Reading and Writing PE-files with PERWAPI

John Gough and Diane Corney

Applies to *PERWAPI* version of August 2005.
Document revision date – 14 September 2005.

1 Introduction

PERWAPI is a component for reading and writing *.NET PE*-files. The name is a compound acronym for *Program Executable – Reader/Writer – Application Programming Interface*. The code was written by one of us (Diane Corney) with some contributions from some of the early users of the tool.

PERWAPI is a managed component, written entirely in safe *C#*. The design of the *writer* part of the component is loosely based on Diane Corney's previous *PEAPI* component. It is open source software, and is released under a "FreeBSD-like" license. The source may be downloaded from "`http://plas.fit.qut.edu.au/perwapi/`"

As of the date of this document the code has facilities for reading and writing *PE*-files compatible with the latest (beta-2) release of the "*Whidbey*" version of *.NET*, that is, the Visual Studio 2005 framework. An invocation option allows earlier versions of the framework to be targetted[1].

1.1 Using PERWAPI

The *PERWAPI* interface provides a number of calls that construct objects corresponding to meaningful entities in a *.NET* program. As will be seen later, there are object types corresponding to types, fields, methods and instructions, to name just some of the members of the object menagerie.

When the *PERWAPI* component is used to *write* a *PE*-file, the process begins by creating an object that will represent the *PE*-file itself. Objects are created corresponding to the various classes and other types that the file will declare, and these are linked to the *PE*-file object. Other objects will be created corresponding to the methods of each of these types. These method objects will be linked to the class objects that declare them. Instruction objects are defined and linked to the method objects, defining the code of the method. Finally, when all of the objects have been created and logically linked together a single method call, "`PeFile::WritePEFile()`", writes out the *PE*-file. This step is often called "baking" the file (after the ingredients have been added one by one and mixed together).

[1] Current users of *PEAPI* will eventually have to make the transition to *PERWAPI*, as *PEAPI* does not support the *Whidbey* framework and is no longer maintained.

Internally *PERWAPI* takes the abstract representation of the program embodied in the structure of linked objects, encodes this into the tables of the *PE*-file format, and writes the tables to the output stream. The output stream does not necessarily need to be a file in the file system, but may be a memory stream ready to be loaded, as in the case of dynamic code generation.

When the *PERWAPI* component is used to *read* a *PE*-file, one of three static method calls is used, depending on the application.

The method "`PeFile::ReadPEFile(`*`filename`*`)`", reads the specified file and produces an abstract representation of the assembly as a *PERWAPI.PEFile* object. This representation is isomorphic to the data structure from which the *PE*-file was originally baked. This is the call that would be used in an application that wished to read in a *PE*-file, do some processing, then write out a new *PE*-file.

The method "`PeFile::ReadPublicClasses(`*`filename`*`)`", reads the specified file and produces an abstract representation of the assembly as a *PERWAPI.PEFile* object. This representation only includes those type and member objects that correspond to public (or at least *visible*) entities of the target *PE*-file. This is the method that would be used to browse the interface to a *PE*-file. However, this is not the ideal method for a compiler to use to link to an external *PE*-file. When a compiler needs to *reference* a class that is defined in another *PE*-file, it needs to create a *ClassRef* object to do this. The structure created by *ReadPublicClasses* will instead contain *ClassDef* objects corresponding to the classes that are *defined* in the *PE*-file that it reads.

The method "`PeFile::ReadExportedInterface(`*`filename`*`)`", reads the specified file and produces an abstract representation of the assembly as a *PERWAPI.-ResolutionScope* object. This representation only includes those type and member objects that correspond to public (or at least *visible*) entities of the target *PE*-file. The result of this call differs from the result of a call to the previous method in that the objects that are created are reference objects corresponding to the definitions in the *PE*-file that was read. This method is thus the one that is most useful to a compiler wishing to construct references to the facilities of other *PE*-files.

1.2 Data Structure Overview

The data structures defined by *PERWAPI* correspond to the various entities of the *.NET* Common Type System (*CTS*). Although there are about 100 public classes defined by the component, understanding the overall structure of the representation requires familiarity with a rather smaller number.

First, however, it should be noted that some of the classes of the component are artifacts of the particular way in which *PE*-files represent metadata. The representation is based on up to 44 tables, very much in the style of a data-base. These tables reference each other, and also hold indexes into a "blob-heap" for unstructured data and a "string-heap" for string data. *Users* of *PERWAPI* never have to deal with either the tables or the heaps, they only deal with the object types that correspond to the entries in the tables. The best policy for a user is to note that almost all the interesting object types derive, directly or indirectly, from an abstract class called *MetaDataElement*, which in turn derives from *TableRow*. Having made this mental note, you may relax, since you only have to deal with the concrete objects that inherit from these types.

Of the 19 public classes that derive from *MetaDataElement* the most important and interesting are —

* *ResolutionScope*. This derives all of the objects that specify the outer level scope

How *gpcp* uses *PERWAPI*

The Gardens Point Component Pascal (*gpcp*) compiler uses both the reading and writing facilities of *PERWAPI*.

The *gpcp* compiler has an option to write *PE*-files directly, rather than emitting textual assembly language and invoking the *IL*-assembler. Early versions of *gpcp* used *PEAPI* for this purpose, but current versions use *PERWAPI*. The compiler is written in *Component Pascal*, and recompiles itself via *PERWAPI* as a test of correctness. The direct-to-*PE*-file option is rather faster than compiling via textual intermediate language.

Whenever *gpcp* compiles a program it emits a metadata "*symbol file*" which specifies the public interface of the compiled module. When a module is compiled that uses the facilities of some separate module, the compiler reads the metadata of the imported module so as to ensure type correctness. It follows that in order for *Component Pascal* programs to access the facilities of any *.NET* base library it is necessary to produce a symbol file corresponding to the public interface of that library. One of the *gpcp* tools, "*PeToCps*", performs this conversion. The program uses the *reader* part of *PERWAPI* to read *PE*-files. The program builds a *Component Pascal* abstract syntax representation corresponding to the exported types of the library. The standard symbol file emitter of *gpcp* then emits the symbol file.

Of course *PERWAPI*, being written in *C#*, is itself is a foreign language library so far as *gpcp* is concerned. Thus, in a conventional bootstrap maneuver, *PeToCps* needed to be run over `PERWAPI.dll` to produce a corresponding symbol file so that *gpcp* could compile *PeToCps*.

within which names are qualified. These objects correspond to all the things that appear within the square brackets at the start of fully qualified *IL* names such as *[mscorlib]System.Exception*. The derived types include *Assembly*, *Module*, *AssemblyRef* and so on.

* *Type*. This derives all of the objects that specify types in the *CTS*. The derived types include reference classes ("`class`" in *C#*), value classes ("`struct`" in *C#*), arrays, primitive types, generic formal types, and so on. Separate object types are used to represent classes that are *defined* in the assembly being constructed, and classes that are defined in external assemblies and *referenced* by the assembly under construction.

* *Member*. This derives all of the object types that define and make reference to the fields and methods of *CTS* classes.

* *Feature*. This derives all of the object types that represent properties and events.

The details of these type hierarchies will be considered in Section 2.

1.3 Performance

Compilers using *PERWAPI* to write their output are fast. As a rough rule of thumb, *gpcp* can write out a *PE*-file using *PERWAPI* in the same time as it writes out a textual-*IL* file. Compared to the conventional approach, this implies a time saving equal to the entire time spent in creating a new process for `ilasm`, reading the *IL* text and creating the output file.

Writing textual IL files	2.9
Writing IL, then invoking `ilasm`	7.0
Writing PE files using *PERWAPI*	3.0

Figure 1: Time in seconds to run **CPMake /all** on *gpcp* source

Figure 1 lists the time taken to compile the roughly 50k lines of *Component Pascal* in the 46 files of the *gpcp* source. The time in seconds is for a single processor 2004-vintage AMD Athlon 1800 machine with 256k of memory. The final row thus corresponds to a compilation speed about one million lines per minute.

Figure 2 lists the compilation times for the three of the largest modules of *gpcp*. In each case the compilation time of each module compiled on it own is about 800mSec. Most of this time is the JIT penalty. The times quoted in the Figure are the result of a multi module compilation, with the measured module being third on the argument list. In this way the module named in the first argument suffers the penalty, and the measured module finds the code already JIT-ed.

Output	Mod-1	Mod-2	Mod-3
Writing textual *IL* files	125	109	109
Writing *IL*, then invoking `ilasm`	313	328	235
Writing *PE*-files using *PERWAPI*	157	125	109

Figure 2: Time to compile each of three large modules, milli-seconds

1.4 Limitations

As of the current release, *PERWAPI* does not emit program debug (*PDB*) files. It is expected that these facilities will be added in the next release, based on calls to the Microsoft unmanaged code API.

Another caution that bears prominence is that the interface to *PERWAPI* is intended to conform to the Common Language Specification (*CLS*). This is done so that tools written in any language that is a *CLS* consumer can access the *API*. However there is a small paradox involved, since it is necessary to deal with (for example) methods that return unsigned values that are not part of the *CLS*. This means that there is some awkwardness in places. Consider, for example, the class *UIntConst*, which constructs and reads unsigned whole-number constants. As expected, there is a method to extract the *value* that the constant represents. Here is the signature of the method: it returns the *unsigned* value that the constant descriptor represents, but as a *signed* long —

public long `GetULongAsLong()`

The signature is *CLS* compliant, but requires the user to attach special semantics to any return value that appears to be a negative *INT64*.

2 Data Structure Details

The containment hierarchy of entities in an assembly is roughly thus —

* Each *PE*-file contains exactly one *Module*
* *PE*-files that declare an *Assembly* contain an assembly manifest
* *Module*s define zero or more *ClassDef*s
* *ClassDef*s contain *FieldDef*s and *MethodDef*s
* *MethodDef*s contain a *CILInstructions* buffer
* *CILInstructions* reference *MethodDef*s and *MethodRef*s
* *CILInstructions* reference *FieldDef*s and *FieldRef*s
* *MethodRef*s and *FieldRef*s are contained in *ClassRef*s
* *ClassRef*s belong to *AssemblyRef*s and *ModuleRef*s

It is the task of *PERWAPI* to allow for the definition of each of these kinds of entity, and their association with their enclosing container.

2.1 Resolution Scope

Resolution scopes are used to define the context within which class names are bound. It is useful to remember that within the *.NET* framework class names are "dotted names". Many languages treat the final identifier of the dotted name, the *simple class identifier*, and the prefix, the *name-space* name, as separate entities. However, within the framework the name is the whole, qualified name. This complete name is resolved within a particular resolution scope corresponding to an assembly or a module of the program.

ResolutionScope is an abstract class of *PERWAPI*, which derives two other abstract classes: *DefiningScope* and *ReferenceScope*. The *DefiningScope* type derives the *Assembly* and *Module* classes, with the very important class *PEFile* being derived from *Module*. A *PEFile* object is the root of every representation of a *PE*-file.

The *ReferenceScope* type derives the *AssemblyRef* and *ModuleRef* classes. Objects of these classes are used to represent imported assemblies and modules that need to be referenced by the module under construction.

