File Format Spec

This document defines the extensions, for NGWS, to the Microsoft PE (Portable Executable) file format that development tools and compilers will generate, and that the NGWS runtime will load and execute. This spec encompasses a description of the following: header and tags; metadata; IL and native code structures; issues of reordering and fix-ups. Generally, all format issues are covered except the actual code sections. The reader is referred to the following additional related runtime specifications: ILInstrSet and Metadata Interfaces for emitting the metadata portion of the runtime file format.

This is preliminary documentation and subject to change

Last updated: 8 June 2000

Table Of Contents

1 Overview

This document specifies a file format for NGWS components that is based on, and is a strict extension of, the current Microsoft Windows Portable Executable (PE) and Common Object File Format (COFF). This extended PE/COFF format enables the Windows operating system to recognize runtime images, accommodates code emitted as runtime Intermediate Language (IL) or native code, and accommodates runtime metadata as an integral part of the emitted code.

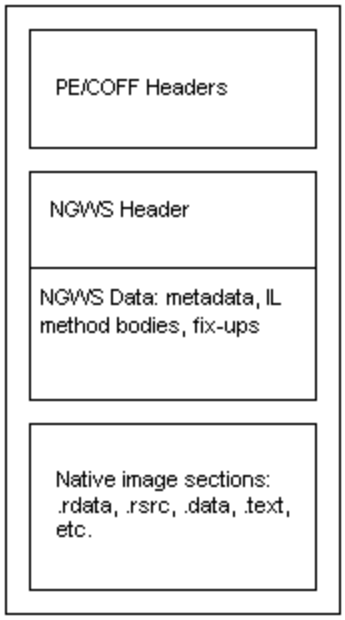
This section provides a brief overview of and motivation for the design approach, including a summary of requirements and constraints. Subsequent sections present the technical specifications as a delta from the current Windows PE/COFF file format, in sufficient detail that a tool or compiler can use the specifications to emit valid runtime images.

The entire document assumes familiarity with the current PE/COFF structure and terminology. For more information, refer to the "Microsoft Portable Executable and Common Object File Format Specification"

1.1 Structure of the Runtime File Format

The figure below provides a high-level view of the NGWS file format. All runtime images contain the following:

- Standard PE/COFF headers, with specific guidelines on how field values should be set in a runtime file
- A runtime header that contains all of the runtime specific data entries. Currently, the runtime header is read-only and may be placed in any read-only section
- Any of the data one currently finds in a valid PE/COFF image, including imports/exports, data, and code. This spec calls out specific areas where we use this data in the runtime



The image is a full PE/COFF file image. The normal PE/COFF headers apply. The runtime header is found using directory entry IMAGE_DIRECTORY_COR_DESCRIPTOR in the PE header. The runtime header in turn contains the address and locations of the runtime data in the rest of the image. Note that the runtime data can be merged into other areas of the PE format with the other data based on the attributes of the sections (such as read only versus execute, etc.).

While the bulk of the file format is generated by tools directly, the metadata portion is emitted through an API that abstracts the tools from the underlying data structures. This is in part because the data structures are many and complex, having been tuned for performance and size, and because we want to be able to do additional tuning of the structures without impacting the tools that are emitting them. And, it is in part because the runtime metadata engine even today supports a number of different formats exposed in a uniform way through the same API. For example, for COM interop, a consumer of runtime metadata can import a typelib as though it were a perfectly valid runtime metadata file. Refer to the Metadata Interfaces spec for details on emitting and consuming the metadata portion of a runtime image.

1.2 Producers and Consumers of the Runtime File Format

Development tools and compilers will emit runtime images that can be packaged and deployed across a range of runtime-enabled platforms. Development tools will range from RAD tools (including scripting languages) to high-level language compilers. The first category of tools will compile and emit files in a single pass from the development environment. Scripting tools may not even have a need to persist the resulting file, but simply regenerate the code every time it's executed. The second category of tools has an incremental approach, first emitting intermediate compilation units and then linking them together with resources into a loadable runtime image.

The file format needs to accommodate not only what the runtime will require in order to load and execute these files, but it needs to make it reasonably straightforward for this range of different tools with different internal data structures and compilation models to emit metadata and code efficiently (along with imports/exports, fix-ups, debugging information, etc.).

Consumers of the runtime file format include the runtime itself as well as development tools and administrative tools. The runtime consumes metadata and IL in order to JIT-compile IL to native code. The loader consumes metadata to load classes and track managed data structures. Development tools will import metadata to enable references to runtime types and members. Administrative tools will consume metadata to browse classes and configure services.

1.3 Requirements Addressed by the Runtime File Format Design

Initial exploration of alternative design approaches ranged from introducing an entirely new file format for the runtime that would co-exist side-by-side with today's PE file format, to ensuring that the runtime format was a natural extension of today's

Windows PE file. In having chosen the latter approach, it may be instructive to review the requirements that drove the design and spec work.

An Option of IL or Native Code

A developer who wants to target a range of runtime platforms may want to build a component or assembly of components once and compile to native when needed for a particular platform. Options for "when needed" range from deployment time to install time to execution time. In this scenario, the code is emitted as IL, plus the metadata that the runtime JIT compiler(s) use to compile the IL to native.

A developer building a runtime component or application in his or her favorite language may have reason to compile code directly to native. For example, if the code is known to target only a specific platform, there may be no perceived benefit from going through an intermediate language. This does not mean that the developer need forego the benefits of the runtime managed services. In the design presented in this document, the target file format is today's PE file, either .exe or .dll.

To be more specific, the runtime recognizes managed native code and unmanaged native code. Both are compiled in any language to the native instruction set of a CPU. Unlike unmanaged native code, managed native code has additional metadata and coding conventions used by the runtime to enable garbage collection, exceptions and other runtime features. The current file format specification does not describe these metadata and file format extensions. Unmanaged native code is fully supported, emitted using all of the structures of today's PE/COFF.

A Combination of IL and Native Code

The runtime will accept a file containing a mixture of IL and native code. The runtime file format accommodates either one or both naturally in a single format, without requiring compilers to emit, and OS loaders to recognize, a range of different formats for specialized purposes.

Self-Contained Environments

Although based on today's Windows PE/COFF, the structure of the sections is intended to be subset-able for self-contained environments that are directly integrated with the runtime. In particular, these environments may be willing to trade off full OS services, like page sharing between processes, for image size. Observe that the structure of the format headers and sections pictured earlier lends itself to a structure that consists solely of the runtime header and the data sections that make up the IL portion of the image.

32- and 64-Bit Support

Support for both 32-bit and 64-bit requires a number of accommodations in the file format design, including:

- Support for agnostic-sized integers
- Data fix-ups

Although 64-bit is not fully supported in this version of the NGWS runtime, the underpinnings are reflected in this specification since moving toward 64-bit is integral to the design of the file format and the runtime.

