0xffffffff).
0x8013121D	Error: TypeDef is marked ValueType but not marked Sealed.
0x8013121E	Error: Parameter has invalid flags set 0xffffffff.
0x8013121F	Error: InterfaceImpl has a duplicate, token=0xffffffff.
0x80131220	Error: MemberRef has no name.
0x80131221	Error: MemberRef name starts with _VtblGap.
0x80131222	Error: MemberRef name starts with _Deleted.
0x80131223	Error: MemberRef parent is Nil but the module is a PE file.
0x80131224	Error: MemberRef signature has invalid calling convention= 0xffffffff.
0x80131225	Error: MemberRef has MethodDef parent, but calling convention is not VARARG (parent:0xffffffff; callconv: 0xffffffff).
0x80131226	Error: MemberRef has different name than parent MethodDef, token=0xffffffff.

Metadata Validation Error Codes and Messages *(continued)*

HRESULT	Error Message
0x80131227	Error: MemberRef has fixed part of signature different from parent MethodDef, token=0xffffffff.
0x80131228	Warning: MemberRef has a duplicate, token=0xffffffff.
0x80131229	Error: ClassLayout has parent TypeDef token=0xffffffff marked AutoLayout.
0x8013122A	Error: ClassLayout has invalid PackingSize; valid set of values is {1,2,4,...,128} (parent: 0xffffffff; PackingSize: 99).
0x8013122B	Error: ClassLayout has a duplicate (parent: 0xffffffff; duplicate rid: 0xffffffff).
0x8013122C	Error: FieldLayout2 record has invalid offset (field: 0xffffffff; offset: 0xffffffff).
0x8013122D	Error: FieldLayout2 record for Field token=0xffffffff has TypeDefNil for parent.
0x8013122E	Error: FieldLayout2 record for field of type that has no ClassLayout record (field: 0xffffffff; type: 0xffffffff).
0x8013122F	Error: Explicit offset specified for field of type marked AutoLayout (field: 0xffffffff; type: 0xffffffff).
0x80131230	Error: FieldLayout2 record has Field token=0xffffffff marked Static.
0x80131231	Error: FieldLayout2 record has a duplicate, rid=0xffffffff.
0x80131232	Error: ModuleRef has no name.
0x80131233	Warning: ModuleRef has a duplicate, token=0xffffffff.
0x80131234	Error: TypeRef has invalid resolution scope.
0x80131235	Error: TypeDef is marked Nested but has no encloser type.
0x80131236	Warning: Type extends TypeRef which resolves to TypeDef in the same module (TypeRef: 0xffffffff; TypeDef: 0xffffffff).
0x80131237	Error: Signature has zero size.
0x80131238	Error: Signature does not have enough bytes left at byte=0xffffffff as indicated by the compression scheme.
0x80131239	Error: Signature has invalid calling convention=0xffffffff.
0x8013123A	Error: Method is marked Static but calling convention=0xffffffff is marked HASTHIS.
0x8013123B	Error: Method is not marked Static, but calling convention=0xffffffff is not marked HASTHIS.
0x8013123C	Error: Signature has no argument count at byte=0xffffffff.
0x8013123D	Error: Signature missing element type after modifier (modifier: 0xff; offset: 0xffffffff).
0x8013123E	Error: Signature missing token after element 0xffff.

(continued)

Metadata Validation Error Codes and Messages *(continued)*

HRESULT	Error Message
0x8013123F	Error: Signature has an invalid token (token: 0xffffffff; offset: 0xffffffff).
0x80131240	Error: Signature missing function pointer at byte=0xffffffff.
0x80131241	Error: Signature has function pointer missing argument count at byte=0xffffffff.
0x80131242	Error: Signature missing rank at byte=0xffffffff.
0x80131243	Error: Signature missing count of sized dimensions of array at byte=0xffffffff.
0x80131244	Error: Signature missing size of dimension of array at byte=0xffffffff.
0x80131245	Error: Signature missing count of lower bounds of array at byte=0xffffffff.
0x80131246	Error: Signature missing lower bound of array at byte=0xffffffff.
0x80131247	Error: Signature has invalid ELEMENT_TYPE_* (element type: 0xffffffff; offset: 0xffffffff).
0x80131248	Error: Signature missing size for VALUEARRAY at byte=0xffffffff.
0x80131249	Error: Field signature has invalid calling convention=0xffffffff.
0x8013124A	Error: Method has no name.
0x8013124B	Error: Method parent is Nil.
0x8013124C	Error: Method has a duplicate, token=0xffffffff.
0x8013124D	Error: Field has no name.
0x8013124E	Error: Field parent is Nil.
0x8013124F	Error: Field has a duplicate, token=0xffffffff.
0x80131250	Error: Multiple assembly records found.
0x80131251	Error: Assembly has no name.
0x80131252	Error: Token 0xffffffff following ELEMENT_TYPE_CLASS (_VALUETYPE) in signature is a ValueType (Class, respectively).
0x80131253	Error: ClassLayout has parent TypeDef token=0xffffffff marked Interface.
0x80131254	Error: AssemblyOS entry has invalid platform id=0xffffffff.
0x80131255	Error: AssemblyRef has no name.
0x80131256	Error: TypeDef is not marked Nested but has an encloser type.
0x80131257	Error: AssemblyRefOS entry for AssemblyRef=0xffffffff has invalid platform id=0xffffffff.