The complete class hierarchy for these classes is shown in Figure 3. In all of these class hierarchy diagrams black rectangles denote abstract classes. Extensible classes are shown unshaded, while sealed classes are shown lightly shaded.

All resolution scopes contain a string name, and a list of classes belonging to that scope. There are public methods for adding and deleting classes from the scope, and for accessing the name. The *Assembly* class is used for *defining* assemblies, and contains a significant amount of additional information. The four-part version number, public key, culture and hash algorithm may be specified at object creation time, or added later with the method *AddAssemblyInfo*. Declarative security information is also stored in *Assembly* objects.

PEFile objects

When a *PEFile* object is created, one of two constructors may be called. The first of these specifies only the filename, and constructs a *PEFile* object with no associated *Assembly* object. The second constructor takes two string arguments. The first is the

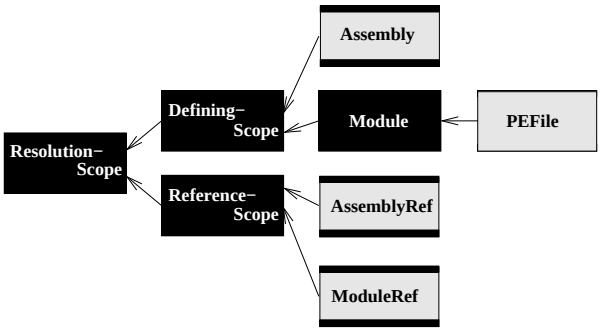

Figure 3: Class hierarchy for the *resolution scope* descriptors

filename; the second is the assembly name. This constructor creates a *PEFile* object with an optional *Assembly* object. This object is accessed by the *GetAssembly* method.

public `PEFile(`**string** `fileName)`

public `PEFile(`**string** `fileName,` **string** `assemblyName)`

Many compilers by default use the same string for both filename prefix and assembly name. Note that specifying a "filename" does not constrain the *PE*-file to be emitted to the file system, as both the output stream and the destination directory may be changed with methods of the class. For both constructors the default value of the "*isDLL*" flag is set depending on whether or not the filename argument ends in ".dll". If the flag is false an "exe" file will be created.

There are a large number of methods to access the data of the object, to add and delete classes and to change the "*isDLL*" flag. There are also the three static methods used for *reading PE*-files, mentioned in Section 1.1.

public static `PEFile ReadPEFile(`**string** `fileName)`

public static `PEFile ReadPublicClasses(`**string** `fileName)`

public static `ResolutionScope ReadExportedInterface(`**string** `fileName)`

An important attribute of a *PE*-file is set by the *SetSubSystem* method of *PEFile*. The enumeration value in the argument to the call specifies whether the *PE*-file (presumed to be an "exe" file) will be a console or GUI application on various platforms. The default value is for a Windows, console application.

Assembly and Module refs

AssemblyRef and *ModuleRef* objects derive from *ReferenceScope*. Objects of these types are automatically created when the static *ReadExportedInterface* method is called. Alternatively, if there is an accessible *Assembly* or *Module* object, a corresponding reference object may be created by a call to the appropriate *MakeExternAssembly* or *MakeExternModule* method.

2.2 Type Descriptors

All *CTS* types in *PERWAPI* are represented in the tree by objects that derive from the abstract class *Type*. The rich hierarchy of derived classes is shown in Figure 4. There are two concrete classes that directly derive from *Type*. These are *GenericParam*

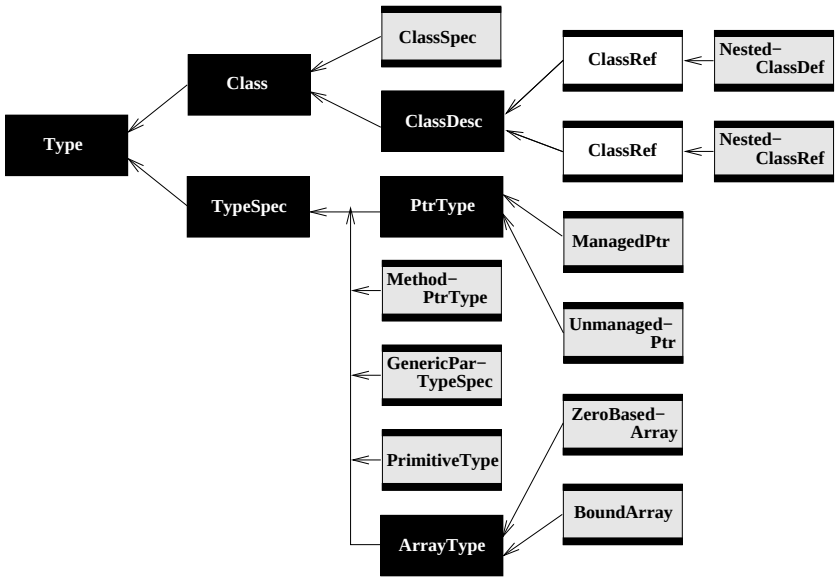

Figure 4: Class hierarchy for the *PERWAPI Type* descriptors

and *CustomModifiedType*. *GenericParam* objects represent the type-formal parameters in generic types and methods. The ways in which *PERWAPI* accesses the generic facilities of the *Whidbey* release are discussed in Section 2.6

Most of the objects derived from *Type* appear in the type hierarchy under the two abstract subclasses: *Class* and *TypeSpec*.

The Class subtree

Class is another abstract type, denoting any kind of class in the type system. In particular this base class has a field holding an array of generic parameter types. Conventional classes will leave this array empty, while generic classes will have one or more array elements. Abstract class *Class* has three directly derived extensions. The first extension, *ClassDesc*, is a abstract "convenience" class that acts as the base class for the *ClassDef* and *ClassRef* classes that define and reference classes, respectively. *ClassDesc* adds no data, but holds the accessor methods for setting and getting the generic parameters of the classes.

ClassSpec objects denote instantiations of generic classes. They contain two relevant pieces of information. The *genClass* field holds a reference to the (necessarily generic) class that the object denotes an instantiation of. Thus, an object denoting, say, *Stack*`<int>` would hold a reference to a class *Stack*`<T>` where *T* is a type formal. The fields that *ClassSpec* inherits from *Class* contain information about the instance itself. In particular, the generic parameter array holds an array of *actual* types that are

substituted for the generic parameter types listed in the referent of the *genClass* field. The class defines public methods to fetch the generic class, and to fetch the generic (actual) arguments.

ClassDef and ClassRef

*ClassDef*s are always associated with the current *PEFile* object or, in the case of a *NestedClassDef*, with another *ClassDef* within which that class is nested. *ClassRef*s are associated with an *AssemblyRef* or with a *ModuleRef*. In the case of a *NestedClassRef* the object parent is another *ClassRef*.

Class refs and defs each contain lists of methods and fields, but the information known for a *ClassDef* is necessarily greater. For a *ClassDef* there is a reference to the super-class, a list of interfaces that the class implements, a list of security demands, field layout declarations, and a *methodImpl* list. None of this information is needed, or included in a *ClassRef*. For example, it is necessary for a *ClassDef* to declare if the field layout of the class is to be different from the default arrangement. However for the *user* of a class, the fields are always referred to by name, so the layout strategy is neither needed nor available.

Both types of descriptors contain a list of generic parameters, usually empty, that is inherited from the *Class* descriptor parent.

The TypeSpec subtree

The *TypeSpec* descriptors denote all of the types that are not simple classes. The type has three concrete subtypes, and two abstract subtypes.

Array is an abstract class, with two concrete subtypes. Zero-based arrays are the array types that are built-in to the framework, while bound arrays are managed by the *System.Array* facilities of *mscorlib*. When array descriptor objects are created, the type descriptor object for the element type is specified.

PtrType is an abstract class, with two concrete subtypes. *ManagedPointers* are the pointers that occur transiently on the evaluation stack of the *.NET* abstract machine. They are pointers to locations that may be data under the control of the garbage collector. Managed pointers are reported to the *JIT* so that if a memory datum is moved while such a value is live the pointer value can be adjusted to point to the moved location. Managed pointer objects are the type descriptors that are associated with parameter values that are passed by reference.

UnmanagedPointers represent address values that are not reported to the garbage collector. Class fields may be declared to be of such a type, but the resulting code will always be unverifiable when that is done. Such types are most usually needed when managed code processes addresses of data allocated in unmanaged (native) code.

The three concrete classes that directly derive from *TypeSpec* are *PrimitiveType*, *MethPtrType* and *GenericParTypeSpec*. Primitive types correspond to the types such as *int, double, string* and *object* that are built-in to the Common Language Infrastructure (*CLI*). These types have built-in type descriptors that are created by *PERWAPI*, with each built-in type corresponding to a named static constant value of the type.

MethPtrType is a type that corresponds to “function pointers” in *ANSI C*. Such values are used in programs that access native code. Objects of these types are most usually needed when code performs “platform-invoke” calls to native code on the execution platform. In general the use of such types makes the resulting code unverifiable.

2.3 Members and Features

Class objects have *Member* and *Feature* objects associated with them. The class hierarchy for the various descriptor classes is shown in Figure 5. Members include methods

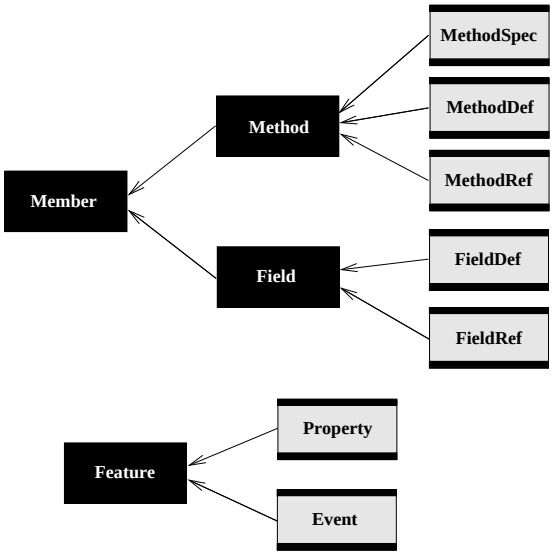

Figure 5: Class hierarchy for the *Members* and *Features* of classes

and fields. *MethodDef* objects are associated with class definitions, and in the case of managed *CIL* methods, will have instruction buffers added to them after their creation. *MethodRef* objects are associated with class references and have various associated attributes, but no code. *FieldDef* and *FieldRef* objects are associated with class definitions and class references respectively.

There are two kinds of *Feature*, both of which are associated with class definitions only. *ClassDef*s may have event or property objects added to them. These features associate particular specified methods with the semantics of the feature. In the case of *Event* features, the conventional methods are the *add_** and *remove_** methods. For *Property* features, the methods include the optional "getter" and "setter" methods.

Method structure

Methods have a variety of attributes associated with them. In a *PE*-file *reader* application *PERWAPI* will produce the method descriptors fully attributed. However, in a *PE*-file *writer* application, the application code will invoke one of the constructors. The attributes may be attached at the time of object creation, or may be attached to the method descriptors *after* the descriptor has been created.