Debugging

It should be possible to emit runtime images that carry debugging information. The Debugging Architecture Specification describes the design goals in detail. The File

Format specification identifies the header tags that are used to indicate when debug information is persisted.

Optimized Code Generation

Optimized native code (and IL for that matter) is important to the quality and speed of the generated code. The file format must not prohibit the use of a code optimizer, or tools that offer post-link optimization of code.

Existing Code Base

There is a broad existing code base that is supported as an integral part of the runtime.

For example, native code that exists without metadata in today's PE files will continue to be loaded and executed by the OS. Runtime IL and runtime native code can import from and forward exports to these types of files, and later sections of this specification describe how such imports and exports are accommodated in the runtime file format.

Native APIs can be expressed as runtime methods, in metadata, allowing runtime IL to make calls to these function exports, with the runtime providing the native mapping services. And, of course, unmanaged COM components are perfectly viable runtime components. COM Type Libraries can be converted into runtime metadata. Details on the metadata portion of the runtime file format are provided in the Metadata Interfaces spec.

Existing Windows Infrastructure

Because OS loaders are tuned for the PE/COFF format, with built-in support for many default features like fix-ups and section mapping, runtime images take advantage of this infrastructure. Pages can be shared between processes, making the overall working set size on the machine smaller and more efficient.

In addition, there are many different tools already designed and shipping which work on the PE format, such as DUMPBIN, and IMAGEHLP.

The NGWS SDK includes additional tools:

- MetaInfo, providing a detailed dump of runtime metadata. This tool works against any file format that can be imported by the runtime metadata engine, including typelib.
- ILDASM, providing a dump of runtime managed code. This tool is useful once the development tool or compiler is known to be emitting valid metadata.

1.4 OS Interactions

Future versions of Windows will be runtime-aware. For backward compatibility with Win9x platforms, runtime images should be marked as x86 images. They should contain an x86 native code stub that will be called by the OS to bootstrap the runtime. See the discussion on Entry Points at the end of this document. One of the benefits of extending the existing PE/COFF for runtime images is that one can create a process with a runtime executable like any other PE/COFF image.

2 Emitting A Valid NGWS Image

This section covers the structure of the file headers, section headers, and extensions to the native PE data that may be used by the runtime.

2.1 File Headers

The image starts with an MS-DOS header, followed by the COFF header, and PE header.

2.1.1 Signature

The PE format calls for an MS-DOS stub to be placed at the front of the module. This stub is then used to tell DOS users that the module cannot be run in DOS mode.

At offset 0x3c is the offset to the PE signature. The signature will remain "PE\0\0" as it is today.

2.1.2 COFF Header

Immediately after the signature is the COFF header consisting of the following:

Offset	Size	Field	Description
0	2	**Machine**	Number identifying type of target machine. See below
2	2	**Number of Sections**	Number of sections; indicates size of the Section Table, which immediately follows the headers
4	4	**Time/Date Stamp**	Time and date the file was created
8	4	**Pointer to Symbol Table**	The COFF symbol table is not used. Set this value to 0
12	4	**Number of Symbols**	Always 0
16	2	**Optional Header Size**	Size of the optional header, the format is described below
18	2	**Characteristics**	Flags indicating attributes of the file

2.1.2.1 Machine Type

If an image is intended to run on a single processor type, then the machine type should be set accordingly. If the image is intended to run on more than one processor type, runtime images will use a machine type of IMAGE_FILE_MACHINE_I386

2.1.2.2 Characteristics

An image that contains native code may have any of the standard flags from the PE file format specification as appropriate.

An IL-only dll has the following characteristics:

Flag	Value	Description
IMAGE_FILE_EXECUTABLE_IMAGE	0x0002	Image only. Indicates that the image file is valid and can be run. If this flag is not set, it generally indicates a linker error
IMAGE_FILE_LINE_NUMS_STRIPPED	0x0004	COFF line numbers have been removed
IMAGE_FILE_LOCAL_SYMS_STRIPPED	0x0008	COFF symbol table entries for local symbols have been removed
IMAGE_FILE_DEBUG_STRIPPED	0x0200	Debugging information removed from image file
IMAGE_FILE_DLL	0x2000	The image file is a dynamic-link library (DLL). Such files are considered executable files for almost all purposes, although they cannot be directly run

Currently we do not anticipate support for IMAGE_FILE_SYSTEM (to produce device drivers and systems level code written in IL)

2.1.3 Optional Header

The PE/COFF Optional Header is required for a runtime image[1]. It is located immediately after the COFF Header and is sometimes referred to as the PE Header. This header contains the following information:

Offset	Size	Header part	Description
0	28	Standard fields	These are defined for all implementations of COFF, including UNIX®.
28	68	NT-specific fields	These include additional fields to support specific features of Windows NT (for example, subsystem)
96	128	Data directories	These fields are address/size pairs for special tables, found in the image file and used by the operating system (for example, Import Table and Export Table)

2.1.3.1 Optional Header Standard Fields

These fields are required for all COFF files. They contain loader information as follows:

Offset	Size	Field	Description
0	2	**Magic**	Unsigned integer identifying the state of the image file. Set this value to 0x10B, meaning an executable file
2	1	**LMajor**	Linker major version number, tool specific
3	1	**LMinor**	Linker minor version number, tool specific

[1] It is called optional because when the COFF format is used for an object file it is not required

4	4	**Code Size**	Size of the code (text) section, or the sum of all code sections if there are multiple sections
8	4	**Initialized Data Size**	Size of the initialized data section, or the sum of all such sections if there are multiple data sections
12	4	**Uninitialized Data Size**	Size of the uninitialized data section (BSS), or the sum of all such sections if there are multiple BSS sections
16	4	**Entry Point RVA**	Address of entry point, relative to image base, when executable file is loaded into memory. See the section below on entry points
20	4	**Base Of Code**	Address, relative to image base, of beginning of code section, when loaded into memory
24	4	**Base Of Data**	Address, relative to image base, of beginning of data section, when loaded into memory

2.1.3.2 Optional Header Windows NT-Specific Fields

These fields are Windows NT specific:

Offset	Size	Field	Description
28	4	**Image Base**	Preferred address of first byte of image when loaded into memory; must be a multiple of 64K
32	4	**Section Alignment**	Alignment (in bytes) of sections when loaded into memory. Must be greater or equal to File Alignment. Default is the page size for the architecture
36	4	**File Alignment**	Alignment factor (in bytes) used to align pages in image file. Valid values are a power of 2 between 512 and 64K. Unless otherwise necessary, use 512
40	2	**OS Major**	Major version number of required OS
42	2	**OS Minor**	Minor version number of required OS
44	2	**User Major**	Major version number of image
46	2	**User Minor**	Minor version number of image
48	2	**SubSys Major**	Major version number of subsystem
50	2	**SubSys Minor**	Minor version number of subsystem
52	4	Reserved	
56	4	**Image Size**	Size, in bytes, of image, including all headers; must be a multiple of Section

