NGWS

Platform Invoke Metadata Guide

This is preliminary documentation and subject to change

Last updated: 8 June 2000

This spec is aimed at compiler writers who will be emitting metadata for Platform Invocation services (PInvoke) For a broad overview of the Platform Invocation Services from a component or application developer's point of view, refer to the Platform Invoke Usage Guide

Table Of Contents

1 Overview of PInvoke Marshalling

Platform Invocation Services, abbreviated throughout this spec to "PInvoke", allows managed code to call unmanaged functions that are implemented in a DLL. PInvoke takes care of finding and invoking the correct function, as well as marshalling its managed arguments to and from their unmanaged counterparts (integers, strings, arrays, structures, etc).

PInvoke was intended primarily to allow managed code to call existing, unmanaged code, typically written in C. A good example is the several thousand functions that comprise the Win32 API.

As mentioned above, PInvoke marshals function arguments between managed and unmanaged code. For simple data types (bytes, integers, floats, etc), or arrays of those simple types, marshalling is straightforward. Even for strings, so long as you specify whether the unmanaged code expects an Ansi string, a Unicode string, or a BSTR, marshalling is again without problems.

But marshalling of structured arguments presents a problem. (Structured types are also known as structs, records, aggregates, etc, depending upon which source language we are discussing. We shall call them "structs" in this spec). Given free-rein, the runtime will lay out the fields of a managed struct in the 'most efficient' way. What is 'most efficient'? Well, it includes making garbage collection fast and space-efficient. It can also take account of access patterns. The point is, that the runtime's choice of layout will rarely match what unmanaged code (typically C) expects, and has hard-wired into its machine code as fixed offsets – where fields of a struct are laid out in the lexical order they were defined in the source code.

[As an aside, you might wonder how user's managed code can ever 'find' the right fields in a class which the runtime lays out in memory at its own whim. The answer is that field access within MSIL is done via metadata tokens; in effect, these provide the 'name' of the field to be accessed, rather than its predefined byte offset within the managed struct]

So, somehow, at runtime, PInvoke must 'manufacture' and hand over a struct, holding fields in the exact order and size that unmanaged code expects. The way it does this is firstly to disallow runtime's normal freedom for how it lays out managed structs (classes or valuetypes); instead, it directs the runtime to lay the struct out in managed memory in the way most-nearly expected by the unmananaged user of this struct. We call such an item a "formatted type".

For many cases, we can achieve an exact, byte-by-byte, match between the managed object and the struct the unmanaged code expects; in these cases, we say the managed and unmanaged struct are "isomorphic". When PInvoke calls the unmanaged code, it can either pin the managed object (so that it will not be moved by garbage collection), and hand a pointer to the managed code; or it can allocate some memory (unmanaged heap or stack) and do a fast, 'blind', byte-by-byte copy from the managed isomorphic object. Either technique results in low overhead.

But there are some cases (non-isomorphic), where PInvoke must carry out marshalling – copying and reformatting of data – at runtime. This is slower than if the struc were isomorphic. The common cases which destroy isomorphism include:

- managed string is Unicode, but the unmanaged code expects Ansi
- managed argument or field is boolean; this occupies 1 byte in managed memory, but 2 or 4 bytes in unmanaged structures

A programmer can avoid the inefficiency incurred with managed boolean fields by declaring them as 2-byte or 4-byte integers instead.

In all other respects, except its predefined field layout in memory, a "formatted" object looks just like a regular managed object. In particular, managed code can read and write all its fields with MSIL instructions.

When it comes to call an unmanaged function, PInvoke locates the DLL where it lives, loads that DLL into process memory, finds the function address in memory, pushes its arguments onto the stack (marshalling if required) and transfers control to the address for the unmanaged code. If the arguments are isomorphic, then no marshalling is required.

2 Overview of PInvoke Metadata

This document specifies what information a tool or compiler must emit into metadata to describe how PInvoke should call an unmanaged function from the Runtime. This information includes the location of the target function (which DLL it lives in) and its signature (number of arguments, their type, and any function return type).

Each compiler provides a construct for its users to decorate methods and arguments with the required PInvoke information. For example, managed C++ provides the "sysimport" attribute, whilst Visual Basic provides the "DECLARE" statement. The compiler parses the decoration and emits the corresponding language-neutral metadata that will be used by PInvoke.