Metadata Validation Error Codes and Messages *(continued)*

HRESULT	Error Message
0x80131258	Error: File has no name.
0x80131259	Error: ExportedType has no name.
0x8013125A	Error: TypeDef extends its own child.
0x8013125B	Error: ManifestResource has no name.
0x8013125C	Error: File has a duplicate, token=0xffffffff.
0x8013125D	Error: File name is fully-qualified, but should not be.
0x8013125E	Error: ExportedType has a duplicate, token=0xffffffff.
0x8013125F	Error: ManifestResource has a duplicate by name, token=0xffffffff.
0x80131260	Error: ManifestResource is not marked Public or Private.
0x80131262	Error: Field value__ (token=0xffffffff) in Enum is marked static.
0x80131263	Error: Field value__ (token=0xffffffff) in Enum is not marked RTSpecialName.
0x80131264	Error: Field (token=0xffffffff) in Enum is not marked static.
0x80131265	Error: Field (token=0xffffffff) in Enum is not marked literal.
0x80131267	Error: Signature of field (token=0xffffffff) in Enum does not match enum type.
0x80131268	Error: Field value__ (token=0xffffffff) in Enum is not the first one.
0x80131269	Error: Field (token=0xffffffff) is marked RTSpecialName but not named value__.
0x8013126A	Error: Field name value__ is reserved for Enums only.
0x8013126B	Error: Instance field in Interface.
0x8013126C	Error: Non-public field in Interface.
0x8013126D	Error: Global item (field,method) must be Public, Private, or PrivateScope.
0x8013126E	Error: Global item (field,method) must be Static.
0x80131270	Error: Type/instance constructor has zero RVA.
0x80131271	Error: Field is marked marshaled but has no marshaling information.
0x80131272	Error: Field has marshaling information but is not marked marshaled.
0x80131273	Error: Field is marked HasDefault but has no const value.
0x80131274	Error: Field has const value but is not marked HasDefault.
0x80131275	Error: Item (field,method) is marked HasSecurity but has no security information.

(continued)

Metadata Validation Error Codes and Messages *(continued)*

HRESULT	Error Message
0x80131276	Error: Item (field,method) has security information but is not marked HasSecurity.
0x80131277	Error: PInvoke item (field,method) must be Static.
0x80131278	Error: PInvoke item (field,method) has no Implementation Map.
0x80131279	Error: Item (field,method) has Implementation Map but is not marked PInvoke.
0x8013127A	Warning: Item (field,method) has invalid Implementation Map.
0x8013127B	Error: Implementation Map has invalid Module Ref, token 0xffffffff.
0x8013127C	Error: Implementation Map has invalid Member Forwarded, token 0xffffffff.
0x8013127D	Error: Implementation Map has no import name.
0x8013127E	Error: Implementation Map has invalid calling convention 0xff.
0x8013127F	Error: Item (field,method) has invalid access flag.
0x80131280	Error: Field marked both InitOnly and Literal.
0x80131281	Error: Literal field must be Static.
0x80131282	Error: Item (field,method) is marked RTSpecialName but not SpecialName.
0x80131283	Error: Abstract method in non-abstract type (token=0xffffffff).
0x80131284	Error: Neither static nor abstract method in interface (token=0xffffffff).
0x80131285	Error: Non-public method in interface (token=0xffffffff).
0x80131286	Error: Instance constructor in interface (token=0xffffffff).
0x80131287	Error: Global constructor.
0x80131288	Error: Static instance constructor in type (token=0xffffffff).
0x80131289	Error: Constructor/initializer in type (token=0xffffffff) is not marked SpecialName,RTSpecialName.
0x8013128A	Error: Virtual constructor/initializer in type (token=0xffffffff).
0x8013128B	Error: Abstract constructor/initializer in type (token=0xffffffff).
0x8013128C	Error: Non-static type initializer in type (token=0xffffffff).
0x8013128D	Error: Method marked Abstract/Runtime/InternalCall/Imported must have zero RVA, and vice versa.
0x8013128E	Error: Method marked Final/NewSlot but not Virtual.
0x8013128F	Error: Static method can not be Final or Virtual.
0x80131290	Error: Method can not be both Abstract and Final.

