TABLE 2-1 Common Definition Metadata Tables

Metadata Definition Table Name	Description
ModuleDef	Always contains one entry that identifies the module. The entry includes the module's file name and extension (without path) and a module version ID (in the form of a GUID created by the compiler). This allows the file to be renamed while keeping a record of its original name. However, renaming a file is strongly discouraged and can prevent the CLR from locating an assembly at run time, so don't do this.
TypeDef	Contains one entry for each type defined in the module. Each entry includes the type's name, base type, and flags (*public*, *private*, etc.) and contains indexes to the methods it owns in the MethodDef table, the fields it owns in the FieldDef table, the properties it owns in the PropertyDef table, and the events it owns in the EventDef table.
MethodDef	Contains one entry for each method defined in the module. Each entry includes the method's name, flags (*private*, *public*, *virtual*, *abstract*, *static*, *final*, etc.), signature, and offset within the module where its IL code can be found. Each entry can also refer to a ParamDef table entry in which more information about the method's parameters can be found.
FieldDef	Contains one entry for every field defined in the module. Each entry includes flags (*private*, *public*, etc.), type, and name.
ParamDef	Contains one entry for each parameter defined in the module. Each entry includes flags (*in*, *out*, *retval*, etc.), type, and name.
PropertyDef	Contains one entry for each property defined in the module. Each entry includes flags, type, and name.
EventDef	Contains one entry for each event defined in the module. Each entry includes flags and name.

As the compiler compiles your source code, everything your code defines causes an entry to be created in one of the tables described in Table 2-1. Metadata table entries are also created as the compiler detects the types, fields, methods, properties, and events that the source code references. The metadata created includes a set of reference tables that keep a record of the referenced items. Table 2-2 shows some of the more common reference metadata tables.

TABLE 2-2 Common Reference Metadata Tables

Metadata Reference Table Name	Description
AssemblyRef	Contains one entry for each assembly referenced by the module. Each entry includes the information necessary to bind to the assembly: the assembly's name (without path and extension), version number, culture, and public key token (normally a small hash value generated from the publisher's public key, identifying the referenced assembly's publisher). Each entry also contains some flags and a hash value. This hash value was intended to be a checksum of the referenced assembly's bits. The CLR completely ignores this hash value and will probably continue to do so in the future.
ModuleRef	Contains one entry for each PE module that implements types referenced by this module. Each entry includes the module's file name and extension (without path). This table is used to bind to types that are implemented in different modules of the calling assembly's module.

Metadata Reference Table Name	Description
TypeRef	Contains one entry for each type referenced by the module. Each entry includes the type's name and a reference to where the type can be found. If the type is implemented within another type, the reference will indicate a TypeRef entry. If the type is implemented in the same module, the reference will indicate a ModuleDef entry. If the type is implemented in another module within the calling assembly, the reference will indicate a ModuleRef entry. If the type is implemented in a different assembly, the reference will indicate an AssemblyRef entry.
MemberRef	Contains one entry for each member (fields and methods, as well as property and event methods) referenced by the module. Each entry includes the member's name and signature and points to the TypeRef entry for the type that defines the member.

There are many more tables than what I listed in Tables 2-1 and 2-2, but I just wanted to give you a sense of the kind of information that the compiler emits to produce the metadata information. Earlier I mentioned that there is also a set of manifest metadata tables; I'll discuss these a little later in the chapter.

Various tools allow you to examine the metadata within a managed PE file. One that I still use frequently is ILDasm.exe, the IL Disassembler. To see the metadata tables, execute the following command line.

```
ILDasm Program.exe
```

This causes ILDasm.exe to run, loading the Program.exe assembly. To see the metadata in a nice, human-readable form, select the View/MetaInfo/Show! menu item (or press Ctrl+M). This causes the following information to appear.

```
============================================================
ScopeName : Program.exe
MVID      : {CA73FFE8-0D42-4610-A8D3-9276195C35AA}
============================================================
Global functions
------------------------------------------------------------

Global fields
------------------------------------------------------------

Global MemberRefs
------------------------------------------------------------

TypeDef #1 (02000002)
------------------------------------------------------------
   TypDefName: Program  (02000002)
   Flags     : [Public] [AutoLayout] [Class] [Sealed] [AnsiClass]
               [BeforeFieldInit]  (00100101)
   Extends   : 01000001 [TypeRef] System.Object
   Method #1 (06000001) [ENTRYPOINT]
```

To build an assembly, you must select one of your PE files to be the keeper of the manifest. Or you can create a separate PE file that contains nothing but the manifest. Table 2-3 shows the manifest metadata tables that turn a managed module into an assembly.