The relevant attributes are —

* *CallConv* attributes, such as *Instance* and *Vararg*
* *ImplAttr* attributes, such as *IL* and *Synchronised*[2]

[2] As a kindness to speakers of American English, the enumeration member "*Synchronized*" has the same value.

* *MethAttr* attributes, including the accessibility attributes such as *Private* and *Public*, and other semantic markers such as *Static, Virtual* and *Final*

All *Method* objects have associated call conventions. However the implementation attributes and method attributes only apply to *MethodDef*s.

When a *Method* object is created and added to a *Class*, a minimum of three arguments need to be supplied to the constructor call. These are the method name, the return type descriptor, and formal argument information. In the case of *MethodRef*s the formal argument information is simply an array of *Type* descriptors. For *MethodDef*s, by contrast, an array of objects of *Param* class is passed. Each element of the formal parameter descriptor array specifies the argument type, as is the case for *MethodRef* creation. However, in the *MethodDef* case the *name* of the formal argument, and some parameter attribute information needs to be included as well.

In the *Whidbey* version of the *CLR* methods may be generic, and all method descriptors have an array of generic parameters. For non-generic methods this array will be empty. Instantiations of generic methods are represented by *MethodSpec* objects. As seen in Figure 5 *MethodSpec* derives from *Method*. Each such object contains an array of the type arguments of the instantiation, as discussed further in Section 2.6.

2.4 MethodDefs and Code Buffers

If a *MethodDef* has the *IL* attribute in its implementation attributes, and does not have the *Abstract* attribute in its method attributes, then it requires a code buffer into which instructions may be placed. The code buffer is of the type *CILInstructions*. Instructions, labels and other markers are added to this buffer in sequence. The type that represents instructions internally in the buffer is not exposed to the *API*.

The instruction enumerations

Instructions are separated into a small number of categories, according to the argument types that the insertion method requires. Each instruction category defines an enumeration for the permitted instruction codes.

Instructions without argument take their operands from the abstract machine evaluation stack. The instructions are defined by the *Op* enumeration. This includes by far the largest number of instruction op-codes.

Instructions that take a label as argument take an operation code from the *BranchOp* enumeration, and include all of the branch operations. The label argument is supplied as an object of the class *CILLabel*. Example instructions are “`beq`”, “`brfalse`” and “`leave`”.

Instructions that load and store fields and field addresses take an operation code from the *FieldOp* enumeration. They take a second argument that is an object of *Field* class. Referring to Figure 5, it may be seen that this object may be either a *FieldDef* or a *FieldRef*.

Instructions that take an integer argument take an operation code from the *IntOp* enumeration. These instructions include constant loading, and the loading and storing of formal arguments and locals. Example instructions are “`ldc.i4`”, “`ldarg`” and “`stloc`”.

Instructions that take a *Method* object as operand take their operation code from the *MethodOp* enumeration. The operand may be either a method reference or definition. These instructions include the “`call`”, “`ldftn`” and “`newobj`” instructions.

Instructions that take a *Type* object as argument take an opcode from the *TypeOp* enumeration. Examples of such instructions include "`castclass`", and "`newarr`".

Finally, there are three special cases that do not fit into any of the previous categories. These are the instructions that load constants of **long, float** and **double** type. The instructions are "`ldc.i8`", "`ldc.r4`" and "`ldc.r8`". Each takes an argument of the specified type.

Labels and branches

Labels in the code buffer are represented by objects of the *CILLabel* class. The creation of these labels, and the placing of the labels in the buffer are separate operations. In the case of a forward branch the label object must be constructed first, and then passed as the argument of a *BranchOp* instruction. At some later stage the label will be added to the buffer to mark the target position.

For backward jumps the order is reversed. The newly constructed label object will be placed in the buffer immediately following construction. The branch instructions that reference the label will be added to the buffer at some later point.

In the *CLR* all branch instructions appear in two versions. One, with a short displacement, may be used for branch offsets that fit in a single byte. The long-displacement version takes up more space in the file. When textual-*CIL* is emitted, it is not easily possible to determine whether the short or long version is required. *PERWAPI* solves this problem by defining only the long-displacement versions in the enumeration. In the implementation it is always the short-displacement version that is initially added to the buffer. During semantic processing *PERWAPI* computes the actual displacements. If a displacement is too large to fit in a single byte the short branch instruction is replaced by the corresponding long version, and all displacements are recomputed.

Note that the labels in the code buffers are an abstraction that is meaningful to the program creating the *PE*-file, but have no concrete existence in the *PE*-file itself. During the creation of the metadata tables, the label positions are transformed into displacements from the locations of all the branch instructions that reference them.

Structured exception blocks

There are two mechanisms within *PERWAPI* to define structured exception handling blocks. The start and end of the blocks may be marked in the instruction buffers. This is probably the simpler mechanism to use, and corresponds to the preferred method of marking such blocks in textual-*CIL*. The alternative mechanism is to mark the beginning and ending of each block with ordinary labels, and declare the boundaries to each region by passing the boundary labels as arguments to the block constructors.

Because **try** and **catch** blocks may be nested, *PERWAPI* keeps a stack of blocks that have been opened but not yet closed. When the end of a block is marked, the limits of the block's extent are finally known, and the block is popped from the stack. If the completed block is declared to be a **try** block a reference to the popped block is returned to the caller, so that later exception handling blocks may be logically associated with the code region contained within the block.

2.5 Constants of Various Kinds

There are 14 different classes that represent constants in *PE*-files. These are —

* *BoolConst : BlobConstant* — true or false values
* *CharConst : BlobConstant* — a unicode character value
* *NullRefConst : BlobConstant* — the “null” value
* *ClassTypeConst : BlobConstant* — a runtime type descriptor
* *BoxedSimpleConst : BlobConstant* — a boxed *SimpleConst*
* *AddressConstant : DataConstant* — an address constant
* *ByteArrConst : DataConstant* — an array of constant bytes
* *RepeatedConst : DataConstant* — a repeated *DataConstant*
* *StringConst : DataConstant* — a string value
* *IntConst : SimpleConst* — a signed integer value
* *UIntConst : SimpleConst* — an unsigned integer value
* *FloatConst : SimpleConst* — a float value
* *DoubleConst : SimpleConst* — a double value

All of these have the expected constructors and value-access methods. *BoolConst* objects can be created with only two different values, while for *NullRefConst* no value is specified in the constructor since there is only one way of being “`null`”.

RepeatedConst is interesting, since it provides a compact way of specifying multiple occurrences of the same constant. Watch out however for usages of *StringConst*. It is possible to construct an object of this type by specifying either a *CLI* string, or an array of bytes. If it is unknown which form a particular constant contains, then user code may need to check so as to know whether to call *GetString* or *GetStringBytes*.

2.6 Representation of Generics

All type descriptors in *PERWAPI* that are derived from *Class* are able to be declared as generic. The super-type *Class* declares an *ArrayList* of generic parameters. This means that both nested and non-nested *ClassDef* and *ClassRef* types may be generic. In the *CTS* “classes” are a broader category than “class” in *C#*. Reference classes, value classes and delegates are all “classes” in the *CLI*, and all may be declared to be generic.

Similarly, all method descriptors are able to be declared as generic. The super-type *Method* declares an *ArrayList* of generic parameters. Both *MethodDef* and *MethodRef* objects may be generic.

The most obvious application of generics involves the use of generic classes. However, the use of generic delegates also provides a useful mechanism. For example, a number of methods in namespace *System.Collections.Generic* take arguments that are delegate values of type —

```
delegate bool Predicate<T>(T elem);
```

from the *System* namespace. *Predicate*`<T>` is thus a generic delegate type. For a particular instantiation of the generic delegate, *Predicate*`<`*Foo*`>` say, a delegate object instance may be created encapsulating any method taking a *Foo* and returning a Boolean.

Generics at the IL-level

There are a few aspects of generic representation in *IL* that need to be understood. First, internally all generic classes have their simple names "mangled" so that the names are unique, even in the presence of overloading on arity. A class, known to the *C#* programmer as "`IComparable<T>`" will have the *IL*-name "`IComparable‘1<T>`". The number at the end of the identifier, after the "back-tick" character encodes the generic arity. Thus the *internal* names of generic classes are not overloaded, and "`IComparable`" and "`IComparable‘1`" are immediately distinguished.

It is a matter of choice for each compiler writer whether the mangled names or the simple names are used in the internal representation of that compiler. If the simple names are used, then resolution of overloading on arity must be implemented, and the names will need to be mangled when the *PERWAPI* objects are constructed. If the mangled names are used internally, then the names need to be "de-mangled" when generating human-readable messages.

The other significant aspect of *IL* representation is that references to type formals use a positional notation. Thus references to the *n*-th generic formal type of the enclosing class will use the notation `!`*N* where *N* is the index of the formal.

Code may occur inside a generic method that is nested inside a generic class. Thus it is necessary to distinguish between the *n*-th class formal type and the *n*-th *method* formal type. For method formal types, the *n*-th generic formal type of the enclosing method uses the notation "`!!`*N*" where *N* is the index of the formal.

When generic nested classes are defined, the generic parameters of the enclosing class are accessible. In this case the parameter indices of the nested class sequentially follow those of the enclosing class. Here is an example —

```
class Foo<A,B> {
   class Bar<C,D> {
      // code from here will translate to IL with ...
      // A, B, C, D denoted !0,!1,!2,!3 respectively
   }
}
```

The GenericParam class

The generic parameters of methods and classes are represented by *GenericParam* objects. Generic parameters derive from *Type*, as seen in Figure 4. Each generic parameter contains an index (a positional marker in the list), a set of constraint flags, and a list of constraining types. For the generation of code, as noted above, the positional index is the critical attribute. There is a subtle trap involved in carelessly trying to bind these indices back to the parameter names, see the sidebox on page 14.

The list of constraining types specifies the contracts that every instantiation of the class or method must fulfill. This provides an important technique when writing generic code. The only operations that may be applied to variables of the parametric types are those that must be available for *every* possible instantiation. Thus, constraining a generic parameter makes all of the operations of the constraining type applicable to variables of the parametric type. Very often the constraining types are interface types that every instantiation must implement.

Concrete constraints on the constraint-list will always be *classes*. However it is also possible that a constraint might be another of the formal parameters. Thus the methods for setting and getting elements from the constraint list deal with *Type* rather than *Class*, since *Type* is the closest ancestor of *ClassDef, ClassRef* and *GenericParam*.

There are also non-class constraints on generic parameters that are specified in the constraint flag value. This value is may specify one or more of *NonVariant, Covariant, Contravariant, ReferenceType, ValueType, RequireDefaultCtor*. Only one of the first three may be specified, and in *C#* all generic parameters are non-variant. The final flag specifies that every instantiation of the type must have a public no-arg constructor. All value classes automatically have such a constructor, so if the *ValueType* flag is set, so will be the *RequireDefaultCtor* flag.

Watch out for this trap!

When *PERWAPI* reads a *PE*-file most references to generic parameter types will be already resolved to the parameter definition. In such cases both the name and the index of the parameter will be available. In other cases the reference will only know the parameter index.