Metadata Validation Error Codes and Messages *(continued)*

HRESULT	Error Message
0x80131291	Error: Abstract method marked ForwardRef.
0x80131292	Error: Abstract method marked PInvokeImpl.
0x80131293	Error: Abstract method not marked Virtual.
0x80131294	Error: Nonabstract method not marked ForwardRef.
0x80131295	Error: Nonabstract method must have RVA or be PInvokeImpl or Runtime.
0x80131296	Error: PrivateScope method has zero RVA.
0x80131297	Error: Global method marked Abstract,Virtual.
0x80131298	Error: Signature contains long form (such as ELEMENT_TYPE_CLASS <token of System.String>).
0x80131299	Warning: Method has multiple semantics.
0x8013129A	Error: Method has invalid semantic association (token=0xffffffff).
0x8013129B	Error: Method has semantic association (token=0xffffffff) that does not exist.
0x8013129C	Error: MethodImpl overrides non-virtual method (token=0xffffffff).
0x8013129E	Error: Method has multiple semantic flags set for association (token=0xffffffff).
0x8013129F	Error: Method has no semantic flags set for association (token=0xffffffff).
0x801312A1	Warning: Unrecognized Hash Algorithm ID (0xffffffff).
0x801312A4	Error: Constant parent token (0xffffffff) is out of range.
0x801312A5	Error: Invalid Assembly flags (0x%ffff).
0x801312A6	Warning: TypeDef (token=0xffffffff) has same name as TypeRef.
0x801312A7	Error: InterfaceImpl has invalid implementing type (0xffffffff).
0x801312A8	Error: InterfaceImpl has invalid implemented type (0xffffffff).
0x801312A9	Error: TypeDef has security information but is not marked Has-Security.
0x801312AA	Error: TypeDef is marked HasSecurity but has no security information.
0x801312AB	Error: Type constructor must have no arguments.
0x801312AC	Error: ExportedType has invalid Implementation (token=0xffffffff).
0x801312AD	Error: MethodImpl has body from another TypeDef (token=0xffffffff).

(continued)

Metadata Validation Error Codes and Messages *(continued)*

HRESULT	Error Message
0x801312AE	Error: Type constructor has invalid calling convention.
0x801312AF	Error: MethodImpl has invalid Type token=0xffffffff.
0x801312B0	Error: MethodImpl declared in Interface (token=0xffffffff).
0x801312B1	Error: MethodImpl has invalid MethodDeclaration token=0xffffffff.
0x801312B2	Error: MethodImpl has invalid MethodBody token=0xffffffff.
0x801312B3	Error: MethodImpl has a duplicate (rid=0xffffffff).
0x801312B4	Error: Field has invalid parent (token=0xffffffff).
0x801312B5	Warning: Parameter out of sequence (parameter: 99; seq.num: 99).
0x801312B6	Error: Parameter has sequence number exceeding number of arguments (parameter: 99; seq.num: 99; num.args: 99).
0x801312B7	Error: Parameter #99 is marked HasFieldMarshal but has no marshaling information.
0x801312B8	Error: Parameter #99 has marshaling information but is not marked HasFieldMarshal.
0x801312BA	Error: Parameter #99 is marked HasDefault but has no const value.
0x801312BB	Error: Parameter #99 has const value but is not marked HasDefault.
0x801312BC	Error: Property has invalid scope (token=0xffffffff).
0x801312BD	Error: Property has no name.
0x801312BE	Error: Property has no signature.
0x801312BF	Error: Property has a duplicate (token=0xffffffff).
0x801312C0	Error: Property has invalid calling convention (0xff).
0x801312C1	Error: Property is marked HasDefault but has no const value.
0x801312C2	Error: Property has const value but is not marked HasDefault.
0x801312C3	Error: Property has related method with invalid semantics (method: 0xffffffff; semantics: 0xffffffff).
0x801312C4	Error: Property has related method with invalid token (0xffffffff).
0x801312C5	Error: Property has related method belonging to another type (method: 0xffffffff; type: 0xffffffff).
0x801312C6	Error: Constant of type (0xff) must have null value.
0x801312C7	Error: Constant of type (0xff) must have non-null value.
0x801312C8	Error: Event has invalid scope (token=0xffffffff).
0x801312CA	Error: Event has no name.