Information required by PInvoke falls into three kinds:

- Define the NGWS method that corresponds to the unmanaged function. This includes its name, location, arguments and return type
- Where the unmanaged code expects a struct argument, define an NGWS class that corresponds to the unmanaged struct – its fields, layout and alignment
- Where the default marshalling provided by PInvoke is not what you want, override with a different marshalling behaviour

Building PInvoke metadata can be quite simple. Here is an example (the source language doesn't matter; its intent should be clear):

```
class C {
    [sysimport(dll = "user32.dll")]
    public static extern int MessageBoxA(int h, string m, string c, int type);
    public static int Main() {
        return MessageBoxA(0, "Hello World!", "Caption", 0);
    }
}
```

To build the corresponding PInvoke metadata, you need only call *DefineMethod*, *DefinePinvokeMap* and *DefineParam* for each parameter. (Individual compilers may choose to structure their definitions differently – a *DefineMethod* followed by a *SetMethodProps*, for example, but the suggested sequence is possible)

On the other hand, building PInvoke metadata can also come quite involved, as witnessed by the size of this spec, and the other specs, listed below, that support it. This happens if your unmanaged function accepts struct arguments, and requires non-default marshalling. Here is a second, more complicated example:

```
[sysstruct(format=ClassFormat.Auto)]
public class LOGFONT {
   public const int LF_FACESIZE = 32;
   public int lfHeight;
   public int lfWidth;
   public int lfEscapement;
   public int lfOrientation;
   public int lfWeight;
   public byte lfItalic;
   public byte lfUnderline;
   public byte lfStrikeOut;
   public byte lfCharSet;
   public byte lfOutPrecision;
   public byte lfClipPrecision;
   public byte lfQuality;
   public byte lfPitchAndFamily;
   [nativetype(NativeType.FixedSysString, size=LF_FACESIZE)]
   public string lfFaceName;
};

class C {
   [sysimport(dll="gdi32.dll",charset=CharacterSet.Auto)]
   public static extern int CreateFontIndirect(
      [in, nativetype(NativeType.NativeTypePtr)]
      LOGFONT lplf   // characteristics
   );
   public static void Main() {
      LOGFONT lf = new LOGFONT();
      lf.lfHeight = 9;
      lf.lfFaceName = "Arial";
      int i = CreateFontIndirectA(lf);
      Console.WriteLine(i);
   }
}
```

To build the PInvoke metadata for this example requires calls to most of the following routines in the *IMetaDataEmit* interface:

DefineMethod: Define a method, with its NGWS method signature

DefinePinvokeMap: Specify PInvoke info for a method

DefineTypeDef: Define an NGWS class or valuetype, used as an argument to a PInvoke-dispatched function

DefineField: Define a data field within a class

SetClassLayout: Supply additional info on the class layout, such as field packing

SetFieldMarshal: Supply non-default marshaling for a function argument, function return value, or a field within a struct

For more info on specific areas touched upon in this spec, see the following documents:

- Platform Invoke Usage Guide for an overview of PInvoke from the user's perspective
- Metadata Interfaces for an overview of metadata, and details of specific routines
- MetadataStructures for the format of signatures (methods and fields)
- DataTypeMarshaling for details of all the field marshalling supported by Pinvoke (much of it shared with COM Interop)

3 Metadata for Methods

You must define a managed method, that describes the target unmanaged function you wish to reach via PInvoke. You may include several methods in a given class that describe unmanaged functions, or you can define a separate class and method for each unmanaged function; the choice is yours.

In the descriptions that follow, all metadata methods are defined on the *IMetaDataEmit* interface.

3.1 DefineMethod for PInvoke

For each unmanaged function you want to call via PInvoke, you must define a managed method, that describes that target unmanaged function. For this, use *DefineMethod.* This routine is used to define all managed methods to the metadata. However, when used for methods that match to PInvoke-dispatched native functions, some of the arguments have particular restrictions. These are listed in the next table.