References to generic parameters always occur in some known context. Thus, when an application meets a reference in the "`!N`" form it can always use the context to bind the reference to some named *GenericParam* object. However, if this is done carelessly then there is a subtle trap.

ClassSpec objects denote instantiations of generic types, and may contain used occurrences of generic parameters as the instantiating type. Further, in some *PE*-files references to such *ClassSpec* objects may be shared between occurrences in different contexts. For example, a reference to an instantiation *ICollection*`<!0>` may occur in the "implements" list of two classes *Ilist*`<T>` and *Dictionary$KeyCollection*`<K,V>`. In one context "`!0`" binds to "`T`", in the other to "`K`".

It follows that for those (rare) applications in which the index-form of the generic parameter needs to be bound to a named generic parameter object, it cannot be presumed that every reference to the *ClassSpec* object will occur in the same context.

3 The Call Interface

This Section discusses the call interface of *PERWAPI*. Many of the key methods of the *API* are discussed here, but coverage is not complete. In order to use the interface additional documentation is needed. This additional information may be found in the source of the component, or in the hypertext documentation derived from it. The relevant html files are part of the distribution package.

3.1 Reading PE-files

Used as a file-reader, the component begins by invoking one or other of the static methods whose signatures were shown on page 6. One or other of these methods is invoked, the choice depending on whether all metadata is needed or just information about the public metadata.

Having obtained a representation of the file, the methods of the *API* may be used to navigate the tree rooted in the *PEFile* object. For example, every *PE*-file has a list of classes that it defines. A corresponding array of *ClassDef* objects may be obtained by a call to the instance method —

```
public ClassDef[] GetClasses()
```

or the *ClassDef* object with a particular name is returned by the instance method —

```
public ClassDef GetClass(stringclsNam)
```

In actual fact both of these methods are inherited from the *Module* class.

Similarly, given a *ClassDef* object all of the methods that it defines may be accessed using methods such as —

```
public MethodDef[] GetMethods()
```

or to access all of the methods with a particular simple name —

```
public MethodDef[] GetMethods(stringmthNam)
```

If it is known that there is only a single method with a particular name, that is, there is no overloading of method names, then the simple method selector —

```
public MethodDef GetMethod(stringmthNam)
```

may be used. There is yet another variant that takes both the name of the method and the array of parameter types. All of these are instance methods that are dispatched on the *PEFile* object.

Other methods of the *ClassDef* type return the fields of the class, the events, the super-type, the implemented interfaces, and so on. Similar methods of the *MethodDef* type return the attributes of the method, including the instruction lists.

3.2 Round-trip Example

When *PERWAPI* reads a *PE*-file, it represents the data of the file as a tree of objects rooted in a *PEFile* object. When *PERWAPI* is used as a file writer a tree is constructed by the application program using exactly the same object types used by the reader.

It is thus possible to read a *PE*-file using the *ReadPEFile* method, perform some kind of transformation on the resulting data structure, then write the tree out using the *WritePEFile* method. This sequence is called “round-tripping the *PE*-file”.

Optimizing branches

When a file is round-tripped through *PERWAPI*, the meaning of the program should be the same, but the bytes of the file may not be identical. For starters, with the current version of the component all debug data will be missing from the output file. More positively, the output file will always have optimized branch instructions. *PE*-files that are assembled using `ilasm`, by constrast, will always contain the branch instructions specified in the input text. Since it is difficult for a writer of textual *IL* to compute the offset distances, *IL* files generally use the long form for all branch instructions. *PERWAPI*, on the other hand, treats branch instructions as *logical branches*, computes the offset in the code buffer, and emits the shortest legal branch instruction into the output file[3].

[3] You may wonder how the reader side of *PERWAPI* deals with branch instructions, since the *PE*-file does not contain labels. During creation of the code buffers all branch destinations are flagged and a *CILLabel* is inserted at each such position in the buffer.

Optimizing load-integer instructions

Most production-quality compilers will optimize the "load-constant-integer" family of instructions to use the smallest instruction that will denote any particular value. Values from -1 to 8 may be encoded in a single byte, values between -128 and 127 use a two byte form, while all others use up five bytes in the *PE*-file. *PERWAPI* allow all of these forms to be used, but also has a method, *PushInt*, that selects the shortest instruction for a particular value (See discussion on page 24). Figure 6 is a simple round-trip program that reads a *PE*-file and then writes it out with all instructions of the "`ldc`" family optimized.

```
using System;
using PERWAPI;
   ...
public static void Process(string fileNm) {                    1
  PEFile peFl = PEFile.ReadPEFile(fileNm);                      2
  ClassDef[] clss = peFl.GetClasses();                          3
  for (int ix = 0; ix < clss.Length; ix++) {                    4
    MethodDef[] mths = clss[ix].GetMethods();                   5
    for (int jx = 0; jx < mths.Length; jx++) {                  6
      CILInstructions code = mths[jx].GetCodeBuffer();          7
      CILInstruction[] list = code.GetInstructions();           8
      for (int kx = 0; kx < list.Length; kx++) {                9
        CILInstruction inst = list[kx];                        10
        if (inst is IntInstr) {                                11
          IntInstr inx = (IntInstr)inst;                       12
          int iOp = (int)inx.GetOp();                          13
          int val = inx.GetInt();                              14
          if (iOp == (int)IntOp.ldc_i4 ||                      15
              iOp == (int)IntOp.ldc_i4_s) {                    16
            code.ReplaceInstruction(kx);  // start insert at kx 17
            code.PushInt(val);            // insert best ldc* instr 18
            code.EndInsert();             // rebuild instr list 19
          }                                                    20
        }                                                      21
      }                                                        22
    }                                                          23
    peFl.SetFileName("Mangle." + fileNm);                      24
    peFl.WritePEFile(false);                                   25
  }
}
```

Figure 6: Optimizing the **ldc*** instructions

The *Process* method begins by reading the *PE*-file the name of which came in as the method argument (line 1). The code at line 2 fetches all of the class descriptors in the *PE*-file, and processes them one by one in the **for** loop at line 4. The code at line 5 fetches all of the methods of the current class, and processes these one by one with the **for** loop beginning at line 6.

For each *MethodDef* the instruction buffer is fetched (line 7), and the associated array extracted (line 8). Each instruction in the buffer is tested to see if it is a *IntInstr*

instruction, and if so it is tested further to see if it is either a short or long version of the "`ldc.i4`" family[4]. All the integer constant load instructions that have been detected are rewritten by the code at lines 17 – 19. The *ReplaceInstruction* method deletes the instruction at the given buffer index, replacing it by all of the instructions that are "inserted" between this call and the subsequent call of *EndInsert*. In this case a single instruction is inserted by the *PushInt* method on line 18. The method is passed the constant value that the *GetInt* call at line 14 extracts from the replaced instruction. As described earlier, the *PushInt* call inserts the shortest legal instruction to load an integer of the given value.

Finally, the modified data structure is "baked" and written out to a different file-name. In this simple example the output name is formed by adding the prefix "`Mangle.`" to the original file name.

It should be emphasised that this code is fully functional, but has been stripped of all of the necessary error checking code to handle such things as file-not-found exceptions. There are also a number of other instruction groups for which short versions exist. It would make sense to optimize all of these groups at once in the inner loop of a method like *Process*.

3.3 Writing PE-files

PERWAPI defines a public interface that allows objects of the various classes to be created and associated with each other. For the most part the object creation methods return references to the newly created objects. This facilitates a style of use where the client of the *API* takes responsibility for retaining some state information. An example may make the pattern clearer.

When a *MethodDef* has an instruction buffer added, it is sensible for the client to retain a reference to the buffer. There is a method to *create* the buffer (of *CILInstructions* class) and attach it to the specified *MethodDef*. However, it would be tedious and inefficient to repeatedly call *GetCodeBuffer* on the *MethodDef* to regenerate a reference to the buffer. In practice clients hold a reference to the buffer, and dispatch their instruction-insertion methods on this reference.

Creating a Root Object

The root object of the tree that *PERWAPI* builds is of *PEFile* class. As noted on page 6 there are separate constructor methods that create these objects with or without an associated assembly manifest and *Assembly* object —

```
public PEFile(string fileName)

public PEFile(string fileName,  string assemblyName)
```

The operations that add children to the root object are dispatched directly on the *PEFile* object. In the event that the file defines an *Assembly* object, this may be retrieved by the call —

```
public Assembly GetThisAssembly()
```

[4]Note that for all instructions any embedded "dots" in the names are replaced by lowline characters in the corresponding *C#* identifier.

Declaring Resolution Scopes

The creation of a *PEFile* object implicitly creates a *Module* resolution scope, and perhaps an *Assembly* resolution scope as well. These are *defining* scopes. Other resolution scopes need to be created in order to be able to refer to external modules or assemblies. These will be *reference* scopes.

External assemblies are attached to the root object by calls to the following instance method —

```
public AssemblyRef MakeExternAssembly(string asmName)
```

The argument specifies the name of the external assembly. *PERWAPI* always creates an assembly reference for the system assembly "`mscorlib`". The *MakeExternAssembly* method checks for this particular string, so that explicit calls to get the "`mscorlib`" assembly do not create duplicate descriptors.

External modules are attached to the root object by calls to the following instance method —

```
public ModuleRef MakeExternModule(string modName)
```

The argument specifies the name of the external module.

If a module defines an assembly, and adds one or more external modules, then there is the possibility of "exporting" any public classes that are defined in the external modules. The *AddExternClass* method called on a *ModuleRef* object adds a class, and adds the new *ClassRef* to the export table of the current assembly.

Creating *ClassDef* and *ClassRef* Objects

Creating class definitions

New class descriptors are created by the *AddClass* methods. Calls that are dispatched on the root object, or on other class definition objects create *ClassDef* objects.

The most frequently used methods to create *ClassDef*s are called on the root object. These create class definition descriptors that are attached to the current resolution scope. There are four such methods —

```
public ClassDef AddClass(TypeAttr at, string ns, string nm)
```

The arguments to this method specify the type attributes, the namespace name and the class name. Classes defined by this method will implicitly derive from *System.Object*.

The second of the methods has the signature —

```
public ClassDef AddValueClass(TypeAttr at, string ns, string nm)
```

The arguments to this method specify the type attributes, the namespace name and the class name. In this case the constructed method will implicitly derive from *System.ValueType*.

The third method for creating an un-nested class definition takes a fourth argument of *Class* type. The actual argument is the explicit *ClassRef* or *ClassDef* that will be the super-type of the class being defined —

```
public ClassDef AddClass(TypeAttr at, string ns, string nm, Class sp)
```

Finally, there is a method that adds an existing *ClassDef* object to the current defining scope.

The type-attribute value is a enumerated type that is partly exclusive values and partly bit-values that may be combined by addition. The attributes declare the visibilty of the class. They also declare whether the class is abstract, sealed, and so on, and the layout kind.

If the *AddNestedClass* class creation methods are called on existing *ClassDef* objects then nested classes are defined. The semantics of these methods otherwise mirror those that are called on the root object.