Metadata Validation Error Codes and Messages *(continued)*

HRESULT	Error Message
0x801312CB	Error: Event has a duplicate (token=0xffffffff).
0x801312CC	Error: Event has invalid EventType (token=0xffffffff).
0x801312CD	Error: Event's EventType (token=0xffffffff) is not a class (flags=0xffffffff).
0x801312CE	Error: Event has related method with invalid semantics (method: 0xffffffff; semantics: 0xffffffff).
0x801312CF	Error: Event has related method with invalid token (0xffffffff).
0x801312D0	Error: Event has related method belonging to another type (method: 0xffffffff; type: 0xffffffff).
0x801312D1	Error: Event has no AddOn related method.
0x801312D2	Error: Event has no RemoveOn related method.
0x801312D3	Error: ExportedType has same namespace+name as TypeDef, token 0xffffffff.
0x801312D4	Error: ManifestResource refers to non-PE file but offset is not 0.
0x801312D5	Error: Decl.Security is assigned to invalid item (token=0xffffffff).
0x801312D6	Error: Decl.Security has invalid action flag (0xffffffff).
0x801312D7	Error: Decl.Security has no associated permission BLOB.
0x801312D8	Error: ManifestResource has invalid Implementation (token=0xffffffff).
0x801312DB	Error: Constructor, initializer must return void.
0x801312DC	Error: Event's Fire method (0xffffffff) must return void.
0x801312DD	Warning: Invalid locale string.
0x801312DE	Error: Constant has parent of invalid type (token=0xffffffff).
0x801312DF	Error: ELEMENT_TYPE_SENTINEL is only allowed in MemberRef signatures.
0x801312E0	Error: Signature containing ELEMENT_TYPE_SENTINEL must be VARARG.
0x801312E1	Error: Multiple ELEMENT_TYPE_SENTINEL in signature.
0x801312E2	Error: Trailing ELEMENT_TYPE_SENTINEL in signature.
0x801312E3	Error: Signature is missing argument # 99.
0x801312E4	Error: Field of byref type.
0x801312E5	Error: Synchronized method in ValueType (token=0xffffffff).
0x801312E6	Error: Full name length exceeds maximum allowed (length: 99; max: 99).
0x801312E7	Error: Duplicate Assembly Processor record (0xffffffff).

(continued)

Metadata Validation Error Codes and Messages *(continued)*

HRESULT	Error Message
0x801312E8	Error: Duplicate Assembly OS record (0xffffffff).
0x801312E9	Error: ManifestResource has invalid flags (0xffffffff).
0x801312EA	Warning: ExportedType has no TypeDefId.
0x801312EB	Error: File has invalid flags (0xffffffff).
0x801312EC	Error: File has no hash BLOB.
0x801312ED	Error: Module has no name.
0x801312EE	Error: Module name is fully-qualified.
0x801312EF	Error: TypeDef marked as RTSpecialName but not SpecialName.
0x801312F0	Error: TypeDef extends an Interface (token=0xffffffff).
0x801312F1	Error: Type/instance constructor marked PInvokeImpl.
0x801312F2	Error: System.Enum is not marked Class.
0x801312F3	Error: System.Enum must extend System.ValueType.
0x801312F4	Error: MethodImpl's Decl and Body method signatures do not match.
0x801312F5	Error: Enum has method(s).
0x801312F6	Error: Enum implements interface(s).
0x801312F7	Error: Enum has properties.
0x801312F8	Error: Enum has one or more events.
0x801312F9	Error: TypeDef has invalid Method List (> Nmethods+1).
0x801312FA	Error: TypeDef has invalid Field List (> Nfields+1).
0x801312FB	Error: Constant has illegal type (0xff).
0x801312FC	Error: Enum has no instance field.
0x80131B00	Error: InterfaceImpl's implemented type (0xffffffff) not marked tdInterface.
0x80131B01	Error: Field is marked HasRVA but has no RVA record.
0x80131B02	Error: Field is assigned zero RVA.
0x80131B03	Error: Method has both RVA!=0 and Implementation Map.
0x80131B04	Error: Extraneous bits in Flags (0xffffffff).
0x80131B05	Error: TypeDef extends itself.
0x80131B06	Error: System.ValueType must extend System.Object.
0x80131B07	Warning: TypeDef extends TypeSpec (0xffffffff), not supported in Version 1.
0x80131B09	Error: Value class has neither fields nor size parameter.
0x80131B0A	Error: Interface is marked Sealed.
0x80131B0B	Error: NestedClass token (0xffffffff) in NestedClass record is not a valid TypeDef.