TABLE 2-3 Manifest Metadata Tables

Manifest Metadata Table Name	Description
AssemblyDef	Contains a single entry if this module identifies an assembly. The entry includes the assembly's name (without path and extension), version (major, minor, build, and revision), culture, flags, hash algorithm, and the publisher's public key (which can be `null`).
FileDef	Contains one entry for each PE and resource file that is part of the assembly (except the file containing the manifest because it appears as the single entry in the AssemblyDef table). The entry includes the file's name and extension (without path), hash value, and flags. If this assembly consists only of its own file, the FileDef table has no entries.
ManifestResourceDef	Contains one entry for each resource that is part of the assembly. The entry includes the resource's name, flags (`public` if visible outside the assembly and `private` otherwise), and an index into the FileDef table indicating the file that contains the resource file or stream. If the resource isn't a stand-alone file (such as a .jpg or a .gif), the resource is a stream contained within a PE file. For an embedded resource, the entry also includes an offset indicating the start of the resource stream within the PE file.
ExportedTypesDef	Contains one entry for each public type exported from all of the assembly's PE modules. The entry includes the type's name, an index into the FileDef table (indicating which of this assembly's files implements the type), and an index into the TypeDef table. Note: To save file space, types exported from the file containing the manifest are not repeated in this table because the type information is available using the metadata's TypeDef table.

The existence of a manifest provides a level of indirection between consumers of the assembly and the partitioning details of the assembly and makes assemblies self-describing. Also, note that the file containing the manifest has metadata information that indicates which files are part of the assembly, but the individual files themselves do not have metadata information that specifies that they are part of the assembly.

Note The assembly file that contains the manifest also has an AssemblyRef table in it. This table contains an entry for all of the assemblies referenced by all of the assembly's files. This allows tools to open an assembly's manifest and see its set of referenced assemblies without having to open the assembly's other files. Again, the entries in the AssemblyRef table exist to make an assembly self-describing.

The C# compiler produces an assembly when you specify any of the following command-line switches: `/t[arget]:exe`, `/t[arget]:winexe`, `/t[arget]: appcontainerexe`, `/t[arget]: library`, or `/t[arget]:winmdobj`.[1] All of these switches cause the compiler to generate a single PE file that contains the manifest metadata tables. The resulting file is either a CUI executable, a GUI executable, a Windows Store executable, a class library, or a WINMD library respectively.

[1] When using `/t[arget]:winmdobj`, the resulting .winmdobj file must be passed to the WinMDExp.exe tool, which massages the metadata a bit in order to expose the assembly's public CLR types as Windows Runtime types. The WinMDExp.exe tool does not touch the IL code in any way.

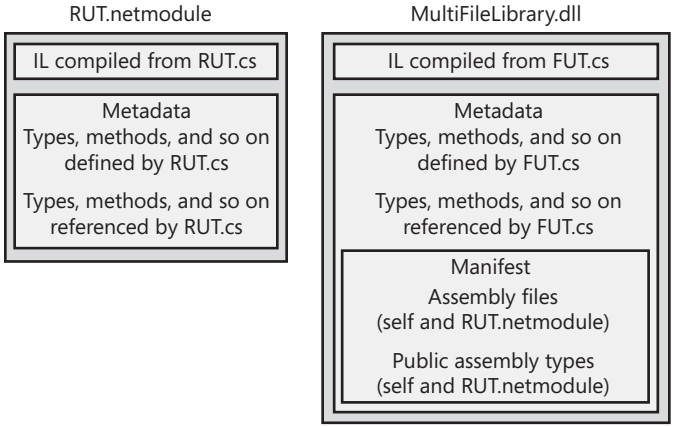

FIGURE 2-1 A multifile assembly consisting of two managed modules, one with a manifest.