All of the methods that create *ClassDef*s provide for an initial value for the type attributes to be specified. There are a number of other methods that allow additional information to be added to an existing *ClassDef* object. Methods allow the attributes to be modified, or for layout information to be supplied in the case of explicit layout being specified.

It will be noticed that a single super-type is able to be nominated at the time that a *ClassDef* object is created. If such a class implements interfaces, then these need to be added later, using the following method —

```
public void AddImplementedInterface(Class iClass)
```

The argument specifies the interface that the *ClassDef* is to implement. The actual parameter may be either a *ClassRef* or a *ClassDef*, but for semantic correctness must correspond to an interface class.

Class definitions have any generic formal types added or fetched by calls to the following methods —

```
public void SetGenericParams(GenericParam[] genPars)

public GenericParam[] GetGenericParams()

public GenericParam GetGenericParam(int index)
```

These methods are all defined on the class *ClassDesc*, and thus apply both the *ClassDef* and *ClassRef* descriptors.

Creating class references

Class references may be attached to *AssemblyRef* and *ModuleRef* objects. The methods to create these *ClassRef* objects are similar to those that create *ClassDef* objects, except that neither type attribute nor a super-type may be specified.

If a class reference from another module is exported from a module that defines an assembly, then it is possible to export nested classes as well. An exported class reference is created by a call of the *AddExternClass* method dispatched on an external *ModuleRef*.

```
cRef = modRef.AddExternClass(att,nsp,"Outer",fil,false);
```

A subsequent call of *AddNestedClass* will declare a nested class from the same module, and add it to the export table of the current assembly.

```
nRef = cRef.AddNestedClass(att,"Inner");
```

Creating descriptors for value classes

In order to create descriptors for value classes it is usual to call one of the *AddValueClass* methods, rather than *AddClass*. This is because the super-type is set at the time of descriptor creation. However, it is also possible to use the *AddClass* method that takes an explicit super class, and pass in the descriptor for *System.ValueType*. Finally the supertype value is a property that is accessible via the usual *get* and *set* operations.

Creating descriptors for other *Type*s

As well as the types that are declared as *CTS* classes, there are a number of other types that need descriptors. Figure 4 on page 7 represents the various possibilties.

Firstly, if the descriptor of a primitive type is needed no method call is necessary. All of the primitive types have their descriptors exposed as static constants of the *PrimitiveType* class.

Managed and unmanaged pointer types are created by calls to the relevant constructor method. In the managed case the constructor has the following signature —

```
public ManagedPointer(Type baseType)
```

Where *baseType* is the bound type of the pointer.

This constructor is frequently called, even when emitting verifiable code. For example, the *CTS* type of a reference-mode formal parameter of type *argTp* will be "managed pointer to type *argTp*", often denoted as "*argTp&*". The type descriptor of this type would need to be constructed as part of the generation of formal argument type-arrays. If this is the type of the *n*-th argument of a method signature, then the formal argument type descriptor would be constructed by a call such as —

```
arg[n] = new ManagedPointer(argTD);
```

where *argTD* is the type descriptor of the type *argTp*.

Array type descriptors are constructed by calls to the appropriate constructor. For example, the descriptor for a zero-based array of the **int** type would be returned by the call —

```
new ZeroBasedArray(PrimitiveType.Int32)
```

Bound arrays are, as expected, more complicated. If all of the lower array bounds of a multi-dimensional array are zero the following constructor method may be used —

```
public BoundArray(Type elTp, int dims, int[] size)
```

where *elTp* is the type descriptor of the (ultimate) element type, *dims* is the number of dimensions, and *size* is the array of lengths of each dimension. Note that not all sizes need be specified, in the event that an array is desired that is "ragged" in one or more final dimensions. For example, an array declared in *C#* as —

```
new int[4,3,2]
```

would require a call to the *BoundArray* constructor with arguments —

```
new BoundArray(PrimitiveType.Int32, 3, lenA)
```

where *lenA* is the array **int**`[] = {4,3,2}`. The bound array descriptor defines a three dimensional array of size $4 \times 3 \times 2$. On the other hand, the array declared in *C#* as —

```
new int[4,3,]
```

would require a call to the *BoundArray* constructor with arguments —

```
new BoundArray(PrimitiveType.Int32, 3, lenA)
```

where lenA is the array **int**`[] = {4,3}`. In this case the bound array descriptor defines a *two* dimensional array of size 4×3 with elements of type **int**`[]`. This is a "ragged" array, since the lengths in the final dimension may be different for each of the twelve rank-2 elements.

There is a corresponding constructor that takes two integer arrays, specifying lower and upper bounds in each dimension rather than size —

```
public BoundArray(Type elTp, int dims, int[] loIx, int[] hiIx)
```

Adding fields

Fields are declared by means of a number of *AddField* methods. Fields are normally added to classes, but may also be declared outside of classes, as static fields of modules.

Fields are declared associated with a particular structure by calling an *AddField* method on the containing object. Calls on *ClassDef* or *PEFile* objects create *FieldDef* objects. Calls on *ClassRef* or *ModuleRef* objects create *FieldRef* objects.

Calls that create *FieldDef* objects may either supply a string with the name of the field and the type descriptor, or may supply a field attribute value from the *FieldAttr* enumeration as well. *FieldDef* objects may have their initial attribute values modified by use of the *AddFieldAttr* method.

Calls that create *FieldRef* objects need only supply the name and type of the field. A typical method, that creates a new *FieldRef* and adds it to an existing *ClassRef* has the signature —

```
public FieldRef AddField(string name, Type fdTp)
```

Adding features to *ClassDef*s

*ClassDef*s may have *Feature*s associated with them. There are two concrete subclasses of the abstract feature class: *Event* and *Property*.

Figure 7 shows the correspondence between the definition of an event in textual-*CIL* and the *PERWAPI* calls that are made to generate the same effect using *PERWAPI*. In this figure *peFl* is the root object of the compilation, and *Et* is the name of the event class, assumed to be defined elsewhere. On the right-hand-side *EtD* is the type descriptor for the *Et* type, and *V* is the type descriptor *PrimitiveType.Void*. The details of the code for the add and remove methods have been elided on both sides of the Figure.

On the right of the figure, there are a number of local variables that hold references to the various *PERWAPI* objects between their definitions and uses. Thus *fD* is the *FieldDef* of the private backing field *f*, *aD* and *rD* are the *MethodDef* descriptors of the add and remove methods, and so on. In the interests of simplicity (and column width) it has been assumed that the class definition in the Figure is not within a

```
                                    ClassDef cD;
                                    FieldDef fD;
                                    MethodDef aD, rD;
                                    Event eD;
.class public Cls {                 cD = peFl.AddClass("","Cls");
    ...                               ...
  .field private class Et 'f'       fD = cD.AddField("f",EtD);
    ...                               ...
                                    // define arg array for add/remove
                                    Param[] arr = new Param[1];
                                    arr[0] = new Param(0,"",EtD);
                                    string aS = "add_f";
  .method public void               // create 'add' MethodDef in ClassDef
              add_f(class Et){      aD = clsD.AddMethod(aS,V,arr);
    ...                               ...
  } // end method
                                    string rS = "remove_f";
  .method public void               // create 'remove' MethodDef in Class
           remove_f(class Et){      rD = clsD.AddMethod(rS,V,arr);
    ...                               ...
  } // end method
                                    // create Event property
  .event Et 'f' {                   eD = cD.AddEvent("f",EtD);
    .addon instance void            // attach AddOn method to Event
         cls::add_f(class Et)       eD.AddMethod(aD,MethodType.AddOn);
    .removeon instance void         // attach RemoveOn method to Event
      cls::remove_f(class Et)       eD.AddMethod(rD,MethodType.RemoveOn);
  } // end event
} // end class
```

Figure 7: Defining an event in textual-*CIL* (left) and *PERWAPI* (right)

.namespace declaration. This is the origin of the empty string as the first argument in the *AddClass* call. The calls of *AddMethod* dispatched on the event descriptor object in the right column of the figure take their second argument from the *MethodType* enumeration. This type enumerates the fixed roles that feature methods have, such as *AddOn, RemoveOn, Getter, Setter* and so on.

The definition of *Property* features is similar, with *PERWAPI* methods that add the "getter", "setter" and "other" methods to the property, analogous to the way that the *AddOn* method is added to an *Event*.

Creating *MethodDef*s and *MethodRef*s

Method definitions

MethodDefs are created by invoking *AddMethod* on an object of *ClassDef* type, or directly on the root *PEFile* object. In the first case the method is defined as belonging to the specified class, while in the second case the method will belong to the current module, but be outside of any class definition.

There is an option to either specify method attributes at the time of creation, or to add them later. Creating a method with the default attributes requires only three arguments —

```
public MethodDef AddMethod(string name, Type retType, Param[] pars)
```

The version with attributes allows the method attributes, belonging to the *MethAttr* enumeration, and implementation attributes, belonging to the *ImplAttr* enumeration, to be specified.

As noted earlier, *Param* objects specify the name type and mode of the parameters. These are usually created by use of the constructor —

```
public Param(ParamAttr mode, string parName, Type parType)
```

The parameter attribute enumeration specifies *Default, In, Out, Opt*, where any combination of the last three may be specified.

Method references

MethodRefs are created by invoking *AddMethod* on an object of *ClassRef* type, or on an object of *ModuleRef* class. In the first case the method is defined as belonging to the specified class, while in the second case the method will belong to the specified module, but be outside of any class definition.

The signature of the *AddMethod* method is the same in each case —

```
public MethodRef AddMethod(string name, Type retType, Type[] pars)
```

Note carefully that in this case only the parameter types are specified, so the names and attributes of the parameters are unspecified.

For all methods, the calling conventions may be specified by a call to a method *AddCallConv*. However, in the case of "vararg" methods this is only part of the story. As well as specifying that the method has the *Vararg* call convention it is necessary to specify *which* arguments are optional. In the case of *MethodDef*s, this is specified by the *Opt* value in the parameter mode declaration. However, for *MethodRef*s there is no mode information associated with the parameters, so other means are required. In this case a separate method is required, in which the types of the mandatory and optional formal parameters are separately listed —

```
public MethodRef AddVarArgMethod
          (string name, Type retType, Type[] pars, Type[] optPars)
```

Setting attributes

Attributes of methods may be added after the *Method* object has been created. There are three classes of attributes for methods, defined by separate enumerations in the interface.

Method attributes, belonging to the *MethAttr* enumeration specify the accessibility of the method, that is whether the method is private, public, family and so on. This attribute also specifies if the method is static, final or abstract, the overriding behaviour of the method, and whether the method name has special significance to the runtime. The *MethAttr* attributes may only be added to *MethodDef*s.

Implementation attributes, belonging to the *ImplAttr* enumeration specify whether the code is managed or unmanged, and if the method is synchronised. It is also possible to mark a method as *not* available for inlining. The *ImplAttr* attributes may only be added to *MethodDef*s.

Call convention attributes, take values defined in the *CallConv* enumeration. The attribute value can specify any of a wide variety of native call conventions, as well as the *Vararg* case discussed earlier. This attribute is also used to specify that the method is an *Instance* method (and hence expects to be passed a **this** reference), or is an explicit instance method in which the **this** reference appears as "arg0" of a conventional argument list. *CallConv* attributes may be added to any *Method*.