			Alignment
60	4	**Header Size**	Combined size of MS-DOS Header, PE Header, and Object Table
64	4	**File Checksum**	Image file checksum. The algorithm for computing is incorporated into IMAGHELP.DLL. The following are checked for validation at load time: all drivers, any DLL loaded at boot time, and any DLL that ends up in the server
68	2	**SubSystem**	Subsystem required to run this image. See note below
70	2	**DLL Flags**	Obsolete
72	4	**Stack Reserve Size**	Size of stack to reserve. Only the Stack Commit Size is committed; the rest is made available one page at a time, until reserve size is reached. Stacks for IL will be handled by the runtime. This value should be set using the same switches as used today
76	4	**Stack Commit Size**	Size of stack to commit
80	4	**Heap Reserve Size**	Size of local heap space to reserve. Only the Heap Commit Size is committed; the rest is made available one page at a time, until reserve size is reached
84	4	**Heap Commit Size**	Size of local heap space to commit
88	4	**Loader Flags**	Obsolete
92	4	**Number of Data Directories**	Number of data-dictionary entries in the remainder of the Optional Header. Each describes a location and size

2.1.3.2.1 SubSystem Settings

The runtime Loader itself does not do anything with the subsystem setting of the PE. The value chosen, however, can impact on what Windows platforms the image may be run. For example, setting this value to IMAGE_SUBSYSTEM_WINDOWS_CE_GUI means the image can't be run on any non-CE device. In addition, IMAGE_SUBSYSTEM_NATIVE is not supported because the runtime cannot run in kernel mode for this release. It is recommend that either IMAGE_SUBSYSTEM_WINDOWS_GUI or IMAGE_SUBSYSTEM_WINDOWS_CUI be used for this setting.

2.1.3.2.2 Stack Reserve Size

For now, the default stack size should be used

Recommended size information will be supplied in a later release.

2.1.3.3 Optional Header Data Directories

Data directories give the address and size of tables used by Windows. Each data directory entry is as follows:

```
typedef struct _IMAGE_DATA_DIRECTORY {

    DWORD   RVA;

    DWORD   Size;

} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

The first field, RVA, is the relative virtual address of the table. The RVA is the address of the table, when loaded, relative to the base address of the image. The second field gives the size in bytes. The data directories, which form the last part of the Optional Header, are listed below.

Offset	Size	Field	Description
96	8	**Export Table**	Export Table address and size
104	8	**Import Table**	Import Table address and size
112	8	**Resource Table**	Resource Table address and size
120	8	**Exception Table**	Exception Table address and size. For Managed Native Code, this table will contain the MIH mapping information for the Code Manager
128	8	**Certificate Table**	Attribute Certificate Table address and size
136	8	**Base Relocation Table**	Base Relocation Table address and size
144	8	**Debug**	Debug data starting address and size
152	8	**Copyright**	Copyright string address and length
160	8	**Global Ptr**	Relative virtual address of the global pointer register. Size member of this structure is set to 0
168	8	**TLS Table**	Thread Local Storage (TLS) Table address and size
176	8	**Load Config Table**	Load Configuration Table address and size
184	8	**Bound Import**	Bound Import Table address and size
192	8	**IAT**	Import Address Table address and size
200	8	**Delay Import Descriptor**	Address and size of the Delay Import Descriptor

208	8	**Runtime Header**	Runtime Header with directories for runtime data
216	8	Reserved	

2.1.4 Storing Runtime Data in Sections

The runtime defines several types of data formats that are used by the engine to execute code. This includes things like the metadata, IL method bodies, garbage-collection encodings, and others. Each type of data may be placed in any part of the PE image so long as the section the data is placed in has the same attributes as required. Each data format described in the following section identifies what type of page attributes the data must have in the final image.

2.1.5 Runtime Header

Directory entry IMAGE_DIRECTORY_COR_DESCRIPTOR contains the location of the runtime Header in the image. This header contains all of the runtime-specific data entries and other information. The header should be placed in a read only, sharable section of the image. This header is defined as follows:

Offset	Size	Field	Description
0	4	**Cb**	Size of the header in bytes
4	2	**MajorRuntimeVersion**	The minimum version of the runtime required to run this program. This value matches the COR_VERSION_MAJOR macro the runtime is compiled with
6	2	**MinorRuntimeVersion**	The minor portion of the version. Use the COR_VERSION_MINOR macro
8	8	**MetaData**	Location and size of the meta data in this image
16	4	**Flags**	Flags describing this runtime image. See "Runtime Flags" below
20	4	**EntryPointToken**	Token for the MethodDef of the entry point for the image
24	8	**Resources**	Location of NGWS resources. These resources are not the same as the WIN32 resource section of the PE file
32	8	**StrongNameSignature**	Location of the hash data for this PE file used by the NGWS loader for binding and versioning
40	8	**CodeManagerTable**	Location and size of the Code Manager Table. See below
48	8	**VTableFixups**	Location and size of an array of locations in the file that contain an

			array of function pointers (e.g., vtable slots). See discussion below
56	8	**ExportAddressTableJumps**	Location and size of an array of RVA's to where jump thunks are written
64	8	**EEInfoTable**	Reserved for IOJ[2]
72	8	**HelperTable**	Reserved for IOJ
80	8	**DynamicInfo**	Reserved for IOJ
88	8	**DelayLoadInfo**	Reserved for IOJ
96	8	**ModuleImage**	Reserved for IOJ
104	8	**ExternalFixups**	Reserved for IOJ
112	8	**RidMap**	Reserved for IOJ
120	8	**DebugMap**	Reserved for IOJ
128	8	**IPMap**	OBSOLETE: this directory is being replaced by the P-Data map already defined in the PE specification

More details on parts of this header are included below. Those directories that are specific to Managed Native Code can be found in the chapter that covers MNC.

2.1.5.1 Runtime Header Definition

Following is the structure definition for the header:

```
// NGWS header structure

typedef struct IMAGE_COR20_HEADER
{
    // Header versioning
    DWORD                   cb;
    WORD                    MajorRuntimeVersion;
    WORD                    MinorRuntimeVersion;

    // Symbol table and startup information
    IMAGE_DATA_DIRECTORY    MetaData;
    DWORD                   Flags;
    DWORD                   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;
```

[2] "Install-O-Jit" – an informal term that describes how you can have the runtime engine compile the IL in PE modules and save the compiled, native code. This would typically be done when you install an application on a computer – thus the name.

