Appendix C

Using PEAPI to Write PE Files

John Gough & Diane Corney

Version 1.0 (1 September 2002)

C.1 Introduction

Chapter 11 of *Compiling for the .NET Common Language Runtime* "Skipping the Assembler: Using Reflection.Emit" discusses the use of the library "`Reflection.Emit`" to create program executable module (*PEM*) files directly. In that Chapter the shortcomings of that approach were discussed.

This appendix details another approach based on a managed component, *PEAPI*, created by Diane Corney. *PEAPI* supplies an application programming interface that allows *PE*-files to be directly created. *PEAPI* is entirely managed code, and is written in *C#*. It has been used as a backend for **gpcp**, with promising results. It is relatively well tested for those parts of the functionality that are required for **gpcp**, but is intended to be able to emit all features of *PE*-files. Testing of those features not directly used by **gpcp** is ongoing. The authors will update the source as required.

Like **gpcp**, *PEAPI* is open source software, and is released under the same license. This Chapter is intended to fulfil two purposes: it is a guide to the use of *PEAPI*, with the interface to **gpcp** as a running example. In addition, this Chapter is intended as a brief introduction to the design and implementation of the component.

Performance

Compilers using *PEAPI* are fast. As a rough rule of thumb, **gpcp** can write out a *PE*-file 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.

Figure C.1 lists the time taken to compile the roughly 40k lines of *Component Pascal* in the 46 files of the **gpcp** source. The time in seconds is for a single processor

Writing textual IL files	4.2
Writing IL, then invoking `ilasm`	11.5
Writing PE files using *PEAPI*	4.0

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

2001 Intel Pentium machine with 512M of memory. The final row thus corresponds to a compilation speed of well over 500k lines per minute.

Figure C.2 lists the compilation times for the three largest single modules of **gpcp**. In each case the compilation time of each module compiled on it own is about 1100mSec. 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 second 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	160	220	161
Writing IL, then invoking `ilasm`	631	691	610
Writing PE files using *PEAPI*	160	241	150

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

Limitations

Currently *PEAPI* does not create program data base (*PDB*) files for the debugger. Since the format of the *PDB* files is not public it does not seem possible to create such files with an open-source managed component. There are several other possibilities that are being considered. *PEAPI* could use the unmanaged interface to create debug files, or it could use the public debug format used by the "Rotor" shared source *CLI* implementation. It is hoped to resolve this issue before the next release of *PEAPI*.

C.2 *PEAPI* Structure

The file format of *PE*-files is based around a set of tables. It is not necessary to understand the format of these files in order to use *PEAPI*. There are at least two excellent references for those who choose to explore under the hood of the file format, and the source code of *PEAPI* is helpful also. Rather, it is the goal of this Section to describe the internal data structures of the component, in order to understand how the *API* is used.

As various calls to the methods of *PEAPI* are made, the component builds a tree-like representation of the module. Classes, fields, methods and code are added to the internal representation by the method calls. Finally, when all the features of the module

have been added a call to the *WritePEFile* method builds the tables and writes the binary file to the output stream.

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 *PEAPI* to allow for the definition of each of these kinds of entity, and their association with their enclosing container.

Assemblies and Modules

The root of the tree-like structure constructed by *PEAPI* is an object of *PEFile* type. This object contains a module definition, and optionally defines an assembly also. Files that define an assembly contain an assembly manifest. This root object of *PEFile* class is the object on which the final call of *WritePEFile* is made.

All name resolution for classes is based on a particular *ResolutionScope*. Resolution scope is represented by an abstract class. There are four concrete classes that derive from this class. The relationship of these types is shown in Figure C.3. In all

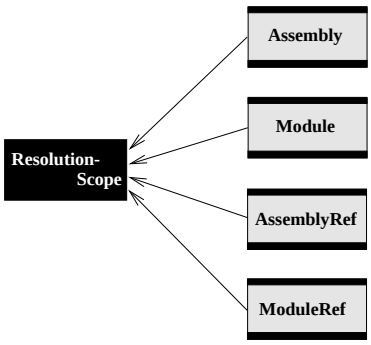


Figure C.3: Class hierarchy for the *resolution scope* descriptors

of these class hierarchy diagrams black rectangles denote abstract classes. Extensible classes are shown unshaded, while sealed classes are shown lightly shaded.