Metadata Validation Error Codes and Messages *(continued)*

HRESULT	Error Message
0x80131B0C	Error: EnclosingClass token (0xffffffff) in NestedClass record is not a valid TypeDef.
0x80131B0D	Error: Duplicate NestedClass record (0xffffffff).
0x80131B0E	Error: Nested type token has multiple EnclosingClass tokens (nested: 0xffffffff; enclosers: 0xffffffff, 0xffffffff).
0x80131B0F	Error: Zero RVA of field 0xffffffff in FieldRVA record.
0x80131B10	Error: Invalid field token in FieldRVA record (field: 0xffffffff; RVA: 0xffffffff).
0x80131B11	Error: Same RVA in another FieldRVA record (RVA: 0xffffffff; field: 0xffffffff).
0x80131B12	Error: Same field in another FieldRVA record(field: 0xffffffff; record: 0xffffffff).
0x80131B13	Error: Invalid token specified as EntryPoint in CLR header.
0x80131B14	Error: Instance method token specified as EntryPoint in CLR header.
0x80131B15	Error: Invalid type of instance field(0xffffffff) of an Enum.
0x80131B16	Error: Method has invalid RVA (0xffffffff).
0x80131B17	Error: Literal field has no const value.
0x80131B18	Error: Class implements interface but not method (class:0xffffffff; interface:0xffffffff; method:0xffffffff).
0x80131B19	Error: CustomAttribute has invalid Parent token (0xffffffff).
0x80131B1A	Error: CustomAttribute has invalid Type token (0xffffffff).
0x80131B1B	Error: CustomAttribute has non-constructor Type (0xffffffff).
0x80131B1C	Error: CustomAttribute's Type (0xffffffff) has invalid signature.
0x80131B1D	Error: CustomAttribute's Type (0xffffffff) has no signature.
0x80131B1E	Error: CustomAttribute's blob has invalid prolog (0xffff).
0x80131B1F	Error: Method has invalid local signature token (0xffffffff).
0x80131B20	Error: Method has invalid header.
0x80131B21	Error: EntryPoint method has more than one argument.
0x80131B22	Error: EntryPoint method must return void, int or unsigned int.
0x80131B23	Error: EntryPoint method must have vector of strings as argument, or no arguments.
0x80131B24	Error: Illegal use of type 'void' in signature.

IL Verification Error Codes and Messages

HRESULT	Error Message
0x80131810	Unknown opcode [0xffffffff].
0x80131811	Unknown calling convention [0xffffffff].
0x80131812	Unknown ELEMENT_TYPE [0xffffffff].
0x80131818	Internal error.
0x80131819	Stack is too large.
0x8013181A	Array name is too long.
0x80131820	fall thru end of the method
0x80131821	try start >= try end
0x80131822	try end > code size
0x80131823	handler >= handler end
0x80131824	handler end > code size
0x80131825	filter >= code size
0x80131826	Try starts in the middle of an instruction.
0x80131827	Handler starts in the middle of an instruction.
0x80131828	Filter starts in the middle of an instruction.
0x80131829	Try block overlap with another block.
0x8013182A	Try and filter/handler blocks are equivalent.
0x8013182B	Try shared between finally and fault.
0x8013182C	Handler block overlaps with another block.
0x8013182D	Handler block is the same as another block.
0x8013182E	Filter block overlaps with another block.
0x8013182F	Filter block is the same as another block.
0x80131830	Filter contains try.
0x80131831	Filter contains handler.
0x80131832	Nested filters.
0x80131833	filter >= code size
0x80131834	Filter starts in the middle of an instruction.
0x80131835	fallthru the end of an exception block
0x80131836	fallthru into an exception handler
0x80131837	fallthru into an exception filter
0x80131838	Leave from outside a try or catch block.
0x80131839	Rethrow from outside a catch handler.
0x8013183A	Endfinally from outside a finally handler
0x8013183B	Endfilter from outside an exception filter block