```
    // Managed Native Code
    IMAGE_DATA_DIRECTORY    EEInfoTable;
    IMAGE_DATA_DIRECTORY    HelperTable;
    IMAGE_DATA_DIRECTORY    DynamicInfo;
    IMAGE_DATA_DIRECTORY    DelayLoadInfo;
    IMAGE_DATA_DIRECTORY    ModuleImage;
    IMAGE_DATA_DIRECTORY    ExternalFixups;
    IMAGE_DATA_DIRECTORY    RidMap;
    IMAGE_DATA_DIRECTORY    DebugMap;
    IMAGE_DATA_DIRECTORY    IPMap;
} IMAGE_COR20_HEADER;
```

2.1.5.2 Runtime Flags

The following flags describe this runtime image and are used by the loader.

Flag	Value	Description
COMIMAGE_FLAGS_ILONLY	0x00000001	Image contains only IL code so that it is not required to run on a specific CPU. Any native code in the module is the x86 startup stub which may safely be ignored if the OS is runtime aware
COMIMAGE_FLAGS_32BITREQUIRED	0x00000002	Image may only be loaded into a 32-bit process
COMIMAGE_FLAGS_TRACKDEBUGDATA	0x00010000	When set, the runtime and JIT are required to track debugging information about methods. See the Debugging Architecture Specification for more details.

2.1.5.3 Entry Point Meta Data Token

The entry point token is always a MethodDef token (refer to the Metadata Interfaces spec for details on metadata tokens). The signature and implementation flags in metadata for the method indicate how the entry is run. The entry point, if given, is always a managed method using the DEFAULT calling convention.

For a DLL image, the entry point is one of:

- int DllMain(HINSTANCE, DWORD, void *)

For an EXE image, the entry point is one of:

- unsigned main(void)
- void DEFAULT main(Microsoft.Runtime.String[])

Although this approach means that the runtime has parsing smarts built into it, it offers the ability for any tool (like the assembler) to have a default implementation. A tool is always free to insert their own entry point wrapper using "unsigned main(void)" which in turn parses the command line and delegates to the user's entry point code of the same signature.

The entry point RVA in the PE header must always be either the x86 entry point stub (which loads and calls the Runtime), or be 0 (for a runtime aware version of

Windows only). If Managed and Unmanaged code are mixed in the same image, this is still the case. To make an unmanaged method the entry point for an image, define a PInvoke MethodDef for the method and use that token. For example, here is a dump of the metadata for a C++ console application with an unmanaged entry point supplied by a static copy of the CRT:

```
Global functions
       Method #1
       -------------------------------------------------------
              MethodName: _mainCRTStartup (06000008)
              Flags     : [Public] [Static] [ReuseSlot] [PinvokeImpl]  (00002011)
              RVA       : 0x00001116
              ImplFlags : [Native] [Unmanaged] [Implemented]  (00000014)
              CallCnvntn: [DEFAULT]
              ReturnType: UI4
              No arguments.
              Pinvoke Map Data:
              Entry point:
              Module ref:       1a000001
              Mapping flags:    [NoMangle] [CharSetNotSpec] [CallConvStdcall]
(00000301)
              Ordinal:          00000000

ModuleRef #1
-----------------------------------------------------
              ModuleRef: (1a000001) :
              GUID : {00000000-0000-0000-0000-000000000000}
              MVID : {00000000-0000-0000-0000-000000000000}
```

2.1.5.4 VTable Fixup

Certain unmanaged C++ classes, which choose not to follow the VOS runtime model, may have virtual functions which need to be represented in a v-table. These v-tables are laid out by the C++ compiler, not by the runtime. Finding the correct v-table slot and calling indirectly through the value held in that slot is also done by the compiler. V-tables are emitted into a *read-write* section of the PE file. (This is different from unmanaged native code, where a v-table can be placed in a read-only section and shared between processes.) It is recommended that a tool try to emit v-tables adjacent to each other in the image in order to minimize the number of entries in this table (ie: you can do all v-tables with 1 entry if they are all adjacent). The runtime header contains the location and size of an array that looks like:

```
#define COR_VTABLE_32BIT    0x01        // V-table slots are 32-bits in size.
#define COR_VTABLE_64BIT    0x02        // V-table slots are 64-bits in size.
#define COR_VTABLE_FROM_UNMANAGED 0x04    // If set, transition from unmanaged.
#define COR_VTABLE_CALL_MOST_DERIVED 0x10 // Call most derived method described by
                                        //  token, only valid for virtuals.

typedef struct _IMAGE_COR_VTABLEFIXUP
{
   ULONG       RVA;            // Offset of v-table array in image.
   USHORT      Count;          // How many entries at location.
   USHORT      Type;           // COR_VTABLE_xxx type of entries.
```

