Introduction

Why was this book written? To tell the truth, I don't think I had much choice in this matter. This book is a revision and extension of my earlier book, *Inside Microsoft .NET IL Assembler*, which hit the shelves in early 2002, about a month after the release of version 1.0 of the .NET common language infrastructure (CLI). So, it is fairly obvious why I had to write this new book now, more than four years later, when the more powerful version 2.0 of the .NET CLI has just been released. And I don't think I had much choice in the matter of writing the first book either, because somebody had to start writing about the .NET CLI inner workings.

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. 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/ISO standardization documents regarding the .NET common language infrastructure. (ILAsm itself, as 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.

The 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 Visual C++/CLI, C#, or Visual Basic. But all .NET developers use the IL disassembler now and then, and many use it on a regular basis. A cyan thunderbolt—the IL disassembler icon (a silent praise for David Drake and his "Hammer's Slammers")—glows on the computer screens of .NET developers regardless of their language preferences and problem areas. And the text output of the IL disassembler is ILAsm source code.

Virtually all books about .NET-based programming that are devoted to high-level programming languages such as C# or Visual Basic 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 managed 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 the IL assembler and disassembler, it was my obligation to see it through all the way.

History of ILAsm, Part I

The first versions of the IL assembler and IL disassembler 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 left the common language runtime team 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 Common Runtime Odds and Ends Development Team (CROEDT). 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, the assembler and disassembler 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. The tools were still considered internal, so 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 the IL assembler input and IL disassembler output were synchronized enough to achieve limited round-tripping. *Round-tripping* means 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. The 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 NetCOBOL, 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, the IL assembler and IL disassembler 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.

ILAsm Marching On

These days the IL assembler is used more and more in the compiler and tool implementation, in education, and in academic research. The following compilers (for example), ranging from purely academic projects to industrial-strength systems, produce ILAsm code as their output and let the IL assembler take care of emitting the managed executables:

- Ada# (USAF Academy, Colorado)
- Alice.NET (Saarland University, Saarbrücken)
- Boo (codehaus.org)
- NetCOBOL (Fujitsu)
- COBOL2002 for .NET Framework (NEC/Hitachi)
- NetExpress COBOL (Microfocus)
- CommonLarceny.NET (Northeastern University, Boston)
- CULE.NET (CULEPlace.com)
- Component Pascal (Queensland University of Technology, Australia)
- Fortran (Lahey/Fujitsu)
- Hotdog Scheme (Northwestern University, Chicago)
- Lagoona.NET (University of California, Irvine)
- LCC (ANSI C) (Microsoft Research, Redmond)
- Mercury (University of Melbourne, Australia)
- Modula-2 (Queensland University of Technology, Australia)
- Moscow ML.NET (Royal Veterinary and Agricultural University, Denmark)
- Oberon.NET (Swiss Federal Institute of Technology, Zürich)
- S# (Smallscript.com)
- SML.NET (Microsoft Research, Cambridge, United Kingdom)

The ability of the IL disassembler and IL assembler to work in tandem gave birth to a slew of interesting tools and techniques based on "creative round-tripping" of managed executables (disassembling—text manipulation—reassembling). For example, Preemptive Software (a company known for its Java and .NET-oriented obfuscators and code optimizers) built its DotFuscator system on this base. The DotFuscator is a commercial, industrial-strength obfuscation and optimization system, well known on the market. I discuss some other interesting examples of application of "creative round-tripping" in Chapter 19.

Practically all academic courses on .NET programming use ILAsm to some extent (how else could the authors of these courses show the innards of .NET managed executables?). Some courses are completely ILAsm based, such as the course developed by Dr. Regeti Govindarajulu at International Institute of Informational Technologies (Hyderabad, India) and the course developed by Drs. Andrey Makarov, Sergey Skorobogatov, and Andrey Chepovskiy at Lomonosov University and Bauman Technical University (Moscow, Russia).

Who Should Read This Book

This book targets all the .NET-oriented developers who, working 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.

This analysis of disassemblies and metadata structuring is crucial in assessing the correctness and efficiency of any .NET-oriented compiler, so 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 the IL assembly language directly, such as 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 1, “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 2, “Underlying Structures,” describes the structure of a managed executable file and general metadata organization. Part 3, “Fundamental Components,” is dedicated to the components that constitute a necessary base of any application: assemblies, modules, classes, methods, fields, and related topics. Part 4, “Inside the Execution Engine,” brings you, yes, inside the execution engine, describing the execution of IL instructions and managed exception handling. Part 5, “Special Components,” discusses metadata representation and the usage of the additional components: events, properties, and custom and security attributes. And Part 6, “Interoperation,” describes the interoperation between managed and unmanaged code and discusses practical applications of the IL assembler and IL disassembler to multilanguage projects.

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