**Metadata and Self-Describing Components**

This page is specific to

**Microsoft Visual Studio 2008/.NET Framework 3.5**

Other versions are also available for the following:

[Microsoft Visual Studio 2003/.NET Framework 1.1](http://msdn.microsoft.com/en-us/library/4y7k7c6k(VS.71).aspx)

[Microsoft Visual Studio 2005/.NET Framework 2.0](http://msdn.microsoft.com/en-us/library/4y7k7c6k(VS.80).aspx)

[.NET Framework 3.0](http://msdn.microsoft.com/en-us/library/4y7k7c6k(VS.85).aspx)

[Microsoft Visual Studio 2010/.NET Framework 4](http://msdn.microsoft.com/en-us/library/4y7k7c6k(VS.100).aspx)

In the past, a software component (.exe or .dll) written in one language could not easily use a software component written in another language. COM provided a step forward in solving this problem. The .NET Framework makes component interoperation even easier by allowing compilers to emit additional declarative information into all modules and assemblies. This information, called metadata, helps components to seamlessly interact.

http://i.msdn.microsoft.com/Global/Images/clear.gif In This Section

[Metadata Overview](http://msdn.microsoft.com/en-us/library/xcd8txaw.aspx)

Provides an overview of the contents and benefits of metadata.

[Structure and Use of Metadata](http://msdn.microsoft.com/en-us/library/2c3d7zbc.aspx)

Describes how metadata is implemented in a .NET Framework portable executable (PE) file.

**Metadata Overview**

Metadata is binary information describing your program that is stored either in a common language runtime portable executable (PE) file or in memory. When you compile your code into a PE file, metadata is inserted into one portion of the file, while your code is converted to Microsoft intermediate language (MSIL) and inserted into another portion of the file. Every type and member defined and referenced in a module or assembly is described within metadata. When code is executed, the runtime loads metadata into memory and references it to discover information about your code's classes, members, inheritance, and so on.

Metadata describes every type and member defined in your code in a language-neutral manner. Metadata stores the following information:

* Description of the assembly.
  + Identity (name, version, culture, public key).
  + The types that are exported.
  + Other assemblies that this assembly depends on.
  + Security permissions needed to run.
* Description of types.
  + Name, visibility, base class, and interfaces implemented.
  + Members (methods, fields, properties, events, nested types).
* Attributes.
  + Additional descriptive elements that modify types and members.

http://i.msdn.microsoft.com/Global/Images/clear.gif Benefits of Metadata

Metadata is the key to a simpler programming model, eliminating the need for Interface Definition Language (IDL) files, header files, or any external method of component reference. Metadata allows .NET languages to describe themselves automatically in a language-neutral manner, unseen by both the developer and the user. Additionally, metadata is extensible through the use of attributes. Metadata provides the following major benefits:

* Self-describing files.

Common language runtime modules and assemblies are self-describing. A module's metadata contains everything needed to interact with another module. Metadata automatically provides the functionality of IDL in COM, allowing you to use one file for both definition and implementation. Runtime modules and assemblies do not even require registration with the operating system. As a result, the descriptions used by the runtime always reflect the actual code in your compiled file, which increases application reliability.

* Language interoperability and easier component-based design.

Metadata provides all the information required about compiled code for you to inherit a class from a PE file written in a different language. You can create an instance of any class written in any managed language (any language that targets the common language runtime) without worrying about explicit marshaling or using custom interoperability code.

* Attributes.

The .NET Framework allows you to declare specific kinds of metadata, called attributes, in your compiled file. Attributes can be found throughout the .NET Framework and are used to control in more detail how your program behaves at run time. Additionally, you can emit your own custom metadata into .NET Framework files through user-defined custom attributes. For more information, see [Extending Metadata Using Attributes](http://msdn.microsoft.com/en-us/library/5x6cd29c.aspx).

**Structure and Use of Metadata**

Although most developers do not need to know the implementation details of metadata, some might want a deeper understanding. This section provides an overview of how metadata is stored in a .NET Framework portable executable (PE) file and an explanation of metadata's role in managed execution. You do not need to read this section to understand .NET programming or how to use attributes.

http://i.msdn.microsoft.com/Global/Images/clear.gif In This Section

[Metadata and the PE File Structure](http://msdn.microsoft.com/en-us/library/8dkk3ek4.aspx)

Describes how metadata is stored separately from Microsoft intermediate language (MSIL) in a PE file.

[Run-Time Use of Metadata](http://msdn.microsoft.com/en-us/library/65a37919.aspx)

Demonstrates the use of metadata at run time with a code example.

**Metadata and the PE File Structure**

Metadata is stored in one section of a .NET Framework portable executable (PE) file, while Microsoft intermediate language (MSIL) is stored in another section of the PE file. The metadata portion of the file contains a series of table and heap data structures. The MSIL portion contains MSIL and metadata tokens that reference the metadata portion of the PE file. You might encounter metadata tokens when you use tools such as the [MSIL Disassembler (Ildasm.exe)](http://msdn.microsoft.com/en-us/library/f7dy01k1.aspx) to view your code's MSIL or the [Runtime Debugger (Cordbg.exe)](http://msdn.microsoft.com/en-us/library/a6zb7c8d.aspx) to perform a memory dump.

http://i.msdn.microsoft.com/Global/Images/clear.gif Metadata Tables and Heaps

Each metadata table holds information about the elements of your program. For example, one metadata table describes the classes in your code, another table describes the fields, and so on. If you have ten classes in your code, the class table will have tens rows, one for each class. Metadata tables reference other tables and heaps. For example, the metadata table for classes references the table for methods.