Adding code to *MethodDef*s

Code is added to *MethodDef*s by attaching a code-buffer to the descriptor. This is done by the calling the following method —

```
public CILInstructions CreateCodeBuffer()
```

on a *MethodDef* object.

As noted previously, the method returns a reference to the code buffer, so that the reference may be the receiver of the various calls that add instructions. The buffer is implemented as an expansible array, so that the buffer length will adjust as required to hold additional instructions.

Instructions are added by the various methods discussed in the Section "The instruction enumerations" on page 10. Except for the branch instructions the instruction opcode specifed in the method call will be precisely the instruction placed in the buffer. In the case of the integer instructions, for example, the short or long form of the instruction must be precisely specified. In the case of the branch instructions, the short-branch form of the instruction is always placed in the buffer initially, and is changed to the long-branch form later, if necessary.

Saying what you mean

In the case of the integer instructions there is an alternative interface that off-loads some processing from the caller. Some users may find these facilities convenient. All of the following methods dispatch on an object of *CILInstructions* type.

The method —

```
public void PushInt(int i)
```

loads the specified integer onto the evaluation stack. The method will choose whichever legal instruction is shortest, whether it be one of the single-byte "`ldc.i4.*`" opcodes, the two-byte "`ldc.i4.s`", or the five-byte "`ldc.i4`" instruction.

Similar methods that automatically choose the best integer instruction are *LoadLocal*, *LoadLocalAdr*, *StoreLocal*, *LoadArg*, *LoadArgAdr*, and *StoreArg*.

Branches and labels

Label objects, of class *CILLabel* are allocated by a call to the method —

```
public CILLabel NewLabel()
```

This method returns a reference to the unique label, so that may be used in subsequent branch instructions. The label is placed into the code buffer by a call to the method —

public void `CodeLabel(CILLabel lab)`

The label appears in the code buffer as a marker only, and does not take up any code space in the subsequent *PE*-file.

In the event that a label needs to be allocated and then immediately placed in the buffer at the current position, the two calls of *NewLabel* and *CodeLabel* may be combined into a single call of the method —

public `CILLabel NewCodedLabel()`

Branch instructions are inserted into the table by means of the following method —

public void `Branch(BranchOp inst, CILLabel lab)`

As mentioned above, the declaration of the *BranchOp* enumeration in *PEAPI* only defines the names of the long-displacement branch instructions. Internally, the component always places the corresponding short-displacement instruction in the *PE*-file, unless the displacement is computed as being outside the single-byte range.

Switch statements

The implementation of switch statements require an *array* of label objects to be allocated, one for each separate branch of the switch, including a separate label for the default branch. The "`switch`" of *CIL* takes an array of label objects as argument, in a call to the method —

public void `Switch(CILLabel[] labs)`

The *labs* array has one element per *case* in the switch. Thus, in general, the labels of the allocated array may appear in multiple positions in the *labs* array.

Consider the sample switch statement in Figure 8. Encoding this statement will

```
switch (exp) {
      case 3: case 6: case  9: Foo(); break;
      case 4: case 7: case 10: Bar(); break;
      case 5: case 8: case 11: Fzz(); break;
      default : Bzz();
}
```

Figure 8: Example switch statement

require allocation of an array of four labels for the four branches, plus another label to be used as the destination of the break statements. The array of labels that is passed to the switch instruction, on the other hand, will have nine elements. This follows from the fact that the ordinal of the smallest case is three, and of the largest case is eleven. It is the responsibility of the *PEAPI* client to construct this array, presumably by traversing the *AST* structure representing the switch statement.

Figure 9 shows the textual-*CIL* on the left, and the corresponding *PERWAPI* method

`ldloc.1` // *exp value*	`buf.LoadLocal(1);`
`ldc.i4.3` // *offset*	`buf.PushInt(3);`
`sub`	`buf.Instr(Op.Sub);`
`switch (`	`buf.Switch(table);`
`lb01, lb02, lb03,`	
`lb01, lb02, lb03,`	
`lb01, lb02, lb03)`	
`br lb04` // *goto default*	
`lb01:`	`buf.CodeLabel(lab[1]);`
`call Cls::Foo()`	`buf.MethInstr(MethOp.Call, fooD);`
`br lb05` // *break*	`buf.Branch(BranchOp.Br, xLab);`
`lb02:`	`buf.CodeLabel(lab[2]);`
`call Cls::Bar()`	`buf.MethInstr(MethOp.Call, barD);`
`br lb05` // *break*	`buf.Branch(BranchOp.Br, xLab);`
`lb03:`	`buf.CodeLabel(lab[3]);`
`call Cls::Fzz()`	`buf.MethInstr(MethOp.Call, fzzD);`
`br lb05` // *break*	`buf.Branch(BranchOp.Br, xLab);`
`lb04:` // *default*	`buf.CodeLabel(lab[0]);`
`call Cls::Bzz()`	`buf.MethInstr(MethOp.Call, bzzD);`
`lb05:` // *exit label*	`buf.CodeLabel(xLab);`

Figure 9: Example switch statement in textual-*CIL* (left) and *PERWAPI* calls (right)

calls on the right. In the Figure it has been assumed that the array of four labels is named *lab*, and the index-zero element is used as the default label. It is also assumed that the table of labels is computed into the array *table*. Note in both cases that the table dispatch indexes from zero, so the case value of the first case, three in the example, is subtracted from the selector expression *exp* before the dispatch.

3.4 Structured Exception Handling

Since structured exception handling blocks may be textually nested, *PERWAPI* maintains a stack of currently open exception handling blocks. At the time that a block is entered it is not necessary to specify what kind of block it is to become. Markers for the start of blocks are pushed on the stack by a call to a method —

public void `StartBlock()`

The receiver for this call is the current code buffer object.

When the end of a block is reached in the code buffer, the current position is marked, and a handler block object is created. At this stage it is necessary to specify what kind of block is to be created. The methods to mark the block-ends are named *End*Block*. These methods take different arguments, depending on the block type.

The end of a try block is marked by a call to the method —

public `TryBlock EndTryBlock()`

This method returns a reference to the try-block object associated with the created handler-block object.

The end of a catch block is marked by a call to the method —

public void `EndCatchBlock(Class exc, TryBlock blk)`

The first argument is the class descriptor for the type that the block is intended to catch. Of course this will normally represent a sub-type of *System.Exception*. The second argument is the try-block with which this catch is to be associated. This reference will have been returned by a previous call to *EndTryBlock*. Fault blocks have similar behaviour to catch blocks, except that no filtering on exception type is performed. In this case only the associated try-block is specified —

public void `EndFaultBlock(TryBlock blk)`

The end of a finally block is marked by a call to the method —

public void `EndFinallyBlock(TryBlock blk)`

In this case the only argument is the try-block with which the finally is to be associated.

The end of a filter block is marked by a call to the method —

public void `EndFilterBlock(CILLabel flt, TryBlock blk)`

The code label *flt* is the starting label of the predicate code that controls entry to the handler block. As usual, the associated try-block needs to be specified. The predicate code that starts at label *flt* is responsible for popping the exception object from the evaluation stack, and computing the Boolean value. The predicate code must always end with the "`endfilter`" instruction, with the evaluation stack empty except for the Boolean filter result value. Of course, as usual, the filter block itself must end with a "`leave`" instruction.

4 Linking *PERWAPI* to a Compiler

The most important design decision, when using *PERWAPI* as a component in a compiler, is the relationship between the Abstract Syntax Tree (*AST*) representation of the compiler and *PERWAPI* classes.

If the compiler will only ever use *PERWAPI* to emit *PE*-files, then the design of the *AST* nodes might provide attribute fields holding corresponding *PERWAPI* object references. If this is the case, then the *AST* nodes might also be designed to correspond as closely as possible to the *PERWAPI* types.

None of this is possible if the *AST* design is already fixed, as would be the case if an existing compiler was being modified to use *PERWAPI*. It is also a little more difficult in the case that the compiler is intended to use several different output engines.

The first compiler to use the original *PEAPI* component was Gardens Point Component Pascal (*gpcp*). This compiler will use *PERWAPI* in all future releases. Some of the experience of using *PERWAPI* in this application is discussed in the next section. The section concentrates particularly on the lessons that seem generally applicable.

4.1 GPCP's *PeUtil* Tree-walker

There are, in effect, four backends for *gpcp*. The compiler can produce either textual *CIL* or *PE*-files for the *.NET* platform. It can also produce either textual *Jasmin* assembler or directly create class files for the *JVM* platform. The choice of output format is determined from command-line options.

The functionality of output file creation is factored between the *Tree-walker* modules and the *File-utility* modules. The compiler driver code creates a target-specific *tree-walker* object depending on the command line options, and calls *Emit*() on this object. The target-specific tree-walker then creates a output format specific *file-emitter* object depending on other command line options. The output behaviour is thus specialized by dispatching virtual methods on the target emitter object, which in turn dispatches virtual methods on the file format object. Figure 10 shows the class hierarchy for the tree-walker modules.

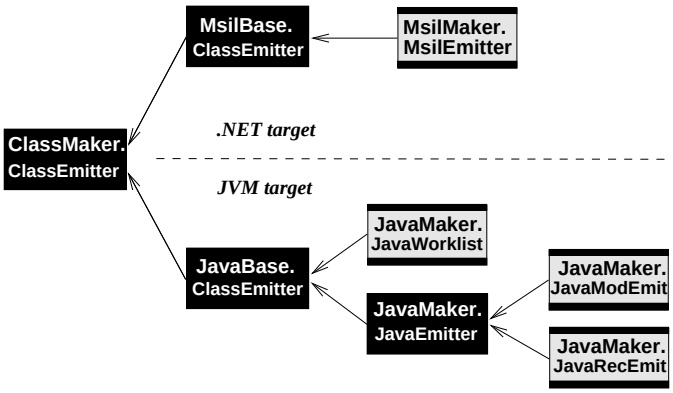

Figure 10: Class hierarchy for the *Tree-walker* classes

In these class hierarchy diagrams, abstract classes are shown in black, while the shaded boxes correspond to sealed classes.

The tree-walker for the *.NET* target is of class *MsilMaker.MsilEmitter*. Only one object of this class is created for each run of the compiler. The situation with the *JVM* target is rather more complicated. An object of the class *JavaMaker.JavaWorklist* is created for each run of the compiler, once the target is known. However, separate objects of class *JavaMaker.JavaModEmitter* or *JavaMaker.JavaRecEmitter* are created for each of the multiple output files arising from each input source file. Our concern in this appendix is only with the *.NET* platform, of course.

The same tree-walker (in file `MsilMaker.cp`) traverses the *AST* to produce either *CIL* or a *PE*-file as output. All actual file *IO* is performed by one of two utility modules *IlasmUtil* and *PeUtil*. The module *MsilUtil* defines an abstract class *MsilFile* with a large number of abstract methods. The abstract class has two concrete extensions *IlasmFile* (in module *IlasmUtil*) and *PeFile* (in module *PeUtil*). The compiler creates a file-emitter object of one or other concrete class, depending on the command line options. All of the calls to the abstract methods of the *MsilFile* class are thus dispatched to the code that emits the appropriate output file. Figure 11 shows this class hierarchy.