When a particular file is compiled, names that are resolved in the scope of the definitions of that file have a *Module* object representing their resolution scope. This corresponds to the default case in textual-*CIL* in which dotted names are unqualified —

```
NameSpaceName.ClassName
```

Names that are resolved with respect to some external assembly have an *AssemblyRef* object to represent their resolution scope. This corresponds to the use of names in textual-*CIL* that are qualified by an assembly name —

```
[SomeAssembly]NameSpaceName.ClassName
```

Names that are resolved with respect to another module in the same assembly as the current module have a *ModuleRef* object to represent their resolution scope. This corresponds to the use of names in textual-*CIL* that are qualified by a module reference —

```
[.module FileName]NameSpaceName.ClassName
```

The call interface for the creation of all of these objects is described in Section C.3.

Type Descriptors

All types in *PEAPI* are represented in the tree by objects that derive from the abstract class *PEAPI.Type*. The rich hierarchy of derived classes is shown in Figure C.4. Figure C.4 is actually slightly simplified by the omission of the *MethPtrType* and *Cus-*

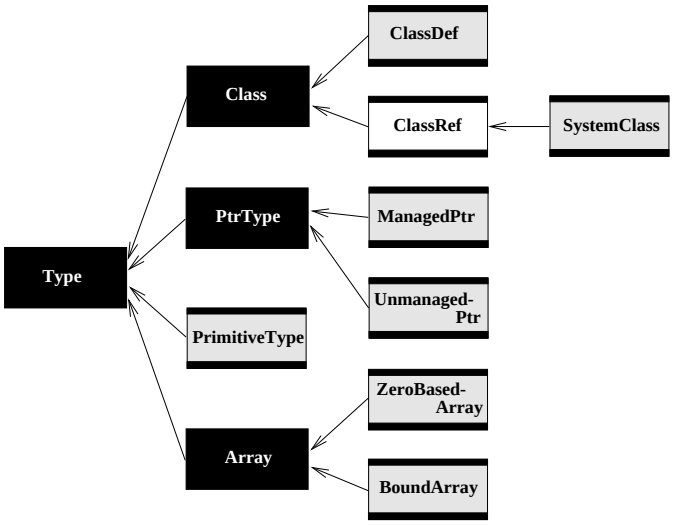


Figure C.4: Class hierarchy for the *PEAPI Type* descriptors

tomModifiedType classes.

The class *Class* is another abstract type, with two directly derived extensions. *ClassDef*s are always associated with the current *PEFile* object or, in the case of a nested classes, with another *ClassDef* within which the class is nested. *ClassRef*s are associated with an *AssemblyRef* or with a *ModuleRef*. System types are a specialization of the *ClassRef* class, and represent the built-in types from the *System* namespace of the "`mscorlib`" assembly. This particular class is not exposed to the *API*.

Primitive types have built-in type descriptors that are created by *PEAPI*, and are named static constant values of the type.

Pointer types occur as objects derived from the abstract *PtrType* class. Only *ManagedPtr* types are used in verifiable code.

Finally, two concrete classes are derived from the abstract *Array* class. 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.

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 C.5. Members include

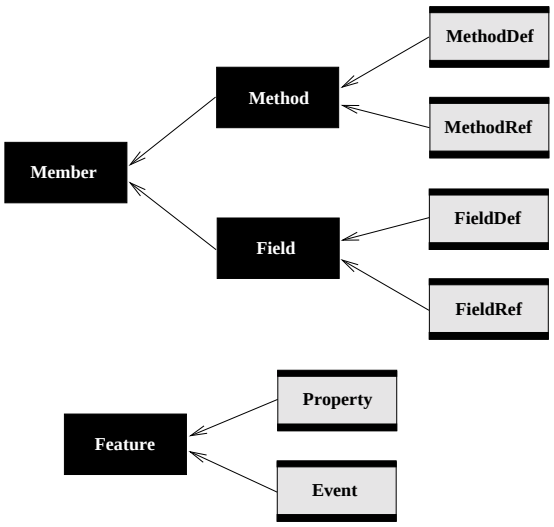


Figure C.5: Class hierarchy for the *Members* and *Features* of classes

methods 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 *PEAPI* these 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*
* *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.

Representing the Code

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 C.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. *PEAPI* 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 *PEAPI* 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.

Structured exception blocks

There are two mechanisms within *PEAPI* to define structured exception handling blocks. The start and end of the blocks may be marked in the instructions buffers. This is proba-

bly 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, *PEAPI* 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 called, so that later exception handling blocks may be logically associated with the code region contained within the block.

C.3 The Call Interface

This Section discusses the call interface of *PEAPI*. 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.