Metadata also stores information in four heap structures: string, blob, user string, and GUID. All the strings used to name types and members are stored in the string heap. For example, a method table does not directly store the name of a particular method, but points to the method's name stored in the string heap.

http://i.msdn.microsoft.com/Global/Images/clear.gif Metadata Tokens

Each row of each metadata table is uniquely identified in the MSIL portion of the PE file by a metadata token. Metadata tokens are conceptually similar to pointers, persisted in MSIL, that reference a particular metadata table.

A metadata token is a four-byte number. The top byte denotes the metadata table to which a particular token refers (method, type, and so on). The remaining three bytes specify the row in the metadata table that corresponds to the programming element being described. If you define a method in C# and compile it into a PE file, the following metadata token might exist in the MSIL portion of the PE file:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl12_ctl00_ctl00_code');" \o "Copy Code)

0x06000004

The top byte (0x06) indicates that this is a **MethodDef** token. The lower three bytes (000004) tells the common language runtime to look in the fourth row of the **MethodDef** table for the information that describes this method definition.

http://i.msdn.microsoft.com/Global/Images/clear.gif Metadata within a PE File

When a program is compiled for the common language runtime, it is converted to a PE file that consists of three parts. The following table describes the contents of each part.

|  |  |
| --- | --- |
| **PE section** | **Contents of PE section** |
| PE header | The index of the PE file's main sections and the address of the entry point.  The runtime uses this information to identify the file as a PE file and to determine where execution starts when loading the program into memory. |
| MSIL instructions | The Microsoft intermediate language instructions (MSIL) that make up your code. Many MSIL instructions are accompanied by metadata tokens. |
| Metadata | Metadata tables and heaps. The runtime uses this section to record information about every type and member in your code. This section also includes custom attributes and security information. |

**Run-Time Use of Metadata**

To better understand metadata and its role in the common language runtime, it might be helpful to construct a simple program and illustrate how metadata affects its run-time life. The following code example shows two methods inside a class called MyApp. The Main method is the program entry point, while the Add method simply returns the sum of two integer arguments.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl01_code');" \o "Copy Code)

Public Class MyApp

Public Shared Sub Main()

Dim ValueOne As Integer = 10

Dim ValueTwo As Integer = 20

Console.WriteLine("The Value is: {0}", Add(ValueOne, ValueTwo))

End Sub

Public Shared Function Add(One As Integer, Two As Integer) As Integer

Return (One + Two)

End Function

End Class

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl02_code');" \o "Copy Code)

using System;

public class MyApp

{

public static int Main()

{

int ValueOne = 10;

int ValueTwo = 20;

Console.WriteLine("The Value is: {0}", Add(ValueOne, ValueTwo));

return 0;

}

public static int Add(int One, int Two)

{

return (One + Two);

}

}

When the code runs, the runtime loads the module into memory and consults the metadata for this class. Once loaded, the runtime performs extensive analysis of the method's Microsoft intermediate language (MSIL) stream to convert it to fast native machine instructions. The runtime uses a just-in-time (JIT) compiler to convert the MSIL instructions to native machine code one method at a time as needed.

The following example shows part of the MSIL produced from the previous code's Main function. You can view the MSIL and metadata from any .NET Framework application using the [MSIL Disassembler (Ildasm.exe)](http://msdn.microsoft.com/en-us/library/f7dy01k1.aspx).

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl04_code');" \o "Copy Code)

.entrypoint

.maxstack 3

.locals ([0] int32 ValueOne,

[1] int32 ValueTwo,

[2] int32 V\_2,

[3] int32 V\_3)

IL\_0000: ldc.i4.s 10

IL\_0002: stloc.0

IL\_0003: ldc.i4.s 20

IL\_0005: stloc.1

IL\_0006: ldstr "The Value is: {0}"

IL\_000b: ldloc.0

IL\_000c: ldloc.1

IL\_000d: call int32 ConsoleApplication.MyApp::Add(int32,int32) /\* 06000003 \*/

The JIT compiler reads the MSIL for the whole method, analyzes it thoroughly, and generates efficient native instructions for the method. At IL\_000d, a metadata token for the Add method (/\* 06000003 \*/) is encountered and the runtime uses the token to consult the third row of the **MethodDef** table.

The following table shows part of the **MethodDef** table referenced by the metadata token that describes the Add method. While other metadata tables exist in this assembly and have their own unique values, only this table is discussed.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Row** | **Relative Virtual Address (RVA)** | **ImplFlags** | **Flags** | **Name**  **(Points to string heap.)** | **Signature (Points to blob heap.)** |
| 1 | 0x00002050 | IL  Managed | Public  ReuseSlot  SpecialName  RTSpecialName  .ctor | .ctor (constructor) |  |
| 2 | 0x00002058 | IL  Managed | Public  Static  ReuseSlot | Main | String |
| 3 | 0x0000208c | IL  Managed | Public  Static  ReuseSlot | Add | int, int, int |

Each column of the table contains important information about your code. The **RVA** column allows the runtime to calculate the starting memory address of the MSIL that defines this method. The **ImplFlags** and **Flags** columns contain bitmasks that describe the method (for example, whether the method is public or private). The **Name** column indexes the name of the method from the string heap. The **Signature** column indexes the definition of the method's signature in the blob heap.

The runtime calculates the desired offset address from the **RVA** column in the third row and returns this address to the JIT compiler, which then proceeds to the new address. The JIT compiler continues to process MSIL at the new address until it encounters another metadata token and the process is repeated.

Using metadata, the runtime has access to all the information it needs to load your code and process it into native machine instructions. In this manner, metadata enables self-describing files and, together with the common type system, cross-language inheritance.