The methods of class *IlasmFile* emit textual *CIL* to a file. The corresponding methods of class *PeFile* call methods of *PERWAPI*.

A consequence of using the same tree-walker for both *CIL* and *PE*-files is that the order in which nodes are visited is necessarily the same for each case. Since the *CIL* emitter writes to the output file "on the fly" as the methods are called, this is the more constrained version. The order in which objects are created by calls to *PERWAPI* is less constrained, since no output is actually produced until the *WritePEFile*() method of class *PEFile* is called.

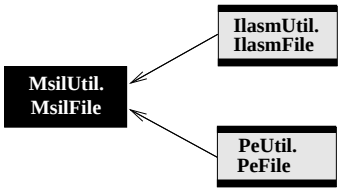

Figure 11: Class hierarchy for the *File-emitter* classes

There are a small number of other types that mirror the relationship between the various **File* types. For example, there is an abstract *Label* class that is sometimes held in objects of the program *AST*. In module *PeUtil* a concrete extension *PeLab* is defined. This new class holds a single field of type *CILLabel*. In the module *IlasmUtil* the corresponding concrete type *ILabel* has a single field of integer type. This integer determines the numeric suffix of the textual *CIL* labels – "`lb`*NNN*".

4.2 *PeFile* Emitter State

The emitter object carries state information about the traversal.

Some nodes of the program *AST* need to hold references to *PERWAPI* objects. For example, when the first reference to an imported class is made, a call to one of the *AddClass* methods of *PERWAPI* creates a class descriptor, and associates it with the appropriate *AssemblyRef* object. The *AddClass* method returns a reference to the newly created *ClassRef* object. Other references to the same class must use the same reference, either as the **this** of a call, or as an argument. It is therefore necessary to associate the returned *PERWAPI ClassRef* reference with the *gpcp*'s *Type* descriptor object in the *AST*.

The mechanism for associating *PERWAPI* object references with *gpcp*'s *AST* descriptors is as follows. Every *Idnt* descriptor object and every *Type* descriptor object contains a target extension field "`tgXtn`". These fields will be of different types, for different target platforms, and will have different types for different concrete subtypes of the abstract *Idnt* and *Type* types. The extension field is declared to be of *Object* type so every use of the field must use a narrowing cast. This design feature is necessary in order to separate the specifics of each target from the shared type declarations of the front-end *AST*.

The target extension fields of *AST* nodes hold *almost* all the state that is needed to call the *PERWAPI* methods. However, there are a small number of *PE*-file entities that have no corresponding *AST* descriptor. References to runtime system (*RTS*) helper routines are of this kind. There are also some shared descriptors that are used so universally that it makes sense to hold them locally, rather than having to repeatedly navigate through the *AST* objects.

The state information held in the emitter object has fields that are inherited from the abstract *MsilFile* parent class, and other fields that are specific to the *PeFile* class.

Inherited fields

Inherited fields hold the name of the module under compilation, and the name of the output file. There is also a field of *ProcInfo* class that holds information about the

current method being emitted.

The *ProcInfo* class contains method state that is used in both textual *CIL* and *PE*-file formats. It is here that the current stack depth is tracked, and the state of the temporary variable allocator is maintained[5].

PE-specific fields

The *PE*-file-specific fields of the emitter state, and their purpose is shown in Figure 12. As described earlier, the elements of the state exist for two main purposes. There are

Field	Type	Purpose
peFl	*PERWAPI.PEFile*	Structure holds file information, and the *AssemblyDef* for this assembly.
clsS	*PERWAPI.ClassDef*	Dummy static class for this assembly.
clsD	*PERWAPI.ClassDef*	Descriptor of class currently being emitted.
pePI	*PeUtil.PProcInfo*	*PE*-file-specific state for the method currently being emitted.
nmSp	*System.String*	Name-string for current namespace.
rts	*PERWAPI.AssemblyRef*	Reference to the *Component Pascal* runtime system assembly `[RTS]`.
cprts	*PERWAPI.ClassRef*	Reference to the *Component Pascal* runtime helper class `[RTS]CP_rts`.
progArgs	*PERWAPI.ClassRef*	Reference to the *Component Pascal* program argument class `[RTS]ProgArgs`.

Figure 12: Components of the *PeFile* state

fields that reference *PERWAPI* objects corresponding to runtime system classes that have no corresponding *AST* objects. Secondly, there are objects that refer to *PERWAPI* descriptors that are used repeatedly. One example is the field *clsD* that holds the *ClassDef* object for the output class currently being emitted. Another is the field that holds a reference to the "dummy static class" to which the static procedures of the *Component Pascal* module are bound. This dummy static class has no concrete representation in the *AST*. The existence of this dummy class in the *PE*-file is an artifact of the mapping from *Component Pascal* to the *CLR*.

The *PeUtil.PProcInfo* object holds state information for the method definition currently being emitted. Of course, this field will be `nil` throughout the *AST* traversal except while a method definition is being emitted.

The information that needs to be persisted while a method definition is being emitted is shown in Figure 13. The field *mthD* holds a reference to the current *MethodDef* descriptor. The field *code* holds a reference to the instruction buffer of the definition. Note that there is no way of extracting the buffer reference from the *ClassDef* reference, so it is necessary to hold this reference in the client.

The final field, *tryB*, holds a reference to the current *TryBlock*, if the code emission sequence is currently in a structured exception handling catch block. These blocks are held on a stack within *PERWAPI*, to account for the possibility of nested blocks.

[5] Recall that all uses of a particular local variable in the *CLR* must be of the same type. The utility that allocates temporary local variables therefore needs to track the currently allocated and free local variables, and the *CLR* data-types to which they have been bound.

Field	Type	Purpose
mthD	*PERWAPI.MethodDef*	The current method definition.
code	*PERWAPI.CILInstructions*	Instruction buffer of *mthD*.
tryB	*PERWAPI.TryBlock*	Current try block (or `nil`).

Figure 13: Components of the *PProcInfo* state

4.3 Creating Descriptors

Descriptors must be generated by calls to *PERWAPI* methods for all of the assemblies, classes and methods that need to appear in the *PE*-file. It is legal to create descriptors for entities that are not referenced in the file. However, it is bad policy to do so, since this practice needlessly expands the file size and slows down loading and JIT-ing.

Most compilers will have many descriptors in their *AST* representation that are unreferenced. This is almost inevitable, given the usual mechanisms for loading metadata from symbol or header files. There are at least two ways to avoid passing on any such unnecessary metadata to the *PE*-file. Firstly, it is possible to mark the used *AST* metadata during the semantic analysis phase of the compilation. Another possibility is to create the *PERWAPI* descriptors in a demand-driven manner. *gpcp* adopts the second approach.

Example – creating descriptors for RTS routines

gpcp emits calls to about 30 different runtime helper routines known to the compiler (as opposed to being explicitly imported by the source code). These include runtime routines that convert between the *CLR* string type and *Component Pascal*'s **array of** *CHAR* type. There are four separate routines that concatenate the various combinations of *String* and character arrays. There are also routines that generate runtime exception messages for failed **case** ("switch") and **with** ("type-case") statements.

It would be possible to generate *MethodRef* descriptors for all of these methods at initialization time[6], but this would insert unneeded metadata in the *PE*-file. Thus a demand driven approach is used. The various runtime helpers are accessed by means of an index value known to the front-end. A call to the method *getMethod* is passed the index of the required method, and returns the corresponding *MethodRef* descriptor.

Generation of a case statement trap[7] in textual *CIL* is shown in Figure 14. The call

```
ldloc.1                    // Push index of erroneous case
call   string [RTS]CP_rts::caseMesg(int32)
newobj instance void
           [mscorlib]System.Exception::.ctor(string)
throw                      // Throw the exception object
```

Figure 14: Textual *CIL* for case trap generation

[6] For an explanation of why it is necessary to repeat this initialization for every new output file, see the sidebox on page 33.

[7] In *Component Pascal* it is a runtime error if no case of a **case** statement is selected, and there is no explicit default case defined.

to the runtime system helper method "`[RTS]CP_rts::caseMesg`" is the instruction of interest here. The tree-walker will make a dispatched call to the abstract method *MsilFile.StaticCall* with the index of *caseMesg* as argument. In the *IlasmFile* override of this abstract method the index will select the text string shown in the second line of the code fragment. The *PeFile* override of the abstract method is shown in Figure 15. The routine simply fetches the required method descriptor by calling *getMethod*. It

```
PROCEDURE (os : PeFile)StaticCall(s : INTEGER);
  VAR mth : PERWAPI.Method;
BEGIN
  mth := os.getMethod(s);
  os.pePI.code.MethInst(opc_call, mth);
END StaticCall;
```

Figure 15: The *PeFile* version of *StaticCall*

then passes the descriptor to the *PERWAPI* method *MethInst*. This method appends a new *MethodOp* instruction to the current code buffer. The demand driven magic is all in the *getMethod* routine.

The procedure *getMethod* is backed by an array of method descriptors. The array initially holds `nil` at each index value. The procedure begins with a simple fetch of the selected array element. If the fetched array element is `nil` a **case** statement selects code that creates the required method descriptor, and stores it in the array. For all subsequent references to the same array element the stored value is returned with no further computation required.

The relevant branch of the **case** statement for our example is shown in Figure 16. In this case the needed routine has a single argument, so it is necessary to create an

```
PROCEDURE (os : PeFile)getMethod(ix : INTEGER) : MethodRef;
                              (* "os" is the named this *)
  VAR tArr : POINTER TO ARRAY OF PERWAPI.Type;
...
  mth := rHelper[ix];         (* look up descriptor array *)
  IF mth = NIL THEN           (* must create new MethodRef *)
    CASE ix OF
    ...
    | caseMesg :
        NEW(tArr, 1);         (* allocate length-one array *)
        tArr[0] := int32D;    (* int32D is TypeRef for int32 *)
        mth := os.cprts.AddMethod("caseMesg",strgD,tArr)
    ...                       (* strgD is TypeRef for String *)
    END; (* case *)
  END; (* if *)
  RETURN mth;
END getMethod;
```

Figure 16: Demand creation of RTS method descriptor

array of *PERWAPI.Type* of length one. The descriptor for the runtime system class is

fetched from the *PeFile* state, as described in Figure 12. The new method descriptor is added to this class with the *AddMethod* call. The first argument is the string holding the method name. The second argument is the descriptor of the method return type, *System.String* in this case. The final argument is the array of parameter types.

The other branches of the **case** statement in *getMethod* are similar. As might be expected, in the real code the creation and initialization of the parameter arrays is abstracted away into another method, rather than being inline as shown in Figure 16.

Finally, it should be noted that references to system routines, such as the constructors for *System.Exception* should be created on demand in a similar way.

Avoid this Nasty Gotcha!