IL Verification Error Codes and Messages *(continued)*

HRESULT	Error Message
0x8013183C	Missing Endfilter.
0x8013183D	Branch into try block.
0x8013183E	Branch into exception handler block.
0x8013183F	Branch into exception filter block.
0x80131840	Branch out of try block.
0x80131841	Branch out of exception handler block.
0x80131842	Branch out of exception filter block.
0x80131843	Branch out of finally block.
0x80131844	Return out of try block.
0x80131845	Return out of exception handler block.
0x80131846	Return out of exception filter block.
0x80131847	jmp / exception into the middle of an instruction
0x80131848	Non-compatible types depending on path.
0x80131849	Init state for this differs depending on path.
0x8013184A	Non-compatible types on stack depending on path.
0x8013184B	Stack depth differs depending on path.
0x8013184C	Instance variable (this) missing.
0x8013184D	Uninitialized this on entering a try block.
0x8013184E	Store into this when it is uninitialized.
0x8013184F	Return from ctor when this is uninitialized.
0x80131850	Return from ctor before all fields are initialized.
0x80131851	Branch back when this is uninitialized.
0x80131852	Expected byref of value type for this parameter.
0x80131853	Non-compatible types on the stack.
0x80131854	Unexpected type on the stack.
0x80131855	Missing stack slot for exception.
0x80131856	Stack overflow.
0x80131857	Stack underflow.
0x80131858	Stack empty.
0x80131859	Uninitialized item on stack.
0x8013185A	Expected I, I4, or I8 on the stack.
0x8013185B	Expected R, R4, or R8 on the stack.
0x8013185C	Unexpected R, R4, R8, or I8 on the stack.
0x8013185D	Expected numeric type on the stack.

(continued)

IL Verification Error Codes and Messages *(continued)*

HRESULT	Error Message
0x8013185E	Expected an Objref on the stack.
0x8013185F	Expected address of an Objref on the stack.
0x80131860	Expected Byref on the stack.
0x80131861	Expected pointer to function on the stack.
0x80131862	Expected single dimension array on the stack.
0x80131863	Expected value type instance on the stack.
0x80131864	Expected address of value type on the stack.
0x80131865	Unexpected value type instance on the stack.
0x80131866	Local variable is unusable at this point.
0x80131867	Unrecognized local variable number.
0x80131868	Unrecognized argument number.
0x80131869	Unable to resolve token.
0x8013186A	Unable to resolve type of the token.
0x8013186B	Expected memberRef/memberDef token.
0x8013186C	Expected memberRef/fieldDef token.
0x8013186D	Expected signature token.
0x8013186E	Instruction can not be verified.
0x8013186F	Operand does not point to a valid string ref.
0x80131870	Return type is BYREF, TypedReference, ArgHandle, or ArgIterator.
0x80131871	Stack must be empty on return from a void function.
0x80131872	Return value missing on the stack.
0x80131873	Stack must contain only the return value.
0x80131874	Return uninitialized data.
0x80131875	Illegal array access.
0x80131876	Store non Object type into Object array.
0x80131877	Expected single dimension array.
0x80131878	Expected single dimension array of pointer types.
0x80131879	Array field access is denied.
0x8013187A	Allowed only in vararg methods.
0x8013187B	Value type expected.
0x8013187C	Method is not visible.
0x8013187D	Field is not visible.
0x8013187E	Item is unusable at this point.
0x8013187F	Expected static field.

IL Verification Error Codes and Messages *(continued)*

HRESULT	Error Message
0x80131880	Expected non-static field.
0x80131881	Address of not allowed for this item.
0x80131882	Address of not allowed for byref.
0x80131883	Address of not allowed for literal field.
0x80131884	Can not change initonly field outside its ctor.
0x80131885	Can not throw this object.
0x80131886	Callvirt on a value type method.
0x80131887	Call signature mismatch.
0x80131888	Static function expected.
0x80131889	Ctor expected.
0x8013188A	Can not use callvirt on ctor.
0x8013188B	Only super::ctor or typeof(this)::ctor allowed here.
0x8013188C	Possible call to ctor more than once.
0x8013188D	Unrecognized signature.
0x8013188E	Can not resolve Array type.
0x8013188F	Array of ELEMENT_TYPE_PTR.
0x80131890	Array of ELEMENT_TYPE_BYREF or ELEMENT_TYPE_TYPEDBYREF.
0x80131891	ELEMENT_TYPE_PTR can not be verified.
0x80131892	Unexpected vararg.
0x80131893	Unexpected Void.
0x80131894	BYREF of BYREF
0x80131896	Code size is zero.
0x80131897	Unrecognized use of vararg.
0x80131898	Missing call/callvirt/calli.
0x80131899	Can not pass byref to a tail call.
0x8013189A	Missing ret.
0x8013189B	Void ret type expected for tail call.
0x8013189C	Tail call return type not compatible.
0x8013189D	Stack not empty after tail call.
0x8013189E	Method ends in the middle of an instruction.
0x8013189F	Branch out of the method.
0x801318A0	Finally handler blocks overlap.
0x801318A1	Lexical nesting.
0x801318A2	Missing ldsfld/stsfld/ldind/stind/ldfld/stfld/ldobj/stobj/initblk/cpblk.