```
} IMAGE_COR_VTABLEFIXUP;
```

Each entry in this array describes a contiguous array of v-table slots of the specified size. Each slot starts out initialized to the metadata token value for the method they need to call. At image load time, the runtime Loader will turn each entry into a pointer to machine code for the CPU and can be called directly.

This structure is also used for managed native code transitions. Please see the chapter on Managed Native Code for more details.

2.1.5.5 Resources

This header entry points to the NGWS managed resources for an image, not to be confused with the traditional Win32 resource format. Please see the resource specification for more details on the format of this data.

2.1.5.6 Strong Name Signature

This header entry points to the strong name hash for an image that can be used to deterministically identify a module from a referencing point. The routines in StrongName.h in the public SDK can be used work with this data. Please see the Design Specification for Assembly Metadata for more details.

2.2 Section Headers

The section headers come immediately after the COFF optional header (PE Header). Refer to the "Microsoft® Portable Executable and Common Object File Format Specification" for settings on standard sections.

The runtime data for a module is always located through the runtime Header. The only requirement on a tool that emits this data is to place the data in a section which has the required characteristics. Currently all of the runtime data for a module (meta data, method bodies, etc.) are read only and may be placed in any read only section. The data may be shared between processes safely.

2.3 Modifications to Existing PE Data

Runtime modules may contain any data one currently finds in a valid PE image, including resources, imports/exports, data, and code. See the PE specification for more details. This section calls out specific areas where we use existing data in the runtime.

2.3.1 Import Address Table (IAT)

The IAT is used for one of the following:

To import mscoree.dll (ie: the runtime engine) for the x86 loader thunk

In a mixed managed/unmanaged image, for legacy imports

The IAT burns into the image slots for methods of a fixed size and is therefore not portable between 32 and 64-bit systems. For this reason, it is ideal to use the metadata to import legacy methods and data through the PInvoke feature. For an

image using the IAT only to import the runtime, the IAT is completely ignored and therefore not a problem (presumes an OS loader which is NGWS-aware).

2.3.2 Export Section (.edata)

Exporting as unmanaged native is interesting when one is implementing an existing interface (such as Win32, print drivers, or ODBC), but you want to write your code using the runtime. The EAT will always have pointers to unmanaged code as down-level clients expect this. There are no plans to use the EAT/IAT to directly export/import managed methods. Rather the metadata should be used to perform this duty.

Please see the Export Address Table Jumps directory entry for more details on how to export methods as unmanaged from an image.

2.3.3 Thread Local Storage Table

The programmer may mark an instance of a static field as TLS. This means that the memory for the instance is associated with the logical thread being executed. Each logical thread has its own copy of the instance of the variable. For an IL image, a compiler generates instructions to access the field as they would any other piece of static data, by specifying the RVA of the data. The runtime does a range check of the RVA on the instruction to see if it falls in the TLS table, and if so, it abstracts the location of the memory relative to the thread automatically.

Per the current Windows standard, only PEs in the originating EXE or explicitly imported DLL's (through the IAT) are included in the calculation for TLS data at process startup. DLLs that are dynamically loaded later may not successfully use the TLS attribute. Runtime does not change this restriction.

TLS storage for native code is unchanged from the current PE specification. The same restrictions apply.

2.3.4 Relocations

Relocations in a pure IL image are required only for the x86 startup stub which access the IAT to load the runtime engine on down level loaders. On an OS which is runtime-aware and marked for IL-only execution, the relocation section, IAT, and any native code is ignored.

When building a mixed IL/native image, relocations may be needed as they are in a traditional image.

Finally, if the image contains embedded RVAs in user data, then relocations must be emitted for these references. Relocations are not required nor recommended for arguments to IL instructions (such as ldptr). These extra relocations are used by tools that need to rewrite or modify the image in some manner, such as a linker which combines images together.

3 Intermediate Language

This section describes how to format and emit IL methods in the image. A method consists of a COR_ILMETHOD method structure and the instructions for the method. A call descriptor contains an RVA to this method structure. There is no required ordering for methods in this section, which allows a tuning tool to reorder the methods if better locality of reference and working set size can be obtained by so doing

The COR_ILMETHOD describes all information about a method that is not needed by callers of the method. This includes:

1. Amount of resources needed for the Operand stack
2. Amount of resources for local variables as well as which local variables contain pointers into the garbage collected heap
3. Amount of resources for the locspace instruction
4. Exception handling information

Unfortunately, the description of the operand stack and the local variable frame is complicated by the fact that stack slots can hold items of unknown size. The size of I and REF types is unknown by the code generator, as well as by-value objects, don't even have a upper bound on their size.

3.1 Local Variable Layout

One function of the COR_ILMETHOD structure is to indicate the layout of the local variables. This is needed for two reasons.

1. To be able to find all pointers into the garbage-collected heap at GC time
2. To indicate the sizes of the local variables so the local frame can be calculated

When local variables are present, use the COR_ILMETHOD_FAT structure definition and provide a value for the LocalVarSigTok field. This is a metadata token for a signature that describes the precise layout of each local on the stack. Please see the MetadataStructures spec for more details on signature tokens.

3.2 File Format Structure Definitions

This section contains the layout of the data structures used to describe an IL method and its exceptions. C language definitions can be found in the CORHDR.H file in the SDK. In addition, some unsupported C++ style declarations can be found in CORHLPR.*, also in the SDK. These helpers greatly simplify emitting and decoding the file format structures and are used in the runtime engine itself.

3.2.1 Method Body

The body of a method is defined in the general form.

Exception handling data is emitted after the method body. If exception-handling data is present, then CorILMethod_MoreSects must be specified in the method header and for each chained item after that.

3.2.1.1 Method Header Type Values

The three least significant bits of the first byte of the method header indicate what type of header is present. These 3 bits will be one and only one of the following:

Value	Value	Description
CorILMethod_TinyFormat	2	The method header is defined by the IMAGE_COR_ILMETHOD_TINY structure and the size of the code is even
CorILMethod_TinyFormat1	6	The method header is defined by the IMAGE_COR_ILMETHOD_TINY structure and the size of the code is odd. Using two values for tiny allows for an extra bit of size data
CorILMethod_FatFormat	3	The method header is defined by the IMAGE_COR_ILMETHOD_FAT structure

3.2.1.2 Tiny Format

The IMAGE_COR_ILMETHOD_TINY structure comes in two flavors with a 5 or 6 bit length encoding. The following is true for all tiny headers:

- No local variables are allowed
- No exceptions
- No extra data sections
- The operand stack need be no bigger than 8 bytes

The first encoding has the following format:

Start Bit	Count of Bits	Description
0	2	CorILMethod_TinyFormat = 0x2
2	6	Size of the method body immediately following this header. Used only when the size of the method is even and under 2^6

The second encoding has the following format:

Start Bit	Count of Bits	Description
0	3	CorILMethod_TinyFormat1 = 0x6
3	5	Size of the method body immediately following this header. Used only when the size of the method is odd and under 2^5

3.2.1.3 Fat Format

The fat format is used whenever the Tiny format won't work. This may be true for one or more of the following reasons:

- The method is too large to encode the size
- There are exceptions
- There are extra data sections
- There are local variables
- The operand stack needs more than 8 bytes

The encoding of the first byte of the header in this case is as follows:

Start Bit	Count of Bits	Description
0	2	CorILMethod_Fat = 0x3
2	1	Reserved
3	1	CorILMethod_MoreSects = 0x8 or 0 to indicate no more sections
4	1	CorILMethod_InitLocals = 0x10 or 0 to indicate no local variable init

The rest of the Fat format is described below

3.2.1.4 IMAGE_COR_ILMETHOD

The metadata points to an instance of this structure. Method headers must be DWORD aligned. There are two possible formats to emit, Tiny or Fat. This structure is a union of these two types. The first byte indicates which type is present (see the "Method Header Type Values" for more details)

```
typedef union
{
    COR_ILMETHOD_TINY       Tiny;
    COR_ILMETHOD_FAT        Fat;
} COR_ILMETHOD;
```

3.2.1.5 IMAGE_COR_ILMETHOD_TINY

This structure must always be DWORD aligned. The layout of this structure is as follows:

```
typedef struct IMAGE_COR_ILMETHOD_TINY
{
    BYTE Flags_CodeSize;
} IMAGE_COR_ILMETHOD_TINY;
```

See the description of the header above.

3.2.1.6 IMAGE_COR_ILMETHOD_FAT

This structure must always be DWORD aligned. The layout of this structure is as follows:

```
typedef struct IMAGE_COR_ILMETHOD_FAT
{
    unsigned Flags    : 12;     // Flags
    unsigned Size     :  4;     // size in DWords of this structure (currently 3)
```