PEAPI 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 necessary 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, there is no "*get_Buffer*" method to *retrieve* the buffer associated with a particular *MethodDef*. 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 *PEAPI* builds is of *PEFile* class. There are three separate constructor methods that create these objects —

```
public PEFile(string name, bool isDLL, bool hasAssembly)
```

This constructor creates the output file for the compilation, and initializes the data. In particular, an object of *Module* class is created as the default resolution scope for the module. The name parameter specifies both the module name and the filename, with the first Boolean argument specifying if the file will have extension "`DLL`" or "`EXE`". The final Boolean specifies if the output file will define an assembly, and therefore contain an assembly manifest. By default, this constructor creates a file in the current working directory.

An alternative constructor has an fourth argument of string type that specifies the directory in which the *PE*-file will be created. If the final argument is the empty string the output file will be created in the current directory.

The final constructor has an alternative fourth argument supplies a reference to an output stream. This constructor does not create a file.

Most of the operations that add children to the root object are dispatched directly on the *PEFile* object. However, if the *Module* object is needed then it may be retrieved by the call —

```
public Module GetThisModule()
```

Similarly, in the event that the file defines an *Assembly* object, this may be retrieved by the call —

```
public Assembly GetThisAssembly()
```

Declaring Resolution Scopes

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

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

```
public AssemblyRef AddExternAssembly(string asmName)
```

The argument specifies the name of the external assembly. *PEAPI* always creates an assembly reference for the system assembly "`mscorlib`". The *AddExternAssembly* 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 AddExternModule(string name)
```

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.

Note that the client code calling these object creation methods must retain the object references that are returned. There are no exposed "lookup" methods to extract such references from the root object.

Creating *ClassDef*s and *ClassRef*s

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 *Module* resolution scope. There are three 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[1]. 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.

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 class creation methods are called on existing *ClassDef* objects then nested classes are defined. The semantics of the creation 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.

ClassDef descriptors for nested classes are defined in the same way as other class definitions, except that the *AddClass* methods are dispatched on the *ClassDef* object of the outer, enclosing class.

[1] It is worth recalling that as far as the *CLI* is concerned, classes have "dotted names". The separation of the name into a possibly dotted namespace name and a simple class identifier is a matter of convenience only.

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, and is not otherwise accessible from the *API*. Of course, it is also possible to use the *AddClass* method that takes an explicit super class, and pass in the descriptor for *System.ValueType*.

Creating descriptors for other *Type*s

As well as the types that are declared as *CLR* classes, there are a number of other types that need descriptors. Figure C.4 on page iv 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. The managed case constructor has the following signature —

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

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

This constructor is frequently called, even in verifiable code. For example, the *CLR* 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 (final) 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 the 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 may supply the name and type of the field only. 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)
```

Using system classes

We refer to classes belonging to the *System* namespace of the "`mscorlib`" assembly as *System Classes*. Classes corresponding to the primitive types are treated specially by *PEAPI*, and are represented internally by objects that belong to a subclass of *ClassRef*.

The reason for this is that *PEAPI* constructs an implicit import of the "`mscorlib`" assembly, and constructs class definitions for the built-in types. Calls to add new class references to the system namespace need to be treated differently, to eliminate the possibility of duplicate table entries.

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 C.6 shows the correspondence between the definition of an event in textual-*CIL* and the *PEAPI* calls that are made to generate the same effect using *PEAPI*. 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.