in/out	**Parameter**	**Description**	**Required?**
in	td	TypeDef token of parent	no
in	wzName	Member name in Unicode	yes
in	dwMethodFlags	Member attributes	yes
in	pvSig	Method signature	yes
in	cbSig	Count of bytes in pvSig	yes
in	ulCodeRVA	Address of code	must be 0
in	dwImplFlags	Implementation flags for method	no, may be all 1s
out	pmd	Member token	

dwMethodFlags is a bitmask from the *CorMethodAttr* enum in CorHdr.h. You must: set *mdStatic*; clear *mdSynchronized*; clear *mdAbstract*

pvSig must be a valid NGWS method signature. Each parameter must be a valid NGWS (as opposed to unmanaged) data type. See the MetadataStructures spec for how to compose an NGWS method signature; take special note of the *CorCallingConvention* enum in CorHdr.h

ulCodeRVA must be zero

dwImplFlags is a bitmask from the *CorMethodImpl* enum in CorHdr.h. You must: set *miNative*; set *miUnmanaged*

3.2 DefineMethodImpl for PInvoke

If you are defining the implementation for a method that is defined by an interface, you use *DefineMethodImpl.* This routine accepts only a subset of what *DefineMethod* accepts, because some of the inherited information cannot be changed (for example, the name of the method). Whilst this is used for regular managed methods, we do not support its use for PInvoke.

3.3 DefinePinvokeMap for PInvoke

Use *DefinePinvokeMap* to provide further information about a method already defined by the *DefineMethod* call above.

in/out	Parameter	Description	Required?
in	tk	Token for target method – a MethodDef or MethodImpl	yes
in	dwMappingFlags	Flags used by Pinvoke to do the mapping	yes
in	wzImportName	Name of target export method in unmanaged DLL	no
in	mdImportDLL	mdModuleRef token for target DLL	yes

dwMappingFlags is a bitmask from the *CorPinvokeMap* enum in CorHdr.h. You can set the following flags:

- *pmNoMangle* – if set, function name is used as-is in searching the target native DLL (ie, no fuzzy matching)
- *pmCharSetAnsi*, *pmCharSetUnicode*, *pmCharSetAuto* – set one as appropriate
- *pmSupportLastError* – if set, user can query last error set within the unmanaged method

3.4 SetPinvokeMap for PInvoke

Use *SetPinvokeMap* to provide further information, or change the information, you supplied in an earlier call to *DefinePinvokeMap*. The arguments, their meanings and restrictions are exactly as for *DefinePinvokeMap*, above.

3.5 Method Signatures for PInvoke

The call to *DefineMethod* includes an argument, called pvSig, that takes the signature of the method. This blob specifies the method's signature – the type for each argument, and for the return type, if any. The format of this blob is defined in the MetadataStructures spec. This section summarizes details of the signature that are specific to its use for PInvoke:

- All data types must be NGWS data types, even though they end up, after PInvoke dispatch, as arguments to an unmanaged function
- PInvoke provides default, automatic marshaling of simple (non-struct) arguments, and of simple fields within struct arguments. The defaults are chosen using heuristics about the NGWS data type declaration, target platform, and method-level ansi/unicode/auto attribute. (This default marshaling can be over-ridden if required – see *SetFieldMarshal*)
- In the NGWS method signature, a struct argument should be declared as an NGWS class or valuetype that carries layout information (what we called a "formatted type" in the discussion above).

4 Metadata for Function Parameters

You should specify the *direction* of each parameter to an unmanaged function. That's to say, whether it is an in, out, or inout parameter. In the cases where a copy of the corresponding argument is made for the unmanaged code to access (typically, for a non-isomorphic struct passed by-reference), the setting of these flags is important. In these cases, PInvoke does the following:

- **in**: make a copy of the managed struct for the unmanaged code to access. This struct is **not** copied back to the managed caller
- **out**: create a freshly-initialized, unmanaged struct for the unmanaged code to access. This struct **is** copied back to the managed caller
- **inout**: make a copy of the managed struct for the unmanaged code to access. This struct **is** copied back to the managed caller

Note that where a struct is isomorphic, but specified only as **in** or as **out**, PInvoke may, for reasons of efficiency, pin the managed struct and pass a reference to that struct to the unmanaged code. In such cases, the behaviour that results will be as if you had asked for **inout**.