When the first version of the *PEAPI*-based emitter for *gpcp* was written the code fell into a plausible but nasty trap. *gpcp* accepts any number of source file names on the command line, compiling each in turn. It seemed a plausible design decision to persist references to the runtime system method and class descriptors between files. The idea was to not have to repeatedly call *AddClass* and *AddMethod* for the same classes and methods for each source file compilation. Unfortunately this plausible strategy does not work. Worse still, it leads to extremely non-intuitive error behaviour.

The problem is that within the *PE*-file every reference is implemented by a table index. Whenever a new output file descriptor is allocated the table index allocation sequence is reset. It follows that descriptors that are persisted between files will have indices that refer to their ordinal position in the previous file. The resulting *PE*-files will almost certainly have totally nonsensical references.

4.4 ClassDefs and ClassRefs

Types that are explicitly referenced in the source code of the file being compiled are represented by nodes in the program *AST*. In this case the nodes themselves are able to hold references to the *PERWAPI* descriptor objects using their generic target extension "`tgXtn`" fields.

As before, these type descriptors are best generated on demand. In the case of *gpcp* the importation of metadata from the symbol files of imported modules clutters the *AST* symbol tables with unreferenced descriptors. Only the used types need have target extension objects allocated to them by calls to *PERWAPI* methods.

There are two functions that do all of the work. A method *typ*(*t*), where *t* is an *AST Type* descriptor, returns the *PERWAPI* type descriptor of its argument. If necessary it creates that descriptor. This method may be called on any *AST* type. The other method, *cls*(*t*), returns the *PERWAPI* class descriptor of its argument. In this case *t* must be an *AST* record type.

Target extensions for *AST* type descriptors

The "`tgXtn`" target extension fields for primitive types, arrays, pointers and enumerations simply hold the *PERWAPI Type* reference. The state for record and procedure types is more complicated, as multiple descriptors need to be created for each *AST* type.

AST record types correspond to *PERWAPI Class* types. These are represented by a structure with the fields shown in Figure 17. In this case, as well as the *ClassDef* or

Field	Type	Purpose
clsD	*PERWAPI.Class*	*CLR* class representing this record.
newD	*PERWAPI.Method*	No-arg constructor for this class.
cpyD	*PERWAPI.Method*	Deep copy method for this class.
boxD	*PERWAPI.Class*	Corresponding boxed class (value class only).
vDlr	*PERWAPI.Field*	Singleton field of boxed class (value class only).

Figure 17: Fields of the *RecXtn* structure for records

ClassRef it is necessary to hold references to the no-arg constructor, and to the field-by-field copy method. These last two fields are `nil` in the event that the semantics of the type forbid these operations. Note that all other methods of these types are explicit in the source code, and thus have their own *AST* descriptors to hold their own target extensions. In *Component Pascal* the no-arg constructor and the value-copy operations are implicit, and do not have concrete representation in the *AST*.

Procedure types in the *AST* correspond to *CLR* delegate types. Delegates are *PERWAPI Class* types, and have two runtime managed methods. The state for these types is represented by a structure with the fields shown in Figure 18. The first of the methods

Field	Type	Purpose
clsD	*PERWAPI.Class*	*CLR* class representing this delegate type.
newD	*PERWAPI.Method*	Constructor for this class.
invD	*PERWAPI.Method*	*Invoke* method for this delegate.

Figure 18: Fields of the *DelXtn* structure for procedure types

is the constructor method, which takes an *Object* as its first argument. As its second argument the constructor takes the *native int* returned by the immediately preceeding "`ldftn`" instruction. The second method is named *Invoke*, and has a signature that matches that of the procedure values that the delegate encapsulates.

Creating the descriptors

As described above, type descriptors are created on demand, as a side-effect of calling the *typ*() and *cls*() functions. The code of the *typ* method is shown in Figure 19. In this code, if the target extension field is `nil` the *MkTyXtn* method is invoked. This allocates a type descriptor of whatever *PERWAPI* type corresponds to the particular *AST* type. Finally, a type-case statement returns the target extension field or the appropriate class of the target extension object.

The code inside *MkTyXtn* is specialized according to the *AST* type. Type descriptors of *Base* type correspond to primitive types of the *CLR*. The target extension fields are assigned by accessing the built-in type descriptors of *PERWAPI*. For example the field for the *AST* base descriptor for the *CHAR* type is assigned by —

```
t.tgXtn := PERWAPI.PrimitiveType.Char;
```

The other base types are similar.

```
PROCEDURE (pf : PeFile)typ(tTy : Api.Type) : PERWAPI.Type;
(* Returns (and maybe creates) the PERWAPI.Type for the AST type tTy *)
  VAR xtn : ANYPTR;          (* aka System.Object *)
BEGIN
  IF tTy.tgXtn = NIL THEN  (* create new descriptor *)
       pf.MkTyXtn(tTy) END;(* MkTyXtn selects on AST type *)
  xtn := tTy.tgXtn;          (* fetch extension field *)
  WITH xtn : PERWAPI.Type DO (* Type-case statement... *)
      RETURN xtn;            (* Base, Array, Pointer, Enum *)
  | xtn : RecXtn DO          (* "elsif xtn is RecXtn do ..." *)
      RETURN xtn.clsD;       (* tTy is an AST Record type *)
  | xtn : DelXtn DO          (* "elsif xtn is DelXtn do ..." *)
      RETURN xtn.clsD;       (* tTy is an AST Procedure type *)
  END;
END typ;
```

Figure 19: Demand creation of type descriptors

AST type descriptors of *Array* type create *PERWAPI* types by calls to the *PERWAPI ZeroBasedArray* constructor. In the *C#* syntax the call would be —

```
t.tgXtn = new PERWAPI.ZeroBasedArray(this.typ(t.elemTp));
```

where the constructor argument is the type descriptor of the element type. Note the recursive call of the *typ* method here.

The creation of the type descriptors for pointer types is slightly more complicated, since it depends on certain artifacts of the *Component Pascal* to *CLR* mapping. It may be helpful to review Chapter 4 of *Compiling for the .NET Common Language Runtime* in this context.

If the bound type of the pointer type is an array type, then the target extension field of the pointer type is simply copied from the target extension field of the bound type[8].

If the bound type is a record type, then two different cases arise. If the bound type is implemented by a reference surrogate, that is, by a *reference* class in the *CLR*, then the record and pointer type share the same runtime representation. In that case, the target extension field of the pointer type is copied from the *clsD* field of the *RecXtn* reference of the record type. On the other hand, if the bound type is represented in the *CLR* by a *value* class then the pointer type is represented by the corresponding named, boxed type. The target extension field of the pointer type is therefore copied from the *boxD* field of the *RecXtn* reference of the record type.

The last of the type kinds with scalar target extension field, is the enumerations. If an enumeration is defined in a different assembly a *TypeRef* must be created. In order to do this it is first necessary to fetch the corresponding *AssemblyRef* descriptor. This is done by another method, *asm*, which returns the *AssemblyRef* reference held in the *AST* module descriptor. In keeping with our demand-driven strategy, the function creates the *AssemblyRef* if necessary, by a call to *AddExternAssembly*. The target extension field is finally created thus —

```
t.tgXtn := this.asm(mod).AddValueClass(nsNm, tyNm);
```

[8] In verifiable code **array of *T*** is implemented by a reference surrogate, and uses the same *CLR* type as **pointer to array of *T***.

In this assignment *mod* is the *AST* module descriptor, and *nsNm*, *tyNm* are strings respectively holding the namespace and typename of the enumeration type. If it is a *TypeDef* that is being created, rather than a *TypeRef*, a different *AddValueClass* method is dispatched, this time on the *PEFile* object. Note carefully that enumerations are *value* classes, so it is convenient to use one of the *AddValueClass* methods, rather than the usual calls to *AddClass*.

4.5 MethodDefs and MethodRefs

As code is generated for methods of a module, method references and method definitions need to be generated. With *gpcp*, even in the case of method definitions it is possible that a used occurrence of the *MethodDef* object might occur before the definition of the method. Therefore, as before, a demand-driven approach is used for the creation of *Method* descriptors.

During *AST* traversal definitions are emitted for every method defined in the source of the module. If no *MethodDef* object has been created for the *AST* procedure descriptor, then a *MethodDef* object is allocated and stored in the target extension field of the *AST* object. In any case, once the *MethodDef* object has been retrieved the *MethAttr* and *ImplAttr* attributes are added, and a code buffer allocated. Subsequent traveral of the *AST* for the procedure body adds instructions to this buffer.

As instructions are added to the code buffer for the current method, references to other methods are used as arguments to *MethodOp* instructions. The *PERWAPI* method descriptor for the target method is extracted from the *AST* descriptor by a call to another utility method *mth*.

For *PE*-files the *Method* descriptor objects are created in a demand-driven way. In the case of textual-*CIL* output the text-strings that hold the signature information of methods are created in a similar demand-driven way. A procedure *MkCallAttr* in the *MsilUtil* module is called from the tree-walker. This procedure calls the abstract procedure *NumberParams*. In module *IlasmUtil* the overriding procedure numbers the formal parameters of the called method, and computes the signature string of the called method. In module *PeUtil* the overriding procedure numbers the formal parameters of the called method, and creates a *MethodDef* or *MethodRef* object, as appropriate.

PeUtil.NumberParams retrieves the *Class* object with which the called procedure is associated. A type-case statement dispatches the appropriate factory procedure —

```
with clsD : PERWAPI.ClassDef do
    methD := MkMethDef(...);
| clsD : PERWAPI.ClassRef do
    methD := MkMethRef(...);
end;
```

If the target procedure is an instance method or a constructor, then the appropriate call convention marker must be added —

```
if ... then methD.AddCallConv(PERWAPI.CallConv.Instance) end;
```

Within the *MkMthDef* and *MkMthRef* procedures the formal parameter arrays are created. These will be arrays of *Param* or *Type* objects respectively. In the case of formal parameters that are passed by reference, it is at this point that the managed pointer descriptor *Type* objects are created from the type descriptors of the formal types in the *AST*, as described on page 20.

4.6 Notes

Details on the structure and format of *PE*-files may be found in Partition II of the *ECMA* standard for the *CLI*. Serge Lidin's excellent book *The ILASM Assembler*, Microsoft Press, 2002, is another invaluable resource.

There is a useful trick to help with debugging compiler backends that use *PERWAPI*. Given that the output files from the current version have no debug information, it is sometimes difficult to find exactly where a problem originates. If the *PE*-file is "round-tripped" through `ildasm` and `ilasm` then the graphical debugger of the Software Development Kit will be able to step through the *PE*-file line-by-line, if necessary. Of course, this will be line-by-line through the textual-*CIL*. But that is usually all that is needed to locate a problem. First, disassemble the file —

`ildasm /out=`*file*`.out` *file*`.DLL`

Then re-assemble the file, using the /debug command-line flag —

`ilasm /DLL /debug` *file*`.out`

Html documentation is supplied as part of the distribution of *PERWAPI*. As well, for those needing to access the component from *Component Pascal*, the *gpcp*-format symbol file is included in the current *gpcp* distribution. This symbol file is named "`PERWAPI_.cps`". The symbol file was created by running the *PeToCps* tool over the "`PERWAPI.dll`" file. A browsable html rendering of this symbol file has been created using the *gpcp* standard *Browse* tool.