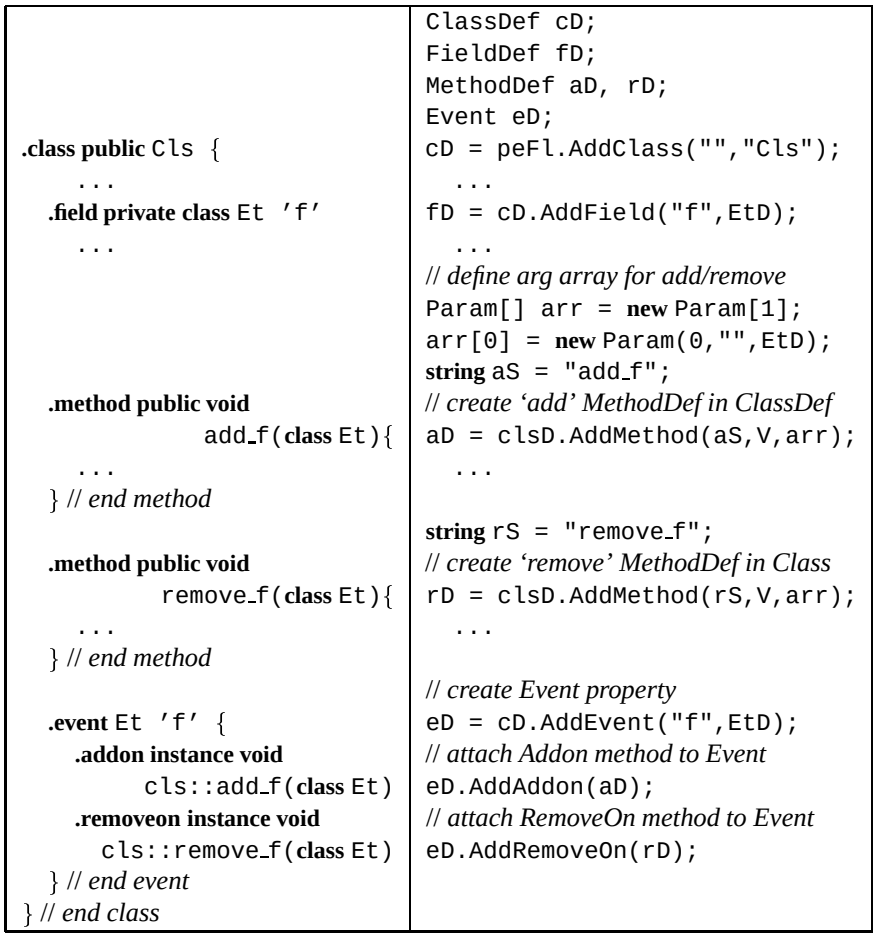
On the right of the Figure, there are a number of local variables that hold references to the various *PEAPI* objects between their definitions and uses. Thus *fD* is the *FieldDef* of the private backing field *f*, *aD* and *rD* are the *MethodDef* descriptors of the add and remove methods, and so on.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

The definition of *Property* features is similar, with *PEAPI* methods that add the "getter", "setter" and "other" methods to the property, analogous to the *AddAddon* method of *Event* features.

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.



```
                                    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.AddAddon(aD);
    .removeon instance void         // attach RemoveOn method to Event
      cls::remove_f(class Et)       eD.AddRemoveOn(rD);
  } // end event
} // end class
```

Figure C.6: Defining an event in textual-*CIL* (left) and *PEAPI* (right)

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 vi. 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 C.7. 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 C.7: 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 C.8 shows the textual-*CIL* on the left, and the corresponding *PEAPI* method 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

`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 C.8: Example switch statement in textual-*CIL* (left) and *PEAPI* calls (right)

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.

Structured Exception Handling

Since structured exception handling blocks may be textually nested, *PEAPI* 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.

C.4 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 C.9 shows the class hierarchy for the tree-walker modules.

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

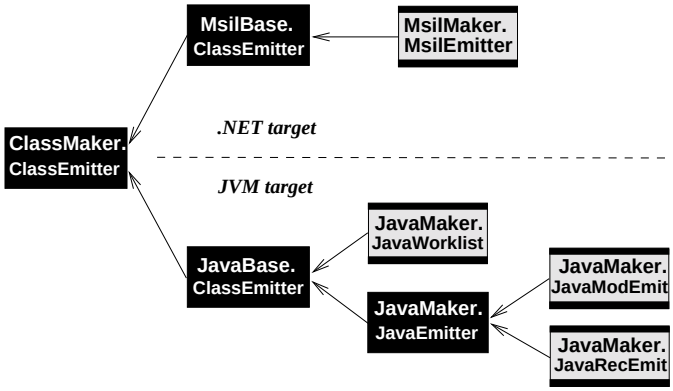


Figure C.9: Class hierarchy for the *Tree-walker* classes

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 C.10 shows this class hierarchy.

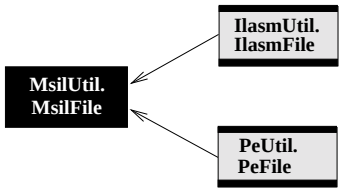


Figure C.10: Class hierarchy for the *File-emitter* classes

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

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 *PEAPI* is less constrained, since no output is actually produced until the *WritePEFile*() method of class *PEAPI.PEFile* is called.

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 *PEAPI.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*".

PeFile Emitter State

The emitter object carries state information about the traversal.

Some nodes of the program *AST* need to hold references to *PEAPI* objects. For example, when the first reference to an imported class is made, a call to one of the *AddClass* methods of *PEAPI* 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 *PEAPI ClassRef* reference with the **gpcp**'s *Type* descriptor object in the *AST*.