4.1 DefineParam for PInvoke

To specify each parameter's *direction*, call *DefineParam*. Do not specify a default value – *dwDefType*, *pValue* or *cbValue*.

in/out	Parameter	Description	Required?
in	md	Token for the method whose parameter is being defined	yes
in	ulParamSeq	Parameter sequence number	yes
in	wzName	Name of parameter in Unicode	no
in	dwParamFlags	Flags for parameter	no
in	dwDefType	ELEMENT_TYPE_* for the constant value	no
in	pValue	Constant value for parameter	no
in	cbValue	Size in bytes of pValue	no
out	ppd	ParamDef token assigned	

ulParamSeq specifies the parameter sequence number, starting at 1. Use a value of 0 to mean the method return value

wzName is the name to give the parameter. If you specify null, this argument is ignored

dwParamFlags is a bitmask from the *CorParamAttr* enumeration in CorHdr.h. Set the *pdIn* and/or *pdOut* bits in this mask

4.2 SetParamProps for PInvoke

As an alternative to *DefineParam*, you may use *SetParamProps*. This is an unlikely scenario, except perhaps during an incremental compilation session. However, for the record, here is the detail – the same restrictions apply as for *DefineParam*

in/out	Parameter	Description	Required?
in	pd	Token for target parameter	yes
in	wzName	Name of parameter in Unicode	no
in	dwParamFlags	Flags for parameter	no
in	dwDefType	ELEMENT_TYPE_* for the constant value	no
in	pValue	Constant value for parameter	no
in	cbValue	Size in bytes of pValue	no

5 Metadata for Struct Arguments

As mentioned previously, a PInvoke-called function can accept struct arguments. Such arguments are expressed as NGWS classes or valuetypes that include layout information (“formatted types”). This section describes details of how to specify layout in the *DefineTypeDef* call you make to define those classes.

5.1 DefineTypeDef for PInvoke

Although supported, it is unlikely that a compiler will define methods or properties for a “formatted type” – that’s to say, for a managed class or valuetype, whose purpose is to describe a matching unmanaged struct argument for a PInvoke-called function. Routinely, the type definition will include only fields – no methods, no properties, no superclass, no interfaces-to-implement. With these simplifications, the arguments to *DefineTypeDef* for a PInvoke struct, are as follows:

in/out	Parameter	Description	Required?
in	wzName	Name of type in Unicode	yes
in	pVer	Version number. Specify as null	no
in	dwTypeDefFlags	Typedef attributes	yes
in	tkExtends	Token of the superclass. Specify as zero	yes
in	rtkImplements[]	Array of tokens specifying the interfaces that this class or interface implements (inherits via interface inheritence). Specify as null	no
out	ptd	TypeDef token assigned	

dwTypeDefFlags is a bitmask from the *CorTypeAttr* enum in CorHdr.h. You must set either *tdLayoutSequential*, or *tdExplicitLayout* (not both). You should set *tdAnsiClass*, *tdUnicodeClass* or *tdAutoClass*.

If your struct has no unions, then set *tdLayoutSequential*, and, if necessary, call *SetClassLayout* to provide more details. If you are in the unfortunate position that your struct includes unions (sometimes called overlays, depending upon source language), or your struct includes weird padding between fields, then you must set *tdExplicitLayout*, and follow with a call to *SetClassLayout* to provide more details.

The string formatting flags say how managed strings (which are always encoded in Unicode) should be marshalled to and from unmanaged code:

- *tdAnsiClass* – PInvoke will marshal to unmanaged Ansi
- *tdUnicodeClass* – PInvoke will pin, or copy, to unmanaged Unicode (no format change of the individual characters required)
- *tdAutoClass* – PInvoke will choose *tdAnsiClass* or *tdUnicodeClass*, by inspecting which platform it is being executed upon

5.2 DefineField for PInvoke

Having defined the struct using *DefineTypeDef*, the next step is to define each field in the struct, using *DefineField.* Just follow the usual rules for using *DefineField*; there are no special rules to apply just because these are fields of a struct that will be used for PInvoke. As a reminder, here are the arguments for the *Definefield* method:

in/out	**Parameter**	**Description**	**Required?**
in	td	Typedef token for the enclosing class	yes
in	wzName	Field name in Unicode	yes
in	dwFieldFlags	Field attributes	yes
in	pvSig	Field signature as a blob	yes
in	cbSig	Count of bytes in pvSig	yes
in	dwDefType	ELEMENT_TYPE_* for the constant value	no
in	pValue	Constant value for field	no
in	cbValue	Size in bytes of pValue	no
out	pmd	FieldDef token assigned	

dwFieldFlags is a bitmask from the *CorFieldAttr* enumeration in CorHdr.h

dwDefType is a value from the *CorElementType* enumeration in CorHdr.h. If you do not want to define any constant value for this field, supply a value of ELEMENT_TYPE_END

For details of how to construct the signature blob, see the MetadataStructures spec