The RUT.netmodule file contains the IL code generated by compiling RUT.cs. This file also contains metadata tables that describe the types, methods, fields, properties, events, and so on that are defined by RUT.cs. The metadata tables also describe the types, methods, and so on that are referenced by RUT.cs. The MultiFileLibrary.dll is a separate file. Like RUT.netmodule, this file includes the IL code generated by compiling FUT.cs and also includes similar definition and reference metadata tables. However, MultiFileLibrary.dll contains the additional manifest metadata tables, making MultiFileLibrary.dll an assembly. The additional manifest metadata tables describe all of the files that make up the assembly (the MultiFileLibrary.dll file itself and the RUT.netmodule file). The manifest metadata tables also include all of the public types exported from MultiFileLibrary.dll and RUT.netmodule.

Note In reality, the manifest metadata tables don't actually include the types that are exported from the PE file that contains the manifest. The purpose of this optimization is to reduce the number of bytes required by the manifest information in the PE file. So statements like "The manifest metadata tables also include all the public types exported from MultiFileLibrary.dll and RUT.netmodule" aren't 100 percent accurate. However, this statement does accurately reflect what the manifest is logically exposing.

After the MultiFileLibrary.dll assembly is built, you can use ILDasm.exe to examine the metadata's manifest tables to verify that the assembly file does in fact have references to the RUT.netmodule file's types. Here is what the FileDef and ExportedTypesDef metadata tables look like.

```
File #1 (26000001)
-------------------------------------------------------
   Token: 0x26000001
   Name : RUT.netmodule
   HashValue Blob : e6 e6 df 62 2c a1 2c 59  97 65 0f 21 44 10 15 96  f2 7e db c2
   Flags : [ContainsMetaData]  (00000000)
```

```
ExportedType #1 (27000001)
------------------------------------------------------
   Token: 0x27000001
   Name: ARarelyUsedType
   Implementation token: 0x26000001
   TypeDef token: 0x02000002
   Flags     : [Public] [AutoLayout] [Class] [Sealed] [AnsiClass]
               [BeforeFieldInit](00100101)
```

From this, you can see that RUT.netmodule is a file considered to be part of the assembly with the token 0x26000001. From the ExportedTypesDef table, you can see that there is a publicly exported type, `ARarelyUsedType`. The implementation token for this type is 0x26000001, which indicates that the type's IL code is contained in the RUT.netmodule file.

Note For the curious, metadata tokens are 4-byte values. The high byte indicates the type of token (0x01=TypeRef, 0x02=TypeDef, 0x23=AssemblyRef, 0x26=File (file definition), 0x27=ExportedType). For the complete list, see the `CorTokenType` enumerated type in the CorHdr.h file included with the .NET Framework SDK. The three lower bytes of the token simply identify the row in the corresponding metadata table. For example, the implementation token 0x26000001 refers to the first row of the File table. For most tables, rows are numbered starting with 1, not 0. For the TypeDef table, rows actually start with 2.

Any client code that consumes the MultiFileLibrary.dll assembly's types must be built using the `/r[eference]:` `MultiFileLibrary.dll` compiler switch. This switch tells the compiler to load the MultiFileLibrary.dll assembly and all of the files listed in its FileDef table when searching for an external type. The compiler requires all of the assembly's files to be installed and accessible. If you were to delete the RUT.netmodule file, the C# compiler would produce the following error: `fatal error CS0009: Metadata file 'C:\MultiFileLibrary.dll' could not be opened—'Error importing module 'RUT.netmodule' of assembly 'C:\MultiFileLibrary.dll'—The system cannot find the file specified'`. This means that to build a new assembly, all of the files from a referenced assembly *must* be present.

As the client code executes, it calls methods. When a method is called for the first time, the CLR detects the types that the method references as a parameter, a return type, or as a local variable. The CLR then attempts to load the referenced assembly's file that contains the manifest. If the type being accessed is in this file, the CLR performs its internal bookkeeping, allowing the type to be used. If the manifest indicates that the referenced type is in a different file, the CLR attempts to load the necessary file, performs its internal bookkeeping, and allows the type to be accessed. The CLR loads assembly files only when a method referencing a type in an unloaded assembly is called. This means that to run an application, all of the files from a referenced assembly do not need to be present.