Understanding CIL and the Role of Dynamic Assemblies

When you are building a full-scale .NET application, you will most certainly make use of C# (or a similar managed language such as Visual Basic), given their inherent productivity and ease of use. However, as you learned in the very first chapter, the role of a managed compiler is to translate *.cs code files into terms of CIL code, type metadata, and an assembly manifest. As it turns out, CIL is a full-fledged .NET programming language, with its own syntax, semantics, and compiler (ildasm.exe).

In this chapter, you will be given a tour of .NET’s mother tongue. Here you will understand the distinction between a CIL directive, CIL attribute, and CIL opcode. You will then learn about the role of round-trip engineering of a .NET assembly and various CIL programming tools. The remainder of the chapter will then walk you through the basics of defining namespaces, types, and members using the grammar of CIL. We’ll wrap up with an examination of the role of the System.Reflection.Emit namespace and examine how it is possible to construct an assembly (with CIL instructions) dynamically at runtime.

Of course, few programmers will ever need to work with raw CIL code on a day-to-day basis. Therefore, I will start up this chapter by examining a few reasons why getting to know the syntax and semantics of this low-level .NET language might be worth your while.

Reasons for Learning the Grammar of CIL

CIL is the true mother tongue of the .NET platform. When you build a .NET assembly using your managed language of choice (C#, VB, F#, COBOL.NET, etc.), the associated compiler translates your source code into terms of CIL. Like any programming language, CIL provides numerous structural and implementation-centric tokens. Given that CIL is just another .NET programming language, it should come as no surprise that it is possible to build your .NET assemblies directly using CIL and the CIL compiler (ildasm.exe) that ships with the .NET Framework 4.5 SDK.

Now while it is true that few programmers would choose to build an entire .NET application directly with CIL, CIL is still an extremely interesting intellectual pursuit. Simply put, the more you understand the grammar of CIL, the better able you are to move into the realm of advanced .NET development. By way of some concrete examples, individuals who possess an understanding of CIL are capable of the following:

- Talking intelligently about how different .NET programming languages map their respective keywords to CIL tokens.
- Disassembling an existing .NET assembly, editing the CIL code, and recompiling the updated code base into a modified .NET binary. For example, there are some scenarios where you might need to modify CIL in order to interoperate with some advanced COM features.
• Building dynamic assemblies using the System.Reflection.Emit namespace. This API allows you to generate an in-memory .NET assembly, which can optionally be persisted to disk.

• Leveraging aspects of the CTS that are not supported by higher-level managed languages, but do exist at the level of CIL. To be sure, CIL is the only .NET language that allows you to access each and every aspect of the CTS. For example, using raw CIL, you are able to define global-level members and fields (which are not permissible in C#).

Again, to be perfectly clear, if you choose not to concern yourself with the details of CIL code, you are still absolutely able to gain mastery of C# and the .NET base class libraries. In many ways, knowledge of CIL is analogous to a C(++) programmer’s understanding of assembly language. Those who know the ins and outs of the low-level “goo” are able to create rather advanced solutions for the task at hand and gain a deeper understanding of the underlying programming (and runtime) environment. So, if you are up for the challenge, let’s begin to examine the details of CIL.

---

**Note** Understand that this chapter is not intended to be a comprehensive treatment of the syntax and semantics of CIL. If you require a full examination of the topic, I’d recommend downloading the official ECMA specification (ecma-335.pdf) from the ECMA International web site (www.ecma-international.org).

---

**Examining CIL Directives, Attributes, and Opcodes**

When you begin to investigate low-level languages such as CIL, you are guaranteed to find new (and often intimidating-sounding) names for very familiar concepts. For example, at this point in the text, if you were shown the following set of items

{new, public, this, base, get, set, explicit, unsafe, enum, operator, partial}

you would most certainly understand them to be keywords of the C# language (which is correct). However, if you look more closely at the members of this set, you might be able to see that while each item is indeed a C# keyword, it has radically different semantics. For example, the enum keyword defines a System.Enum-derived type, while the this and base keywords allow you to reference the current object or the object’s parent class, respectively. The unsafe keyword is used to establish a block of code that cannot be directly monitored by the CLR, while the operator keyword allows you to build a hidden (specially named) method that will be called when you apply a specific C# operator (such as the plus sign).

In stark contrast to a higher-level language such as C#, CIL does not just simply define a general set of keywords, per se. Rather, the token set understood by the CIL compiler is subdivided into the following three broad categories based on semantics:

• CIL directives

• CIL attributes

• CIL operation codes (opcodes)