The mechanism for associating *PEAPI* 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 *PEAPI* 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[2].

PE-specific fields

The *PE*-file-specific fields of the emitter state, and their purpose is shown in Figure C.11. As described earlier, the elements of the state exist for two main purposes.

Field	Type	Purpose
peFl	*PEAPI.PEFile*	Structure holds file information, and the *AssemblyDef* for this assembly.
clsS	*PEAPI.ClassDef*	Dummy static class for this assembly.
clsD	*PEAPI.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	*PEAPI.AssemblyRef*	Reference to the *Component Pascal* runtime system assembly `[RTS]`.
cprts	*PEAPI.ClassRef*	Reference to the *Component Pascal* runtime helper class `[RTS]CP_rts`.
progArgs	*PEAPI.ClassRef*	Reference to the *Component Pascal* program argument class `[RTS]ProgArgs`.

Figure C.11: Components of the *PeFile* state

There are fields that reference *PEAPI* objects corresponding to runtime system classes that have no corresponding *AST* objects. Secondly, there are objects that refer to *PEAPI* descriptors that are used repeatedly. One example is the field *clsD* that holds the *PEAPI.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 C.12. 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 *PEAPI.TryBlock*, if the code emission sequence is currently in a structured exception handling catch block. These blocks are held on a stack within *PEAPI*, to account for the possibility of nested blocks.

[2]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	*PEAPI.MethodDef*	The current method definition.
code	*PEAPI.CILInstructions*	Instruction buffer of *mthD*.
tryB	*PEAPI.TryBlock*	Current try block (or **nil**).

Figure C.12: Components of the *PProcInfo* state

Creating Descriptors

Descriptors must be generated by calls to *PEAPI* 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 meta-data 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 *PEAPI* 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 *PEAPI.MethodRef* descriptors for all of these methods at initialization time[3], 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[4] in textual *CIL* is shown in Figure C.13. The call 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 C.14.

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

[4]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.

```
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 C.13: Textual *CIL* for case trap generation

The routine simply fetches the required method descriptor by calling *getMethod*. It

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

Figure C.14: The *PeFile* version of *StaticCall*

then passes the descriptor to the *PEAPI* 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 C.15. In this case the needed routine has a single argument, so it is necessary to create an array of *PEAPI.Type* of length one. The descriptor for the runtime system class is fetched from the *PeFile* state, as described in Figure C.11. 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 C.15.

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.

```
PROCEDURE (os : PeFile)getMethod(ix : INTEGER) : MethodRef;
                              (* "os" is the named this *)
  VAR tArr : POINTER TO ARRAY OF PEAPI.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 C.15: Demand creation of RTS method descriptor

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.

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 *PEAPI* 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 *PEAPI* methods.

There are two functions that do all of the work. A method *typ*(*t*), where *t* is an *AST Type* descriptor, returns the *PEAPI* 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 *PEAPI* 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 *PEAPI 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 *PEAPI Class* types. These are represented by a structure with the fields shown in Figure C.16. In this case, as well as the *ClassDef* or

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

Figure C.16: 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 *PEAPI Class* types, and have two runtime managed methods. The state for these types is represented by a structure with the fields shown in Figure C.17. The first of the methods

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

Figure C.17: 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 C.18. In this code, if the target extension field is **nil** the *MkTyXtn* method is invoked. This

```
PROCEDURE (pf : PeFile)typ(tTy : Api.Type) : PEAPI.Type;
(* Returns (and maybe creates) the PEAPI.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 : PEAPI.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 C.18: Demand creation of type descriptors

allocates a type descriptor of whatever *PEAPI* 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 *PEAPI*. For example the field for the *AST* base descriptor for the *CHAR* type is assigned by —

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

The other base types are similar.

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

```
t.tgXtn = new PEAPI.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 [5].

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);
```

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*.

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 *PEAPI* method descriptor for the target method is extracted from the *AST* descriptor by a call to another utility method *mth*.

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

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 : PEAPI.ClassDef do
   methD := MkMethDef(...);
| clsD : PEAPI.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(PEAPI.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 xi.

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 *PEAPI*. Given that the output files 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 *PEAPI*. 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 "`PEAPI_.cps`". The symbol file was created by running the *N2CPS* tool over the "`PEAPI.dll`" file. A browsable html rendering of this symbol file has been created using the **gpcp** standard *Browse* tool.