(continued)

IL Verification Error Codes and Messages *(continued)*

HRESULT	Error Message
0x801318A3	Missing ldind/stind/ldfld/stfld/ldobj/stobj/initblk/cpblk.
0x801318A4	Innermost exception blocks should be declared first.
0x801318A5	Calli not allowed on virtual methods.
0x801318A6	Call not allowed on abstract methods.
0x801318A7	Unexpected array type on the stack.
0x801318A9	Attempt to enter a try block with nonempty stack.
0x801318AA	Unrecognized arguments for delegate ctor.
0x801318AB	Delegate ctor not allowed at the start of a basic block when the function pointer argument is a virtual method.
0x801318AC	Dup, ldvirtftn, newobj delegate::.ctor() pattern expected (in the same basic block).
0x801318AD	Ldftn/ldvirtftn instruction required before call to a delegate ctor.
0x801318AE	Attempt to load address of an abstract method.
0x801318AF	ELEMENT_TYPE_CLASS ValueClass in signature.
0x801318B0	ELEMENT_TYPE_VALUETYPE non-ValueClass in signature.
0x801318B1	Box operation on TypedReference, ArgHandle, or ArgIterator.
0x801318B2	Byref of TypedReference, ArgHandle, or ArgIterator.
0x801318B3	Array of TypedReference, ArgHandle, or ArgIterator.
0x801318B4	Stack not empty when leaving an exception filter.
0x801318B5	Unrecognized delegate ctor signature; expected I.
0x801318B6	Unrecognized delegate ctor signature; expected Object.
0x801318B7	Mkrefany on TypedReference, ArgHandle, or ArgIterator.
0x801318B8	Value type not allowed as catch type.
0x801318B9	filter block should immediately precede handler block
0x801318BA	ldvirtftn on static
0x801318BB	callvirt on static
0x801318BC	initlocals must be set for verifiable methods with one or more local variables.
0x801318BD	branch/leave to the beginning of a catch/filter handler
0x801318F0	Unverifiable PE Header/native stub.
0x801318F1	Unrecognized metadata, unable to verify IL.
0x801318F2	Unrecognized appdomain pointer.
0x801318F3	Type load failed.
0x801318F4	Module load failed.

Index

Send feedback about this index to *mspindex@microsoft.com*

Serge Lidin

Serge Lidin has worked as a software developer for about 20 years—in different countries, in different languages and platforms, and in different areas, including astrophysic models, industrial process simulations, transaction processing in financial systems, and many others. For the last three years he has worked on the Microsoft .NET Common Language Runtime team. His responsibilities include design and development of the IL Assembler, IL Disassembler, Metadata Validator, and run-time metadata validation in the execution engine. He still has nostalgic memories of the times when he lugged around a bagful of punched cards but likes his present IBM laptop much better: it weighs less and holds more stuff. When not writing software or sleeping, he plays tennis and reads books (his literary taste is below any criticism). Serge shares his time between Vancouver, BC, where his heart is, and Redmond, WA, where his brain is.

Serge Lidin shaves head of the development lead Mike Magruder (Summer 2000). Runtime development manager Jason Zander and leads Larry Sullivan and Mike Magruder made a bet with the rest of the runtime team that the team would blow an important deadline. As a wager, they proposed to dye Jason's hair yellow and Larry's hair green and to shave Mike's head. They lost.