5.3 SetClassLayout for PInvoke (Sequential)

If you told *DefineTypeDef* that your struct was tdLayoutSequential, then you should call *SetClassLayout* to further define the field layout.

in/out	**Parameter**	**Description**	**Required?**
in	td	Token for the class being laid out	yes
in	dwPackSize	Packing size: 1, 2, 4, 8 or 16 bytes	no
in	rFieldOffsets	Array of mdFieldDef / ululByteOffset values for each field. Specify as zero	no
in	ulClassSize	Overall size of these class objects, in bytes	no

dwPackSize is the packing size between adjacent fields. For each field in sequence, the runtime looks at its size, and current offset within the struct. It lays the field down to start at its natural offset, or the pack size, whichever results in the *smaller* offset. This matches precisely the semantics of the C and C++ #pragma pack compiler directive

rFieldOffsets is not required in this instance. Specify it as zero

ulClassSize is optional. If you specify this argument, then PInvoke will marshal this struct argument by making a blind, byte-by-byte copy of the managed object. [This technique is used by Visual C++]

5.4 SetClassLayout for PInvoke (Explicit)

If you told *DefineTypeDef* that your struct was tdExplicitLayout, then you must call *SetClassLayout* to further define the field layout.

in/out	Parameter	Description	Required?
in	td	Token for the class being laid out	yes
in	dwPackSize	Packing size. Specify as zero	no
in	rFieldOffsets	Array of mdFieldDef / ululByteOffset values for each field on the class for which sequence or offset information is specified. Terminate array with mdTokenNil.	no
in	ulClassSize	Overall size of these class objects. Specify as zero	no

rFieldOffsets is an array of *COR_FIELD_OFFSET*s. The *COR_FIELD_OFFSET* struct is defined in CorHdr.h, but repeated here for convenience:

```
typedef struct COR_FIELD_OFFSET {
    mdFieldDef  tokField;
    ULONG       ulOffset;
} COR_FIELD_OFFSET;
```

The *tokField* is the token for the target field; the *ulOffset* is the byte offset within the struct at which it starts. The struct is assumed to start at offset 0. (So, if you specify just one field, 4 bytes wide, with a *ulOffset* of 1000, then you create a managed struct that is 1004 bytes long). Terminate the *rFieldOffsets[]* array with a field token of *mdTokenNil*.

6 Metadata for Explicit Marshalling

If the default marshaling provided by PInvoke is just what you need, then you can skip this section. However, if you want to specify non-default marshaling for any of the following items:

- function return value
- function argument
- field within a struct that is a function argument

then you must specify the requested behaviour using *SetFieldMarshal*.

Both PInvoke and COM Interop provide marshaling of data between managed and unmanaged code. And for most data types, they share the same marshalling code. The marshalling behaviour is specified in the DataTypeMarshaling spec. Please consult this spec for details of PInvoke's default marshaling (in most cases the same as for COM Interop), as well as the valid alternatives you may specify for non-default marshalling.

6.1 SetFieldMarshal for PInvoke

For each item that requires non-default marshaling, call *SetFieldMarshal* and specify which native type the item should be marshalled to.

in/out	**Parameter**	**Description**	**Required?**
in	tk	Token for target item	yes
in	pvUnmgdType	Signature for unmanaged type	yes
in	cbUnmgdType	Count of bytes in pvUnmgdType	yes

For details of how to build the native type signature, see the MetadataStructures spec

7 Custom Attributes

Compilers can alternatively set marshaling information by emitting certain pre-defined Custom Attributes (eg the *MarshalAsAttribute*). This enables compilers to use their generic code, that parses and handles all Custom Attributes, to be used to direct the operation of the runtime for PInvoke marshalling.

For details on Custom Attributes in general, see the Metadata Interfaces spec. For details of those specific Custom Attributes used to define PInvoke marshaling information, see the DataTypeMarshalling spec

Each compiler is free to choose which way it emits metadata – depending upon the tradeoff it chooses among the following factors:

1. compile-time speed and efficiency
2. ease of use
3. good argument checking
4. good isolation or *genericity*
5. whether the compiler itself is written in managed or unmanaged code

Broadly speaking, using unmanaged metadata-emit APIs is good for 1. Using unmanaged metadata-emit APIs, together with pre-defined Custom Attributes provides some level of 4, at the cost of slowing compilation. Use of Reflection Emit (only works for case 5) is good for 2, 3 and 4.