```
    unsigned MaxStack : 16;      // maximum number of items (I4, I, I8, obj ...),
                                 //   on the operand stack
    DWORD    CodeSize;           // size of the code
    DWORD    LocalVarSigTok;     // token that indicates the signature of the local
                                 //   vars (0 means none)
} IMAGE_COR_ILMETHOD_FAT;
```

Offset	Size	Field	Description
0	12 bits	**Flags**	Flags (described below)
12 bits	4 bits	**Size**	Size of this header expressed as the count of DWORDs occupied
16 bits	16 bits	**MaxStack**	Maximum number of items on the operand stack
4	4	**CodeSize**	Size in bytes of the actual method body
8	4	**LocalVarSigTok**	Meta Data token for a signature describing the layout of the local variables for the method. 0 means there are no local variables present

3.2.1.6.1 Flags for Method Headers

The first byte of a method header may also contain the following flags, valid only for the Fat format, that indicate how the method is to be executed:

Flag	Value	Description
CorILMethod_InitLocals	0x0010	Call default constructor on all local variables
CorILMethod_MoreSects	0x0008	Set to indicate a section attribute follows the current attribute. Used to chain in exception information among other things

3.2.2 Section Data

Certain types of data may be found after a method header. Currently only exceptions are described in such a way, but other usages are envisioned. These sections are only valid when the method is encoded with an IMAGE_COR_ILMETHOD_FAT header with the CorILMethod_MoreSects bit set. Section data must be DWORD aligned after the end of the method body as described by the method header.

Every section is at least two bytes. The first byte of the section data contains the flags describing the section as follows:

Flag	Value	Description
CorILMethod_Sect_Reserved	0	Reserved for future use
CorILMethod_Sect_EHTable	1	Exception handling data. See the IMAGE_COR_ILMETHOD_SECT_EH

		structure for more encoding details
CorILMethod_Sect_OptILTable	2	Reserved for future use
CorILMethod_Sect_FatFormat	0x40	Data format is of the fat variety, meaning there is a 3 byte length. If not set, the header is small with a 1 byte length
CorILMethod_Sect_MoreSects	0x80	Another data section occurs after this current section

3.2.3 IMAGE_COR_ILMETHOD_SECT_EH

Exceptions are declared as an extra section of data that comes after a method header. This structure defines an element of the Exception Handling information for a method. The structure must be emitted on a DWORD boundary. This structure is a union of the IMAGE_COR_ILMETHOD_SECT_EH_SMALL and IMAGE_COR_ILMETHOD_SECT_EH_FAT structure types, depending on how much data is present.

3.2.3.1 IMAGE_COR_ILMETHOD_SECT_EH_SMALL

The layout of this structure as is a follows:

```
typedef struct IMAGE_COR_ILMETHOD_SECT_EH_SMALL
{
    IMAGE_COR_ILMETHOD_SECT_SMALL SectSmall;
    WORD Reserved;
    IMAGE_COR_ILMETHOD_SECT_EH_CLAUSE_SMALL Clauses[1];   // actually variable size
} IMAGE_COR_ILMETHOD_SECT_EH_SMALL;
```

Offset	Size	Field	Description
0	1	**Kind**	CorExceptionFlag type for this block
1	1	**DataSize**	Size of the data for the block not including the header
2	2	**Reserved**	DWORD padding
4	*n*	**Clauses**	One or more clauses as defined by DataSize

3.2.3.2 CorExceptionFlag Values

The following flag values are used for each exception-handling clause:

Flag	Value	Description
COR_ILEXCEPTION_CLAUSE_NONE	0x0000	
COR_ILEXCEPTION_CLAUSE_FILTER	0x0001	Entry is for an exception filter
COR_ILEXCEPTION_CLAUSE_FINALLY	0x0002	A finally clause
COR_ILEXCEPTION_CLAUSE_FAULT	0x0004	Fault clause (finally that is called

		on exception only)

3.2.3.3 IMAGE_COR_ILMETHOD_SECT_EH_CLAUSE_SMALL

The small form of the exception clause should be used whenever the code size for the try block and handler code is smaller than or equal to 256 bytes. The format for a small exception clause is as follows:

```
typedef struct IMAGE_COR_ILMETHOD_SECT_EH_CLAUSE_SMALL
{
    CorExceptionFlag    Flags         : 16;
    unsigned            TryOffset     : 16;
    unsigned            TryLength     : 8;  // relative to start of try block
    unsigned            HandlerOffset : 16;
    unsigned            HandlerLength : 8;  // relative to start of handler
    union {
        DWORD       ClassToken;
        DWORD       FilterOffset;
    };
} IMAGE_COR_ILMETHOD_SECT_EH_CLAUSE_SMALL;
```

Offset	Size	Field	Description
0	2	**Flags**	CorExceptionFlag type for this block
2	2	**TryOffset**	Offset of a try block
4	1	**TryLength**	Length in bytes of the try block
5	2	**HandlerOffset**	Location of the handler for this try block
7	1	**HandlerLength**	Size of the handler code in bytes
8	4	**ClassToken**	Meta data token for a type-based exception handler
8	4	**FilterOffset**	Offset in method body for filter-based exception handler

3.2.3.4 IMAGE_COR_ILMETHOD_SECT_EH_FAT

Used to describe a large exception clause that will not fit in the COR_ILMETHOD_SECT_EH_SMALL structure.

```
typedef struct IMAGE_COR_ILMETHOD_SECT_EH_FAT
{
    IMAGE_COR_ILMETHOD_SECT_FAT   SectFat;
    IMAGE_COR_ILMETHOD_SECT_EH_CLAUSE_FAT Clauses[1];     // actually variable size
} IMAGE_COR_ILMETHOD_SECT_EH_FAT;
```

Offset	Size	Field	Description
0	1	**Kind**	Which type of exception block is being used

1	3	**DataSize**	How big is the data excluding the header size
4	*n*	**Clauses**	One or more clauses describing exception handling

3.2.4 IMAGE_COR_ILMETHOD_SECT_EH_CLAUSE_FAT

Use this structure when the smaller clause cannot handle the size of the data. The layout of this structure is as follows:

```
typedef struct IMAGE_COR_ILMETHOD_SECT_EH_CLAUSE_FAT
{
    CorExceptionFlag    Flags;
    DWORD               TryOffset;
    DWORD               TryLength;      // relative to start of try block
    DWORD               HandlerOffset;
    DWORD               HandlerLength;  // relative to start of handler
    union {
        DWORD           ClassToken;     // use for type-based exception handlers
        DWORD           FilterOffset;   // use for filter-based exception handlers
                                        //   (COR_ILEXCEPTION_FILTER is set)
    };
} IMAGE_COR_ILMETHOD_SECT_EH_CLAUSE_FAT;
```

Offset	Size	Field	Description
0	4	**Flags**	CorExceptionFlag type for this block
4	4	**TryOffset**	Offset of a try block
8	4	**TryLength**	Length in bytes of the try block
12	4	**HandlerOffset**	Location of the handler for this try block
16	4	**HandlerLength**	Size of the handler code in bytes
20	4	**ClassToken**	Meta data token for a type-based exception handler
20	4	**FilterOffset**	Offset in method body for filter-based exception handler

4 Code Transitions

For first release of the runtime product, one is allowed to mix unmanaged native code and managed IL code in the same image. Not supported but planned for a future release is "Managed Native Code", which is managed code emitted using the native instructions for a given CPU instead of MSIL. This section of the specification documents the layout of these mixed images.

Following is a definition of terms:

Managed	Code that requires the runtime for execution. A managed program can use the VOS and take advantage of other runtime features that an unmanaged program cannot. Additional metadata is required in the image to be managed. This includes information about garbage collection, exception handling, and the locations of methods in the file. A managed program cannot run without the assistance, at runtime, of the execution engine
Unmanaged	Code that does not require the runtime for execution. This code may not use the VOS or other features of the runtime. Traditional native code (before the NGWS runtime) is considered unmanaged
Code Manager	A code manager contains the code required to walk the state of a running program (such as tracing the stack and track GC references). NGWS supports pluggable code managers, and provides default versions for JIT'ed code, IL, and F-Jit
Thunk	A (typically) small piece of code used to provide a transition between two pieces of code where special handling is required

4.1 Call Transitions

4.1.1 Transition Types

A Transition is any boundary between types of code that requires special handling. When the caller and the callee code have the same properties, then no special transition is required to make the call. There are three main categories discussed where transitions may be required:

Runtime Type	A method may be ***mnaged*** or ***unmanaged***. Transitions between the two types in either direction require a thunk
Code Type	A method body may be stored in ***IL***, ***Opt-IL***[3], or n***ative*** byte codes. Transition between IL and Opt-IL is handled by the Runtime.

[3] Opt-IL is not supported for first release of the product. Some file format elements allow this format for future use.

	Transitions between any form of IL and Native require a thunk
Location Type	The implementation for a method may appear either in the ***local*** image, or may be ***imported*** from another image. When calling an imported method, a thunk may be required.

Any permutation of types above can result in the requirement for a thunk. This thunk is generated by the runtime during execution and will set up the correct state for the call. For example, when making a transition from managed to unmanaged, the GC state for the thread is marked as pre-emptive (meaning it will not read or write GC reference) and an exception filter is put in place to protect the managed code from errors that occur. It is beyond the scope of this document to describe the different types of thunks and their contents.

Effects on Like Pieces of Code

If the caller and callee have the same properties (say managed-native calling managed-native in the same module), then there is no special handling required. Only the minimal amount of metadata should be emitted in this case, and normal code generation rules apply (ie: just make a PC relative call to the target).

Affects on IL Code

When the caller is any form of IL, the runtime has complete control over the execution of the code. It will almost always be JIT-compiled, in which case the jitter may generate any transition thunks inline with the code.

Affects on Native Code

When the caller is native code, there is no opportunity to change code once the compiler has emitted it[4]. Because of this, a mechanism is required which allows the runtime to provide the thunk capable of setting up the right state. The rest of this section describes this mechanism in detail.

4.2 Runtime Header Support for Transitions

This section contains descriptions of directory entries in the header to support managed native code.

4.2.1 VTableFixups

The VTableFixups directory entry is used to declare transitions. In order to have a transition, for example from managed to unmanaged, one does the following:

1. Allocate a slot in a read/write portion of the file
2. Point a IMAGE_COR_VTABLEFIXUP entry at the slot
3. Set the Type to indicate managed to unmanaged
4. Place the token of the target in the slot

[4] Note that we could consider modifying the code in the .text section, but this causes the pages to be marked writable in the process using up extra working set size. This solution was discarded.

The following transition types are interesting:

Combination	Description
managed to managed Token	For non-VOS v-table fix-ups. For example, a C++ compiler creating its own v-tables
managed to unmanaged Token	For calling an unmanaged call site within a PE image where full PInvoke metadata is not required
COR_VTABLE_FROM_UNMANAGED to managed Token	For calling a managed call site from unmanaged code, such as in a classic v-table

Unmanaged to unmanaged through a transition is not interesting and not supported.

4.2.2 Export Address Table Fix-ups

Exporting a managed method using an unmanaged entry point is supported through the EAT. Only unmanaged entry points may be found in the EAT to provide compatibility with down level loaders. One uses the metadata to import/export managed methods in managed code.

In a pure IL image, the EAT itself would be sufficient to find all of these fix-up entries. But in a mixed managed/unmanaged image, only those EAT entries that need a thunk are to be fixed up and touching other entries would cause corruption. Therefore, the ExportAddressTableJumps table is used to identify which entries in the EAT are for managed methods.

Building an export table amounts to the following steps:

1. Use the VTableFixup support described above to allocate an unmanaged to managed slot
2. Put the token of the managed method to be exported into this slot
3. Allocate a 32-byte piece of memory, called the "EAT Jump Thunk", in the image that will not move. Put the address of this memory in the EAT of the PE format for the method.
4. Initialize the first 4 bytes of the EAT Jump Thunk with the RVA of the slot allocated in step #1. Initialize the next 4 bytes after this RVA to zero (reserved for future use)
5. Create the ExportAddressTableJumps table and put the RVA of the EAT Jump Thunk into this table.

At runtime, the following sequence of events takes place:

1. The OS loader will do the transitive closure of all IAT's of all images being loaded. This will take the address from each EAT and place it into the IAT before any user code runs.
2. When the runtime loader runs, it will process all VTableFixup table entries and will replace the token from step #2 above with a pointer to an unmanaged thunk for the managed method.
3. Next the runtime loader walks the ExportAddressTableJumps entries to find EAT Jump Thunks that need to get replaced. It uses the RVA stored in the jump

thunk to find the address of the unmanaged thunk from the previous step. Finally, it replaces the EAT Jump Thunk contents with a native jump instruction which jumps to the unmanaged thunk.

It is worth pointing out a couple of caveats that drove this design:

- On down-level OS loaders, the runtime loader runs only when the OS turns control over to the PE entry point. Therefore, the EAT cannot be initialized until that point. By requiring the EAT Jump Thunk, this requirement is met. On future OS's that cannot encode the jump thunk in the predetermined sized limit, an OS change is required.
- Calling an EAT entry for a managed function in your DLL's initialization routine is forbidden.
- The runtime loader does not track the location of the thunks it builds. Because of this, using the VTableFixup table allows the same thunk to be shared many times. The second 4 bytes of the EAT Jump Thunk are set to zero in case this restriction is removed in the future and an alternate way of building the data is required.

5 Entry Points

This section contains an overview of entry point handling in loaded images.

5.1 Runtime API's

The following API's are exported from mscoree.dll and are used to bootstrap the runtime.

5.1.1 _CorExeMain

This method is called to start execution of an executable program. The API will retrieve the handle of the image using GetModuleHandle(NULL). The module is then added to the loaded module list.