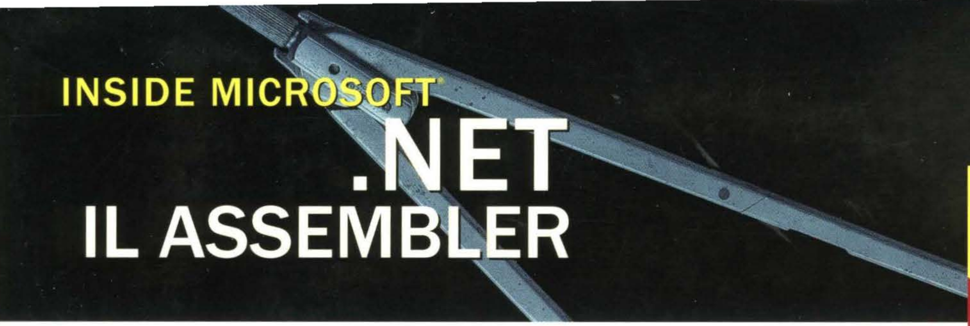
Drafting Compass

The drafting compass, an instrument for taking measurements and drawing circles, consists of two straight and equal legs connected at one end by a movable joint. The word *encompass*, "to encircle something," comes from the same source. The famous poem "A Valediction: Forbidding Mourning" by the 17th-century metaphysical poet John Donne uses the metaphor of twin compass legs to represent separated lovers.

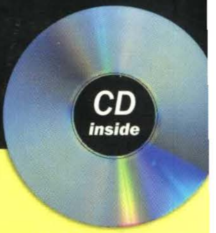
Tools are central to the progress of the human race. People are adept at building and using tools to accomplish important (and unimportant) tasks. Software is among the most powerful of tools moving us forward, and Microsoft is proud to create tools used by millions worldwide and to contribute to continuing innovation.

The manuscript for this book was prepared and galleyed using Microsoft Word. Pages were composed by Microsoft Press using Adobe FrameMaker+SGML for Windows, with text in Garamond and display type in Helvetica Condensed. Composed pages were delivered to the printer as electronic prepress files.

Cover Designer:	Methodologie, Inc.
Interior Graphic Designer:	James D. Kramer
Principal Compositor:	Kerri DeVault
Interior Artist:	Michael Kloepfer
Principal Copy Editor:	Lisa Pawlewicz
Indexer:	Hugh Maddocks



INSIDE MICROSOFT .NET IL ASSEMBLER



Includes a searchable eBook and sample code

Drill down into the details of the .NET runtime and the IL assembly language with help from one of the language's designers!

Most books about .NET programming are devoted to high-level languages in RAD environments. But what about the plumbing? This book drills down into the internal structures and operations of the .NET common language runtime and teaches you how to master the IL assembly language (ILAsm), which fully describes these structures and operations. The rule of thumb is that if the runtime can do it, ILAsm must be able to express it. You'll learn everything you need to know about this low-level language from the developer who designed and implemented the IL Assembler, IL Disassembler, and metadata validation tools. The book is ideal for compiler developers, developers working on multilanguage projects, and anyone else who wants to develop tighter, faster code for the .NET Framework platform.

Topics covered include:

> Overview of ILAsm and the common language runtime, plus a sample program

> Enhancing your code

> Managed executable file structure

> Metadata tables organization

> Modules and assemblies

> Namespaces and classes

> Primitive types and signatures

> Fields and data constants

> Methods

> IL instructions

> Structured exception handling

> Events and properties

> Custom and security attributes

> Managed and unmanaged code interoperation

> Multilanguage projects

> Appendixes: ILAsm grammar, metadata organization, the IL instruction set, compiler options, and the offline metadata validation tool

PHASE | Analyzing Business Requirements | Defining Technical Architecture | Solution Design | Coding/ Implementing | Testing/ Debugging | Deployment | Maintenance/ Troubleshooting

ISBN 0-7356-1547-0

Part No. X08-56869

7 90145 15470 5 9 780735 615472 90000

U.S.A. $49.99
Canada $72.99
[Recommended]

Programming/Microsoft Visual Studio .NET

CD-ROM features:

- A searchable electronic version of the book
- Sample code in IL assembly language

*For **System Requirements**, see the book's Introduction.*

About the Author:

Serge Lidin, a Russian-born Canadian with 20 years in the computer industry, has programmed in more languages and for more platforms that he can recall, in areas varying from astrophysics models to industrial process simulations to transaction processing in financial systems. For the last three years, he has worked on the Microsoft .NET Common Language Runtime team, where he has designed and developed the IL Assembler, IL Disassembler, Metadata Validator, and run-time metadata validation in the execution engine.

To see the full line of Microsoft Press® developer resources, go to:
microsoft.com/mspress/developer

Microsoft®