```
__int32 __stdcall _CorExeMain();
```

5.1.2 _CorDllMain

This method is called for the entry point of a DLL. This routine uses the handle to the module passed in to add the module to the loaded list.

```
__int32 __stdcall _CorDllMain(HINSTANCE, DWORD, void *);
```

5.1.3 Entry Points for Windows CE

Windows CE throws out the PE header to reduce overall working set. Because of this, all required data from the PE header is passed to the runtime entry points at runtime. The prototypes on this platform are therefore different.

5.2 Shut Down Requirements

The runtime must be given a chance to shut itself down in the process in order to do things like GC finalization, and cleanup for COM interop. This is done as transparently as possible from a developer's point of view. However, some problems can occur with ill-behaved applications. One such example is creating an unmanaged executable image (ie: a .exe) that supplies its own entry point but does not call ExitProcess. Such an application may work if it were single threaded, but would fail to work with the runtime as the runtime system threads do not get a notification to shut down.

5.3 Entry Point Stubs

5.3.1 Runtime Aware OS Loader

A runtime-aware operating system can recognize a runtime image by checking directory entry 14 in the PE header. The general loading algorithm is as follows:

1. If directory entry 14 is not set, the image is not runtime-aware and should be loaded using normal load sequences.

2. If the CPU type is not x86, and does not match the current CPU, the OS should reject the image with an error.
3. If the CPU type is x86, check the flags in the heading for COMIMAGE_FLAGS_ILONLY. When this bit is set, it means the x86 code in the image is just a loader stub, and should be ignored. If this flag is not set, the image requires an x86 capable CPU to run.

If the image is a 32-bit PE format, and the flags in the runtime header have the COMIMAGE_FLAGS_32BITREQUIRED bit set, the image cannot be loaded in a 64-bit process. In this case, the OS should reject the load with an error.

If the image is a PE 32+ PE format (64-bit), then the image must be loaded into a 64-bit process.

5.3.2 Non Runtime Aware OS Loader

Existing supported platforms include Win '9x and x86 NT 4.0 and above[5]. For this reason, runtime images should emit an x86 startup stub that these OS platforms will understand and load. The load algorithm is therefore:

1. The OS will validate the image is an x86 image (Intel only), and page the image into the process.
2. The OS will invoke the native entry point, which is an x86 load stub. This stub will call the entry point API of mscoree (_CorExeMain or _CorDllMain) through the IAT section of the image.
3. The mscoree entry point code will use the module handle to load the runtime metadata out of the image, including the user's entry point specified in the runtime header, if one was given.
4. The runtime will then invoke the user's entry point code, jitting it if required.

5.3.3 Sample x86 Stubs

The following stubs are used by runtime tools when emitting applications:

```
//*****************************************************************************
// Stubs.h
//
// This file contains a template for the default entry point stubs of a runtime
// IL only program.  One can emit these stubs (with some fix-ups) and make
// the code supplied the entry point value for the image.  The fix-ups will
// in turn cause mscoree.dll to be loaded and the correct entry point to be
// called.
//
// Copyright (c) 1997-1998, Microsoft Corp.  All rights reserved.
//*****************************************************************************
#pragma once

//*****************************************************************************
// This stub is designed for a Windows application.  It will call the
// _CorExeMain function in mscoree.dll.  This entry point will in turn load
```

[5] Windows CE version 3 is a runtime-aware OS loader. The runtime will not work on a Windows CE platform before this version.

```
// and run the IL program.
//
// void ExeMain(void)
// {
//    _CorExeMain();
// }
//
// The code calls the imported functions through the iat, which must be
// emitted to the PE file.  The two addresses in the template must be replaced
// with the address of the corresponding iat entry which is fixed up by the
// loader when the image is paged in.
//*****************************************************************************
static const BYTE ExeMainTemplate[] =
{
   // Jump through IAT to _CorExeMain
   0xFF, 0x25,                // jmp iat[_CorDllMain entry]
      0x00, 0x00, 0x00, 0x00, //   address to replace

};

#define ExeMainTemplateSize         sizeof(ExeMainTemplate)
#define CorExeMainIATOffset         2

//*****************************************************************************
// This stub is designed for a Windows application.  It will call the
// _CorDllMain function in mscoree.dll with with the base entry point
// for the loaded DLL.
// This entry point will in turn load and run the IL program.
//
// BOOL APIENTRY DllMain( HANDLE hModule,
//                        DWORD ul_reason_for_call,
//                        LPVOID lpReserved )
// {
//         return _CorDllMain(hModule, ul_reason_for_call, lpReserved);
// }

// The code calls the imported function through the iat, which must be
// emitted to the PE file.  The address in the template must be replaced
// with the address of the corresponding iat entry which is fixed up by the
// loader when the image is paged in.
//*****************************************************************************

static const BYTE DllMainTemplate[] =
{
   // Call through IAT to CorDllMain
   0xFF, 0x25,                 // jmp iat[_CorDllMain entry]
      0x00, 0x00, 0x00, 0x00,  //   address to replace
};

#define DllMainTemplateSize      sizeof(DllMainTemplate)
#define CorDllMainIATOffset      2

#elif defined(_ALPHA_)
```

```
const BYTE ExeMainTemplate[] =
{
   // load the high half of the address of the IAT
    0x00, 0x00, 0x7F, 0x27,     // ldah t12,_imp__CorExeMain(zero)
   // load the contents of the IAT entry into t12
   0x00, 0x00, 0x7B, 0xA3,     // ldl    t12,_imp__CorExeMain(t12)
   // jump to the target address and don't save a return address
   0x00, 0x00, 0xFB, 0x6B,     // jmp    zero,(t12),0
};

#define ExeMainTemplateSize      sizeof(ExeMainTemplate)
#define CorExeMainIATOffset      0

const BYTE DllMainTemplate[] =
{
   // load the high half of the address of the IAT
    0x42, 0x00, 0x7F, 0x27,     // ldah t12,_imp__CorDLLMain(zero)
   // load the contents of the IAT entry into t12
   0x04, 0x82, 0x7B, 0xA3,     // ldl    t12,_imp__CorDLLMain(t12)
   // jump to the target address and don't save a return address
   0x00, 0x00, 0xFB, 0x6B,     // jmp    zero,(t12),0
};

#define DllMainTemplateSize      sizeof(DllMainTemplate)
#define CorDllMainIATOffset      0
```